JAKARTA EE

Jakarta EE Platform

Jakarta EE Platform Team, https://projects.eclipse.org/projects/ee4j.jakartaee-platform

8, August 26, 2019

Table of Contents

```
Specification: Jakarta EE Platform

Version: 8

Status: Final Release

Release: August 26, 2019
```

Copyright

Eclipse Foundation Specification License

Disclaimers

Chapter 1. Introduction

Enterprises today need to extend their reach, reduce their costs, and lower the response times of their services to 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. The 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 ensure that business transactions are accurately and promptly processed.

In most cases, enterprise services are implemented as multitier __ applications. The middle tiers integrate existing EISs with the business functions and data of the new service. Maturing web technologies are used to provide first tier users with easy access to business complexities, and eliminate or drastically reduce user administration and training.

The Jakarta™ EE Platform reduces the cost and complexity of developing multitier, enterprise services. Jakarta EE applications can be rapidly deployed and easily enhanced as the enterprise responds to competitive pressures.

Jakarta EE achieves these benefits by defining a standard architecture with the following elements:

- **Jakarta EE Platform** - A standard platform for hosting Jakarta EE applications.
- **Jakarta EE Compatibility Test Suite** - A suite of compatibility tests for verifying that a Jakarta EE platform product complies with the Jakarta EE platform standard.
- **Jakarta Compatible Implementations** - Certified implementations for building and deploying Jakarta EE applications.

This document is the Jakarta EE platform specification. It sets out the requirements that a Jakarta EE platform product must meet.

1.1. Acknowledgements for the Initial Version of Java EE

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.

1.2. Acknowledgements for Java EE 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, Borland, Bull S.A., Exoffice, Fujitsu Limited, GemStone Systems, Inc., IBM, Inline Software, 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. Alfred Towell provided significant editorial assistance with this version.

1.3. Acknowledgements for Java EE Version 1.4

Version 1.4 of this specification was created under the Java Community Process as JSR-151. The JSR-151 Expert Group included the following members: Larry W. Allen (SilverStream Software), Karl Avedal (Individual), Charlton Barreto (Borland Software Corporation), Edward Cobb (BEA), Alan Davies (SeeBeyond Technology Corporation), Sreeram Duvvuru (iPlanet), B.J. Fesq (Individual), Mark Field (Macromedia), Mark Hapner (Sun Microsystems, Inc.), Pierce Hickey (IONA), Hemant Khandelwal (Pramati Technologies), Jim Knutson (IBM), Elika S. Kohen (Individual), Ramesh Loganathan (Pramati Technologies), Jasen Minton (Oracle Corporation), Jeff Mischkinsky (Oracle Corporation), Richard Monson-Haefel (Individual), Sean Neville (Macromedia), Bill Shannon (Sun Microsystems, Inc.), Simon Tuffs (Lutris Technologies), Jeffrey Wang (Persistence Software, Inc.), and Ingo Zenz (SAP AG). My colleagues at Sun provided invaluable assistance: Umit Yalcinalp converted the deployment descriptors to XML Schema; Tony Ng and Sanjeev Krishnan helped with transaction requirements; Jonathan Bruce helped with JDBC requirements; Suzette Pelouch, Eric Jendrock, and Ian Evans provided editorial assistance. Thanks also to all the external reviewers, including Jeff Estefan (Adecco Technical Services).

1.4. Acknowledgements for Java EE Version 5

Version 5 (originally known as version 1.5) of this specification was created under the Java Community Process as JSR-244. The JSR-244 Expert Group included the following members: Kilinc Alkan (Individual), Rama Murthy Amar Pratap (Individual), Charlton Barreto (Individual), Michael Bechauf (SAP AG), Florent Benoit (INRIA), Bill Burke (JBoss, Inc.), Muralidharan Chandrasekaran (Individual), Yongmin Chen (Novell, Inc.), Jun Ho Cho (TmaxSoft), Ed Cobb (BEA), Ugo Corda (SeeBeyond Technology Corporation), Scott Crawford (Individual), Arulazi Dhesiaseelan (Hewlett-Packard Company), Bill Dudney (Individual), Francois Exertier (INRIA), Jeff Genender (The Apache Software Foundation), Evan Ireland (Sybase, Inc.), Vishy Kasar (Borland Software Corporation), Michael Keith (Oracle Corporation), Wonseok Kim (TmaxSoft, Inc.), Jim Knutson (IBM), Elika Kohen (Individual), Felipe Leme (Individual), Geir Magnusson Jr. (The Apache Software Foundation), Scott Marlow (Novell, Inc.), Jasen Minton (Oracle Corporation), Jishnu Mitra (Borland Software Corp), David Morandi (E.piphany), Nathan Pahucki (Novell, Inc.), David Morandi (E.piphany, Inc.), Ricardo Morin (Intel Corporation), Nathan Pahucki (Novell, Inc.), Matt Raible (Individual), Dirk Reinshagen (Individual), Narinder Sahota (Cap Gemini), Suneet Shah (Individual), Bill Shannon (Sun Microsystems, Inc.), Rajiv Shivane (Pramati

Technologies), Scott Stark (Jboss, Inc), Hani Suleiman (Ironflare AB), Kresten Krab Thorup (Trifork), Ashish Kumar Tiwari (Individual), Sivasundaram Umapathy (Individual), Steve Weston (Cap Gemini), Seth White (BEA Systems), and Umit Yalcinalp (SAP AG). Once again, my colleagues at Sun provided invaluable assistance: Roberto Chinnici provided draft proposals for many issues related to dependency injection.

1.5. Acknowledgements for Java EE Version 6

Version 6 of this specification was created under the Java Community Process as JSR-316. The spec leads for the JSR-316 Expert Group were Bill Shannon (Sun Microsystems, Inc.) and Roberto Chinnici (Sun Microsystems, Inc.). The expert group included the following members: Florent Benoit (Inria), Adam Bien (Individual), David Blevins (Individual), Bill Burke (Red Hat Middleware LLC), Larry Cable (BEA Systems), Bongjae Chang (Tmax Soft, Inc.), Rejeev Divakaran (Individual), Francois Exertier (Inria), Jeff Genender (Individual), Antonio Goncalves (Individual), Jason Greene (Red Hat Middleware LLC), Gang Huang (Peking University), Rod Johnson (SpringSource), Werner Keil (Individual), Michael Keith (Oracle), Wonseok Kim (Tmax Soft, Inc.), Jim Knutson (IBM), Elika S. Kohen (Individual), Peter Kristiansson (Ericsson AB), Changshin Lee (NCsoft Corporation), Felipe Leme (Individual), Ming Li (TongTech Ltd.), Vladimir Pavlov (SAP AG), Dhanji R. Prasanna (Google), Reza Rahman (Individual), Rajiv Shivane (Pramati Technologies), Hani Suleiman (Individual).

1.6. Acknowledgements for Java EE Version 7

Version 7 of this specification was created under the Java Community Process as JSR-342. The Expert Group work for this specification was conducted by means of the *http://javaee-spec.java.net* project in order to provide transparency to the Java community. The specification leads for the JSR-342 Expert Group were Bill Shannon (Oracle) and Linda DeMichiel (Oracle). The expert group included the following members: Deepak Anupalli (Pramati Technologies), Anton Arhipov (ZeroTurnaround), Florent Benoit (OW2), Adam Bien (Individual), David Blevins (Individual), Markus Eisele (Individual), Jeff Genender (Individual), Antonio Goncalves (Individual), Jason Greene (Red Hat, Inc.), Alex Heneveld (Individual), Minehiko Iida (Fujitsu), Jevgeni Kabanov (Individual), Ingyu Kang (Tmax Soft, Inc.), Werner Keil (Individual), Jim Knutson (IBM), Ming Li (TongTech Ltd.), Pete Muir (Red Hat, Inc.), Minoru Nitta (Fujitsu), Reza Rahman (Caucho Technology, Inc), Kristoffer Sjogren (Ericsson AB), Kevin Sutter (IBM), Spike Washburn (Individual), Kyung Koo Yoon (TmaxSoft).

1.7. Acknowledgements for Java EE Version 8

Version 8 of this specification was created under the Java Community Process as JSR-366. The Expert Group work for this specification was conducted by means of the *http://javaee-spec.java.net* and *https:javaee.github.io/javaee-spec* projects in order to provide transparency to the Java community. The specification leads for the JSR-366 Expert Group were Bill Shannon (Oracle) and Linda DeMichiel (Oracle). The expert group included the following members: Florent Benoit (OW2), David Blevins (Tomitribe), Jeff Genender (Savoir Technologies), Antonio Goncalves (Individual), Jason Greene (Red Hat), Werner Keil (Individual), Moon Namkoong (TmaxSoft, Inc.) Antoine Sabot-Durand (Red Hat), Kevin Sutter (IBM), Ruslan Synytsky (Jelastic, Inc.), Markus Winkler (oparco - open architectures &

consulting). Reza Rahman (Individual) participated as a contributor.

1.8. Acknowledgements for Jakarta EE 8

The Jakarta EE 8 specification was created by the Jakarta EE Working Group within the Eclipse Foundation *https://jakarta.ee/*

Chapter 2. Platform Overview

This chapter provides an overview of the Jakarta™ EE Platform.

2.1. Architecture

The required relationships of architectural elements of the Java EE platform are shown in *Jakarta EE Architecture Diagram*. Note that this figure shows the logical relationships of the elements; it is not meant to imply a physical partitioning of the elements into separate machines, processes, address spaces, or virtual machines.

The Containers, denoted by the separate rectangles, are Jakarta EE runtime environments that provide required services to the application components represented in the upper half of the rectangle. The services provided are denoted by the boxes in the lower half of the rectangle. For example, the Application Client Container provides Jakarta Messaging APIs to Application Clients, as well as the other services represented. All these services are explained below. See Jakarta EE Standard Services.

The arrows represent required access to other parts of the Jakarta EE platform. The Application Client Container provides Application Clients with direct access to the Jakarta EE required Database through the Java API for connectivity with database systems, the JDBC™ API. Similar access to databases is provided to server pages, server faces applications, and servlets by the Web Container, and to enterprise beans by the Enterprise Beans Container.

As indicated, the APIs of the Java™ Platform, Standard Edition (Java SE), are supported by Java SE runtime environments for each type of application component.

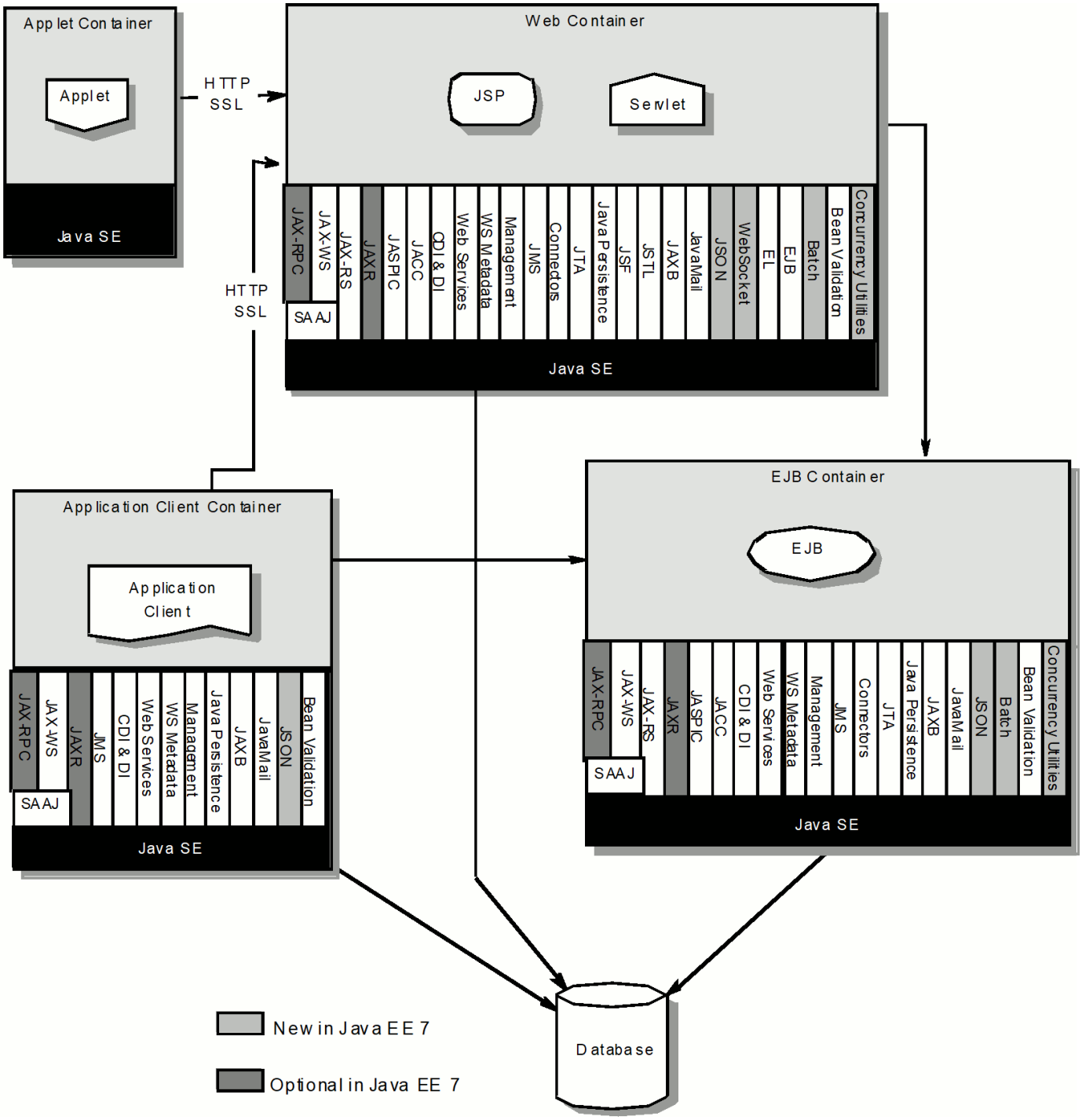

Figure 1. Jakarta EE Architecture Diagram

The following sections describe the Jakarta EE Platform requirements for each kind of Jakarta EE platform element.

2.2. Profiles

The Java EE 6 specification introduced the notion of “profiles” (see Profiles”).

A profile is a configuration of the Jakarta EE platform targeted at a specific class of applications.

Profiles are not a new concept, nor are they unique to the Jakarta EE platform. The Jakarta EE Specification process: “A Specification that includes by reference a collection of Specifications and possibly additional requirements. APIs from the referenced Platform Edition must be included according to the referencing rules set out in that Platform Edition Specification. Other referenced specifications must be referenced in their entirety.”

All Jakarta EE profiles share a set of common features, such as naming and resource injection, packaging rules, security requirements, etc. This guarantees a degree of uniformity across all products and, indirectly, applications that fall under the “Jakarta EE platform” umbrella. This also ensures that developers who are familiar with a certain profile, or with the full platform, can move easily to other profiles, avoiding excessive compartmentalization of skills and experience.

Beyond the basic functionality outlined above, profiles are free to include any set of technologies that are part of the platform, provided that all rules in the present specification that pertain to the included technologies—either alone or in combination with others—are followed.

This last point is worth stressing. If profiles only included pointwise technologies, they would be little more than bundles of APIs with few or no tie-ins. Instead, the definition of profiles adopted here guarantees that whenever this specification defines requirements on combinations of technologies, these requirements will be honored in all products based on Jakarta EE profiles.

As a concrete example, consider the use of transactions in a servlet container. In isolation, neither the Jakarta Servlet specification nor the Jakarta Transactions specification defines a complete programming model for portable applications. This specification fills that gap by introducing its own set of requirements that pertain to the combination of servlets and Jakarta Transactions. These requirements must be satisfied by any Jakarta EE profile-based product that includes those two technologies, thus offering application developers a more complete programming model shared across all relevant Jakarta EE profiles.

A separate specification, the Jakarta EE Web Profile Specification, defines the Jakarta EE Web Profile, the first profile of the Jakarta EE platform.

Additional profiles may be defined in accordance with the rules of the Jakarta EE Specification Process and those contained in the present specification. In particular, profiles are initiated by submitting a Project Proposal to the Jakarta EE Specification Process and are released at completion on their own schedule, independently of any concurrent revision of the platform itself or of other profiles. This ensures maximum flexibility in defining and releasing a new profile or an updated version of an existing one.

In accordance with the definition of profiles given above, a profile may end up being either a proper subset or a proper superset of the platform, or it may overlap with it to a certain extent. This flexibility guarantees that future profiles will be able to cover uses well beyond those originally envisioned by the platform specification.

As the previous paragraphs made clear, creating a new profile is a significant undertaking. The decision to create a profile should take into account its potential drawbacks, especially in terms of

fragmentation and developer confusion. In general, a profile should be created only when there is a natural developer constituency and a well-understood class of applications that can benefit from it. It is also recommended that a profile cast a comprehensive net on its area of interest, to minimize the occurrence of overlapping or competing profiles. Jakarta EE platform features such as optional components and extensibility can be used by profiles to achieve a better fit to their intended target.

2.3. Application Components

The Jakarta EE runtime environment defines four application component types that a Jakarta EE 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 Jakarta EE 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 Jakarta EE applications. (Simple HTML pages can also be used to provide a more limited user interface for Jakarta EE applications.)
- servlets, server pages, server faces applications, filters, and web event listeners typically execute in a web container and may respond to HTTP requests from web clients. Servlets, server pages, server faces applications, and filters 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. A special kind of servlet provides support for web services using the SOAP/HTTP protocol. Servlets, pages created with the Jakarta Server Pages technology or Jakarta Server Faces technology, web filters, and web event listeners are referred to collectively in this specification as "web components." Web applications are composed of web components and other data such as HTML pages. Web components execute in a web container. A web server includes a web container and other protocol support, security support, and so on, as required by Jakarta EE specifications.
- Jakarta Enterprise Beans components execute in a managed environment that supports transactions. Enterprise beans typically contain the business logic for a Jakarta EE application. Enterprise beans may directly provide web services using the SOAP/HTTP protocol.

2.3.1. Jakarta EE Server Support for Application Components

The Jakarta EE servers provide deployment, management, and execution support for conforming application components. Application components can be divided into three categories according to their dependence on a Jakarta EE server:

- Components that are deployed, managed, and executed on a Jakarta EE server. These components include web components and Jakarta Enterprise Beans components. See the separate specifications for these components.
- Components that are deployed and managed on a Jakarta EE server, but are loaded to and executed

on a client machine. These components include web resources such as HTML pages and applets embedded in HTML pages.

- Components whose deployment and management is not completely defined by this specification. Application Clients fall into this category. Future versions of this specification may more fully define deployment and management of Application Clients. See Application Clients," for a description of Application Clients.

2.4. Containers

Containers provide the runtime support for Jakarta EE application components. Containers provide a federated view of the underlying Jakarta EE APIs to the application components. Jakarta EE application components never interact directly with other Jakarta EE application components. They use the protocols and methods of the container for interacting with each other and with platform services. Interposing a container between the application components and the Jakarta EE services allows the container to transparently inject the services required by the component, such as declarative transaction management, security checks, resource pooling, and state management.

A typical Jakarta EE product will provide a container for each application component type: application client container, applet container, web component container, and enterprise bean container.

2.4.1. Container Requirements

This specification requires that containers provide a Java Compatible™ runtime environment, as defined by the Java Platform, Standard Edition, v8 specification (Java SE). 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 JDK™ 1.1 APIs is outside the scope of this specification.

The container tools must understand the file formats for the packaging of application components for deployment.

The containers are implemented by a Jakarta EE Product Provider. See the description of the Product Provider role in Jakarta EE Product Provider".

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

2.4.2. Jakarta EE Servers

Underlying a Jakarta EE container is the server of which it is a part. A Jakarta EE Product Provider typically implements the Jakarta EE server-side functionality using an existing transaction processing infrastructure in combination with Java Platform, Standard Edition (Java SE) technology. The Jakarta EE client functionality is typically built on Java SE technology.

2.5. Resource Adapters

A resource adapter is a system-level software component that typically implements network connectivity to an external resource manager. A resource adapter can extend the functionality of the Jakarta EE platform either by implementing one of the Java SE service APIs (such as a JDBC™ driver), or by defining and implementing a resource adapter for a connector to an external application system. Resource adapters may also provide services that are entirely local, perhaps interacting with native resources. Resource adapters interface with the Jakarta EE platform through the Jakarta EE service provider interfaces (Jakarta EE SPI). A resource adapter that uses the Jakarta EE SPIs to attach to the Jakarta EE platform will be able to work with all Jakarta EE products.

2.6. Database

The Jakarta EE platform requires 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 Jakarta EE Product Provider must also provide a preconfigured, default data source for use by the application in accessing this database. See Default Data Source".

2.7. Jakarta EE Standard Services

The Jakarta EE standard services include the following (specified in more detail later in this document). Some of these standard services are actually provided by Java SE.

2.7.1. HTTP

The HTTP client-side API is defined by the *java.net* package. The HTTP server-side API is defined by the Jakarta Servlet, Jakarta Server Pages, and Jakarta Server Faces interfaces and by the web services support that is a part of the Jakarta EE platform.

2.7.2. HTTPS

Use of the HTTP protocol over the SSL protocol is supported by the same client and server APIs as HTTP.

2.7.3. Jakarta Transaction API (JTA)

The Jakarta Transactions 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 Jakarta EE SPI level.

2.7.4. RMI-IIOP (Proposed Optional)

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 those APIs that supports both the Java SE native RMI protocol (JRMP) and the CORBA IIOP protocol. Jakarta EE applications can use RMI-IIOP, with IIOP protocol support, to access CORBA services that are compatible with the RMI programming restrictions (see the RMI-IIOP specification for details). Such CORBA services would typically be defined by components that live outside of a Jakarta EE product, usually in a legacy system. Only Jakarta EE 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.

Jakarta EE products must be capable of exporting Jakarta Enterprise Beans components using the IIOP protocol and accessing enterprise beans using the IIOP protocol, as specified in the Jakarta Enterprise Beans specification. The ability to use the IIOP protocol is required to enable interoperability between Jakarta EE products, however a Jakarta EE product may also use other protocols. Requirements for use of the RMI-IIOP APIs when accessing Jakarta Enterprise Beans components have been relaxed as of EJB 3.0. See the Jakarta Enterprise Beans specification for details.

Support for CORBA, including use of IIOP and Java IDL, is Proposed Optional as of Jakarta EE 8. See Pruned Java Technologies.”

2.7.5. Java IDL (Proposed Optional)

Java IDL allows Jakarta EE 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 Jakarta EE product. Jakarta EE applications may use Java IDL to act as clients of CORBA services, but only Jakarta EE application clients are required to be allowed to use Java IDL directly to present CORBA services themselves.

2.7.6. JDBC™ API

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 Jakarta EE platform. Support for the service provider interface is not required in Jakarta EE products. Instead, JDBC drivers should be packaged as resource adapters that use the facilities of the Connector API to interface with a Jakarta EE product. The JDBC API is included in Java SE, but this specification includes additional requirements on JDBC device drivers.

2.7.7. Jakarta Persistence API

Jakarta Persistence is the standard API for the management of persistence and object/relational mapping. It provides an object/relational mapping facility for application developers using a Java domain model to manage a relational database. Jakarta Persistence is required to be supported in

Jakarta EE. It can also be used in Java SE environments.

2.7.8. Jakarta™ Messaging

Jakarta Messaging is a standard API for messaging that supports reliable point-to-point messaging as well as the publish-subscribe model. This specification requires a Jakarta Messaging provider that implements both point-to-point messaging as well as publish-subscribe messaging. The Jakarta EE Product Provider must also provide a preconfigured, default Jakarta Messaging connection factory for use by the application in accessing this JMS provider. See Default Jakarta Messaging Connection Factory".

2.7.9. 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. The JNDI API is included in Java SE, but this specification defines additional requirements.

2.7.10. Jakarta™ Mail

Many Internet applications require the ability to send email notifications, so the Jakarta EE platform includes the Jakarta Mail API along with a Jakarta Mail service provider that allows an application component to send Internet mail. The Jakarta Mail API has two parts: an application-level interface used by the application components to send mail, and a service provider interface used at the Jakarta EE SPI level.

2.7.11. JavaBeans™ Activation Framework (JAF)

The JAF API provides a framework for handling data in different MIME types, originating in different formats and locations. The JavaMail API makes use of the JAF API. The JAF API is included in Java SE and so is available to Jakarta EE applications.

2.7.12. XML Processing

The Java™ API for XML Processing (JAXP) provides support for the industry standard SAX and DOM APIs for parsing XML documents, as well as support for XSLT transform engines. The Streaming API for XML (StAX) provides a pull-parsing API for XML. The JAXP and StAX APIs are included in Java SE and so are available to Jakarta EE applications.

2.7.13. Jakarta Connector Architecture

The Connector architecture is a Jakarta EE SPI that allows resource adapters that support access to Enterprise Information Systems to be plugged in to any Jakarta EE product. The Connector architecture defines a standard set of system-level contracts between a Jakarta EE server and a resource adapter. The standard contracts include:

- A connection management contract that lets a Jakarta EE 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 Jakarta EE 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.
- A thread management contract that allows a resource adapter to delegate work to other threads and allows the application server to manage a pool of threads. The resource adapter can control the security context and transaction context used by the worker thread.
- A contract that allows a resource adapter to deliver messages to message driven beans independent of the specific messaging style, messaging semantics, and messaging infrastructure used to deliver messages. This contract also serves as the standard message provider pluggability contract that allows a message provider to be plugged into any Jakarta EE server via a resource adapter.
- A contract that allows a resource adapter to propagate an imported transaction context to the Jakarta EE server such that its interactions with the server and any application components are part of the imported transaction. This contract preserves the ACID (atomicity, consistency, isolation, durability) properties of the imported transaction.
- An optional contract providing a generic command interface between an application program and a resource adapter.

2.7.14. Security Services

The Java™ Authentication and Authorization Service (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 supports user-based authorization. Jakarta™ Authorization defines a contract between a Jakarta EE application server and an authorization service provider, allowing custom authorization service providers to be plugged into any Jakarta EE product. Jakarta™ Authentication defines an SPI by which authentication providers implementing message authentication mechanisms may be integrated in client or server message processing containers or runtimes. Jakarta Security leverages Jakarta Authentication, but provides an easier to use SPI for authentication of users of web applications and defines identity store APIs for authentication and authorization.

2.7.15. Web Services

Jakarta EE provides full support for both clients of web services as well as web service endpoints. Several Java technologies work together to provide support for web services. JAX-WS and Jakarta XML-

based RPC both provide support for web service calls using the SOAP/HTTP protocol. JAX-WS, which is included in Java SE, is the primary API for web services and is a follow-on to Jakarta XML-based RPC. JAX-WS offers extensive web services functionality, with support for multiple bindings/protocols. JAX-WS and Jakarta XML-based RPC are fully interoperable when using the SOAP 1.1 over HTTP protocol as constrained by the WS-I Basic Profile specification. Support for Jakarta XML-based RPC has been made optional as of Java EE 7. See Pruned Java Technologies".

JAX-WS and the Java Architecture for XML Binding (JAXB) define the mapping between Java classes and XML as used in SOAP calls, and provide support for 100% of XML Schema. JAXB is included in Java SE. The SOAP with Attachments API for Java (SAAJ), which is also included in Java SE, provides support for manipulating low level SOAP messages. The Web Services for Jakarta EE specification fully defines the deployment of web service clients and web service endpoints in Java EE, as well as the implementation of web service endpoints using enterprise beans. The Web Services Metadata specification defines Java language annotations that make it easier to develop web services. The Jakarta XML Registries provides client access to XML registry servers. Support for JAXR has been made optional as of Java EE 7. See Pruned Java Technologies".

Jakarta JSON Processing provides a convenient way to process (parse, generate, transform, and query) JSON text. Jakarta JSON Binding provides a convenient way to convert between JSON text and Java objects. Jakarta WebSocket is a standard API for creating WebSocket applications.

Jakarta RESTful Web Services provides support for web services using the REST style. RESTful web services better match the design style of the web and are often easier to access using a wide variety of programming languages. Jakarta RESTful Web Services provides a simple high-level API for writing such web services as well as a low-level API that can be used to control the details of the web service interaction.

2.7.16. Jakarta Concurrency

Jakarta Concurrency is a standard API for providing asynchronous capabilities to Jakarta EE application components through the following types of objects: managed executor service, managed scheduled executor service, managed thread factory, and context service.

2.7.17. Jakarta Batch

The Jakarta Batch API (Batch) provides a programming model for batch applications and a runtime for scheduling and executing jobs.

2.7.18. Jakarta Management

The Jakarta Management Specification defines APIs for managing Jakarta EE servers using a special management enterprise bean. The Java™ Management Extensions (JMX) API is also used to provide some management support.

2.7.19. Jakarta Deployment

The Jakarta Deployment Specification defines a contract between deployment tools and Jakarta EE products. The Jakarta EE products provide plug-in components that run in the deployment tool and allow the deployment tool to deploy applications into the Jakarta EE product. The deployment tool provides services used by these plug-in components. Support for the Deployment Specification has been made optional as of Java EE 7. See Pruned Jakarta Technologies."

2.8. Interoperability

Many of the APIs described above provide interoperability with components that are not a part of the Jakarta EE platform, such as external web or CORBA services.

Jakarta EE Interoperability illustrates the interoperability facilities of the Jakarta EE platform. (The directions of the arrows indicate the client/server relationships of the components.)

.

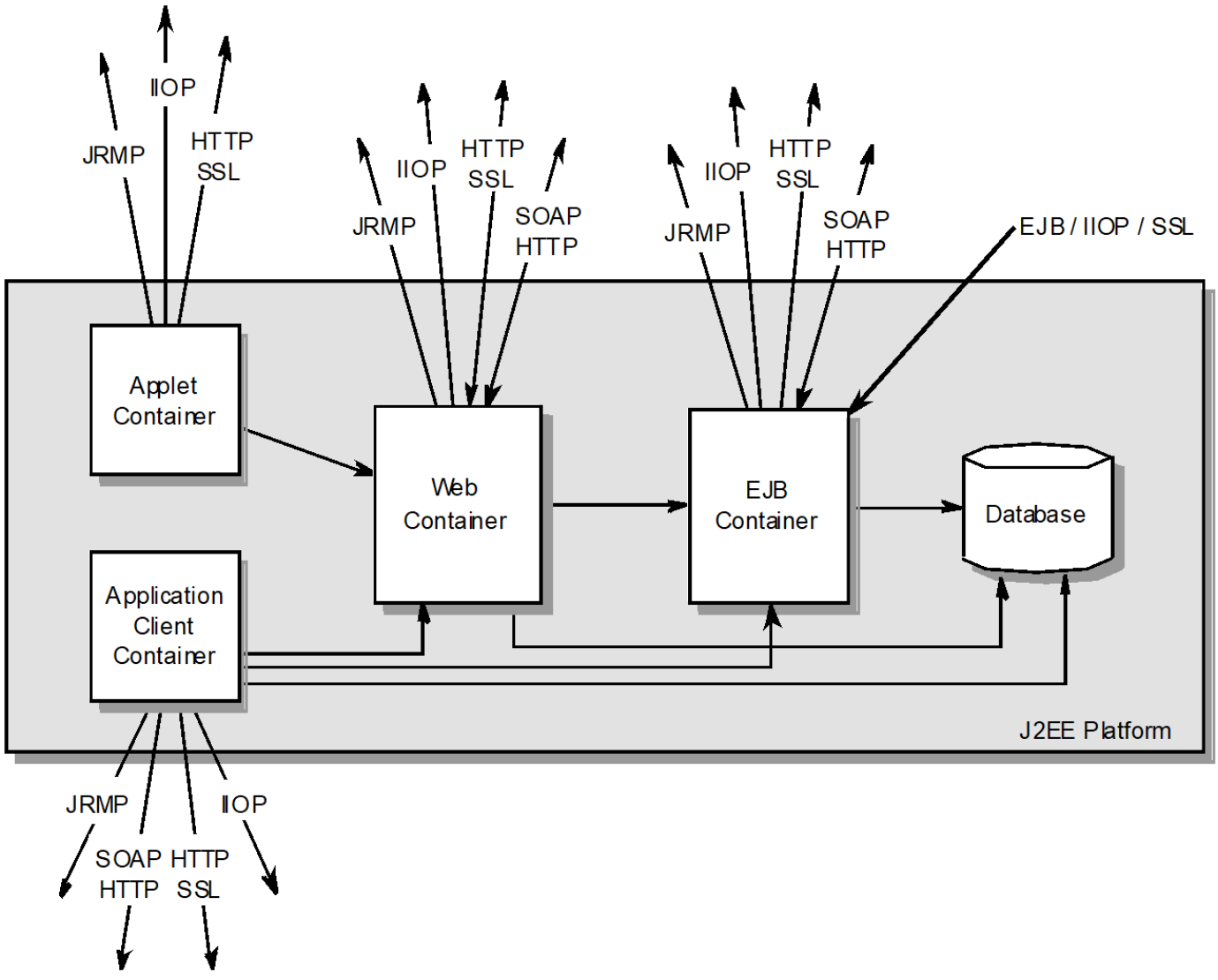

Figure 2. Java EE Interoperability

2.9. Flexibility of Product Requirements

This specification doesn't require that a Jakarta EE 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, or processes. As long as the requirements in this specification are met, Jakarta EE Product Providers can partition the functionality however they see fit. A Jakarta EE product must be able to deploy application components that execute with the semantics described by this specification.

A typical low end Jakarta EE product will support applets using the Java Plugin 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 Jakarta EE product might split the server components into multiple servers, each of which can be distributed and load-balanced across a collection of machines. While such machines might exist on-site in an enterprise, they might also reside, for example, in a public cloud. This specification does not prescribe or preclude any of these configurations.

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

2.10. Jakarta EE Product Packaging

This specification doesn't include requirements for the packaging of a Jakarta EE product. A Jakarta EE product might be provided on distribution media, for download on the web, or as a service available only on the web, for example. A Jakarta EE product must include implementations of all the APIs required by this specification. These implementations might depend on other software or services not included in the Jakarta EE product. The customer may be required to combine or configure the product with other software or services that are necessary to meet the requirements of this specification. The documentation for the Jakarta EE product must fully describe all the required software and configuration.

For example, a Jakarta EE product might depend on a database server, a naming service, a mail service, and/or a messaging service. All configurations in which the product is defined to operate must include all the software and services necessary to meet the requirements of this specification.

Whether these services are available (running, accessible on the network, properly configured, operating correctly, etc.) may be controlled independently of the Jakarta EE product — they may be unavailable when the Jakarta EE server is started, or they may fail while the Jakarta EE server is running. This specification does not require the Jakarta EE product to assure the availability of these services. However, if such a service is needed to meet the requirements of this specification, the Jakarta EE product must ensure that the service has been configured for use and will be usable when it is available.

For example, this specification requires that applications can use a database. If the Jakarta EE product

requires a database server to be separately installed, and requires the Jakarta EE product to be configured to use that database, such configuration must be done before applications are deployed. This ensures that the operational environment of applications includes all the required services.

2.11. Jakarta EE Product Extensions

This specification describes a minimum set of facilities available to all Jakarta EE products. A Jakarta EE profile may include some or all of these facilities, as described in Profiles". Products implementing the full Jakarta EE platform must provide all of them (see Full Jakarta EE Product Requirements"). Most Jakarta EE 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 Java SE on extensions to Java APIs. A Jakarta EE product must not add classes to the Java programming language packages included in this specification, and must not add methods or otherwise alter the signatures of the specified classes.

However, many other extensions are allowed. A Jakarta EE product may provide additional Java APIs, either other Java optional packages or other (appropriately named) packages. A Jakarta EE product may include support for additional protocols or services not specified here. A Jakarta EE 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.

We expect Jakarta EE products to vary widely and compete vigorously on various aspects of quality of service. Products will provide different levels of performance, scalability, robustness, availability, and security. In some cases this specification requires minimum levels of service. Future versions of this specification may allow applications to describe their requirements in these areas.

2.12. Platform Roles

This section describes typical Java Platform, Enterprise Edition roles. In an actual instance, an organization may divide role functionality differently to match that organization's application development and deployment workflow.

The roles are described in greater detail in later sections of this specification.

2.12.1. Jakarta EE Product Provider

A Jakarta EE Product Provider is the implementor and supplier of a Jakarta EE product that includes the component containers, Jakarta EE platform APIs, and other features defined in this specification. A Jakarta EE Product Provider is typically an application server vendor, a web server vendor, a database system vendor, or an operating system vendor. A Jakarta EE Product Provider must make available the Jakarta EE APIs to the application components through containers. A Product Provider frequently

bases their implementation on an existing infrastructure.

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

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

2.12.2. Application Component Provider

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

2.12.3. Application Assembler

The Application Assembler takes a set of components developed by Application Component Providers and assembles them into a complete Jakarta EE application delivered in the form of an 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.

2.12.4. Deployer

The Deployer is responsible for deploying application clients, web applications, and Enterprise JavaBeans components into a specific operational environment. The Deployer uses tools supplied by the Jakarta EE Product Provider to carry out deployment tasks. Deployment is typically a three-stage process:

1. During Installation the Deployer moves application 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 application components, and additional classes and interfaces, into the appropriate Jakarta EE containers.
2. During Configuration, external dependencies declared by the Application Component Provider are resolved and application assembly instructions defined by the Application Assembler are followed. For example, the Deployer is responsible for mapping security roles defined by the Application Assembler onto user groups and accounts that exist in the target operational environment.
3. Finally, the Deployer starts up Execution of the newly installed and configured application.

In some cases, a specially qualified Deployer may customize the business logic of the application's

components at deployment time. For example, using tools provided with a Jakarta EE product, the Deployer may provide simple application code that wraps an enterprise bean's business methods, or customizes the appearance of a Jakarta Server Pages or Jakarta Server Faces page.

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 Jakarta EE container.

For example, in the case of cloud deployments, the Deployer would be responsible for configuring the application to run in the cloud environment. The Deployer would install the application into the cloud environment, configure its external dependencies, and might handle aspects of provisioning its required resources.

2.12.5. 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 Jakarta EE applications. The System Administrator typically uses runtime monitoring and management tools provided by the Jakarta EE Product Provider to accomplish these tasks.

For example, in a cloud scenario, the System Administrator would be responsible for installing, configuring, managing, and maintaining the cloud environment, including the resources that are made available to applications running in the environment.

2.12.6. 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 types of application components supported by the Jakarta EE platform. Platform independent tools can be used for all phases of development through the deployment of an application and the management and monitoring of an application server.

2.12.7. System Component Provider

A variety of system level components may be provided by System Component Providers. The Connector Architecture defines the primary APIs used to provide resource adapters of many types. These resource adapters may connect to existing enterprise information systems of many types, including databases and messaging systems. Another type of system component is an authorization policy provider as defined by the Jakarta Authorization specification.

2.13. Platform Contracts

This section describes the Jakarta EE contracts that must be fulfilled by a Jakarta EE Product Provider implementing the full Jakarta EE platform. Jakarta EE profiles may include some or all of these facilities, as described in Profiles".

2.13.1. Jakarta EE APIs

The Jakarta EE APIs define the contract between the Jakarta EE application components and the Jakarta EE platform. The contract specifies both the runtime and deployment interfaces.

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

2.13.2. Jakarta EE Service Provider Interfaces (SPIs)

The Jakarta EE Service Provider Interfaces (SPIs) define the contract between the Jakarta EE platform and service providers that may be plugged into a Jakarta EE product. The connector APIs define service provider interfaces for integrating resource adapters with a Jakarta EE application server. Resource adapter components implementing the connector APIs are called Connectors. The Jakarta Authorization APIs define service provider interfaces for integrating security authorization mechanisms with a Jakarta EE application server.

The Jakarta EE Product Provider must implement the Jakarta EE 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.

2.13.3. Network Protocols

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

The Jakarta EE Product Provider is required to publish the installed application components on the industry-standard protocols. This specification defines the mapping of servlets and server pages to the HTTP and HTTPS protocols, and the mapping of Jakarta Enterprise Beans components to IIOP and SOAP protocols.

2.13.4. Deployment Descriptors and Annotations

Deployment descriptors and Java language annotations are used to communicate the needs of application components to the Deployer. The deployment descriptor and class file annotations are 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, and so forth in the component's deployment descriptor or through class file annotations. The Jakarta EE Product Provider is required to provide a deployment tool that interprets the Jakarta EE deployment descriptors and class file annotations and allows the Deployer to map the application component's requirements to the capabilities of a specific Jakarta EE product and environment.

2.14. Changes in J2EE 1.3

The J2EE 1.3 specification extends the J2EE platform with additional enterprise integration facilities. The Connector API supports integration with external enterprise information systems. A JMS provider is now required. The JAXP API provides support for processing XML documents. The JAAS API provides security support for the Connector API. The EJB specification now requires support for interoperability using the IIOP protocol.

Significant changes have been made to the EJB specification. The EJB specification has a new container-managed persistence model, support for message driven beans, and support for local enterprise beans.

Other existing J2EE APIs have been updated as well. See the individual API specifications for details. Finally, J2EE 1.3 requires support for J2SE 1.3.

2.15. Changes in J2EE 1.4

The primary focus of J2EE 1.4 is support for web services. The JAX-RPC and SAAJ APIs provide the basic web services interoperability support. The Web Services for J2EE specification describes the packaging and deployment requirements for J2EE applications that provide and use web services. The EJB specification was also extended to support implementing web services using stateless session beans. The JAXR API supports access to registries and repositories.

Several other APIs have been added to J2EE 1.4. The J2EE Management and J2EE Deployment APIs enable enhanced tool support for J2EE products. The JMX API supports the J2EE Management API. The J2EE Authorization Contract for Containers provides an SPI for security providers.

Many of the existing J2EE APIs have been enhanced in J2EE 1.4. J2EE 1.4 builds on J2SE 1.4. The JSP specification has been enhanced to simplify the development of web applications. The Connector API now supports integration with asynchronous messaging systems, including the ability to plug in JMS providers.

Changes in this J2EE platform specification include support for deploying class libraries independently of any application and the conversion of deployment descriptor DTDs to XML Schemas.

Other J2EE APIs have been enhanced as well. For additional details, see each of the referenced specifications.

2.16. Changes in Java EE 5

With this release, the platform has a new name – Java Platform, Enterprise Edition, or Java EE for short. This new name gets rid of the confusing “2” while emphasizing even in the short name that this is a Java platform. Previous versions are still referred to using the old name “J2EE”.

The focus of Java EE 5 is ease of development. To simplify the development process for programmers

just starting with Java EE, or developing small to medium applications, Java EE 5 makes extensive use of Java language annotations, which were introduced by J2SE 5.0. Annotations reduce or eliminate the need to deal with Java EE deployment descriptors in many cases. Even large applications can benefit from the simplifications provided by annotations.

One of the major uses of annotations is to specify injection of resources and other dependencies into Java EE components. Injection augments the existing JNDI lookup capability to provide a new simplified model for applications to gain access to the resources needed from the operational environment. Injection also works with deployment descriptors to allow the deployer to customize or override resource settings specified in the application's source code.

The use of annotations is made even more effective by providing better defaults. Better default behavior and better default configuration allows most applications to get the behavior they want most of the time, without the use of either annotations or deployment descriptors in many cases. When the default is not what the application wants, a simple annotation can be used to specify the required behavior or configuration.

The combination of annotations and better defaults has greatly simplified the development of applications using Enterprise JavaBeans technology and applications defining or using web services. Enterprise beans are now dramatically simpler to develop. Web services are much easier to develop using the annotations defined by the Web Services Metadata specification.

The area of web services continues to evolve at a rapid pace. To provide the latest web services support, the JAX-RPC technology has evolved into the JAX-WS technology, which makes heavy use of the JAXB technology to bind Java objects to XML data. Both JAX-WS and JAXB are new to this version of the platform.

Major additions to Java EE 5 include the JSTL and JSF technologies that simplify development of web applications, and the Java Persistence API developed by the EJB 3.0 expert group, which greatly simplifies mapping Java objects to databases.

Minor additions include the StAX API for XML parsing. Most APIs from previous versions have been updated with small to medium improvements.

2.17. Changes in Java EE 6

Java EE 6 continues the "ease of development" focus of Java EE 5.

One of the major improvements introduced in Java EE 6 is the Contexts and Dependency Injection (CDI) technology, which provides a uniform framework for the dependency injection and lifecycle management of "managed beans".

The Java EE 6 Managed Bean specification defines the commonalities across the spectrum of Java EE managed objects, extending from basic managed beans through EJB components.

The Bean Validation specification, introduced in this release, provides a facility for validation of

managed objects. Bean Validation is integrated into the Java Persistence API, where it provides an automated facility for the validation of JPA entities.

Java EE 6 adds the JAX-RS API as a required technology of the Java EE Platform. JAX-RS is the API for the development of Web services built according to the Representational State Transfer (REST) architectural style.

Java EE 6 also introduces the Java EE Web Profile, the first new profile of the Java EE Platform.

2.18. Changes in Java EE 7

Since its inception, the Java EE platform has been targeted at offloading the developer from common infrastructure tasks through its container-based model and abstraction of resource access. In recent releases the platform has considerably simplified the APIs for access to container services while broadening the range of the services available. In this release we continue the direction of improved simplification, while extending the range of the Java EE platform to encompass emerging technologies in the web space.

The Java EE 7 platform adds first-class support for recent developments in web standards, including Web Sockets and JSON, which provide the underpinnings for HTML 5 support in Java EE. Java EE 7 also adds a modern HTTP client API as defined by JAX-RS 2.0.

Also new in the Java EE 7 platform is the Batch API, which provides a programming model for batch applications and a runtime for scheduling and executing jobs, and the Concurrency Utilities API, which provides asynchronous capabilities by means of managed executor service, managed scheduled executor service, managed thread factory, and context service.

The CDI dependency injection facility introduced in Java EE 6 is enhanced as well as more broadly utilized by the Java EE 7 platform technologies, and the managed bean model is further aligned to remove inconsistencies among Java EE component classes in aspects of CDI injection and interceptor support. The declarative transaction functionality introduced by EJB is been made available in a more general way through CDI interceptors, so that it may be leveraged by other managed beans. The Bean Validation facility is extended to the automatic validation of method invocations and likewise made available via CDI interceptors.

Java EE 7 also continues the "ease of development" focus of Java EE 5 and Java EE 6. Most notably, Java EE 7 includes a revised and greatly simplified JMS 2.0 API. Ease of development encompasses ease of configuration as well. To that end, Java EE 7 broadens the resource definition facilities introduced in Java EE 6 to encompass more of the standard platform resource types, and also provides default database and JMS connection factory resources. It also improves the configuration of application security, including new descriptors for security permissions. Java EE 7 further simplifies the platform by making optional the technologies that were identified as candidates for pruning in Java EE 6, namely: EJB Entity Beans, JAX-RPC 1.1, JAXR 1.0, and JSR-88 1.2.

Finally, Java EE 7 lays groundwork for enhancements to the platform for use in cloud environments in a future release. Such features include resource definition metadata, improved security configuration,

and support for database schema generation via the Java Persistence API.

2.19. Changes in Java EE 8

Java EE 8 continues the focus on modern web applications of Java EE 7 and broadening the range of such applications. Java EE 8 introduces the JSON Binding API (JSON-B) for mapping between JSON text and Java objects, building on the JSON Processing API (JSON-P) introduced in Java EE 7. The JSON Processing API itself is updated to reflect additional JSON standards. Servlet undergoes major enhancement with the addition of support for the new HTTP/2 protocol. JAX-RS adds support for server-sent events and, building on concurrency facilities added in Java SE 8, a reactive client API. The new Java EE Security API provides enhanced support for authentication and authorization in web modules, and also introduces APIs for access to identity stores. The Bean Validation facility is updated to reflect enhancements made in Java SE 8 and to extend the range of validated objects. While the focus of CDI in this release is to extend its scope beyond Java EE with the introduction of a bootstrapping API, CDI also includes enhancements for event processing and alignment on Java SE 8 features.

2.20. Changes in Jakarta EE 8

Jakarta EE 8 is the migration of Java EE 8 from the JCP to the Eclipse Foundation.

Chapter 3. Security

This chapter describes the security requirements for the Jakarta™ Enterprise Edition (Jakarta EE) that must be satisfied by Jakarta EE products.

In addition to the Jakarta EE requirements, each Jakarta EE Product Provider will determine the level of security and security assurances that will be provided by their implementation.

3.1. Introduction

Almost every enterprise has security requirements and specific mechanisms and infrastructure to meet them. Sensitive resources that can be accessed by many users or that often traverse unprotected open networks (such as the Internet) need to be protected.

Although the quality assurances and implementation details may vary, they all share some of the following characteristics:

- *Authentication* : The means by which communicating entities (for example, client and server) prove to one another that they are acting on behalf of specific identities that are authorized for access.
- *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 has not 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 made available only 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 Jakarta EE platform requirements address security requirements, and identifies requirements that may be addressed by Jakarta EE Product Providers. Finally, issues being considered for future versions of this specification are briefly mentioned in Future Directions.

3.2. A Simple Example

The security behavior of a Jakarta EE environment may be better understood by examining what happens in a simple application with a web client, a JSP user interface, and enterprise bean business logic. (The 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.

Initial Request

The web client requests the main application URL, shown in *Initial Request* .

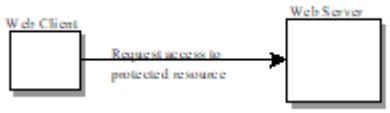

Figure 3. Initial Request

Since the 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.

Initial Authentication

The web server returns a form that the web client uses to collect authentication data (for example, 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 *Initial Authentication* .

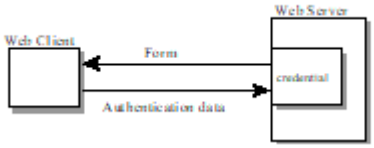

Figure 4. Initial Authentication

The validation mechanism may be local to the server, or it may leverage the underlying security services. On the basis of the validation, the web server sets a credential for the user.

URL Authorization

The credential is used for future determinations of whether the user is authorized to access restricted resources it may request. The web server consults 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 credential against each role to determine if it can map the user to the role. *URL Authorization* shows this process.

Figure 5. URL Authorization

The web server’s evaluation stops with an “is authorized” outcome when the web server is able to map the user to a role. A “not authorized” outcome is reached if the web server is unable to map the user to

any of the permitted roles.

Fulfilling the Original Request

If the user is authorized, the web server returns the result of the original URLrequest, as shown in *Fulfilling the Original Request* .

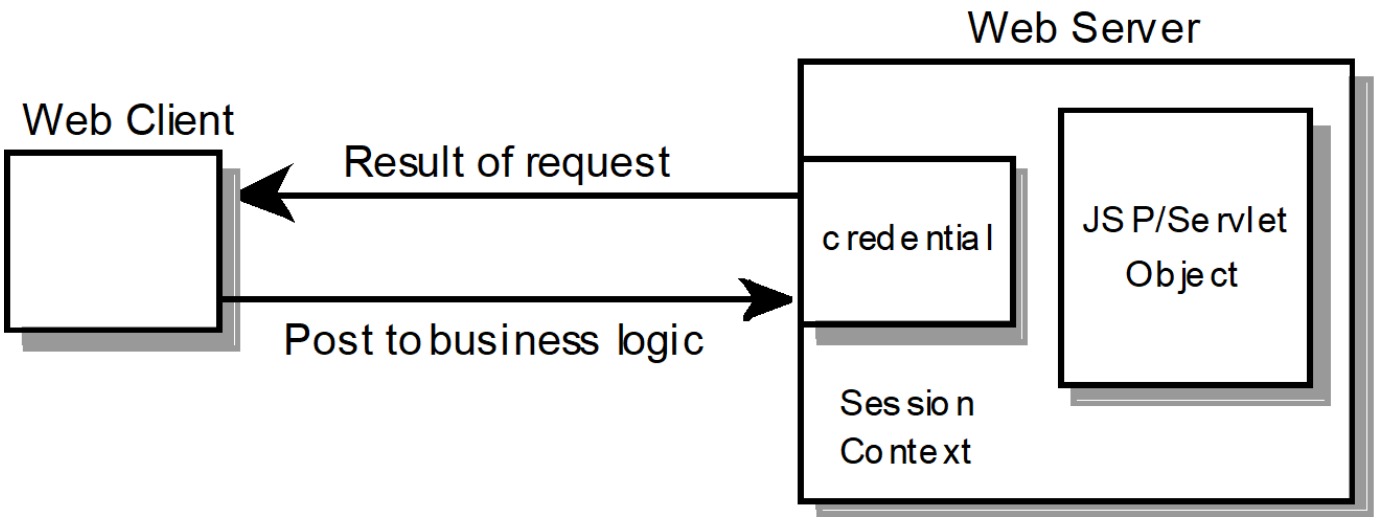

Figure 6. Fulfilling the Original Request

In our example, the response URL of a JSP page is returned, enabling the user to post form data that needs to be handled by the business logic component of the application.

Invoking Enterprise Bean Business Methods

The JSP page performs the remote method call to the enterprise bean, using the user's credential to establish a secure association between the JSP page and the enterprise bean (as shown in *Invoking an Enterprise Bean Business Method*). The association is implemented as two related security contexts, one in the web server and one in the Jakarta Enterprise Beans container.

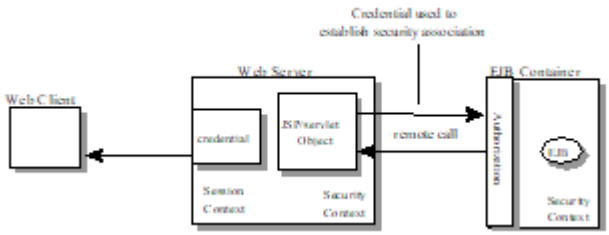

Figure 7. Invoking an Enterprise Bean Business Method

The Jakarta Enterprise Beans container is responsible for enforcing access control on the enterprise bean method. It consults 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. For each role, the Jakarta Enterprise Beans container uses the security context associated with the call to determine if it can map the caller to the role.

The container's evaluation stops with an "is authorized" outcome when the container is able to map the caller's credential to a role. A "not authorized" outcome is reached if the container is unable to map the caller to any of the permitted roles. A "not authorized" result causes an exception to be thrown by

the container, and propagated back to the calling JSP page.

If the call “is authorized”, the container dispatches control to the enterprise bean method. The result of the bean’s execution of the call is returned to the JSP, and ultimately to the user by the web server and the web client.

3.3. Security Architecture

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

3.3.1. Goals

The following are goals for the Jakarta EE security architecture:

1. Portability: The Jakarta EE 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 Jakarta EE platform should be able to perform authentication and access control according to instructions established by the Deployer using deployment attributes, and managed by the System Administrator.

Note that divorcing the application from responsibility for security ensures greater portability of Jakarta EE applications.

1. Extensibility: The use of platform services by security-aware applications must not compromise application portability.

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

1. Flexibility: The security mechanisms and declarations used by applications under this specification should not impose a particular security policy, but facilitate the implementation of security policies specific to the particular Jakarta EE installation or application.
2. Abstraction: An application component’s security requirements will be logically specified using Java language annotations or deployment descriptors. Java language annotations or deployment descriptors will specify how security roles and access requirements are to be mapped into environment-specific security roles, users, and policies. A Deployer may choose to modify the security properties in ways consistent with the deployment environment. The annotations or deployment descriptor should document which security properties can be modified and which cannot.
3. Independence: Required security behaviors and deployment contracts should be implementable

using a variety of popular security technologies.

4. Compatibility testing: The Jakarta EE security requirements architecture must be expressed in a manner that allows for an unambiguous determination of whether or not an implementation is compatible.
5. Secure interoperability: Application components executing in a Jakarta EE product must be able to invoke services provided in a Jakarta EE product from a different vendor, whether with the same or a different security policy. The services may be provided by web components or enterprise beans.

3.3.2. Non Goals

The following are not goals for the Jakarta EE security architecture:

1. This specification does not dictate a specific security policy. Security policies for applications and for enterprise information systems vary for many reasons unconnected with this specification. Product Providers can provide the technology needed to implement and administer desired security policies while adhering to the requirements of this specification.
2. This specification does not mandate a specific security technology, such as Kerberos, PK, NIS+, or NTLM.
3. This specification does not require that the Jakarta EE security behaviors be universally implementable using any or all security technologies.
4. This specification does not provide any warranty or assurance of the effective security of a Jakarta EE product.

3.3.3. Terminology

This section introduces the terminology that is used to describe the security requirements of the Jakarta EE 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 can vary depending upon the authentication protocol.

Security Policy Domain

A security policy domain, also referred to as a security domain, is a scope over which a common security policy is defined and enforced by the security administrator of the security service.

A security policy domain is also sometimes referred to as a realm. This specification uses the security policy domain, or security domain, terminology.

Security Technology Domain

A security technology domain is the scope over which the same security mechanism (for example 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 (for example, access to protected resources and auditing of users). Security attributes can be associated with a principal by an authentication protocol and/or by the Jakarta EE Product Provider.

The Jakarta EE platform does not specify what security attributes are associated with a principal.

Credential

A credential contains or references information (security attributes) used to authenticate a principal for Jakarta EE product 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. The contents and format of a credential can vary widely.

3.3.4. Container Based Security

Security for components is provided by their containers in order to achieve the goals for security specified above in a Jakarta EE environment. A container provides two kinds of security (discussed in the following sections):

- Declarative security
- Programmatic security

3.3.4.1. Declarative Security

Declarative security refers to the means of expressing an application's security structure, including security roles, access control, and authentication requirements in non-programmatic form. Java language annotations and the deployment descriptor are the primary vehicles for declarative security in the Jakarta EE platform.

Java language annotations and the deployment descriptor are a contract between an Application Component Provider and a Deployer or Application Assembler. They can be used by an application programmer to represent an application's security related environmental requirements. A deployment descriptor can be associated with groups of components.

A Deployer maps the declarative representation of the application's security policy to a security structure specific to the particular environment. A Deployer uses a deployment tool to process the annotations and deployment descriptor.

At runtime, the container uses the security policy security structure derived from the declarative

security information expressed in annotations and the deployment descriptor and configured by the Deployer to enforce authorization (see Authorization Model).

3.3.4.2. Programmatic Security

Programmatic security refers to security decisions made by security aware applications. Programmatic security is useful when declarative security alone is not sufficient to express the security model of the application. The API for programmatic security consists of methods of the Jakarta Security *SecurityContext* interface, and methods of the Jakarta Enterprise Beans *EJBContext* interface and the servlet *HttpServletRequest* interface. The methods of the Jakarta Security *SecurityContext* interface are intended to supersede the corresponding methods of the *EJBContext* and *HttpServletRequest* interfaces.

These methods allow components to make business logic decisions based on the security role of the caller or remote user. For example they allow the component to determine the principal name of the caller or remote user to use as a database key. (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. Due to principal name mapping, the same logical principal may have different names in different containers, although usually it will be possible to configure a single product to use consistent principal names. In particular, if a principal name is used as a key into a database table, and that database table is accessed from multiple components, containers, or products, the same logical principal may map to different entries in the database.)

3.3.5. Distributed Security

Some Product Providers may produce Jakarta EE products in which the containers for various component types are distributed. In a distributed environment, communication between Jakarta EE 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 that establishes the basis of a secure communication between 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. Setting up a security context for the association between the components.

Since a container provides security in Jakarta EE, 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 Jakarta Enterprise Beans specification.

Product Providers may allow for control over the quality of protection or other aspects of secure association at deployment time. Applications can specify their requirements for access to web resources using annotations or elements in their deployment descriptor.

This specification does not define mechanisms that an Application Component Provider can use to

communicate requirements for secure associations with an enterprise bean.

3.3.6. Authorization Model

The Jakarta EE 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. A Deployer maps roles to security identities (for example principals, and groups) in the operational environment. Security roles are used with both declarative security and programmatic security.

Declarative authorization can be used to control access to an enterprise bean method and is specified in annotations or in the enterprise bean deployment descriptor. The *RolesAllows* , *PermitAll* , and *DenyAll* annotations are used to specify method permissions. An enterprise bean method can also 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.

Security roles are used in the *SecurityContext* method *isCallerInRole* , the *EJBContext* method *isCallerInRole* , and the *HttpServletRequest* method *isUserInRole* . Each method returns *true* if the calling principal is in the specified security role.

3.3.6.1. Role Mapping

Enforcement of security constraints on web resources or enterprise beans, whether programmatic or declarative, depends upon determination of whether the principal associated with an incoming request is in a given security role. A container makes this determination based on the security attributes of the calling principal. For example,

1. A Deployer may have mapped a security role to a user group in the operational environment or may depend on the default mapping of security roles to user groups as defined by the Jakarta Security specification. In this case, the user group of the calling principal is retrieved from its security attributes. The principal is in the security role if the principal's user group matches a user group to which the security role has been mapped.
2. A Deployer may 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 name is the same as a 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 Jakarta EE platform. Security attributes may be transmitted in the calling principal's credential or in the security context. In other cases, security attributes may be retrieved from an identity store, or from a trusted third party, such as a directory service or a security service.

3.3.7. HTTP Login Gateways

Secure interoperability between enterprise beans in different security policy domains is addressed in the Jakarta Enterprise Beans specification. In addition, a component may choose to log in to a foreign server via HTTP. An application component can be configured to use SSL mutual authentication for security when accessing a remote resource using HTTP. Applications using HTTP in this way may choose to use XML or some other structured format, rather than HTML.

We call the use of HTTP with SSL mutual authentication to access a remote service an HTTP Login Gateway. Requirements in this area are specified in Authentication by Web Clients.

3.3.8. 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 Jakarta EE application components. An end user can authenticate using either of the two supported client types:

- Web client
- Application client

3.3.8.1. Authentication by Web Clients

It is required that a web client be able to authenticate a user to a web server using any of the following mechanisms. The Deployer or System Administrator determines which method to apply to an application or to a group of applications.

- 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 in simple base64 encoding. The target server is not authenticated. Additional protection can be applied to overcome these weaknesses. The password may be protected by applying security at the transport layer (for example HTTPS) or at the network layer (for example, IPSEC or VPN).

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

- HTTPS Client 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, a PKC is rarely used

by end users on the Internet. However, it is useful for e-commerce applications and also for a single-signon from within the browser. For these reasons, HTTPS client authentication is a required feature of the Jakarta EE platform.

- Form Based Authentication

The look and feel of a login screen cannot be varied using the web browser's built-in authentication mechanisms. This specification introduces the ability to package standard HTML or servlet/JSP/JSF based forms for logging in, allowing customization of the user interface. The form based authentication mechanism introduced by this specification is described in the Servlet specification.

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

A web client can employ a web server as its authentication proxy. In this case, a client's credential is established in the server, where it may be used by the server for various purposes: 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.

3.3.8.2. 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 he or she accesses.
3. Require re-authentication of users only when a security policy domain boundary has been crossed.

Credentials that are acquired through a web login process are associated with a session. The container uses the 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 (including enterprise beans).

3.3.8.3. Login Session

In the Jakarta EE platform, login session support is provided by a web 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]

3.3.8.4. Authentication by Application Clients

Application clients (described in detail in Application Clients) are client programs that may interact with enterprise beans directly (that is, without the help of a web browser and without traversing a web server). Application clients may also access web resources.

Application clients, like the other Jakarta EE 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 are expected to communicate with a human user.

Application clients are used to authenticate end users to the Jakarta EE platform, when the users access protected web resources or enterprise beans.

3.3.9. 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 use lazy authentication, that is, to perform authentication only when it is needed. With lazy authentication, a user is not required to authenticate until there is a request to access a protected resource.

Lazy authentication can be used with first-tier clients (applets, application clients) when they request access to protected resources that require authentication. At that point the user can be asked to provide appropriate authentication data. If a user is successfully authenticated, the user is allowed to access the resource.

3.4. User Authentication Requirements

The Jakarta EE Product Provider must meet the following requirements concerning user authentication.

3.4.1. Login Sessions

All Jakarta EE web servers must maintain a login session for each web user. It must be possible for a login session to span more than one application, allowing a user to log in once and access multiple applications. The required login session support is described in the Servlet specification. This requirement of a session for each web user supports single signon.

Applications can remain independent of the details of implementing the security and maintenance of login information. The Jakarta EE Product Provider has the flexibility to choose authentication mechanisms independent of the applications secured by these mechanisms.

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

3.4.2. Required Login Mechanisms

All Jakarta EE 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.

3.4.2.1. HTTP Basic Authentication

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

3.4.2.2. SSL Mutual Authentication

TLS 1.2 and the means to perform mutual (client and server) certificate-based authentication are required by this specification.

All Jakarta EE products must also support TLS 1.1 and TLS 1.0, to ensure interoperable secure communications with clients; however, TLS 1.0 should be disabled if not needed for a given deployment, and TLS 1.1 may be disabled if not needed.

Similarly, all Jakarta EE products must support the following cipher suites, to ensure interoperable secure communications with clients:

- *TLS_RSA_WITH_AES_128_CBC_SHA*
- *TLS_DHE_RSA_WITH_AES_128_CBC_SHA*
- *TLS_ECDH_RSA_WITH_AES_128_CBC_SHA*
- *TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA*
- *TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA*
- *TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA*

However, it is recommended to use the strongest possible cipher suite that can be negotiated between client and server, and the above cipher suites may be disabled in favor of stronger cipher suites, if not needed for a given deployment.

Note that previous versions of this specification required support for SSL 3.0, and for the following cipher suites:

- *TLS_RSA_WITH_RC4_128_MD5*
- *SSL_RSA_WITH_RC4_128_MD5*
- *TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA*
- *SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA*
- *TLS_RSA_EXPORT_WITH_RC4_40_MD5*
- *SSL_RSA_EXPORT_WITH_RC4_40_MD5*
- *TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA*
- *SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA*

SSL 3.0 was officially deprecated by RFC 7568 in 2015, and is unsupported or disabled by default in many TLS implementations. None of the above cipher suites is currently considered secure, and may

be unsupported or disabled by default. In extreme cases, it may be necessary to use SSL 3.0, or to negotiate one of the above cipher suites, in order to interoperate with an older client or a previous version of Jakarta EE. However, it is recommended to use TLS 1.0 or higher, and to negotiate a stronger cipher suite, whenever possible. SSL 3.0, and the above listed cipher suites, should be disabled if not needed for interoperability in a given deployment.

3.4.2.3. Form Based Login

The web application deployment descriptor contains an element that causes a Jakarta EE 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 specification.

3.4.3. 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 *SecurityContext* method *getCallerPrincipal* or the *HttpServletRequest* method *getUserPrincipal* . This is different than the result of the *EJBContext* method *getCallerPrincipal* . The Jakarta Enterprise Beans specification requires that the *EJBContext* method *getCallerPrincipal* always return a valid *Principal* object. This method can never return *null* . The *SecurityContext* method *getCallerPrincipal* can also be called in the Jakarta Enterprise Beans container, and still returns *null* for anonymous users.

In Jakarta EE products that contain both a web container and an Jakarta Enterprise Beans container, components running in a web container must 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 Jakarta EE product must provide a principal for use in the call.

A Jakarta EE 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.
- Allow the deployer or system administrator to choose which principal to use through the Run As capability of the web and enterprise bean containers.

This specification does not specify how a Jakarta EE product should choose a principal to represent unauthenticated users, although future versions of this specification may add requirements in this area. Note that the Jakarta Enterprise Beans specification does include requirements in this area when using the Jakarta Enterprise Beans interoperability protocol. Applications are encouraged to use the Run As capability in cases where the web component may be unauthenticated and needs to call Jakarta

Enterprise Beans components.

3.4.4. Application Client User Authentication

The application client container must provide authentication of application users to satisfy the authentication and authorization constraints enforced by the enterprise bean containers and web containers. The techniques used may vary with the implementation of the application client container, and are beyond the control of the application. The application client container may integrate with a Jakarta EE 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 there is a request to access a protected resource or enterprise bean.

The container will provide an appropriate user interface for interactions with the user to gather authentication data. 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 Jakarta EE Application Client XML Schema 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 may execute in an environment controlled by a Java SE security manager and are subject to the security permissions defined in Java Platform, Standard Edition (Java SE) Requirements. Although this specification does not define the relationship between the operating system identity associated with a running application client and the authenticated user identity, support for single signon requires that the Jakarta EE product be able to relate these identities. Additional application client requirements are described in CHApter of this specification.

3.4.5. Resource Authentication Requirements

Resources within an enterprise are often deployed in security policy domains different from the security policy domain of the application component. The wide variance of authentication mechanisms used to authenticate the caller to resources leads to the requirement that a Jakarta EE product provide the means to authenticate in the security policy domain of the resource.

A Product Provider must support both of the following:

1. Configured Identity. A Jakarta EE container must be able to authenticate for access to the resource using a principal and authentication data specified by a Deployer at deployment time.The authentication must not depend in any way on data provided by the application components. Providing for the confidential storage of the authentication information is the responsibility of the Product Provider.
2. Programmatic Authentication. The Jakarta EE product must provide for specification of the

principal and authentication data for a resource by the application component at runtime using appropriate APIs. 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, and so forth.

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

1. Principal Mapping. A resource can have a principal and attributes that are determined by a mapping from the identity and security attributes of the requesting principal. In this case, a resource principal is not based on inheritance of the identity or security attributes from a requesting principal, but gets its identity and security attributes based on the mapping.
2. Caller Impersonation. A resource principal acts on behalf of a requesting principal. Acting on behalf of a caller principal requires delegation of the caller's identity and credentials to the underlying resource manager. In some scenarios, a requesting principal can be a delegate of an initiating principal and the resource principal is transitively impersonating 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.)

1. Credentials Mapping. This technique may be used when an application server and an EIS support different authentication domains. For example:
2. The initiating principal may have been authenticated and have public key certificate-based credentials.
3. The security environment for the resource manager may be 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 Jakarta EE products.

3.5.1. Code Authorization

A Jakarta EE product may restrict the use of certain Java SE classes and methods to secure and ensure proper operation of the system. The minimum set of permissions that a Jakarta EE product is required to grant to a Jakarta EE application is defined in Java Platform, Standard Edition (Java SE) Requirements. All Jakarta EE products must be capable of deploying application components with exactly these permissions.

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

The *permissions.xml* descriptor (see Declaring Permissions Required by Application Components) makes it possible to express permissions that a component needs for access.

3.5.2. Caller Authorization

A Jakarta EE product must enforce the access control rules specified at deployment time (see Deployment Requirements) and more fully described in the Jakarta Enterprise Beans and Servlet specifications.

3.5.3. Propagated Caller Identities.

In a Jakarta EE product that contains an Jakarta Enterprise Beans container, it must be possible to configure the Jakarta EE product so that a propagated caller identity is used in all authorization decisions. With this configuration, for all calls to all enterprise beans from a single application within a single Jakarta EE product, the principal name returned by the *EJBContext* method *getCallerPrincipal* or the *SecurityContext* 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* or the *SecurityContext* method *getCallerPrincipal* in the calling servlet or JSP page. (However, if the *HttpServletRequest* or *SecurityContext* method *getCallerPrincipal* 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. A single principal must be the principal used in authorization decisions for access to all enterprise beans in the call chain. The requirements in this section apply only when a Jakarta EE product has been configured to propagate caller identity.

3.5.4. Run As Identities

Jakarta EE 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 it is the Run As identity that is propagated to subsequent Jakarta Enterprise Beans components, rather than the original caller identity.

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. However, the Jakarta Authorization specification does specify the relationship between the Run As identity and the access control context used by the Java SE security manager.

3.6. Deployment Requirements

All Jakarta EE products must implement the access control semantics described in all included component specifications, such as the Jakarta Enterprise Beans, Jakarta Server Pages, and Jakarta Servlet specifications; provide a means of mapping the security roles specified in metadata annotations or the deployment descriptor to the actual roles exposed by a Jakarta EE product; and support the default mapping from user groups to roles defined by the Jakarta Security specification.

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

As described in the Jakarta Enterprise Beans specification and the Servlet specification, a Jakarta EE product must provide a deployment tool or tools capable of assigning the security roles in metadata annotations or 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 metadata annotations or 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 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.

3.7.2. Instance-based Access Control

Some applications need to control access to their data based on the content of the data, rather than simply the type of the data. We refer to this as "instance-based" rather than "class-based" access control. We hope to address this in a future release.

3.7.3. User Registration

Web-based internet applications often need to manage a set of customers dynamically, allowing users to register themselves as new customers. This scenario was widely discussed in the Servlet expert group (JSR-53) but we were unable to achieve consensus on the appropriate solution. We had to abandon this work for J2EE 1.3, and were not able to address it for J2EE 1.4, but hope to pursue it further in a future release.

---

[1] While the client is stateless with respect to authentication, the client requires that the server act as its proxy and maintain its login context. A reference to the login session state is made available to the client through cookies or URL re-writing. If SSL mutual authentication is used as the authentication protocol, the client can manage its own authentication context, and need not depend on references to the login session state.

Chapter 4. Transaction Management

This chapter describes the required Jakarta Enterprise Edition (Jakarta EE) transaction management and runtime environment.

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

If the following components are included in a Jakarta EE product, they are considered transactional resources and must behave as specified here:

- JDBC connections
- Jakarta Messaging sessions
- Resource adapter connections for resource adapters specifying the *XATransaction* transaction level

4.1. Overview

A Jakarta EE Product that includes both a servlet container and an enterprise bean container must support a transactional application comprised of combinations of web application components accessing multiple enterprise beans within a single transaction. If the Jakarta EE product also includes support for the Connectors specification, each component may also acquire one or more connections to access one or more transactional resource managers.

For example, in *Servlets/Server Pages Accessing Enterprise Beans* , the call tree starts from a servlet or server pages accessing multiple enterprise beans, which in turn may access other enterprise beans. The components access resource managers via connections.

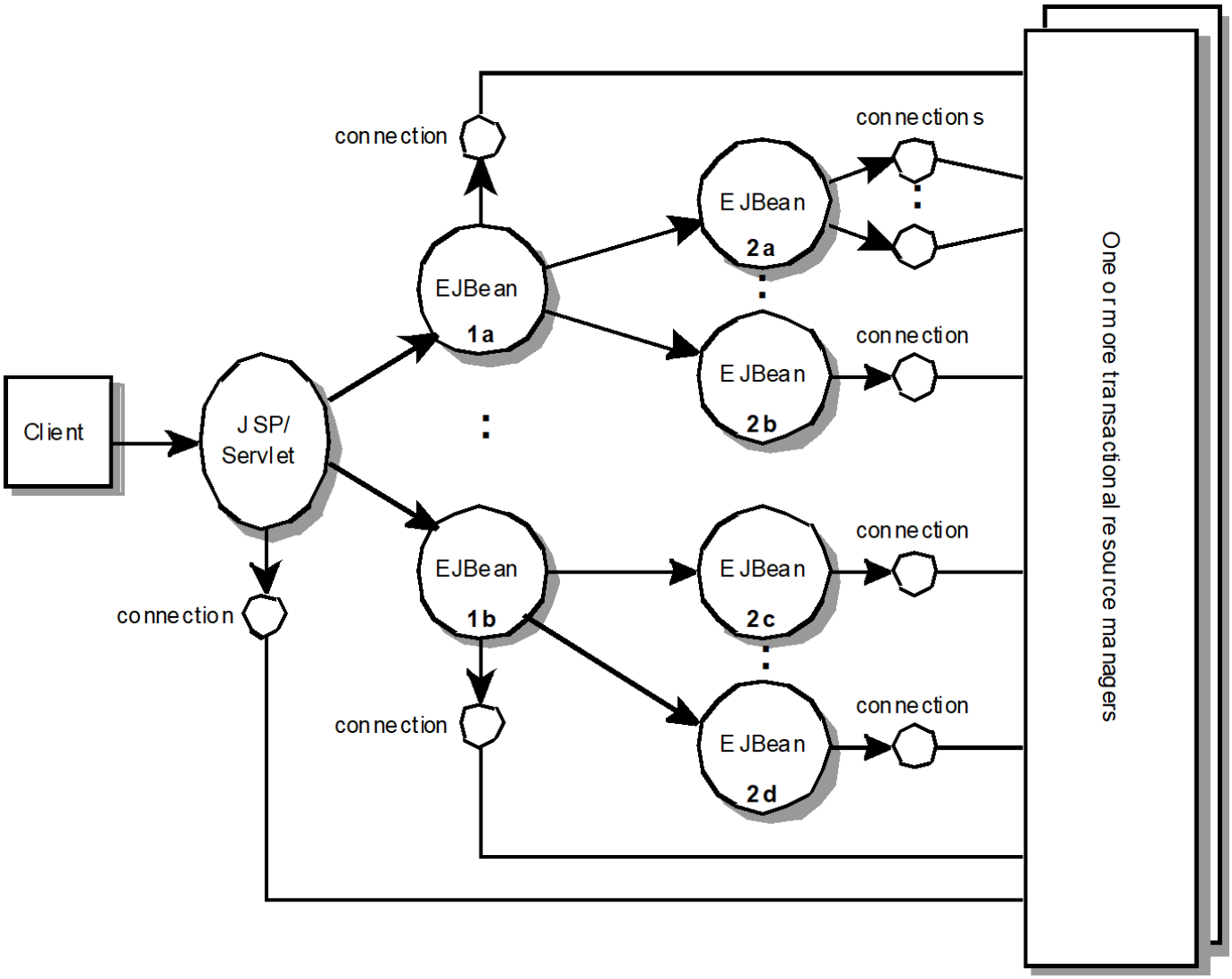

Figure 8. Servlets/Server Pages Accessing Enterprise Beans

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 *Servlets/Server Pages Accessing Enterprise Beans* 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 Jakarta EE product. In order to achieve the transactional semantics required by the application, the Jakarta EE 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, in accordance with the requirements of the component specifications.

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

4.2. Requirements

This section defines the transaction support requirements of Jakarta EE Products that must be supported by Product Providers.

4.2.1. Web Components

Web components may demarcate transactions using the *javax.transaction.UserTransaction* interface or transactional interceptors, which are defined in the Jakarta Transactions specification. They may access multiple resource managers and invoke multiple enterprise beans within a single transaction. The specified transaction context is automatically propagated to the enterprise beans and transactional resource managers. The result of the propagation may be subject to the enterprise bean transaction attributes (for example, a bean may be required to use Container Managed Transactions).

Web application event listeners and upgrade handlers must not demarcate transactions using the *javax.transaction.UserTransaction* interface or transactional interceptors. Servlet filters may use transactional resources within their *doFilter* methods but should not use any transactional resources in the methods of any objects used to wrap the request or response objects.

4.2.1.1. Transaction Requirements

The Jakarta EE platform must meet the following requirements:

- The Jakarta EE platform must provide an object implementing the *javax.transaction.UserTransaction* interface to all web components. 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*.
- The Jakarta EE platform must provide classes that implement the transactional interceptors, as defined by the Jakarta Transactions specification.
- If a web component invokes an enterprise bean from a thread associated with a transaction defined by the Jakarta Transactions specification, the Jakarta EE 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 Jakarta Enterprise Beans specification.

Note that this transaction propagation requirement applies only to invocations of enterprise beans in the same Jakarta EE product instance [2] as the invoking component. Invocations of enterprise beans in another Jakarta EE product instance (for example, using the Jakarta Enterprise Beans interoperability protocol) need not propagate the transaction context. See the Jakarta Enterprise Beans specification for details.

- If a web component accesses a transactional resource manager from a thread associated with a transaction defined by the Jakarta Transactions specification, the Jakarta EE platform must ensure that the resource access is included as part of the transaction defined by the Jakarta Transactions specification.

- If a web component creates a thread, the Jakarta EE platform must ensure that the newly created thread is not associated with any transaction defined by the Jakarta Transactions specification.

4.2.1.2. Transaction Non-Requirements

The Product Provider is not required to support the importing of a transaction context from a client to a web component.

The Product Provider is not required to support transaction context propagation via an HTTP request across web components. The HTTP protocol does not support such transaction context propagation. When 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.

However, when a web component is invoked through the *RequestDispatcher* interface, any active transaction context must be propagated to the called servlet or server pages.

4.2.2. Transactions in Web Component Life Cycles

Transactions may not span web requests from a client on the network. If a web component starts a transaction in the *service* or *doFilter* method (or transactional interceptor of *service* or *doFilter* method), it must be completed before the *service* or *doFilter* method returns to the network client.[3] Returning from the *service* or *doFilter* method to the network client with an active transaction context is an error. The web container is required to detect this error and abort the transaction.

As specified above in Transaction Non-Requirements, requests made within a web container using the *RequestDispatcher* must propagate any transaction context to the called class. Unless the called class commits or aborts the transaction, the transaction must remain active when the called class returns.

If a servlet that is called via the *RequestDispatcher* starts a transaction, the behavior of the container with regard to that transaction is unspecified when the servlet returns from its service method. The web container may throw an exception to the caller, abort the transaction and return to the caller without error, or propagate the transaction context back to the caller. Portable servlets will complete any transaction they start before returning from the service method.

4.2.3. Transactions and Threads

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.

- Transactions defined by the Jakarta Transactions specification should be started and completed in the thread in which the *service* method is called. Additional threads that are created for any purpose should not attempt to start transactions defined by the Jakarta Transactions specification.
- 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 (for example, JDBC *Connection* objects) should not be stored in static fields. Such objects can only be associated with one transaction at a time. Storing them in static fields would make it easy to erroneously share them between threads in different transactions.
- Web components implementing *SingleThreadModel* may store top-level transactional resource objects in class instance fields. A top-level object is one acquired directly from a container managed connection factory object (for example, a JDBC *Connection* acquired from a JDBC *ConnectionFactory*), as opposed to other objects acquired from these top-level objects (for example, a JDBC *Statement* acquired from a JDBC *Connection*). The web container ensures that requests to a *SingleThreadModel* servlet are serialized and thus only one thread and one transaction will be able to use the object at a time, and that the top-level object will be enlisted in any new transaction started by the component.
- In web components not implementing *SingleThreadModel* , transactional resource objects, as well as Java Persistence *EntityManager* objects, should not be stored in class instance fields, and should be acquired and released within the same invocation of the *service* method.
- Web components that are called by other web components (using the *forward* or *include* methods) should not store transactional resource objects in class instance fields.
- Enterprise beans may be invoked from any thread used by a web component. Transaction context propagation requirements are described above and in the Jakarta Enterprise Beans specification.

4.2.4. Jakarta Enterprise Beans Components

The Jakarta EE Product Provider must provide support for transactions as defined in the Jakarta Enterprise Beans specification.

4.2.5. Application Clients

The Jakarta EE Product Provider is not required to provide transaction management support for application clients.

4.2.6. Applet Clients

The Jakarta EE Product Provider is not required to provide transaction management support for applets.

4.2.7. Transactional JDBC™ Technology Support

A Jakarta EE product must support a JDBC technology database as a transactional resource manager. The platform must enable transactional JDBC API access from web components 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.

A Jakarta EE product must provide a transaction manager that is capable of coordinating two-phase commit operations across multiple XA-capable JDBC databases. If a JDBC driver supports the Jakarta Transactions API's XA interfaces (in the *javax.transaction.xa* package), then the Jakarta EE product must be capable of using the XA interfaces provided by the JDBC driver to accomplish two-phase commit operations. The Jakarta EE product may discover the XA capabilities of JDBC drivers through product-specific means, although normally such JDBC drivers would be delivered as resource adapters using the Connector API.

4.2.8. Transactional Jakarta Messaging Support

A Jakarta EE product must support a Jakarta Messaging provider as a transactional resource manager. The platform must enable transactional Jakarta Messaging access from servlets, server pages, and enterprise beans.

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

4.2.9. Transactional Resource Adapter (Connector) Support

A Jakarta EE product must support resource adapters that use *XATransaction* mode as transactional resource managers. The platform must enable transactional access to the resource adapter from servlets, server 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 Jakarta EE Platform Interoperability

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

We recommend that Jakarta EE Product Providers use the IIOP transaction propagation protocol defined by OMG and described in the OTS specification, transaction interoperability when using the Jakarta Enterprise Beans interoperability protocol based on RMI-IIOP.

4.3.2. Support for Transactional Resource Managers

This specification requires all Jakarta EE products to support the *javax.transaction.xa.XAResource* interface, as specified in the Connector specification. This specification also requires all Jakarta EE products to support the *javax.transaction.xa.XAResource* interface for performing two-phase commit operations on JDBC drivers that support the JTA XA APIs. This specification does not require that JDBC drivers or Jakarta Messaging providers use the *javax.transaction.xa.XAResource* interface, although they may use this interface and in all cases they must meet the transactional resource manager requirements described in this chapter. In particular, it must be possible to combine operations on one or more JDBC databases, one or more Jakarta Messaging sessions, one or more enterprise beans, and multiple resource adapters supporting the *XATransaction* mode in a single transaction defined by the Jakarta Transactions specification.

4.4. Local Transaction Optimization

4.4.1. Requirements

If a transaction uses a single resource manager, performance may be improved by using a resource manager specific local optimization. A local transaction is typically more efficient than a global transaction and provides better performance. Local optimization is not available for transactions that are imported from a different container.

Containers may choose to provide local transaction optimization, but are not required to do so. Local transaction optimization must be transparent to a Jakarta EE application.

The following section describes a possible mechanism for local transaction optimization by containers.

4.4.2. A Possible Design

This section illustrates how the previously described requirements might be implemented.

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 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:

- If only a single resource manager had been accessed as part of the transaction, the transaction is completed using the resource manager specific local transaction mechanism.
- If a global transaction had been started, 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 Jakarta EE application use the same resource manager, containers may choose to provide connection sharing within the same transaction scope. Sharing connections typically results in efficient usage of resources and better performance. Containers are required to provide connection sharing in certain situations; see the Connector specification for details.

Connections to resource managers acquired by Jakarta EE applications are considered potentially shared or shareable. A Jakarta EE application component that intends to use a connection in an unshareable way must provide deployment information to that effect, to prevent the connection from being shared by the container. Examples of when this may be needed include situations with changed security attributes, isolation levels, character settings, and localization configuration. Containers must not attempt to share connections that are marked unshareable. If a connection is not marked unshareable, it must be transparent to the application whether the connection is actually shared or not.

Jakarta EE application components may use the optional *shareable* element of the *Resource* annotation or the optional deployment descriptor element *res-sharing-scope* to indicate whether a connection to a resource manager is shareable or unshareable. Containers must assume connections to be shareable if no deployment hint is provided. Jakarta EE Application Client XML Schema, the Jakarta Enterprise Beans specification, and the Servlet specification provide descriptions of the deployment descriptor element.

Jakarta EE application components may cache connection objects and reuse them across multiple transactions. Containers that provide connection sharing must 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. JDBC and Jakarta Messaging Deployment Issues

The JDBC transaction requirements in Transactional JDBC™ Technology Support and the Jakarta Messaging transaction requirements in Transactional Jakarta Messaging Support may impose some restrictions on a Deployer's configuration of an application's JDBC and Jakarta Messaging resources. Jakarta EE Product Providers may impose the restrictions described in this section to meet these requirements.

If the deployer configures a non-XA-capable JDBC resource manager in a transaction, then a Jakarta EE Product Provider may restrict all JDBC access within that transaction to that non-XA-capable JDBC resource manager. Otherwise, a Jakarta EE Product Provider must support use of multiple XA-capable JDBC resource managers within a transaction. In addition, a Jakarta EE Product Provider may restrict the security configuration of all JDBC connections within a transaction to a single user identity. A Jakarta EE Product Provider is not required to support transactions where more than one JDBC identity is used. Specifically, this means that transactions that require the use of more than one JDBC security identity (which can be done explicitly via component provided user name and password) may not be portable.

A Jakarta EE Product Provider may make the same restrictions as above, resulting in a transaction being restricted to a single Jakarta Messaging resource manager and user identity.

In addition, when both a JDBC resource manager and a Jakarta Messaging resource manager are used in the same transaction, a Jakarta EE Product Provider may restrict both to a pairing that allows their combination to deliver the full transactional semantics required by the application, and may restrict the security identity of both to a single identity. To fully support such usage, portable applications that wish to include JDBC and Jakarta Messaging access in a single global transaction must not mark the corresponding transactional resources as “unshareable”.

Although these restrictions are allowed, it is recommended that Jakarta EE Product Providers support JDBC and Jakarta Messaging resource managers that provide full two-phase commit functionality and, as a result, do not impose these restrictions.

4.7. Two-Phase Commit Support

A Jakarta EE product must support the use of multiple XA-capable resource adapters in a single transaction. To support such a scenario, full two-phase commit support is required. A Jakarta Messaging provider may be provided as an XA-capable resource adapter. In such a case, it must be possible to include Jakarta Messaging operations in the same global transaction as other resource adapters. While JDBC drivers are not required to be XA-capable, a JDBC driver may be delivered as an XA-capable resource adapter. In such a case, it must be possible to include JDBC operations in the same global transaction as other XA-capable resource adapters. See also Transactional JDBC™ Technology Support.

4.8. System Administration Tools

Although there are no compatibility requirements for system administration capabilities, the Jakarta EE 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).

[2] A product instance corresponds to a single installation of a Jakarta™ EE product. A single product instance might use multiple operating system processes, or might support multiple host machines as part of a distributed container. In contrast, it might be possible to run multiple instances of a product on a single host machine, or possibly even in a single Java virtual machine, for example, as part of a virtual hosting solution. The transaction propagation requirement applies within a single product instance and is independent of the number of Java virtual machines, operating system processes, or host machines used by the product instance.
[3] For a Jakarta™ Server Pages page, this requirement applies to the *service* method of the equivalent Jakarta™ Server Pages page Implementation Class.

Chapter 5. Resources, Naming, and Injection

This chapter describes how applications declare dependencies on external resources and configuration parameters, and how those items are represented in the Jakarta EE naming system and can be injected into application components. These requirements are based on annotations defined in the Java Metadata specification and features defined in the Java Naming and Directory Interface™ (JNDI) specification. The *Resource* annotation described here is defined in more detail in the Jakarta Annotations specification. The *EJB* annotation described here is defined in more detail in the Enterprise JavaBeans specification. The *PersistenceUnit* and *PersistenceContext* annotations described here are defined in more detail in the Java Persistence specification. The *Inject* annotation described here is defined in the Dependency Injection for Java specification, and its usage in Jakarta EE applications is defined in the CDI specification.

5.1. Overview

The requirements defined in this chapter address the following two issues:

- The Application Assembler and Deployer should be able to customize the behavior of an application's business logic without accessing the application's source code. Typically this will involve specification of parameter values, connection to external resources, and so on. Deployment descriptors provide this capability
- Applications must be able to access resources and external information in their operational environment without knowledge of how the external information is named and organized in that environment. The JNDI naming context and Java language annotations provide this capability.

5.1.1. Chapter Organization

The following sections contain the Jakarta EE platform solutions to the above issues:

- JNDI Naming Context," defines general rules for the use of the JNDI naming context and its interaction with Java language annotations that reference entries in the naming context.
- Responsibilities by Jakarta EE Role," defines the general responsibilities for each of the Jakarta EE roles such as Application Component Provider, Application Assembler, Deployer, and Jakarta EE Product Provider.
- Simple Environment Entries," defines the basic 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.
- Jakarta Enterprise Beans References," defines the interfaces for obtaining the business interface, no-interface view, or home interface of an enterprise bean using a Jakarta Enterprise Bean reference. A Jakarta Enterprise Bean reference is a special entry in the application component's environment.
- Web Service References," refers to the specification for web service references.

- Resource Manager Connection Factory References," 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.
- Resource Environment References," defines the interfaces for obtaining an administered object that is associated with a resource using a resource environment reference. A resource environment reference is a special entry in the application component's environment.
- Message Destination References," defines the interfaces for declaring and using message destination references.
- UserTransaction References," describes the use by eligible application components of references to a *UserTransaction* object in the component's environment to start, commit, and abort transactions.
- TransactionSynchronizationRegistry References," describes the use by eligible application components of references to a *TransactionSynchronizationRegistry* object in the component's environment.
- ORB References," describes the use by eligible application components of references to a CORBA *ORB* object in the component's environment.
- Persistence Unit References," describes the use by eligible application components of references to an *EntityManagerFactory* object in the component's environment.
- Persistence Context References," describes the use by eligible application components of references to an *EntityManager* object in the component's environment.
- Application Name and Module Name References," describes the use by eligible application components of references to the names of the current application and module.
- Application Client Container Property," describes the use by eligible application components of references to the application client container property.
- Validator and Validator Factory References," describes the use by eligible application components of references to the *Validator* and *ValidatorFactory* objects in the component's environment.
- Resource Definition and Configuration," describes the use by eligible application components of metadata that may be used to define resources in the component's environment.
- DataSource Resource Definition," describes the use by eligible application components of references to *DataSource* resources in the component's environment.
- Jakarta Messaging Connection Factory Resource Definition," describes the use by eligible application components of references to Jakarta Messaging *ConnectionFactory* resources in the component's environment.
- Jakarta Messaging Destination Definition," describes the use by eligible application components of references to Jakarta Messaging *Destination* resources in the component's environment.
- Mail Session Definition," describes the use by eligible application components of references to Mail *Session* resources in the component's environment.
- Connector Connection Factory Definition," describes the use by eligible application components of

references to Connector connection factory resources in the component's environment.

- Connector Administered Object Definition," describes the use by eligible application components of references to Connector administered object resources in the component's environment.
- Default Data Source," describes the use by eligible application components of references to the default DataSource in the component's environment.
- Default Jakarta Messaging Connection Factory," describes the use by eligible application components of references to the default Jakarta Messaging ConnectionFactory in the component's environment.
- Default Jakarta Concurrency Objects," describes the use by eligible application components of references to the default Jakarta Concurrency objects in the component's environment.
- Managed Bean References," describes the use by eligible application components of references to Managed Beans.
- Bean Manager References," describes the use by eligible application components of references to a *BeanManager* object in the component's environment.
- Support for Dependency Injection," describes support for the use of the Dependency Injection APIs.

5.1.2. Required Access to the JNDI Naming Environment

Jakarta EE application clients, enterprise beans, and web components are required to have access to a JNDI naming environment.[4] The containers for these application component types are required to provide the naming environment support described here.

Annotations and deployment descriptors are the main vehicles for conveying access information to the Application Assembler and Deployer about application components' requirements for customization of business logic and access to external information. The annotations described here are available for use by all application component types. The deployment descriptor entries described here are present in identical form in the deployment descriptor schemas for each of these application component types. See the corresponding specification of each application component type for the details.

5.2. 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. Use of 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.

5.2.1. The Application Component's Environment

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 make use of entries from the environment. The business methods may access the environment using the JNDI interfaces or lookup methods on component-specific context objects. Also, entries from the environment may be injected into the application component's fields or methods. The Application Component Provider declares in the deployment descriptor, or via annotations, all the environment entries that the application component expects to be provided in its environment at runtime. For each environment entry, the Application Component Provider can also specify in the deployment descriptor, or via annotations, the JNDI name of another environment entry whose value should be used to initialize the environment entry being defined ("lookup" functionality).
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 or via annotations. The Deployer can set and modify the values of the environment entries. As part of this process, the Deployer is allowed to override any "lookup" information associated with the application component.
4. The container injects entries from the environment into application component fields or methods as specified by the application component's deployment descriptor or by annotations on the application component class.
5. The container also makes the environment naming context available to the application component instances at runtime. The application component's instances may use the JNDI interfaces or component context lookup methods to obtain the values of the environment entries. __

5.2.2. Application Component Environment Namespaces

The application component's naming environment is composed of four logical namespaces, representing naming environments with different scopes. The four namespaces are: __

- *java:comp* – Names in this namespace are per-component (for example, per enterprise bean). Except for components in a web module, each component gets its own *java:comp* namespace, not shared with any other component. Components in a web module do not have their own private component namespace. __ See note below.
- *java:module* – Names in this namespace are shared by all components in a module (for example, all enterprise beans in a single enterprise bean module, or all components in a web module). __
- *java:app* – Names in this namespace are shared by all components in all modules in a single application, where "single application" means a single deployment unit, such as a single ear file, a single module deployed standalone, etc. For example, a war file and a Jakarta Enterprise Beans jar file in the same ear file would both have access to resources in the *java:app* namespace. __
- *java:global* – Names in this namespace are shared by all applications deployed in an application server instance. Note that an application server instance may represent a single server, a cluster of servers, an administrative domain containing many servers, or even more. The scope of an application server instance is product-dependent, but it must be possible to deploy multiple

applications to a single application server instance.

Note that in environments in which an application is deployed multiple times—such as, for example, in cloud environments, where multiple instances of the same application might be deployed on behalf of multiple tenants—the namespace for each application instance would be disjoint from the namespace of any other instance of that same application.

For historical reasons, the *java:comp* namespace is shared by all components in a web module. To preserve compatibility, this specification doesn't change that. In a web module, *java:comp* refers to the same namespace as *java:module* . It is recommended that resources in a web module that are intended to be shared by more than one component be declared in the *java:module/env* namespace.

Note that an application client is a module with only a single component.

Note also that resource adapter (connector) modules may not define resources in any of the component namespaces, but may look up resources defined by other components. All the *java:* namespaces accessible in a resource adapter are the namespaces of the component that called the resource adapter (when called in the context of a component).

If multiple application components declare an environment entry in one of the shared namespaces, all attributes of that entry must be identical in each declaration. For example, if multiple components declare a resource reference with the same *java:app* name, the *authentication* and *shareable* attributes must be identical.

If all attributes of each declaration of a shared environment entry are not identical, this must be reported as a deployment error to the Deployer. The deployment tool may allow the Deployer to correct the error and continue deployment.

The default JNDI namespace for resource references and resource definitions must always be *java:comp/env* . Note that this applies to both the case where no name has been supplied so the rules for choosing a default name are used, and the case where a name has been supplied explicitly but the name does not specify a *java:* namespace. Since the *java:comp* namespace is not available in some contexts, use of that namespace in such a context should result in a deployment error. Likewise, the *java:module* namespace is not valid in some contexts; use of that namespace in such contexts should result in a deployment error. Environment entries may be declared in any one of the defined namespaces by explicitly including the namespace prefix before the name.

It is recommended but not required that environment entries be created in the *env* subcontext of the corresponding naming context. For example, entries shared within a module should be declared in the *java:module/env* context. Note that names that are not under the *env* subcontext may conflict with the current or future versions of this specification, with server-defined names, such as the names of applications or modules, or with server-defined resources. Names in the *env* subcontexts of any of the namespaces must only be created by an explicit declaration in an application or by an explicit action by an administrator; the application server must not predefine any names in the *env* subcontext of any of the namespaces, or in any subcontext of any such *env* context.

An environment entry declared in the *application.xml* descriptor must specify a JNDI name in the

java:app or *java:global* namespace, for example: *java:app/env/myString* or *java:global/someValue* . The specification of a *java:comp* or *java:module* name for an environment entry declared in the *application.xml* descriptor must be reported as a deployment error to the Deployer.

A Jakarta EE product may impose security restrictions on access of resources in the shared namespaces. However, it must be possible to deploy applications that define resources in the shared namespaces that are usable by different entities at the given scope. For example, it must be possible to deploy an application that defines a resource, using various forms of metadata declaration, in the *java:global* namespace that is usable by a separate application.

5.2.3. Accessibility of Environment Entry Types

All objects defined in environment entries of any kind (either in deployment descriptors or through annotations) must be specified to be of a Java type that is accessible to the component. Accessibility of Java classes is specified in section Class Loading Requirements." If the object is of type *java.lang.Class* , the *Class* object must refer to a class that is accessible to the component. Note that in cases where the container may return an implementation subtype of the requested type, the implementation subtype might not be accessible to the component.

5.2.4. Sharing of Environment Entries

Each application component defines its own set of dependencies that must appear as entries in the application component's environment. All instances of an application component within the same application instance within the same container share the same environment entries. Application component instances are not allowed to modify the environment at runtime.

In general, lookups of objects in the JNDI *java:* namespace are required to return a new instance of the requested object every time. Exceptions are allowed for the following:

- The container knows the object is immutable (for example, objects of type *java.lang.String*), or knows that the application can't change the state of the object.
- The object is defined to be a singleton, such that only one instance of the object may exist in the JVM.
- The name used for the lookup is defined to return an instance of the object that might be shared. The names *java:comp/ORB* , *java:comp/ValidatorFactory* , and *java:comp/BeanManager* are such names.

In these cases, a shared instance of the object may be returned. In all other cases, a new instance of the requested object must be returned on each lookup. Note that, in the case of resource adapter connection objects, it is the resource adapter's *ManagedConnectionFactory* implementation that is responsible for satisfying this requirement.

Each injection of an object corresponds to a JNDI lookup. Whether a new instance of the requested object is injected, or whether a shared instance is injected, is determined by the rules described above.

—

5.2.5. Annotations and Injection

As described in the following sections, a field or method of certain container-managed component classes may be annotated to request that an entry from the application component's environment be injected into the class. The specifications for the different containers indicate which classes are considered container-managed classes; not all classes of a given type are necessarily managed by the container.

Any of the types of resources described in this chapter may be injected. Injection may also be requested using entries in the deployment descriptor corresponding to each of these resource types. The field or method may have any access qualifier (*public* , *private* , etc.). For all classes except application client main classes, the fields or methods must not be *static* . Because application clients use the same lifecycle as Java SE applications, no instance of the application client main class is created by the application client container. Instead, the *static main* method is invoked. To support injection for the application client main class, the fields or methods annotated for injection must be *static* .

A field of a class may be the target of injection. The field must not be *final* . By default, the name of the field is combined with the fully qualified name of the class and used directly as the name in the application component's naming context. For example, a field named *myDatabase* in the class *MyApp* in the package *com.example* would correspond to the JNDI name *java:comp/env/com.example.MyApp/myDatabase* . The annotation also allows the JNDI name to be specified explicitly. When a deployment descriptor entry is used to specify injection, the JNDI name and the field name are both specified explicitly. Note that, by default, the JNDI name is relative to the *java:comp/env* naming context.

Environment entries may also be injected into a class through methods that follow the naming conventions for JavaBeans properties. The annotation is applied to the *set* method for the property, which is the method that is called to inject the environment entry into the class. The JavaBeans property name (not the method name) is used as the default JNDI name. For example, a method named *setMyDatabase* in the same *MyApp* class would correspond to the same JNDI name *java:comp/env/com.example.MyApp/myDatabase* as the field *myDatabase* .

Each resource may only be injected into a single field or method of a given name in a given class. Requesting injection of the *java:comp/env/com.example.MyApp/myDatabase* resource into both the *setMyDatabase* method and the *myDatabase* field is an error. Note, however, that either the field or the method could request injection of a resource of a different (non-default) name. By explicitly specifying the JNDI name of a resource, a single resource may be injected into multiple fields or methods of multiple classes.

The specifications for the various application component types describe which classes may be annotated for injection, as summarized in Component classes supporting injection.

The component classes listed in Component classes supporting injection with support level "Standard" all support Jakarta EE resource injection, as well as PostConstruct and PreDestroy callbacks. In addition, if CDI is enabled—which it is by default—these classes also support CDI injection, as described in Support for Dependency Injection", and the use of interceptors.[5] The component classes

listed with support level “Limited” only support Jakarta EE field injection and the PostConstruct callback. Note that these are application client main classes, where field injection is into static fields.

The specifications for the various application component types also describe when injection occurs in the lifecycle of the component. Typically injection will occur after an instance of the class is constructed, but before any business methods are called. If the container fails to find a resource needed for injection, initialization of the class must fail, and the class must not be put into service.

Table 1. Component classes supporting injection

Spec	Classes supporting injection	Support level
Servlet	servlets	Standard
	servlet filters	Standard
	event listeners	Standard
	HTTP upgrade handlers	Standard
Jakarta Server Pages	tag handlers	Standard
	tag library event listeners	Standard
Jakarta Server Faces	managed classes [6]	Standard
JAX-WS	service endpoints	Standard
	handlers	Standard
Jakarta RESTful Web Services	Jakarta RESTful Web Services components [7]	Standard
WebSocket	endpoints	Standard
Jakarta Enterprise Beans	beans	Standard
Interceptor	interceptors [8]	Standard
Java Persistence	entity listeners	Standard
Managed Beans	managed beans	Standard
CDI [9]	CDI-style managed beans [10]	Standard
	decorators [11]	Standard
Jakarta EE platform	main class (static)	Limited
	login callback handler	Standard

Annotations may also be applied to the class itself. These annotations declare an entry in the application component’s environment but do not cause the resource to be injected. Instead, the

application component is expected to use JNDI or a component context lookup method to lookup the entry. When the annotation is applied to the class, the JNDI name and the environment entry type must be specified explicitly. __

Resource annotations may appear on any of the classes listed above, or on any superclass of any class listed above. A resource annotation on any class in the inheritance hierarchy defines a resource needed by the application component. However, injection of resources follows the Java language overriding rules for visibility of fields and methods. A method definition that overrides a method on a superclass defines the resource, if any, to be injected into that method. An overriding method may request injection even though the superclass method does not request injection, it may request injection of a different resource than is requested by the superclass, or it may request no injection even though the superclass method requests injection.

In addition, fields or methods that are not visible in or are hidden (as opposed to overridden) by a subclass may still request injection. This allows, for example, a private field to be the target of injection and that field to be used in the implementation of the superclass, even though the subclass has no visibility into that field and doesn't know that the implementation of the superclass is using an injected resource. Note a declaration of a field in a subclass with the same name as a field in a superclass always causes the field in the superclass to be hidden.

In some cases a class may need to perform initialization of its own after all resources have been injected. To support this case, one method of the class may be annotated with the *PostConstruct* annotation (or, equivalently, specified using the *post-construct* entry of a deployment descriptor). This method will be called after all injections have occured and before the class is put into service. This method will be called even if the class doesn't request any resources to be injected. Similarly, for classes whose lifecycle is managed by the container, the *PreDestroy* annotation (or, equivalently, the *pre-destroy* entry of a deployment descriptor) may be applied to one method that will be called when the class is taken out of service and will no longer be used by the container. Each class in a class hierarchy may have *PostConstruct* and *PreDestroy* methods. The order in which the methods are called matches the order of the class hierarchy with methods on a superclass being called before methods on a subclass.

The *PostConstruct* and *PreDestroy* annotations are specified by the Jakarta Annotations specification. All classes that support injection also support the *PostConstruct* annotation. All classes for which the container manages the full lifecycle of the object also support the *PreDestroy* annotation.

Starting with Java EE 7, CDI support is enabled by default. CDI bean-defining annotations and the *beans.xml* descriptor are used to determine which classes are CDI beans and eligible for injection into other objects. Similarly, the annotation metadata and the *beans.xml* descriptor are used by CDI to determine which interceptors are eligible to be applied. See the CDI specification and the Interceptors specification for the rules that determine which classes are CDI beans and the treatment of interceptors.

5.2.6. Annotations and Deployment Descriptors

Environment entries may be declared by use of annotations, without need for any deployment

descriptor entries. Environment entries may also be declared by deployment descriptor entries. The same environment entry may be declared using both an annotation and a deployment descriptor entry. In this case, the information in the deployment descriptor entry may be used to override some of the information provided in the annotation. This approach may be used by an Application Assembler or Deployer to override information provided by the Application Component Developer. Applications should not use deployment descriptor entries to request injection of a resource into a field or method that has not been designed for injection.

The following list describes the rules for how a deployment descriptor entry may override a *Resource* annotation.

- The relevant deployment descriptor entry is located based on the JNDI name used with the annotation (either defaulted or provided explicitly).
- The type specified in the deployment descriptor must be assignable to the type of the field or property.
- The description, if specified, overrides the description element of the annotation.
- The injection target, if specified, defines additional injection points for the resource.
- The *mapped-name* element, if specified, overrides the *mappedName* element of the annotation.
- The *res-sharing-scope* element, if specified, overrides the *shareable* element of the annotation. In general, the Application Assembler or Deployer should not change this value as doing so is likely to break the application.
- The *res-auth* element, if specified, overrides the *authenticationType* element of the annotation. In general, the Application Assembler or Deployer should not change this value as doing so is likely to break the application.
- The *lookup-name* element, if specified, overrides the *lookup* element of the annotation.

It is an error to request injection of two resources into the same target. The behavior of an application that does so is undefined.

The rules for how a deployment descriptor entry may override an *EJB* annotation are included in the Jakarta Enterprise Beans specification. The rules for how a deployment descriptor entry may override a *WebServiceRef* annotation are included in the Web Services for Jakarta EE specification.

A PostConstruct method may be specified using either the *PostConstruct* annotation on the method or the *post-construct* deployment descriptor entry. Similarly, a PreDestroy method may be specified using either the *PreDestroy* annotation on the method or the *pre-destroy* deployment descriptor entry.

5.2.7. Other Naming Context Entries

In addition to environment entries declared by application components, other items will appear in the naming context, as specified by this and other specifications. Following are some of these entries. This is not an exhaustive list; consult the corresponding specification for details.

- All enterprise beans in an application are given entries in the shared namespaces. See the Jakarta Enterprise Beans specification for details.
- All web applications are given names in the shared namespaces. The names correspond to the complete URL of the web application. See the Servlet specification for details.
- Objects representing several container services are defined in the *java:comp* namespace. See, for example, UserTransaction References," TransactionSynchronizationRegistry References," and ORB References."
- Strings providing the current module name and application name are defined in the *java:comp* namespace. See Application Name and Module Name References."

5.3. Responsibilities by Jakarta EE Role

This section describes the responsibilities for each Jakarta EE role that apply to all uses of the Jakarta EE naming context. The sections that follow describe the responsibilities that are specific to the different types of objects that may be stored in the naming context.

5.3.1. Application Component Provider's Responsibilities

The Application Component Provider may make use of three techniques for accessing and managing the naming context. First, the Application Component Provider may use Java language annotations to request injection of a resource from the naming context, or to declare elements that are needed in the naming context. Second, the component may use the JNDI APIs to access entries in the naming context. Third, deployment descriptor entries may be used to declare entries needed in the naming context, and to request injection of these entries into application components. Deployment descriptor entries may also be used to override information provided by annotations.

As part of the declaration of elements in the naming context, the Application Component Provider can specify the JNDI name of a resource to be looked up in the naming context to initialize the element being declared. The JNDI name in question may belong to any of the namespaces that compose the application component environment.

To ensure that it has access to the correct *javax.naming.InitialContext* implementation provided by the container, a portable application component must not specify the *java.naming.factory.initial* property, must not specify a *URLContextFactory* for the *"java"* scheme-id, and must not call the *javax.naming.spi.NamingManager.setInitialContextFactoryBuilder* method.

5.3.2. Application Assembler's Responsibilities

The Application Assembler is allowed to modify the entries in the naming context set by the Application Component Provider, and is allowed to set the values of those entries for which the Application Component Provider has not specified any values. The Application Assembler may use the deployment descriptor to override settings made by the Application Component Provider in the source code using annotations.

5.3.3. Deployer's Responsibilities

The Deployer must ensure that all the entries declared by an application component are created and properly initialized.

The Deployer can modify the entries that have been previously set by the Application Component Provider and/or Application Assembler, and must set the values of those entries for which a required value has not been specified. If an annotation contains the *lookup* element or a deployment descriptor entry includes the *lookup-name* element, the Deployer should bind it to the entry specified as the target of the lookup. Deployment should fail if the *lookup* element of an annotation or the *lookup-name* element in a deployment descriptor entry does not specify a name with an explicit *java:* namespace. The Deployer may also use product-specific resource mapping tools, deployment descriptors, rules, or capabilities to bind resource reference entries to resources in the target operational environment.

The *description* deployment descriptor elements and annotation elements provided by the Application Component Provider or Application Assembler help the Deployer with this task.

5.3.4. Jakarta EE Product Provider's Responsibilities

The Jakarta EE Product Provider has the following responsibilities:

- Provide a deployment tool that allows the Deployer to set and modify the entries of the application component's naming context.
- Implement the *java:comp* , *java:module* , *java:app* and *java:global* environment naming contexts, and provide them to the application component instances at runtime. The naming context must include all the 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. Certain entries in the naming context may have to be initialized with the values of other entries, specifically when the "lookup" facility is used. In this case, it is an error if there are any circular dependencies between entries. Similarly, it is an error if looking up the specified JNDI name results in a resource whose type is not compatible with the entry being created. The deployment tool may allow the deployer to correct either of these classes of errors and continue the deployment.
- Ensure that, in the absence of any properties specified by the application, the *javax.naming.InitialContext* implementation meets the requirements described in this specification.
- Inject entries from the naming environment into the application component, as specified by the deployment descriptor or annotations on the application component classes.
- The container must ensure that the application component instances have only read access to their naming context. 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.

5.4. Simple Environment Entries

A simple environment entry is a configuration parameter used to customize an application component's business logic. The environment entry values may be one of the following Java types: *String* , *Character* , *Byte* , *Short* , *Integer* , *Long* , *Boolean* , *Double* , *Float* , *Class* , and any subclass of *Enum* .

The following subsections describe the responsibilities of each Jakarta EE Role.

5.4.1. 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 three sections, the first describing annotations for injecting environment entries, the second describing the API for accessing environment entries, and the third describing syntax for declaring the environment entries in a deployment descriptor.

Injection of Simple Environment Entries

A field or a method of an application component may be annotated with the *Resource* annotation. The name and type of the environment entry are as described above. Note that the container will unbox the environment entry as required to match it to a primitive type used for the injection field or method. The *authenticationType* and *shareable* elements of the *Resource* annotation must not be specified; simple environment entries are not shareable and do not require authentication.

The following code example illustrates how an application component uses annotations to declare environment entries.

```
// The maximum number of tax exemptions, configured by the Deployer.
@Resource int maxExemptions;
// The minimum number of tax exemptions, configured by the Deployer.
@Resource int minExemptions;

public void setTaxInfo(int numberOfExemptions,...)
       throws InvalidNumberOfExemptionsException {
  ...
  // Use the environment entries to
  // customize business logic.
  if (numberOfExemptions > maxExemptions ||
         numberOfExemptions < minExemptions)
    throw new InvalidNumberOfExemptionsException();
 ...
}
```

The following code example illustrates how an environment entry can be assigned a value by referring

to another entry, potentially in a different namespace.

```
// an entry that gets its value from an application-wide entry
@Resource(lookup="java:app/env/timeout") int timeout;
```

Programming Interfaces for Accessing Simple Environment Entries

In addition to the injection based approach described above, an application component may access environment entries dynamically. 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 its direct or indirect subcontexts.

Environment entries have 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 (numberOfExemptions > 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");
  ...
}
```

Declaration of Simple 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 either annotations on the application component's code, or 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, which by default is relative to the *java:comp/env* context, the expected Java programming language type of the environment entry value (the type of the object returned from the JNDI *lookup* method), and an optional environment entry value.

An environment entry is scoped to the application component whose declaration contains the *env-*

entry element. This means that the environment entry is not accessible from other application components at runtime, and that other application components may define *env-entry* elements with the same *env-entry-name* without causing a name conflict.

If the Application Component Provider provides a value for an environment entry using the *env-entry-value* element, 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, or in the case of *Character* , a single character.

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>
<env-entry>
  <env-entry-name>helperClass</env-entry-name>
  <env-entry-type>java.lang.Class</env-entry-type>
  <env-entry-value>com.acme.helper.Helper</env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>timeUnit</env-entry-name>
  <env-entry-type>java.util.concurrent.TimeUnit</env-entry-type>
  <env-entry-value>NANOSECONDS</env-entry-value>
</env-entry>
<env-entry>
  <env-entry-name>bar</env-entry-name>
  <env-entry-type>java.lang.Integer</env-entry-type>
  <lookup-name>java:app/env/appBar</lookup-name>
</env-entry>
...
```

Injection of environment entries may also be specified using the deployment descriptor, without need for Java language annotations. The following example is the declaration of environment entries corresponding to the earlier injection example.

```
...
<env-entry>
  <description>
     The maximum number of tax exemptions
     allowed to be set.
  </description>
  <env-entry-name>
     com.example.PayrollService/maxExemptions
  </env-entry-name>
  <env-entry-type>java.lang.Integer</env-entry-type>
  <env-entry-value>15</env-entry-value>
  <injection-target>
    <injection-target-class>
       com.example.PayrollService
    </injection-target-class>
    <injection-target-name>
       maxExemptions
    </injection-target-name>
  </injection-target>
</env-entry>
<env-entry>
  <description>
     The minimum number of tax exemptions
     allowed to be set.
  </description>
  <env-entry-name>
     com.example.PayrollService/minExemptions
  </env-entry-name>
  <env-entry-type>java.lang.Integer</env-entry-type>
  <env-entry-value>1</env-entry-value>
  <injection-target>
    <injection-target-class>
       com.example.PayrollService
    </injection-target-class>
    <injection-target-name>
       minExemptions
    </injection-target-name>
  </injection-target>
</env-entry>
...
```

It's often convenient to declare a field or method as an injection target, but specify a default value in the code, as illustrated in the following example.

```
// The maximum number of tax exemptions, configured by the Deployer.
@Resource int maxExemptions = 4;        // defaults to 4
```

To support this case, the container must only inject a value for this resource if the deployer has specified a value to override the default value. The *env-entry-value* element in the deployment descriptor is optional when an injection target is specified. If the element is not specified, no value will be injected. In addition, if the element is not specified, the named resource is not initialized in the naming context; explicit lookups of the named resource will fail.

The deployment descriptor equivalent of the *lookup* element of the *@Resource* annotation is *lookup-name* . The following deployment descriptor fragment is equivalent to the earlier example that used *lookup* .

```
<env-entry>
  <env-entry-name>somePackage.SomeClass/timeout</env-entry-name>
  <env-entry-type>java.lang.Integer</env-entry-type>
  <injection-target>
    <injection-target-class>
       somePackage.SomeClass
    </injection-target-class>
    <injection-target-name>timeout</injection-target-name>
  </injection-target>
  <lookup-name>java:app/env/timeout</lookup-name>
</env-entry>
```

It is an error for both the *env-entry-value* and *lookup-name* elements to be specified for a given *env-entry* element. If either element exists, an eventual *lookup* element of the corresponding *Resource* annotation (if any) must be ignored. In other words, assignment of a value to an environment entry via a deployment descriptor, either directly (*env-entry-value*) or indirectly (*lookup-name*), overrides any assignments made via annotations.

5.5. Jakarta Enterprise Beans References

This section describes the programming and deployment descriptor interfaces that allow the Application Component Provider to refer to the homes of enterprise beans or to enterprise bean instances using “logical” names called Jakarta Enterprise Beans references. The Jakarta Enterprise Beans references are special entries in the application component’s naming environment. The Deployer binds the Jakarta Enterprise Beans reference to the enterprise bean’s business interface, no-interface view, or home interface in the target operational environment.

The deployment descriptor also allows the Application Assembler to link a Jakarta Enterprise Bean reference declared in one application component to an enterprise bean contained in an ejb-jar file in the same Jakarta EE application. The link is an instruction to the tools used by the Deployer describing

the binding of the Jakarta Enterprise Beans reference to the business interface, no-interface view, or home interface of the specified target enterprise bean. The same linking can also be specified by the Application Component Provider using annotations in the source code of the component.

The requirements in this section only apply to Jakarta EE products that include a Jakarta Enterprise Beans container.

5.5.1. Application Component Provider's Responsibilities

This subsection describes the Application Component Provider's view and responsibilities with respect to Jakarta Enterprise Beans references. It does so in three sections, the first describing annotations for injecting Jakarta Enterprise Beans references, the second describing the API for accessing Jakarta Enterprise Beans references, and the third describing the syntax for declaring the Jakarta Enterprise Beans references in a deployment descriptor

5.5.1.1. Injection of Jakarta Enterprise Beans Entries

A field or a method of an application component may be annotated with the *EJB* annotation. The *EJB* annotation represents a reference to a Jakarta Enterprise Beans session bean or entity bean. The reference may be to a session bean's business interface, to a session bean's no-interface view, or to the local or remote home interface of a session bean or entity bean.

The following example illustrates how an application component uses the *EJB* annotation to reference an instance of an enterprise bean. The referenced bean is a stateful session bean. The enterprise bean reference will have the name *java:comp/env/com.example.ExampleBean/myCart* in the naming context, where *ExampleBean* is the name of the class of the referencing bean and *com.acme.example* is its package. The target of the reference is not named and must be resolved by the Deployer, unless there is only one session bean component within the application that exposes a client view type that matches the Jakarta Enterprise Bean reference.

```
package com.acme.example;

@Stateless public class ExampleBean implements Example {
  ...
  @EJB private ShoppingCart myCart;
  ...
}
```

The following example illustrates use of almost all elements of the *EJB* annotation.

```
@EJB(
  name = "ejb/shopping-cart",
  beanName = "cart1",
  beanInterface = ShoppingCart.class,
  description = "The shopping cart for this application"
)
private ShoppingCart myCart;
```

As an alternative to *beanName* , a reference to an enterprise bean can use the global JNDI name for that enterprise bean, or any of the other names mandated by the Jakarta Enterprise Beans specifications, by means of the *lookup* annotation element. The following example uses a JNDI name in the application namespace.

```
@EJB(
  lookup="java:app/cartModule/ShoppingCart",
  description = "The shopping cart for this application"
)
private ShoppingCart myOtherCart;
```

If the *ShoppingCart* bean were instead written to the Jakarta Enterprise Beans 2.x client view, the Jakarta Enterprise Bean reference would be to the bean's home interface. For example:

```
@EJB(
  name="ejb/shopping-cart",
  beanInterface=ShoppingCartHome.class,
  beanName="cart1",
  description="The shopping cart for this application"
)
private ShoppingCartHome myCartHome;
```

If the *ShoppingCart* bean were instead written to the no-interface client view and implemented by bean class *ShoppingCartBean.class* , the Jakarta Enterprise Bean reference would have type *ShoppingCartBean.class* . For example:

```
@EJB(
  name="ejb/shopping-cart",
  beanInterface=ShoppingCartBean.class,
  beanName="cart1",
  description="The shopping cart for this application"
)
private ShoppingCartBean myCart;
```

5.5.1.2. Programming Interfaces for Jakarta Enterprise Beans References

The Application Component Provider may use Jakarta Enterprise Beans references to locate the business interface, no-interface view, or home interface of an enterprise bean as follows.

- Assign an entry in the application component's environment to the reference. (See subsection Declaration of Jakarta Enterprise Beans References for information on how Jakarta Enterprise Beans references are declared in the deployment descriptor.)
- This specification recommends, but does not require, that references to enterprise beans be organized in the *ejb* subcontext of the application component's environment (that is, in the *java:comp/env/ejb* JNDI context). Note that enterprise bean references declared via annotations will not, by default, be in any subcontext.
- Look up the business interface, no-interface view, or home interface of the referenced enterprise bean in the application component's environment using JNDI.

The following example illustrates how an application component uses a Jakarta Enterprise Bean 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 Jakarta Enterprise Bean reference name to refer to the remote home interface of an enterprise bean.

5.5.1.3. Declaration of Jakarta Enterprise Beans References

Although the Jakarta Enterprise Bean 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 Jakarta Enterprise Beans references using either annotations on the application component's code or the *ejb-ref* or *ejb-local-ref* elements of the deployment descriptor. This allows the consumer of the application component's JAR file (the

Application Assembler or Deployer) to discover all the Jakarta Enterprise Beans references used by the application component. Deployment descriptor entries may also be used to specify injection of a Jakarta Enterprise Bean reference into an application component.

Each *ejb-ref* or *ejb-local-ref* element describes the interface requirements that the referencing application component has for the referenced enterprise bean. The *ejb-ref* element is used for referencing an enterprise bean that is accessed through its remote business interface or remote home and component interfaces. The *ejb-local-ref* element is used for referencing an enterprise bean that is accessed through its local business interface, no-interface view, or local home and component interfaces. The *ejb-ref* element contains a *description* element and the *ejb-ref-name* , *ejb-ref-type* , *home* , and *remote* elements. The *ejb-local-ref* element contains a *description* element and the *ejb-ref-name* , *ejb-ref-type* , *local-home* , and *local* elements

The *ejb-ref-name* element specifies the Jakarta Enterprise Bean reference name. Its value is the environment entry name used in the application component code. The optional *ejb-ref-type* element specifies the expected type of the enterprise bean. Its value must be either *Entity* or *Session* . The *home* and *remote* or *local-home* and *local* elements specify the expected Java programming language types of the referenced enterprise bean's interface(s). If the reference is to a Jakarta Enterprise Beans 2.x remote client view interface, the *home* element is required. Likewise, if the reference is to a Jakarta Enterprise Beans 2.x local client view interface, the *local-home* element is required. The *remote* element of the *ejb-ref* element refers to either the business interface type or the component interface, depending on whether the reference is to a bean's Jakarta Enterprise Beans 3.x or Jakarta Enterprise Beans 2.x remote client view. Likewise, the *local* element of the *ejb-local-ref* element refers to either the business interface type, bean class type, or the component interface type, depending on whether the reference is to a bean's Jakarta Enterprise Beans 3.x local business interface, no-interface view, or Jakarta Enterprise Beans 2.x local client view respectively.

A Jakarta Enterprise Bean reference is scoped to the application component whose declaration contains the *ejb-ref* or *ejb-local-ref* element. This means that the Jakarta Enterprise Bean reference is not accessible from other application components at runtime and that other application components may define *ejb-ref* or *ejb-local-ref* elements with the same *ejb-ref-name* without causing a name conflict.

The lookup-name element specifies the JNDI name of an environment entry that provides a value for the reference.

The following example illustrates the declaration of Jakarta Enterprise Beans 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>
  <lookup-name>java:global/personnel/retirement/PensionPlan</lookup-name>
</ejb-ref>
...
```

5.5.2. Application Assembler's Responsibilities

The Application Assembler can use the *ejb-link* element in the deployment descriptor to link a Jakarta Enterprise Beans reference to a target enterprise bean.

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* or *ejb-local-ref* element of the referencing application component. The value of the *ejb-link* element is the name of the target enterprise bean. This is the name as defined by the metadata annotation (or default) on the bean class or in the *ejb-name* element for the target enterprise bean. The target enterprise bean can be in any ejb-jar file or war file in the same Jakarta EE application as the referencing application component.
- Alternatively, to avoid the need to rename enterprise beans to have unique names within an entire Jakarta EE application, the Application Assembler may use either of the following two syntaxes in the *ejb-link* element of the referencing application component.
- The Application Assembler specifies the module name of the ejb-jar file or war file containing the

referenced enterprise bean and appends the *ejb-name* of the target bean separated by “/”. The module name is the base name of the bundle with no filename extension, unless specified in the deployment descriptor.

- 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. In this manner, multiple beans with the same *ejb-name* may be uniquely identified when the Application Assembler cannot change _ejb-name_s.
- Alternatively to the use of *ejb-link* , the Application Assembler may use the *lookup-name* element to reference the target enterprise bean component by means of one of its JNDI names. It is an error for both *ejb-link* and *lookup-name* to appear inside an *ejb-ref* element.
- The Application Assembler must ensure that the target enterprise bean is type-compatible with the declared Jakarta Enterprise Beans reference. This means that the target enterprise bean must be of the type indicated in the *ejb-ref-type* element, if present, and that the business interface, no-interface view, or home and remote interfaces of the target enterprise bean must be Java type-compatible with the type declared in the Jakarta Enterprise Bean 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 Jakarta EE 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 Jakarta EE 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>
...
```

The following example illustrates using the *ejb-link* element to indicate an enterprise bean reference to the *ShoppingCart* enterprise bean that is in the same Jakarta EE application unit but in a different ejb-jar file. The reference was originally declared in the application component's code using an annotation. The Assembler provides only the link to the bean.

```
...
<ejb-ref>
  <ejb-ref-name>ShoppingService/myCart</ejb-ref-name>
  <ejb-link>../products/product.jar#ShoppingCart</ejb-link>
</ejb-ref>
...
```

The same effect can be obtained by using the *lookup-name* element instead, using an appropriate JNDI name for the target bean.

```
...
<ejb-ref>
  <ejb-ref-name>ShoppingService/myCart</ejb-ref-name>
  <lookup-name>java:app/products/ShoppingCart</lookup-name>
</ejb-ref>
...
```

5.5.3. Deployer's Responsibilities

The Deployer is responsible for the following:

- The Deployer must ensure that all the declared Jakarta Enterprise Beans references are bound to the business interfaces, no-interface views, or home interfaces 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.
- The Deployer must ensure that the target enterprise bean is type-compatible with the types declared for the Jakarta Enterprise Bean reference. This means that the target enterprise bean must be of the type indicated in the *ejb-ref-type* element or specified via the *EJB* annotation, and that the business interface, no-interface view, or home and remote interfaces of the target enterprise bean must be Java type-compatible with the type declared in the Jakarta Enterprise Bean reference (if specified).
- If a Jakarta Enterprise Bean reference declaration includes the *ejb-link* element, the Deployer should bind the enterprise bean reference to the enterprise bean specified as the link's target. If an *EJB* annotation includes the *lookup* element or the Jakarta Enterprise Beans reference declaration includes the *lookup-name* element, the Deployer should bind the enterprise bean reference to the enterprise bean specified as the target of the lookup. It is an error for a Jakarta Enterprise Bean reference declaration to include both an *ejb-link* and a *lookup-name* element.

The following example illustrates the use of the *lookup-name* element to bind an *ejb-ref* to a target enterprise bean in the operational environment. The reference was originally declared in the bean's code using an annotation. The target enterprise bean has *ejb-name ShoppingCart* and is deployed in the stand-alone module *products.jar* .

```
...
<ejb-ref>
  <ejb-ref-name>ShoppingService/myCart</ejb-ref-name>
  <lookup-name>java:global/products/ShoppingCart</lookup-name>
</ejb-ref>
```

5.5.4. Jakarta EE Product Provider's Responsibilities

The Jakarta EE 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 Jakarta EE Product Provider must be able to process the information supplied in class file annotations and in the *ejb-ref* and *ejb-local-ref* elements in the deployment descriptor.

At the minimum, the tools must be able to:

- Preserve the application assembly information in annotations or in the *ejb-link* elements by binding a Jakarta Enterprise Bean reference to the business interface, no-interface view, or home interface of the specified target enterprise bean.
- Inform the Deployer of any unresolved Jakarta Enterprise Beans references, and allow him or her to resolve a Jakarta Enterprise Bean reference by binding it to a specified compatible target enterprise bean.

5.6. Web Service References

A web service reference is similar to an Jakarta Enterprise Bean reference, but is used to reference a web service. Web service references are fully specified in the Web Service specification and the JAX-WS specification.

5.7. 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 (for example, 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.

5.7.1. 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 three sections, the first describing the annotations used to inject resource manager connection factory references, the second describing the API for accessing resource manager connection factory references, and the third describing the syntax for declaring the factory references in a deployment descriptor

5.7.1.1. Injection of Resource Manager Connection Factory References

A field or a method of an application component may be annotated with the *Resource* annotation. The name and type of the factory are as described above. The *authenticationType* and *shareable* elements of the *Resource* annotation may be used to control the type of authentication desired for the resource and the shareability of connection acquired from the factory, as described in the following sections.

The following code example illustrates how an application component uses annotations to declare resource manager connection factory references.

```
// The employee database.
@Resource javax.sql.DataSource employeeAppDB;

public void changePhoneNumber(...) {
  ...
  // 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 = employeeAppDB.getConnection();
  ...
}
```

It is possible to specify as part of the *@Resource* annotation the JNDI name of an entry to which the resource being defined will be bound.

```
// The customer database, looked up in the application environment.
@Resource(lookup="java:app/env/customerDB")
javax.sql.DataSource customerAppDB;
```

The data source object being looked up in the previous example may have been declared as follows.

```
@Resource(name="java:app/env/customerDB",
          type=javax.sql.DataSource.class)
public class AnApplicationClass {
  ...
}
```

From a practical standpoint, declaring a commonly used data source at the application level and referring to it using lookup from multiple components may simplify the task of deploying the application, since now the Deployer will have to perform a single binding operation for the application-level resource, instead of multiple ones. The task can be further simplified by using a data source resource definition, see DataSource Resource Definition". Of course, nothing prevents the Deployer from separately binding each data source reference if necessary.

5.7.1.2. Programming Interfaces for Resource Manager Connection Factory References

The Application Component Provider may 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 Declaration of Resource Manager Connection Factory References in Deployment Descriptor 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 Jakarta Messaging connection factories in the *java:comp/env/jms* subcontext, all Jakarta Mail connection factories in the *java:comp/env/mail* subcontext, and all URL connection factories in the *java:comp/env/url* subcontext. Note that resource manager connection factory references declared via annotations will not, by default, appear in any 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 object 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 can control the shareability of the connections acquired from the resource manager connection factory. By default, connections to a resource manager are shareable across other application components in the application that use the same resource in the same transaction context. The Application Component Provider can specify that connections obtained from a resource manager connection factory reference are not shareable by specifying the value of the *shareable* annotation element to *false* or the *res-sharing-scope* deployment descriptor element to be *Unshareable* . The sharing of connections to a resource manager allows the container to optimize the use of connections and enables the container's use of local transaction optimizations.

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 sign on 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 sign on information as method parameters.

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

We expect that the first form (that is letting the Deployer set up the resource sign on 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();
  ...
}
```

5.7.1.3. 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 using either annotations on the application component's code or in the deployment descriptor using the *resource-ref* elements. This allows the consumer of the application component's JAR file (the Application Assembler or Deployer) to discover all the resource manager connection factory references used by an application component. Deployment descriptor entries may also be used to specify injection of a resource manager connection factory reference into an application component.

Each *resource-ref* element describes a single resource manager connection factory reference. The *resource-ref* element consists of the *description* element, the mandatory *res-ref-name* element, and the optional *res-sharing-scope* , *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 (for example, the name should be *jdbc/EmployeeAppDB* rather than *java:comp/env/jdbc/EmployeeAppDB*). The *res-type* element contains the Java programming language type of the resource manager connection factory that the application component code expects. The *res-type* element is optional if an injection target is specified for this resource; in this case the *res-type* defaults to the type of the injection target. The *res-auth* element indicates whether the application component code performs resource sign on 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 sign on responsibility by setting the value of the *res-auth* element to *Application* or *Container* . If not specified, the default is *Container* . The *res-sharing-scope* element indicates whether connections to the resource manager obtained

through the given resource manager connection factory reference can be shared or whether connections are unshareable. The value of the *res-sharing-scope* element is *Shareable* or *Unshareable* . If the *res-sharing-scope* element is not specified, connections are assumed to be shareable.

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 the resource reference 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>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
...
```

The following example modifies the previous one by linking the resource reference being defined to another one, using a well-known JNDI name for the latter.

```
<resource-ref>
  <res-ref-name>jdbc/EmployeeAppDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
  <lookup-name>java:app/env/TheEmployeeDB</lookup-name>
</resource-ref>
```

5.7.1.4. 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.ConnectionFactory* , the *javax.jms.QueueConnectionFactory* , or the *javax.jms.TopicConnectionFactory* for obtaining Jakarta Messaging connections.

The Application Component Provider must use the *javax.mail.Session* resource manager connection factory type for obtaining Jakarta Mail API 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 Jakarta Messaging connection factories in the *java:comp/env/jms* subcontext, all Jakarta Mail API connection factories in the *java:comp/env/mail* subcontext, and all URL connection factories in the *java:comp/env/url* subcontext. Note that resource manager connection factory references declared via annotations will not, by default, appear in any subcontext.

The Jakarta EE Connector Architecture allows an application component to use the annotation or API described in this section to obtain resource objects that provide access to additional back-end systems.

5.7.2. 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 source code or in the *res-type* element. If the resource manager connection factory references includes a *lookup* annotation element or a *lookup-name* deployment descriptor element, the Deployer may choose whether to honor it and have the corresponding lookup be performed, or override it with a binding of his or her own choosing.
- 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 *Resource* annotation *authenticationType* element is *AuthenticationType.CONTAINER* or the deployment descriptor's *res-auth* element is *Container* , the Deployer is responsible for configuring the sign on 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.

5.7.3. Jakarta EE Product Provider's Responsibilities

The Jakarta EE 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 sets the *authenticationType* element of the *Resource* annotation to *AuthenticationType.APPLICATION* or the *res-auth* of a resource reference to *Application* , the container must allow the application component to perform explicit programmatic sign on using the resource manager's API.
- If the Application Component Provider sets the *shareable* element of the *Resource* annotation to *false* or sets the *res-sharing-scope* of a resource manager connection factory reference to *Unshareable* , the container must not attempt to share the connections obtained from the resource manager connection factory reference.[12]
- The container must provide tools that allow the Deployer to set up resource sign on information for the resource manager references whose *authenticationType* is set to *AuthenticationType.CONTAINER* or whose *res-auth* element is set to *Container* . The minimum requirement is that the Deployer must be able to specify the username/password information for each resource manager connection factory reference declared by the application component, and the container must be able to use the username/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 sign on 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 Jakarta EE products will provide the following features:

- A tool to allow the System Administrator to add, remove, and configure a resource manager for the Jakarta EE 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.

5.7.4. System Administrator's Responsibilities

The System Administrator is typically responsible for the following:

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

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

5.8. 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 a resource (for example, a Connector CCI *InteractionSpec* instance) 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.

5.8.1. Application Component Provider's Responsibilities

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

5.8.1.1. Injection of Resource Environment References

A field or a method of an application component may be annotated with the *Resource* annotation to request injection of a resouce environment reference. The name and type of the resource environment reference are as described earlier. The *authenticationType* and *shareable* elements of the *Resource* annotation must not be specified; resource environment entries are not shareable and do not require authentication. The use of the *Resource* annotation to declare a resource environment reference differs from the use of the *Resource* annotation to declare other environment references only in that the type of a resource environment reference is not one of the Java language types used for other environment references.

5.8.1.2. Resource Environment Reference Programming Interfaces

The Application Component Provider may use resource environment references to locate administered objects that are associated with resources as follows.

- Assign an entry in the application component's environment to the reference. (See subsection Declaration of Resource Environment References in Deployment Descriptor 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. Note that resource environment references declared via annotations will not, by default, appear in any subcontext.

- Look up the administered object in the application component's environment using JNDI.

5.8.1.3. 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 either annotations on the application component's code or 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. Deployment descriptor entries may also be used to specify injection of a resource environment reference into an 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 optional *description* and *resource-env-ref-type* elements and the mandatory *resource-env-ref-name* element. The *resource-env-ref-type* element is optional if an injection target is specified for this resource; in this case the *resource-env-ref-type* defaults to the type of the injection target.

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 resource environment reference is relative to the *java:comp/env* context. The *resource-env-ref-type* element specifies the expected type of the referenced object.

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.

A resource environment reference may specify a *lookup-name* to link the reference being defined to another one via a JNDI name.

5.8.2. 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 may override the linkage preferences of a resource environment reference that includes a *lookup* annotation element or *lookup-name* deployment descriptor element.
- 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* annotation or the *resource-env-ref-type* element.

5.8.3. Jakarta EE Product Provider's Responsibilities

The Jakarta EE 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 Jakarta EE Product Provider must be able to process the information supplied in the class file annotations and 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.

5.9. Message Destination References

This section describes the programming and deployment descriptor interfaces that allow the Application Component Provider to refer to message destination objects by using "logical" names called message destination references. Message destination references are special entries in the application component's environment. The Deployer binds the message destination references to administered message destinations in the target operational environment.

The requirements in this section only apply to Jakarta EE products that include support for Jakarta Messaging.

5.9.1. Application Component Provider's Responsibilities

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

5.9.1.1. Injection of Message Destination References

A field or a method of an application component may be annotated with the *Resource* annotation to request injection of a message destination reference. The name and type of the resource environment reference are as described earlier. The *authenticationType* and *shareable* elements of the *Resource* annotation must not be specified; message destination references are not shareable and do not require authentication.

Note that when using the *Resource* annotation to declare a message destination reference it is not possible to link the reference to other references to the same message destination or to specify whether the message destination is used to produce or consume messages. The deployment descriptor entries described later do provide a way to associate many message destination references with a single message destination and to specify whether each message destination reference is used to produce, consume, or both produce and consume messages, so that the entire message flow of an application may be specified. The Application Assembler may use these message destination links to link together message destination references that have been declared using the *Resource* annotation. A message destination reference declared via the *Resource* annotation is assumed to be used to both produce and consume messages; this default may be overridden using a deployment descriptor entry.

The following example illustrates how an application component uses the Resource annotation to request injection of a message destination reference.

```
@Resource javax.jms.Queue stockQueue;
```

The following example illustrates how a message destination reference can be linked to another one by specifying its JNDI name, perhaps in a different namespace, as a value for the lookup element.

```
@Resource(lookup="java:app/env/TheOrderQueue")
javax.jms.Queue orderQueue;
```

5.9.1.2. Message Destination Reference Programming Interfaces

The Application Component Provider may use message destination references to locate message destinations, as follows.

- Assign an entry in the application component's environment to the reference. (See subsection Declaration of Message Destination References in Deployment Descriptor for information on how message destination references are declared in the deployment descriptor.)
- This specification recommends, but does not require, that all message destination references be organized in the appropriate subcontext of the component's environment for the resource type (for example, in the *java:comp/env/jms* JNDI context for Jakarta Messaging Destinations). Note that message destination references declared via annotations will not, by default, appear in any subcontext.
- Look up the administered object in the application component's environment using JNDI.

The following example illustrates how an application component uses a message destination reference to locate a Jakarta Messaging Destination.

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

// Look up the Jakarta Messaging 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 message destination reference name to refer to a Jakarta Messaging queue.

5.9.1.3. Declaration of Message Destination References in Deployment Descriptor

Although the message destination 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 should declare all references to message destinations using either the *Resource* annotation in the application component's code or the *message-destination-ref* elements of the deployment descriptor. This allows the application component's JAR file consumer to discover all the message destination references used by the application component. Deployment descriptor entries may also be used to specify injection of a message destination reference into an application component.

Each *message-destination-ref* element describes the requirements that the referencing application component has for the referenced destination. The *message-destination-ref* element contains optional *description* , *message-destination-type* , and *message-destination-usage* elements and the mandatory *message-destination-ref-name* element.

The *message-destination-ref-name* element specifies the message destination reference name. Its value is the environment entry name used in the application component code. By default, the name of the message destination reference is relative to the *java:comp/env* context (for example, the name should be *jms/StockQueue* rather than *java:comp/env/jms/StockQueue*). The *message-destination-type* element specifies the expected type of the referenced destination. For example, in the case of a Jakarta Messaging Destination, its value might be *javax.jms.Queue* . The *message-destination-type* element is optional if an injection target is specified for this message destination reference; in this case the *message-destination-type* defaults to the type of the injection target. The *message-destination-usage* element specifies whether messages are consumed from the message destination, produced for the destination, or both. If not specified, messages are assumed to be both consumed and produced.

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

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

```
...
<message-destination-ref>
  <description>
     This is a reference to a Jakarta Messaging queue used in the
     processing of Stock info
  </description>
  <message-destination-ref-name>
     jms/StockInfo
  </message-destination-ref-name>
  <message-destination-type>
     javax.jms.Queue
  </message-destination-type>
  <message-destination-usage>
     Produces
  </message-destination-usage>
</message-destination-ref>
...
```

5.9.2. Application Assembler's Responsibilities

By means of linking message consumers and producers to one or more common logical destinations specified in the enterprise bean deployment descriptor, the Application Assembler can specify the flow of messages within an application. The Application Assembler uses the *message-destination* element, the *message-destination-link* element of the *message-destination-ref* element, and the *message-destination-link* element of an ejb-jar's *message-driven* element to link message destination references to a common logical destination.

The Application Assembler specifies the link between message consumers and producers as follows:

- The Application Assembler uses the *message-destination* element to specify a logical message destination within the application. The *message-destination* element defines a *message-destination-name* , which is used for the purpose of linking.
- The Application Assembler uses the *message-destination-link* element of the *message-destination-ref* element of an application component that produces messages to link it to the target destination. The value of the *message-destination-link* element is the name of the target destination, as defined in the *message-destination-name* element of the *message-destination* element. The *message-destination* element can be in any module in the same Jakarta EE application as the referencing component. The Application Assembler uses the *message-destination-usage* element of the *message-destination-ref* element to indicate that the referencing application component produces messages to the referenced destination.
- If the consumer of messages from the common destination is a message-driven bean, the Application Assembler uses the *message-destination-link* element of the *message-driven* element to reference the logical destination. If the Application Assembler links a message-driven bean to its source destination, he or she should use the *message-destination-type* element of the *message-*

driven element to specify the expected destination type. Otherwise, the Application Assembler uses the *message-destination-link* element of the *message-destination-ref* element of the application component that consumes messages to link to the common destination. In the latter case, the Application Assembler uses the *message-destination-usage* element of the *message-destination-ref* element to indicate that the application component consumes messages from the referenced destination.

- To avoid the need to rename message destinations to have unique names within an entire Jakarta EE application, the Application Assembler may use the following syntax in the *message-destination-link* element of the referencing application component. The Application Assembler specifies the path name of the JAR file containing the referenced message destination and appends the *message-destination-name* of the target destination separated from the path name by #. The path name is relative to the referencing application component JAR file. In this manner, multiple destinations with the same *message-destination-name* may be uniquely identified.
- When linking message destinations, the Application Assembler must ensure that the consumers and producers for the destination require a message destination of the same or compatible type, as determined by the messaging system.

5.9.3. Deployer's Responsibilities

The Deployer is responsible for the following:

- The Deployer must ensure that all the declared message destination 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 may override the linkage preferences of a message destination reference that includes a *lookup-name* element.
- The Deployer must ensure that the target object is type-compatible with the type declared for the message destination reference. This means that the target object must be of the type indicated in the *message-destination-type* element.
- The Deployer must observe the message destination links specified by the Application Assembler.

5.9.4. Jakarta EE Product Provider's Responsibilities

The Jakarta EE 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 Jakarta EE Product Provider must be able to process the information supplied in the *message-destination-ref* elements in the deployment descriptor.

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

5.10. UserTransaction References

Certain Jakarta EE application component types are allowed to use the Jakarta Transactions *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* or by requesting injection of a *UserTransaction* object using the *Resource* annotation. The *authenticationType* and *shareable* elements of the *Resource* annotation must not be specified. The container is only required to provide the *java:comp/UserTransaction* name, or inject a *UserTransaction* object, for those components that can validly make use of it. Any such reference to a *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 via injection.

```
@Resource UserTransaction tx;
public void updateData(...) {
  ...
  // Start a transaction.
  tx.begin();
  ...
  // Perform transactional operations on data.
  ...
  // Commit the transaction.
  tx.commit();
  ...
}
```

The following example illustrates how an application component acquires and uses a *UserTransaction* object using a JNDI lookup.

```
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();
  ...
}
```

A *UserTransaction* object reference may also be declared in a deployment descriptor in the same way as a resource environment reference. Such a deployment descriptor entry may be used to specify injection of a *UserTransaction* object.

The requirements in this section only apply to Jakarta EE products that include support for Jakarta Transactions.

5.10.1. Application Component Provider's Responsibilities

The Application Component Provider is responsible for requesting injection of a *UserTransaction* object using a *Resource* annotation, or using the defined name to look up the *UserTransaction* object.

Only some application component types are required to be able to access a *UserTransaction* object; see *Jakarta EE Technologies* in this specification and the Jakarta Enterprise Beans specification for details.

5.10.2. Jakarta EE Product Provider's Responsibilities

The Jakarta EE Product Provider is responsible for providing an appropriate *UserTransaction* object as required by this specification.

5.11. TransactionSynchronizationRegistry References

The Jakarta Transactions *TransactionSynchronizationRegistry* interface may be used by system level components such as persistence managers that may be packaged with enterprise bean or web application components. Such components can find an appropriate object implementing the *TransactionSynchronizationRegistry* interface by looking up the JNDI name

java:comp/TransactionSynchronizationRegistry or by requesting injection of a *TransactionSynchronizationRegistry* object using the *Resource* annotation. The *authenticationType* and *shareable* elements of the *Resource* annotation must not be specified. The container is only required to provide the *java:comp/TransactionSynchronizationRegistry* name, or inject a *TransactionSynchronizationRegistry* object, for those components that can validly make use of it. Any such reference to a *TransactionSynchronizationRegistry* object is only valid within the component instance that performed the lookup. See the individual component definitions for further information.

A *TransactionSynchronizationRegistry* object reference may also be declared in a deployment descriptor in the same way as a resource environment reference. Such a deployment descriptor entry may be used to specify injection of a *TransactionSynchronizationRegistry* object.

The requirements in this section only apply to Jakarta EE products that include support for Jakarta Transactions.

5.11.1. Application Component Provider's Responsibilities

The Application Component Provider is responsible for requesting injection of a *TransactionSynchronizationRegistry* object using a *Resource* annotation, or using the defined name to look up the *TransactionSynchronizationRegistry* object.

Only some application component types are required to be able to access a *TransactionSynchronizationRegistry* object; see *Jakarta EE Technologies* in this specification for details.

5.11.2. Jakarta EE Product Provider's Responsibilities

The Jakarta EE Product Provider is responsible for providing an appropriate *TransactionSynchronizationRegistry* object as required by this specification.

5.12. ORB References

Some Jakarta EE applications will need to make use of the CORBA ORB to perform certain operations. Such applications can find an appropriate object implementing the *ORB* interface by looking up the JNDI name *java:comp/ORB* or by requesting injection of an *ORB* object. The container is required to provide the *java:comp/ORB* name for all components except applets. Any such reference to a *ORB* object is only valid within the component instance that performed the lookup.

The following example illustrates how an application component acquires and uses an *ORB* object via injection.

```
@Resource ORB orb;

public void method(...) {
  ...
  // Get the POA to use when creating object references.
  POA rootPOA = (POA)orb.resolve_initial_references("RootPOA");
  ...
}
```

The following example illustrates how an application component acquires and uses an *ORB* object using a JNDI lookup.

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

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

  // Get the POA to use when creating object references.
  POA rootPOA = (POA)orb.resolve_initial_references("RootPOA");
  ...
}
```

An *ORB* object reference may also be declared in a deployment descriptor in the same way as a resource manager connection factory reference. Such a deployment descriptor entry may be used to specify injection of an *ORB* object.

The *ORB* instance available under the JNDI name *java:comp/ORB* may always be a shared instance. By default, the *ORB* instance injected into a component or declared via a deployment descriptor entry may also be a shared instance. However, the application may set the *shareable* element of the *Resource* annotation to *false* , or may set the *res-sharing-scope* element in the deployment descriptor to *Unshareable* , to request a non-shared *ORB* instance.

The requirements in this section only apply to Jakarta EE products that include support for interoperability using CORBA.

5.12.1. Application Component Provider's Responsibilities

The Application Component Provider is responsible for requesting injection of the *ORB* object using the Resource annotation, or using the defined name to look up the *ORB* object. If the *shareable* element of the *Resource* annotation is set to *false* , the ORB object injected will not be the shared instance used by other components in the application but instead will be a private ORB instance used only by this

component.

5.12.2. Jakarta EE Product Provider's Responsibilities

The Jakarta EE Product Provider is responsible for providing an appropriate *ORB* object as required by this specification.

5.13. Persistence Unit References

This section describes the metadata annotations and deployment descriptor elements that allow the application component code to refer to the entity manager factory for a persistence unit using a logical name called a *persistence unit reference* . Persistence unit references are special entries in the application component's environment. The Deployer binds the persistence unit references to entity manager factories that are configured in accordance with the *persistence.xml* specification for the persistence unit, as described in the Java Persistence specification.

The requirements in this section only apply to Jakarta EE products that include support for the Java Persistence API.

5.13.1. Application Component Provider's Responsibilities

This subsection describes the Application Component Provider's view of locating the entity manager factory for a persistence unit and defines his or her responsibilities. The first subsection describes annotations for injecting references to an entity manager factory for a persistence unit; the second describes the API for accessing an entity manager factory using a persistence unit reference; and the third describes syntax for declaring persistence unit references in a deployment descriptor.

5.13.1.1. Injection of Persistence Unit References

A field or a method of an application component may be annotated with the *PersistenceUnit* annotation. The *name* element specifies the name under which the entity manager factory for the referenced persistence unit may be located in the JNDI naming context. The optional *unitName* element specifies the name of the persistence unit as declared in the *persistence.xml* file that defines the persistence unit.

The following code example illustrates how an application component uses annotations to declare persistence unit references.

```
@PersistenceUnit
EntityManagerFactory emf;

@PersistenceUnit(unitName="InventoryManagement")
EntityManagerFactory inventoryEMF;
```

5.13.1.2. Programming Interfaces for Persistence Unit References

The Application Component Provider must use persistence unit references to obtain references to entity manager factories as follows.

- Assign an entry in the application component's environment to the persistence unit reference. (See subsection Declaration of Persistence Unit References in Deployment Descriptor for information on how persistence unit references are declared in the deployment descriptor.) It is recommended that the Application Component Provider organize all persistence unit references in the *java:comp/env/persistence* subcontext of the component's environment.
- Lookup the entity manager factory for the persistence unit in the application component's environment using JNDI.
- Invoke the appropriate method on the entity manager factory to obtain an entity manager instance.

The following code sample illustrates obtaining an entity manager factory.

```
@PersistenceUnit(name="persistence/InventoryAppDB")
@Stateless
public class InventoryManagerBean implements InventoryManager {
  EJBContext ejbContext;
  ...
  public void updateInventory(...) {
    ...
    // obtain the initial JNDI context
    Context initCtx = new InitialContext();

    // perform JNDI lookup to obtain entity manager factory
    EntityManagerFactory = (EntityManagerFactory)
       initCtx.lookup(
          "java:comp/env/persistence/InventoryAppDB");

    // use factory to obtain application-managed entity manager
    EntityManager em = emf.createEntityManager();
    ...
  }
}
```

5.13.1.3. Declaration of Persistence Unit References in Deployment Descriptor

Although a persistence unit 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, if metadata annotations are not used, the Application Component Provider must declare all the persistence unit references in the deployment descriptor using the *persistence-unit-ref* elements.

This allows the Application Assembler or Deployer to discover all the persistence unit references used by an application component. Deployment descriptor entries may also be used to specify injection of a persistence unit reference into an application component.

Each *persistence-unit-ref* element describes a single entity manager factory reference for the persistence unit. The *persistence-unit-ref* element consists of the optional *description* and *persistence-unit-name* elements, and the mandatory *persistence-unit-ref-name* element.

The *persistence-unit-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 *persistence/InventoryAppDB* rather than *java:comp/env/persistence/InventoryAppDB*). The *persistence-unit-name* element is the name of the persistence unit, as specified in the *persistence.xml* file for the persistence unit.

The following example is the declaration of a persistence unit reference used by the *InventoryManager* enterprise bean illustrated in the previous subsection.

```
...
  <persistence-unit-ref>
    <description>
       Persistence unit for the inventory management
       application.
    </description>
    <persistence-unit-ref-name>
       persistence/InventoryAppDB
    </persistence-unit-ref-name>
    <persistence-unit-name>
       InventoryManagement
    </persistence-unit-name>
  </persistence-unit-ref>
...
```

5.13.2. Application Assembler's Responsibilities

The Application Assembler can use the *persistence-unit-name* element in the deployment descriptor to disambiguate a reference to a persistence unit.The Application Assembler (or Application Component Provider) may use the following syntax in the *persistence-unit-name* element of the referencing application component to avoid the need to rename persistence units to have unique names within a Jakarta EE application. The Application Assembler specifies the path name of the root of the *persistence.xml* file for the referenced persistence unit and appends the name of the persistence unit separated from the path name by # . The path name is relative to the referencing application component jar file. In this manner, multiple persistence units with the same persistence unit name may be uniquely identified when the Application Assembler cannot change persistence unit names.

For example,

```
...
  <persistence-unit-ref>
    <description>
       Persistence unit for the inventory management
       application.
    </description>
    <persistence-unit-ref-name>
       persistence/InventoryAppDB
    </persistence-unit-ref-name>
    <persistence-unit-name>
       ../lib/inventory.jar#InventoryManagement
    </persistence-unit-name>
  </persistence-unit-ref>
...
```

The Application Assembler uses the *persistence-unit-name* element to link the persistence unit name *InventoryManagement* declared in the *InventoryManagerBean* to the persistence unit named *InventoryManagement* defined in *inventory.jar* .

The following rules apply to how a deployment descriptor entry may override a *PersistenceUnit* annotation:

- The relevant deployment descriptor entry is located based on the JNDI name used with the annotation (either defaulted or provided explicitly).
- The *persistence-unit-name* overrides the *unitName* element of the annotation. The Application Assembler or Deployer should exercise caution in changing this value, if specified, as doing so is likely to break the application.
- The injection target, if specified, must name exactly the annotated field or property method.

5.13.3. Deployer's Responsibility

The Deployer uses deployment tools to bind a persistence unit reference to the actual entity manager factory configured for the persistence unit in the target operational environment.

The Deployer must perform the following tasks for each persistence unit reference declared in the metadata annotations or deployment descriptor:

- Bind the persistence unit reference to an entity manager factory configured for the persistence unit 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 entity manager factory.
- If the persistence unit name is specified, the Deployer should bind the persistence unit reference to the entity manager factory for the persistence unit specified as the target.
- Provide any additional configuration information that the entity manager factory needs for managing the persistence unit, as described in the Java Persistence specification.

5.13.4. Jakarta EE Product Provider's Responsibility

The Jakarta EE 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 entity manager factory classes for the persistence units that are configured with the container. The implementation of the entity manager factory classes may be provided by the container directly or by the container in conjunction with a third-party persistence provider, as described in the Java Persistence specification.

5.13.5. System Administrator's Responsibility

The System Administrator is typically responsible for the following:

- Add, remove, and configure entity manager factories in the server environment.

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

5.14. Persistence Context References

This section describes the metadata annotations and deployment descriptor elements that allow the application component code to refer to a container-managed entity manager of a specified persistence context type using a logical name called a *persistence context reference* . Persistence context references are special entries in the application component's environment. The Deployer binds the persistence context references to container-managed entity managers for persistence contexts of the specified type and configured in accordance with their persistence unit, as described in the Java Persistence specification.

The requirements in this section only apply to Jakarta EE products that include support for the Java Persistence API.

5.14.1. Application Component Provider's Responsibilities

This subsection describes the Application Component Provider's view of locating container-managed entity managers and defines his or her responsibilities. The first subsection describes annotations for injecting references to container-managed entity managers; the second describes the API for accessing references to container-managed entity managers; and the third describes syntax for declaring these references in a deployment descriptor.

5.14.1.1. Injection of Persistence Context References

A field or a method of an application component may be annotated with the *PersistenceContext* annotation. The *name* element specifies the name under which a container-managed entity manager for the referenced persistence unit may be located in the JNDI naming context. The optional *unitName*

element specifies the name of the persistence unit as declared in the *persistence.xml* file that defines the persistence unit. The optional *type* element specifies whether a transaction-scoped or extended persistence context is to be used. If the type is not specified, a transaction-scoped persistence context will be used. References to container-managed entity managers with extended persistence contexts can only be injected into stateful session beans. The optional *synchronization* element specifies whether the persistence context is always automatically synchronized with the current transaction or whether it must be explicitly joined to the transaction. If the *synchronization* element is not specified, the persistence context will be automatically synchronized. The optional *properties* element specifies configuration properties to be passed to the persistence provider when the entity manager is created.

The following code example illustrates how an application component uses annotations to declare persistence context references.

```
@PersistenceContext(type=EXTENDED)
EntityManager em;
```

Programming Interfaces for Persistence Context References

The Application Component Provider may use a persistence context reference to obtain a reference to a container-managed entity manager configured for a persistence unit as follows:

- Assign an entry in the application component's environment to the persistence context reference. (See subsection Declaration of Persistence Context References in Deployment Descriptor for information on how persistence context references are declared in the deployment descriptor.) It is recommended that the Application Component Provider organize all persistence context references in the *java:comp/env/persistence* subcontext of the component's environment.
- Lookup the container-managed entity manager for the persistence unit in the application component's environment using the JNDI API.

The following code sample illustrates obtaining an entity manager for a persistence context.

```
@PersistenceContext(name="persistence/InventoryAppMgr")
@Stateless
public class InventoryManagerBean implements InventoryManager {

  public void updateInventory(...) {
    ...

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

    // JNDI lookup to obtain container-managed entity manager
    EntityManager = (EntityManager)
       initCtx.lookup(
          "java:comp/env/persistence/InventoryAppMgr");
    ...
  }
}
```

5.14.1.2. Declaration of Persistence Context References in Deployment Descriptor

Although a persistence context 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, if metadata annotations are not used, the Application Component Provider must declare all the persistence context references in the deployment descriptor using the *persistence-context-ref* elements. This allows the Application Assembler or Deployer to discover all the persistence context references used by an application component. Deployment descriptor entries may also be used to specify injection of a persistence context reference into a bean.

Each *persistence-context-ref* element describes a single container-managed entity manager reference. The *persistence-context-ref* element consists of the optional *description* , *persistence-unit-name* , *persistence-context-type* , *persistence-context-synchronization* , and *persistence-property* elements, and the mandatory *persistence-context-ref-name* element.

The *persistence-context-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 *persistence/InventoryAppMgr* rather than *java:comp/env/persistence/InventoryAppMgr*). The *persistence-unit-name* element is the name of the persistence unit, as specified in the *persistence.xml* file for the persistence unit. The *persistence-context-type* element specifies whether a transaction-scoped or extended persistence context is to be used. Its value is either *Transaction* or *Extended* . If the persistence context type is not specified, a transaction-scoped persistence context will be used. The optional *persistence-context-synchronization* element specifies whether the persistence context is automatically synchronized with the current transaction. Its value is either *Synchronized* or *Unsynchronized* . If the persistence context synchronization is not specified, the persistence context will be automatically synchronized. The optional *persistence-property*

elements specify configuration properties that are passed to the persistence provider when the entity manager is created.

The following example is the declaration of a persistence context reference used by the *InventoryManager* enterprise bean illustrated in the previous subsection.

```
...
  <persistence-context-ref>
    <description>
       Persistence context for the inventory management
       application.
    </description>
    <persistence-context-ref-name>
       persistence/InventoryAppDB
    </persistence-context-ref-name>
    <persistence-unit-name>
       InventoryManagement
    </persistence-unit-name>
  </persistence-context-ref>
...
```

5.14.2. Application Assembler's Responsibilities

The Application Assembler can use the *persistence-unit-name* element in the deployment descriptor to specify a reference to a persistence unit using the syntax described in Application Assembler's Responsibilities." In this manner, multiple persistence units with the same persistence unit name may be uniquely identified when the persistence unit names cannot be changed.

For example,

```
...
  <persistence-context-ref>
    <description>
       Persistence context for the inventory management
       application.
    </description>
    <persistence-context-ref-name>
       persistence/InventoryAppDB
    </persistence-context-ref-name>
    <persistence-unit-name>
       ../lib/inventory.jar#InventoryManagement
    </persistence-unit-name>
  </persistence-context-ref>
...
```

The Application Assembler uses the *persistence-unit-name* element to link the persistence unit name *InventoryManagement* declared in the *InventoryManagerBean* to the persistence unit named *InventoryManagement* defined in *inventory.jar* .

The following rules apply to how a deployment descriptor entry may override a *PersistenceContext* annotation:

- The relevant deployment descriptor entry is located based on the JNDI name used with the annotation (either defaulted or provided explicitly).
- The *persistence-unit-name* overrides the *unitName* element of the annotation. The Application Assembler or Deployer should exercise caution in changing this value, if specified, as doing so is likely to break the application.
- The *persistence-context-type* , if specified, overrides the *type* element of the annotation. In general, the Application Assembler or Deployer should never change the value of this element, as doing so is likely to break the application.
- The *persistence-context-synchronization* , if specified, overrides the *synchronization* element of the annotation. In general, the Application Assembler or Deployer should never change the value of this element, as doing so is likely to break the application.
- Any *persistence-property* elements are added to those specified by the *PersistenceContext* annotation. If the name of a specified property is the same as one specified by the *PersistenceContext* annotation, the value specified in the annotation is overridden.
- The injection target, if specified, must name exactly the annotated field or property method.

5.14.3. Deployer's Responsibility

The Deployer uses deployment tools to bind a persistence context reference to the container-managed entity manager for the persistence context of the specified type and configured for the persistence unit in the target operational environment.

The Deployer must perform the following tasks for each persistence context reference declared in the metadata annotations or deployment descriptor:

- Bind the persistence context reference to a container-managed entity manager for a persistence context of the specified type and configured for the persistence unit as specified in the *persistence.xml* file for the persistence unit 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 entity manager.
- If the persistence unit name is specified, the Deployer should bind the persistence context reference to an entity manager for the persistence unit specified as the target.
- Provide any additional configuration information that the entity manager factory needs for creating such an entity manager and for managing the persistence unit, as described in the Java Persistence specification.

5.14.4. Jakarta EE Product Provider's Responsibility

The Jakarta EE 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 entity manager classes for the persistence units that are configured with the container. This implementation may be provided by the container directory or by the container in conjunction with a third-party persistence provider, as described in the Java Persistence specification.

5.14.5. System Administrator's Responsibility

The System Administrator is typically responsible for the following:

- Add, remove, and configure entity manager factories in the server environment.

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

5.15. Application Name and Module Name References

A component may access the name of the current application using the pre-defined JNDI name *java:app/AppName* . A component may access the name of the current module using the pre-defined JNDI name *java:module/ModuleName* . Both of these names are represented by *String* objects.

5.15.1. Application Component Provider's Responsibilities

The Application Component Provider is responsible for requesting injection of the application name or module name using a *Resource* annotation on a *String* method or field, or using the defined name to look up the application name or module name.

5.15.2. Jakarta EE Product Provider's Responsibilities

The Jakarta EE Product Provider is responsible for providing the correct application name and module name *String* objects as required by this specification.

5.16. Application Client Container Property

An application may determine whether it is executing in a Jakarta EE application client container by using the pre-defined JNDI name *java:comp/InAppClientContainer* . This property is represented by a *Boolean* object. If the application is running in a Jakarta EE application client container, the value of this property is true. If the application is running in a Jakarta EE web or enterprise bean container, the value of this property is false.

5.16.1. Application Component Provider's Responsibilities

The Application Component Provider is responsible for requesting injection of the application client container property using a *Resource* annotation on a *Boolean* or *boolean* method or field, or using the defined name to look up the application client container property.

5.16.2. Jakarta EE Product Provider's Responsibilities

The Jakarta EE Product Provider is responsible for providing the correct application client container property as required by this specification.

5.17. Validator and Validator Factory References

This section describes the metadata annotations and deployment descriptor entries that allow an application to obtain instances of the Bean Validation *Validator* and *ValidatorFactory* types.

Applications that need to use those interfaces can find appropriate objects by looking up the name *java:comp/Validator* for *Validator* and *java:comp/ValidatorFactory* for *ValidatorFactory* , or by requesting the injection of an object of the appropriate type via the *Resource* annotation. The *authenticationType* and *shareable* elements of the *Resource* annotation must not be specified.

```
@Resource ValidatorFactory validatorFactory;

@Resource Validator validator;
```

For Validator objects, the default validation context is used. This means that all such Validators will be equivalent to those obtained by first acquiring a ValidatorFactory and then invoking the getValidator method on it with no arguments.

In other words, the following two code snippets are equivalent:

```
// obtaining a Validator directly
Context initCtx = new InitialContext();
Validator validator = (Validator)initCtx.lookup(
                            "java:comp/Validator");

// obtaining a Validator from a ValidatorFactory
Context initCtx = new InitialContext();
Validator validator =
       ((ValidatorFactory) initCtx.lookup(
                             "java:comp/ValidatorFactory"))
       .getValidator();
```

A *Validator* or *ValidatorFactory* object reference may also be declared in a deployment descriptor in

the same way as a resource environment reference.

In order to customize the returned *ValidatorFactory* , a Jakarta Enterprise Beans, web or application client module may specify a Bean Validation XML deployment descriptor, as described in the Bean Validation specification.

A validation deployment descriptor only affects *ValidatorFactory* instances in that module.

There is no per-application validation deployment descriptor.

5.17.1. Application Component Provider's Responsibilities

The Application Component Provider is responsible for requesting injection of a *Validator* or of a *ValidatorFactory* using a *Resource* annotation, or using the defined names to look up a *Validator* or *ValidatorFactory* instance.

The Application Component Provider may customize the *ValidatorFactory* and (indirectly) *Validator* instances by including a Bean Validation deployment descriptor inside a specific module of the application.

5.17.2. Jakarta EE Product Provider's Responsibilities

The Jakarta EE Product Provider must make a default *ValidatorFactory* available at *java:comp/ValidatorFactory* . The default *ValidatorFactory* available at *java:comp/ValidatorFactory* must support use of CDI if CDI is enabled for the module. In particular, all of the classes specified by the *javax.validation.BootstrapConfiguration* interface must be created as non-contextual objects using CDI, as described in Support for Dependency Injection". These objects must be used to configure the default *ValidatorFactory* available at *java:comp/ValidatorFactory* in accordance with the bootstrapping APIs described by the Bean Validation specification.

The default *ValidatorFactory* is a single instance per module; each lookup of *java:comp/ValidatorFactory* returns the same instance.

The default *Validator* is created by the default *ValidatoryFactory* using the *getValidator* method. Each lookup of *java:comp/Validator* returns a new *Validator* instance.

5.18. Resource Definition and Configuration

In addition to referencing resources as defined in this chapter, an application may specify the definition and configuration of resources that it requires in its operational environment.

Each application has a set of "physical" resources and services that it depends on (database storage, queueing, mail, etc.) and which need to be made available to it when it is deployed. Such resources may be scoped to the application instance or may be shareable. An application may define a dependency upon such resources in its environment by means of resource definition metadata.

The specification of resource definition metadata provides information that can be used at the application's deployment to provision and configure the required resource. Further, resource definitions allow an application to be deployed into a Jakarta EE environment with more minimal administrative configuration.

Resources may be defined in any of the JNDI namespaces described in Application Component Environment Namespaces". For example, a resource may be defined:

in the *java:comp* namespace, for use by a single component;

- in the *java:module* namespace, for use by all components in a module;
- in the *java:app* namespace, for use by all components in an application;
- in the *java:global* namespace, for use by all applications.

The following annotations (and corresponding XML deployment descriptor elements) define resources: *DataSourceDefinition* , *JMSConnectionFactoryDefinition* , *JMSDestinationDefinition* , *MailSessionDefinition* , *ConnectionFactoryDefinition* , and *AdministeredObjectDefinition* .

Once defined, a resource may be referenced by a component using the *lookup* element of the *Resource* annotation or the *lookup-name* element of the *resource-ref* deployment descriptor element in order to bind the logical reference to the resource as referenced in the application code to the resource defined in the environment. __

The specificity of the resource definition elements as provided by the Application Component Provider may vary according to the needs of the application. For example:

- An application may require an instance of a resource, but its needs may be general in that while it requires a resource with certain properties, it does not require a particular instance of the resource. It may expect the resource to be provisioned and configured for it by the Deployer or System Administrator.
- An application may require a particular instance of a resource (with specific configuration properties) that already exists. For example, the resource may previously have been created and configured by the Deployer or System Administrator.

The values specified for required annotation elements (and corresponding XML deployment descriptor elements) must be observed when the application is deployed. Changing a value that has been specified for some optional elements (e.g., *transactional*) may cause the application to work incorrectly. Changing a value that has been specified for an optional element related to quality of service (e.g., pool size, idle time, etc.) may affect the performance of the application.

The following default values used in the *DataSourceDefinition* , *JMSConnectionFactoryDefinition* , *JMSDestinationDefinition* , *MailSessionDefinition* , and *ConnectionFactoryDefinition* annotations indicate that an element value is optional and has not been set:

- integer-valued elements: *-1*

- string-valued elements: “”
- array-valued elements: *{}*

5.18.1. Guidelines

The following guidelines should be observed with regard to the specification of values for resource definition elements.

- In general, the Application Component Provider or Assembler should specify values for elements which, if changed, would cause the application to break—for example, JNDI name, isolation level. If multiple resource definitions are specified for a given resource, they must be consistent.
- The Jakarta EE Product Provider may choose suitable server-specific default values for optional elements for which values have not been specified.

5.18.2. Requirements Common to All Resource Definition Types

The following requirements apply to the resource definitions described in Sections DataSource Resource Definition through Connector Administered Object Definition.

When an Application Component Provider or Application Assembler specifies connectivity information to a “physical” resource through a resource definition annotation or deployment descriptor element, it is assumed that the physical resource exists. The automatic provisioning of resources may be supported by an implementation of this specification, but support for this functionality is not required. If automatic provisioning of resources is not supported, it is the Deployer’s responsibility (possibly in conjunction with the System Administrator) to insure that the physical resource is provisioned for use by the application.

5.18.2.1. JNDI Name

The Deployer and Jakarta EE Product Provider must not alter the specified JNDI name. The requested resource must be made available in JNDI under the specified name.

5.18.2.2. Resource Address

If the Application Component Provider or Application Assembler has specified an address for a resource (server name, port, etc.), a resource at the specified location should already exist. If it does not, and if the automatic provisioning of resources is not supported, it is the Deployer’s responsibility (possibly in conjunction with the System Administrator) to insure that the resource is provisioned for use by the application.[13]

If the resource has not been otherwise provisioned and if automatic provisioning of resources is supported, the Jakarta EE Product Provider is responsible for provisioning the resource. If the requested resource cannot be made available or created, the application must fail to deploy.

5.18.2.3. Quality of Service Elements

Quality of service elements may be altered by the Deployer. The Jakarta EE Product Provider is permitted to impose restrictions upon quality of service elements in accordance with its implementation limits and quality of service guarantees. If quality of service values that have been specified do not meet these restrictions, the Product Provider must not reject the deployment (but must instead use appropriate values).

5.18.2.4. Properties

All resource definition annotations and XML elements support the use of property elements (elements named " *properties* " or " *property* "). A Jakarta EE Product Provider is permitted to reject a deployment if a property that it recognizes has a value that it does not support. A Jakarta EE Product Provider must not reject a deployment on the basis of a property that it does not recognize.

5.18.3. DataSource Resource Definition

An application may define a *DataSource* resource. A *DataSource* resource is used to access a database using a JDBC driver.

The *DataSource* resource may be defined in any of the JNDI namespaces described in Application Component Environment Namespaces".

A *DataSource* resource may be defined in a web module, enterprise bean module, application client module, or application deployment descriptor using the *data-source* element.

For example:

```
<data-source>
  <description>Sample DataSource definition</description>
  <name>java:app/MyDataSource</name>
  <class-name>com.example.MyDataSource</class-name>
  <server-name>myserver.com</server-name>
  <port-number>6689</port-number>
  <database-name>myDatabase</database-name>
  <user>lance</user>
  <password>secret</password>
  <property>
    <name>Property1</name>
    <value>10</value>
  </property>
  <property>
    <name>Property2</name>
    <value>20</value>
  </property>
  <login-timeout>0</login-timeout>
  <transactional>false</transactional>
  <isolation-level>TRANSACTION_READ_COMMITTED</isolation-level>
  <initial-pool-size>0</initial-pool-size>
  <max-pool-size>30</max-pool-size>
  <min-pool-size>20</min-pool-size>
  <max-idle-time>0</max-idle-time>
  <max-statements>50</max-statements>
</data-source>
```

A *DataSource* resource may also be defined using the *DataSourceDefinition* annotation on a container-managed class, such as a servlet or enterprise bean class.

For example:

```
  @DataSourceDefinition(
     name="java:app/MyDataSource",
     className="com.example.MyDataSource",
     portNumber=6689,
     serverName="myserver.com",
     user="lance",
     password="secret")
```

(Of course, we do not recommend including passwords to production systems in the code, but it's often useful while testing. Passwords, or other parts of the *DataSource* definition, can be overridden by a deployment descriptor when the application is deployed.)

Once defined, a *DataSource* resource may be referenced by a component using the *resource-ref*

deployment descriptor element or the *Resource* annotation. For example, the above *DataSource* could be referenced as follows:

```
@Stateless
public class MySessionBean {
  @Resource(lookup = "java:app/MyDataSource")
  DataSource myDB;
  ...
}
```

The following *DataSourceDefinition* annotation elements (and corresponding XML deployment descriptor elements) are considered to specify an address for a *DataSource* resource: *serverName* , *portNumber* , *databaseName* , *url* .

The following *DataSourceDefinition* annotation elements (and corresponding XML deployment descriptor elements) are considered to be quality of service elements: *loginTimeout* , *initialPoolSize* , *maxPoolSize* , *minPoolSize* , *maxIdleTime* , *maxStatements* .

5.18.3.1. Application Component Provider's Responsibilities

The Application Component Provider is responsible for the definition of a *DataSource* resource using a *DataSourceDefinition* annotation or the *data-source* deployment descriptor element.

If the database has been previously provisioned for the application (e.g., by administrative action), it is the responsibility of the Application Component Provider to specify the class name of the data source implementation class and the server and port at which the database is to be accessed.

A URL should not be specified in conjunction with address elements such as server name and port. If it is, the precedence order is undefined and implementation specific.

5.18.3.2. Deployer's Responsibilities

Requirements common to all resource definition types are described in Requirements Common to All Resource Definition Types". The following additional requirements apply:

If specified, user name and password should be used as specified.

- The transactional specification and isolation level must be used as specified.

5.18.3.3. Jakarta EE Product Provider's Responsibilities

Requirements common to all resource definition types are described in Requirements Common to All Resource Definition Types". The following additional requirements apply:

- If a class name is specified, a resource with the specified implementation class (or a subclass) must be provided. If the class name is specified as *XADataSource* , an XA datasource must be provided.

- If an isolation level is specified, the Product Provider must satisfy the request or provide a higher level of isolation. If the request cannot be satisfied, the Product Provider must reject the deployment.

5.18.4. Jakarta Messaging Connection Factory Resource Definition

An application may define a Jakarta Messaging *ConnectionFactory* resource.

The Jakarta Messaging *ConnectionFactory* resource may be defined in any of the JNDI namespaces described in Application Component Environment Namespaces".

A Jakarta Messaging *ConnectionFactory* resource may be defined in a web module, enterprise bean module, application client module, or application deployment descriptor using the jms-connection-factory element.

For example:

```
<jms-connection-factory>
  <description>
     Sample Jakarta Messaging ConnectionFactory definition
  </description>
  <name>java:app/MyJMSCF</name>
  <interface-name>
     javax.jms.QueueConnectionFactory
  </interface-name>
  <resource-adapter>myJMSRA</resource-adapter>
  <user>scott</user>
  <password>secret</password>
  <client-id>MyId</client-id>
  <property>
    <name>Property1</name>
    <value>10</value>
   </property>
   <property>
     <name>Property2</name>
     <value>20</value>
   </property>
   <transactional>false</transactional>
   <max-pool-size>30</max-pool-size>
   <min-pool-size>20</min-pool-size>
</jms-connection-factory>
```

A Jakarta Messaging *ConnectionFactory* resource may also be defined using the *JMSConnectionFactoryDefinition* annotation on a container-managed class, such as a servlet or enterprise bean class.

For example:

```
@JMSConnectionFactoryDefinition(
   name="java:app/MyJMSCF",
   interfaceName="javax.jms.QueueConnectionFactory",
   resourceAdapter="myJMSRA")
```

(As with the *DataSource* definition, we do not recommend including passwords to production systems in the code, but it's often useful while testing. Passwords, or other parts of the *JMSConnectionFactoryDefinition* annotation, can be overridden by a deployment descriptor when the application is deployed.)

Once defined, a Jakarta Messaging *ConnectionFactory* resource may be referenced by a component using the *resource-ref* deployment descriptor element or the *Resource* annotation. For example, the above Jakarta Messaging *ConnectionFactory* could be referenced as follows:

```
@Stateless_
public class MySessionBean {
  @Resource(lookup = "java:app/MyJMSCF")
  ConnectionFactory myCF;
  ...
}
```

The following *JMSConnectionFactoryDefinition* annotation elements (and corresponding XML deployment descriptor elements) are considered to specify an address for a Jakarta Messaging *ConnectionFactory* resource: *resourceAdapter* .

The following *JMSConnectionFactoryDefinition* annotation elements (and corresponding XML deployment descriptor elements) are considered to be quality of service elements: *maxPoolSize* , *minPoolSize* .

5.18.4.1. Application Component Provider's Responsibilities

The Application Component Provider is responsible for the definition of a Jakarta Messaging *ConnectionFactory* using a *JMSConnectionFactoryDefinition* annotation or the *jms-connection-factory* deployment descriptor element.

5.18.4.2. Deployer's Responsibilities

Requirements common to all resource definition types are described in Requirements Common to All Resource Definition Types". The following additional requirements apply:

A resource of the specified interface type (or of the default interface type, if not specified) must be provided.

- If specified, user name and password should be used as specified.
- The transactional specification must be used as specified.
- If specified, the client id should be used as specified.

5.18.4.3. Jakarta EE Product Provider's Responsibilities

Requirements common to all resource definition types are described in Requirements Common to All Resource Definition Types".

5.18.5. Jakarta Messaging Destination Definition

An application may define a Jakarta Messaging *Destination* resource. A Jakarta Messaging *Destination* resource is a Jakarta Messaging Queue or Topic.

The Jakarta Messaging *Destination* resource may be defined in any of the JNDI namespaces described in Application Component Environment Namespaces".

A Jakarta Messaging *Destination* resource may be defined in a web module, enterprise bean module, application client module, or application deployment descriptor using the *jms-destination* element.

For example:

```
<jms-destination>
  <description>Sample Jakarta Messaging Destination definition</description>
  <name>java:app/MyJMSDestination</name>
  <interface-name>javax.jms.Queue</interface-name>
  <resource-adapter>myJMSRA</resource-adapter>
  <destination-name>myQueue1</destination-name>
  <property>
    <name>Property1</name>
    <value>10</value>
  </property>
  <property>
    <name>Property2</name>
    <value>20</value>
  </property>
</jms-destination>
```

A Jakarta Messaging *Destination* resource may also be defined using the *JMSDestinationDefinition* annotation on a container-managed class, such as a servlet or enterprise bean class.

For example:

```
@JMSDestinationDefinition(
   name="java:app/MyJMSQueue",
   interfaceName="javax.jms.Queue",
   destinationName="myQueue1")
```

The *JMSDestinationDefinition* annotation can be overridden by a deployment descriptor when the application is deployed.

Once defined, a Jakarta Messaging *Destination* resource may be referenced by a component using either the *resource-env-ref* or *message-destination-ref* deployment descriptor element or the *Resource* annotation. For example, the above *Destination* could be referenced as follows:

```
@Stateless
public class MySessionBean {
  @Resource(lookup = "java:app/MyJMSQueue")
  Queue myQueue;
  ...
}
```

The following *JMSDestinationDefinition* annotation elements (and corresponding XML deployment descriptor elements) are considered to specify an address for a Jakarta Messaging *Destination* resource: *resourceAdapter* , *destinationName* .

5.18.5.1. Application Component Provider's Responsibilities

The Application Component Provider is responsible for the definition of a Jakarta Messaging *Destination* using a *JMSDestinationDefinition* annotation or the *jms-destination* deployment descriptor element.

5.18.5.2. Deployer's Responsibilities

Requirements common to all resource definition types are described in Requirements Common to All Resource Definition Types". The following additional requirements apply:

A resource of the specified interface type must be provided.

5.18.5.3. Jakarta EE Product Provider's Responsibilities

Requirements common to all resource definition types are described in Requirements Common to All Resource Definition Types".

5.18.6. Mail Session Definition

An application may define a Mail *Session* resource.

The Mail *Session* resource may be defined in any of the JNDI namespaces described in Application Component Environment Namespaces".

A Mail *Session* resource may be defined in a web module, enterprise bean module, application client module, or application deployment descriptor using the *mail-session* element.

For example:

```
<mail-session>
  <description>Sample Mail Session definition</description>
  <name>java:app/mail/MySession</name>
  <store-protocol>imap</store-protocol>
  <transport-protocol>smtp</transport-protocol>
  <host>somewhere.myco.com</host>
  <user>linda</user>
  <password>secret</password>
  <from>some.body@myco.com</from>
  <property>
    <name>mail.smtp.starttls.enable</name>
    <value>true</value>
  </property>
  <property>
    <name>mail.imap.connectiontimeout</name>
    <value>500</value>
  </property>
</mail-session>
```

A Mail *Session* resource may also be defined using the *MailSessionDefinition* annotation on a container-managed class, such as a servlet or enterprise bean class.

For example:

```
  @MailSessionDefinition(_
       name="java:app/mail/MySession",
       host="somewhere.myco.com",
       from="some.body@myco.com")
```

The *MailSessionDefinition* annotation can be overridden by a deployment descriptor when the application is deployed.

Once defined, a Mail *Session* resource may be referenced by a component using the *resource-ref* deployment descriptor element or the *Resource* annotation. For example, the above *Destination* could be referenced as follows:

```
@Stateless
public class MySessionBean {
  @Resource(lookup = "java:app/mail/MySession")
  Session myMailSession;
  ...
}
```

The following *MailSessionDefinition* annotation elements (and corresponding XML deployment descriptor elements) are considered to specify an address for a Mail *Session* resource: *host* .

5.18.6.1. Application Component Provider's Responsibilities

The Application Component Provider is responsible for the definition of a Mail Session using a *MailSessionDefinition* annotation or the *mail-session* deployment descriptor element.

If a mail server resource has been previously provisioned for the application (e.g., by administrative action), it is the responsibility of the Application Component Provider to specify the mail server host name.

5.18.6.2. Deployer's Responsibilities

Requirements common to all resource definition types are described in Requirements Common to All Resource Definition Types". The following additional requirements apply:

- If store protocol, store protocol class, transport protocol, or transport protocol class has been specified, a resource with the specified property or properties should be provided.
- If specified, the user name and password should be used as specified.
- If specified, the from address should be used as specified.

5.18.6.3. Jakarta EE Product Provider's Responsibilities

Requirements common to all resource definition types are described in Requirements Common to All Resource Definition Types".

5.18.7. Connector Connection Factory Definition

An application may define Connector connection factory resources.

The resource may be defined in any of the JNDI namespaces described in Application Component Environment Namespaces".

A Connector connection factory resource may be defined in a web module, enterprise bean module, or application deployment descriptor using the *connection-factory* element.

For example:

```
<connection-factory>
  <description>Sample Connector resource definition</description>
  <name>java:app/myConnectionFactory</name>
  <interface-name>
     com.eis.ConnectionFactory
   </interface-name>
   <resource-adapter>MyEISRA</resource-adapter>
   <max-pool-size>20</max-pool-size>
   <min-pool-size>10</min-pool-size>
   <transaction-support>XATransaction</transaction-support>
   <property>
     <name>Property1</name>
     <value>prop1val</value>
   </property>
   <property>
     <name>Property2</name>
     <value>prop2val</value>
   </property>
</connection-factory>
```

A Connector connection factory resource may also be defined using the *ConnectionFactoryDefinition* annotation on a container-managed class, such as a servlet or enterprise bean class.

For example:

```
  @ConnectionFactoryDefinition(
       name="java:app/myConnectionFactory",
       interfaceName="com.eis.ConnectionFactory",
       resourceAdapter="MyESRA")
```

The *ConnectionFactoryDefinition* annotation can be overridden by a deployment descriptor when the application is deployed.

Once defined, a Connector connection factory resource may be referenced by a component using the *resource-ref* deployment descriptor element or the *Resource* annotation. For example, the above Connector connection factory resource could be referenced as follows:

```
  @Stateless
  public class MySessionBean {
    @Resource(lookup = "java:app/myConnectionFactory")
    ConnectionFactory myCF;
    ...
  }
```

5.18.7.1. Application Component Provider's Responsibilities

The Application Component Provider is responsible for the definition of a Connector connection factory resource using a *ConnectionFactoryDefinition* annotation or the *connection-factory* deployment descriptor element.

5.18.7.2. Deployer's Responsibilities

Requirements common to all resource definition types are described in Requirements Common to All Resource Definition Types". The following additional requirements apply:

- A resource of the specified type must be provided.

5.18.7.3. Jakarta EE Product Provider's Responsibilities

Requirements common to all resource definition types are described in Requirements Common to All Resource Definition Types".

5.18.8. Connector Administered Object Definition

An application may define a Connector administered object resource. The administered object resource may be defined in any of the JNDI namespaces described in Application Component Environment Namespaces".

An administered object resource may be defined in a web module, enterprise bean module, or application deployment descriptor using the *administered-object* element. Properties that are specified are used in the configuration of the administered object, as described in the Connector specification.

For example:

```
<administered-object>
  <description>Sample Admin Object definition</description>
  <name>java:app/MyAdminObject</name>
  <class-name>com.extraServices.AdminObject</class-name>
  <resource-adapter>myESRA</resource-adapter>
  <property>
    <name>Property1</name>
    <value>10</value>
  </property>
  <property>
    <name>Property2</name>
    <value>20</value>
  </property>
</administered-object>
```

An administered object resource may also be defined using the *AdministeredObjectDefinition*

annotation on a container-managed class, such as a servlet or enterprise bean class.

For example:

```
@AdministeredObjectDefinition(
    name="java:app/myAdminObject",
    className="com.extraServices.AdminObject",
    resourceAdapter="myESRA")
```

The *AdministeredObjectDefinition* annotation can be overridden by a deployment descriptor when the application is deployed.

Once defined, an administered object resource may be referenced by a component using the *resource-env-ref* deployment descriptor element or the *Resource* annotation. For example, the above administered object resource could be referenced as follows:

```
  @Stateless public class MySessionBean {
    @Resource(lookup="java:app/myAdminObject")
    AdminObject myAdminObject;
    ...
  }
```

5.18.8.1. Application Component Provider's Responsibilities

The Application Component Provider is responsible for the definition of an administered object resource using an *AdministeredObjectDefinition* annotation or the *administered-object* deployment descriptor element.

5.18.8.2. Deployer's Responsibilities

Requirements common to all resource definition types are described in Requirements Common to All Resource Definition Types". The following additional requirements apply:

If a class name is specified, an administered object resource of the specified class (or a subclass) must be provided.

5.18.8.3. Jakarta EE Product Provider's Responsibilities

Requirements common to all resource definition types are described in Requirements Common to All Resource Definition Types."

5.19. Default Data Source

The Jakarta EE Platform requires that a Jakarta EE Product Provider provide a database in the

operational environment (see Database"). The Jakarta EE Product Provider must also provide a preconfigured, default data source for use by the application in accessing this database.

The Jakarta EE Product Provider must make the default data source accessible to the application under the JNDI name *java:comp/DefaultDataSource* .

The Application Component Provider or Application Assembler may explicitly bind a DataSource resource reference to the default data source using the *lookup* element of the *Resource* annotation or the *lookup-name* element of the *resource-ref* deployment descriptor element. For example,

```
@Resource(lookup="java:comp/DefaultDataSource")
DataSource myDS;
```

In the absence of such a binding, or an equivalent product-specific binding, the mapping of the reference will default to the product's default data source.

For example, the following will map to a preconfigured data source for the product's default database:

```
@Resource
DataSource myDS;
```

5.19.1. Jakarta EE Product Provider's Responsibilities

The Jakarta EE Product Provider must provide a database in the operational environment. The Jakarta EE Product Provider must also provide a preconfigured, default data source for use by the application in accessing this database under the JNDI name *java:comp/DefaultDataSource* .

If a DataSource resource reference is not mapped to a specific data source by the Application Component Provider, Application Assembler, or Deployer, it must be mapped by the Jakarta EE Product Provider to a preconfigured data source for the Jakarta EE Product Provider's default database.

5.20. Default Jakarta Messaging Connection Factory

The Jakarta EE Platform requires that a Jakarta EE Product Provider provide a Jakarta Messaging provider in the operational environment (see Jakarta™ Message Service (Jakarta Messaging)") . The Jakarta EE Product Provider must also provide a preconfigured, Jakarta Messaging ConnectionFactory for use by the application in accessing this Jakarta Messaging provider.

The Jakarta EE Product Provider must make the default Jakarta Messaging connection factory accessible to the application under the JNDI name *java:comp/DefaultJMSConnectionFactory* .

The Application Component Provider or Application Assembler may explicitly bind a Jakarta Messaging ConnectionFactory resource reference to the default connection factory using the *lookup*

element of the *Resource* annotation or the *lookup-name* element of the *resource-ref* deployment descriptor element. For example,

```
@Resource(name="myJMSCF",
          lookup="java:comp/DefaultJMSConnectionFactory")
ConnectionFactory myJMScf;
```

In the absence of such a binding, or an equivalent product-specific binding, the mapping of the reference will default to a Jakarta Messaging connection factory for the product's Jakarta Messaging provider.

For example, the following will map to a preconfigured connection factory for the product's default Jakarta Messaging provider:

```
@Resource(name="myJMSCF")
ConnectionFactory myJMScf;
```

5.20.1. Jakarta EE Product Provider's Responsibilities

The Jakarta EE Product Provider must provide a Jakarta Messaging provider in the operational environment. The Jakarta EE Product Provider must also provide a preconfigured, default Jakarta Messaging connection factory for use by the application in accessing this provider under the JNDI name *java:comp/DefaultJMSConnectionFactory* .

If a Jakarta Messaging ConnectionFactory resource reference is not mapped to a specific Jakarta Messaging connection factory by the Application Component Provider, Application Assembler, or Deployer, it must be mapped by the Jakarta EE Product Provider to a preconfigured Jakarta Messaging connection factory for the Jakarta EE Product Provider's default Jakarta Messaging provider.

5.21. Default Jakarta Concurrency Objects

The Jakarta EE Platform requires that a Jakarta EE Product Provider provide a preconfigured default managed executor service, a preconfigured default managed scheduled executor service, a preconfigured default managed thread factory, and a preconfigured default context service for use by the application.

The Jakarta EE Product Provider must make the default Jakarta Concurrency objects accessible to the application under the following JNDI names:

- *java:comp/ DefaultManagedExecutorService* for the preconfigured managed executor service
- *java:comp/DefaultManagedScheduledExecutorService* for the preconfigured managed scheduled executor service
- *java:comp/DefaultManagedThreadFactory* for the preconfigured managed thread factory

- *java:comp/DefaultContextService* for the preconfigured context service

The Application Component Provider or Application Assembler may explicitly bind a resource reference to a default Jakarta Concurrency object using the *lookup* element of the *Resource* annotation or the *lookup-name* element of the *resource-ref* deployment descriptor element. For example,

```
@Resource(name="myManagedExecutorService",
          lookup="java:comp/DefaultManagedExecutorService")
ManagedExecutorService myManagedExecutorService;
```

In the absence of such a binding, or an equivalent product-specific binding, the mapping of the reference will default to the product's default managed executor service.

For example, the following will map to a preconfigured default managed executor service for the product:

```
@Resource(name="myManagedExecutorService")
ManagedExecutorService myManagedExecutorService;
```

5.21.1. Jakarta EE Product Provider's Responsibilities

The Jakarta EE Product Provider must provide the following:

a preconfigured, default managed executor service for use by the application in accessing this service under the JNDI name *java:comp/DefaultManagedExecutorService* ;

- a preconfigured, default managed scheduled executor service for use by the application in accessing this service under the JNDI name *java:comp/DefaultManagedScheduledExecutorService* ;
- a preconfigured, default managed thread factory for use by the application in accessing this factory under the JNDI name *java:comp/DefaultManagedThreadFactory* ;
- a preconfigured, default context service for use by the application in accessing this service under the JNDI name *java:comp/DefaultContextService* .

If a Jakarta Concurrency object resource environment reference is not mapped to a specific configured object by the Application Component Provider, Application Assembler, or Deployer, it must be mapped by the Jakarta EE Product Provider to a preconfigured Jakarta Concurrency object for the Jakarta EE Product Provider.

5.22. Managed Bean References

This section describes the metadata annotations and deployment descriptor entries that allow an application to obtain instances of a Managed Bean.

An instance of a named Managed Bean can be obtained by looking up its name in JNDI using the same naming scheme used for Jakarta Enterprise Beans components:

```
java:app/<module-name>/<bean-name>

java:module/<bean-name>
```

The latter will only work within the module the Managed Bean is declared in.

Each such lookup must return a new instance.

Alternatively, the *Resource* annotation can be used to request the injection of a Managed Bean given either its type or its name. If a name is specified using the *lookup* element then the type of the resource can be any of the types that the Managed Bean class implements, including any of its interfaces. If no name is specified, the type must be the Managed Bean class itself. (Note that the *name* element of the *Resource* annotation serves an entirely different purpose than the *lookup* element, consistently with other uses of *Resource* in this specification.) The *authenticationType* and *shareable* elements of the *Resource* annotation must not be specified.

For example, given a *ShoppingCartBean* bean named *"cart"* defined in the same module as the client code and implementing the *ShoppingCart* interface, a client may use any of the following methods to obtain an instance of the bean class:

```
@Resource ShoppingCartBean cart;

@Resource(lookup="java:module/cart") ShoppingCart cart;

ShoppingCart cart = (ShoppingCart) context.lookup("java:module/cart");
```

References to managed beans can be declared in the deployment descriptor using the *resource-ref* element. The *res-type* element must contain a type that the managed bean implements. The *lookup-name* must be present and refer to a managed bean by name. The *res-sharing-scope* and *res-auth* elements may be omitted; if present, they must have the values *Shareable* and *Container* respectively, so as to match the default values of the corresponding elements of the *Resource* annotation.

The following example shows how to declare references to the shopping cart bean of the previous example, this time using descriptors. (To make the example somewhat more realistic, one should add an *injection-target* child element to *resource-ref* .)

```
<resource-ref>
  <res-ref-name>bean/cart</ref-ref-name>
  <ref-type>com.acme.ShoppingCart</ref-type>
  <lookup-name>java:module/cart</lookup-name>
</resource-ref>
```

5.22.1. Application Component Provider's Responsibilities

The Application Component Provider is responsible for requesting injection of a Managed Bean or for looking it up in JNDI using an appropriate name.

5.22.2. Jakarta EE Product Provider's Responsibilities

The Jakarta EE Product Provider is responsible for providing appropriate instances of the requested Managed Bean class as required by this specification.

5.23. Bean Manager References

This section describes the metadata annotations and deployment descriptor entries that allow an application to obtain instances of the CDI *BeanManager* type.

Typically, only portable extensions using the CDI SPI need to access a *BeanManager* . Application code may occasionally require access to that interface; in that case, the application should either look up a *BeanManager* instance in JNDI under the name *java:comp/BeanManager* , or request the injection of an object of type *javax.enterprise.inject.spi.BeanManager* via the *Resource* annotation. If the latter, the *authenticationType* and *shareable* elements of the *Resource* annotation must not be specified.

```
@Resource BeanManager manager;
```

Per the CDI specification, a bean can also request the injection of a *BeanManager* using the *Inject* annotation.

```
@Inject BeanManager manager;
```

A *BeanManager* object reference may also be declared in a deployment descriptor in the same way as a resource environment reference.

5.23.1. Application Component Provider's Responsibilities

The Application Component Provider is responsible for requesting injection of a *BeanManager* instance using a *Resource* annotation, or using the defined name to look up an instance in JNDI.

5.23.2. Jakarta EE Product Provider's Responsibilities

The Jakarta EE Product Provider is responsible for providing appropriate *BeanManager* instances as required by this specification.

5.24. Support for Dependency Injection

In Jakarta EE, support for dependency injection annotations as specified in the Dependency Injection for Java specification is mediated by CDI. Containers must support injection points annotated with the *javax.inject.Inject* annotation only to the extent dictated by CDI.

Per the CDI specification, dependency injection is supported on managed beans. There are currently three ways for a class to become a managed bean:

1. Being a Jakarta Enterprise Beans session bean component.
2. Being annotated with the *ManagedBean* annotation.
3. Satisfying the conditions described in the CDI specification.

Classes that satisfy at least one of these conditions will be eligible for full dependency injection support as described in the CDI specification.

Component classes listed in Component classes supporting injection that satisfy the third condition above, but neither the first nor the second condition, can also be used as CDI managed beans if they are annotated with a CDI bean-defining annotation or contained in a bean archive for which CDI is enabled. However, if they are used as CDI managed beans (e.g., injected into other managed classes), the instances that are managed by CDI may not be the instances that are managed by the Jakarta EE container.

Therefore, to make injection support more uniform across all Jakarta EE component types, Jakarta EE containers are required to support field, method, and constructor injection using the *javax.inject.Inject* annotation into all component classes listed in Component classes supporting injection as having the "Standard" level of injection support, as well as the use of interceptors for these classes. Such injection must be performed in the same logical phase as resource injection of fields and methods annotated with the *Resource* annotation. In particular, dependency injection must precede the invocation of any methods annotated with the *PostConstruct* annotation. In supporting such injection points, the container must behave as if it carried out the following steps, involving the use of the CDI SPI. Note that using these steps causes the container to create a non-contextual instance, which is not managed by CDI but rather by the Jakarta EE container.

1. Obtain a *BeanManager* instance.
2. Create an *AnnotatedType* instance for the component into which injection is to occur.
3. Create an *InjectionTarget* instance for the annotated type.
4. Create a *CreationalContext* , passing in *null* to the *BeanManager createCreationalContext* method.

5. Instantiate the component by calling the *InjectionTarget produce* method.
6. Inject the component instance by calling the *InjectionTarget inject* method on the instance.
7. Invoke the *PostConstruct* callback, if any, by calling the *InjectionTarget postConstruct* method on the instance.

When such a non-contextual instance is to be destroyed, the container should behave as if it carried out the following steps.

1. Invoke the *PreDestroy* callback, if any, by calling the *InjectionTarget preDestroy* method on the instance.
2. Invoke the *InjectionTarget dispose* method on the instance.
3. Invoke the *CreationalContext release* method to destroy any dependent objects of the instance.

Containers may optimize the steps above, e.g., by avoiding calls to the actual CDI SPI and relying on container-specific interfaces instead, as long as the outcome is the same.

---

[4] Note that Jakarta™ Managed Beans are required to have access to the JNDI naming environment of their calling component.
[5] Note that the use of interceptors defined by means of the Interceptors annotation is supported in the absence of CDI for Jakarta™ Enterprise Beans and Jakarta™ Managed Bean components.
[6] See the Jakarta™ Server Faces specification section Jakarta™ Server Faces Managed Classes and Jakarta™ Annotations" for a list of these managed classes.
[7] Resource and CDI injection is supported only in Jakarta™ RESTful Web Services components managed by CDI.
[8] Interceptors cannot be bound to other interceptors.
[9] See the CDI specification for requirements related to resource injection in CDI managed beans.
[10] We use this term to refer to classes that become managed beans per the rules in the CDI specification, thus excluding managed beans declared using the *ManagedBean* annotation as well as Jakarta™ Enterprise Beans session beans, both of which would be managed beans even in the absence of CDI.
[11] Interceptors cannot be bound to decorators.
[12] Connections obtained from the same resource manager connection factory through a different resource manager connection factory reference may be shareable.
[13] Note that the Deployer is not prohibited from overriding the resource address.

Chapter 6. Application Programming Interface

This chapter describes API requirements for the Jakarta™ Platform, Enterprise Edition (Jakarta EE). Jakarta EE requires the provision of a number of APIs for use by Jakarta EE applications, starting with the core Java APIs and including many additional Java technologies.

6.1. Required APIs

Jakarta EE application components execute in runtime environments provided by the containers that are a part of the Jakarta EE platform. The full Jakarta EE platform supports four types of containers corresponding to Jakarta EE application component types: application client containers; applet containers; web containers for servlets, Jakarta Server Pages, Jakarta Server Faces applications, Jakarta RESTful Web Services applications; and enterprise bean containers. A Jakarta EE profile may support only a subset of these component types, as defined by the individual Jakarta EE profile specification.

The per-technology requirements in this chapter apply to any Jakarta EE product that includes the technology. Note that even though a Jakarta EE profile might not require support for a particular technology, a Jakarta EE product based on that Jakarta EE profile might nonetheless include support for the technology. In such a case, the requirements for that technology described in this chapter would apply.

6.1.1. Java Compatible APIs

The containers provide all application components with at least the Java Platform, Standard Edition, v8 (Java SE) APIs. Containers may provide newer versions of the Java SE platform, provided they meet all the Jakarta EE platform requirements. The Java SE platform includes the following enterprise technologies:

- Java IDL
- JDBC
- RMI-IIOP
- JNDI
- JAXP
- StAX
- JAAS
- JMX
- JAX-WS
- JAXB

- JAF
- SAAJ
- Common Annotations

In particular, the applet execution environment must be Java SE 8 compatible. Since typical browsers don't yet provide such support, Jakarta EE products may make use of the Java Plugin to provide the required applet execution environment. Use of the Java Plugin is not required, but is one method of meeting the requirement to provide a Java SE 8 compatible applet execution environment. This specification adds no requirements to the applet container beyond those specified by Java SE.

Some of the enterprise technologies that are included in Java SE 8 are also available independently of the Java SE platform, and this specification requires newer versions of some of these technologies, as described in the following section.

The specifications for the Java SE APIs are available at *http://docs.oracle.com/javase/8/docs/* .

6.1.2. Required Java Technologies

The full Jakarta EE platform also provides a number of Java technologies in each of the containers defined by this specification. *Jakarta EE Technologies* indicates the technologies with their required versions, which containers include the technologies, and whether the technology is required (REQ), proposed optional (POPT), or optional (OPT). Each Jakarta EE profile specification will include a similar table describing which technologies are required for the profile. Note that some technologies are marked Optional, as described in the next section.

Table 2. Jakarta EE Technologies

Java Technology	App Client	Web	Enterprise Beans	Status
Enterprise Beans 3.2	Y [14]	Y	Y	REQ, OPT [15], POPT [16]
Servlet 4.0	N	Y	N	REQ
Server Pages 2.3	N	Y	N	REQ
Expression Language 3.0	N	Y	N	REQ
Messaging 2.0	Y	Y	Y	REQ
Transactions 1.3	N	Y	Y	REQ
Mail 1.6	Y	Y	Y	REQ
Connector 1.7	N	Y	Y	REQ
Enterprise Web Services 1.4	Y	Y	Y	REQ
XML RPC 1.1	Y	Y	Y	OPT

Java Technology	App Client	Web	Enterprise Beans	Status
RESTful Web Services 2.1	N	Y	N	REQ
WebSocket 1.1	N	Y	N	REQ
JSON Processing 1.1	Y	Y	Y	REQ
JSON Binding 1.0	Y	Y	Y	REQ
Concurrency 1.1	N	Y	Y	REQ
Batch 1.0	N	Y	Y	REQ
XML Registries 1.0	Y	Y	Y	OPT
Management 1.1	Y	Y	Y	REQ
Deployment 1.7 [17]	N	N	N	OPT
Authorization 1.5	N	Y	Y	REQ
Authentication 1.1	N	Y	Y	REQ
Security API 1.0	N	Y	Y	REQ
Server Pages Debugging 1.0	N	Y	N	REQ
Standard Tag Library 1.2	N	Y	N	REQ
Server Faces 2.3	N	Y	N	REQ
Common Annotations 1.3	Y	Y	Y	REQ
Persistence 2.2	Y	Y	Y	REQ
Bean Validation 2.0	Y	Y	Y	REQ
Managed Beans 1.0	Y	Y	Y	REQ
Interceptors 1.2	Y	Y	Y	REQ
Contexts and Dependency Injection 2.0	Y	Y	Y	REQ
Dependency Injection 1.0	Y	Y	Y	REQ

All classes and interfaces required by the specifications for the APIs must be provided by the Jakarta EE containers indicated above. In some cases, a Jakarta EE product is not required to provide objects that implement interfaces intended to be implemented by an application server, nevertheless, the definitions of such interfaces must be included in the Jakarta EE platform. If an implementation

includes support for a technology marked as Optional, that technology must be supported in the containers specified above. If a product implementation does not support a technology marked as Optional, it must not include the APIs for that technology.[18]

6.1.3. Pruned Java Technologies

As the Jakarta EE specification has evolved, some of the technologies originally included in Jakarta EE are no longer as relevant as they were when they were introduced to the platform. The Jakarta EE expert group follows a process first defined by the Java SE expert group (*http://mreinhold.org/blog/removing-features*) to prune technologies from the platform in a careful and orderly way that minimizes the impact to developers using these technologies, while allowing the platform to grow even stronger. In short, the process defines two steps:

1. The Umbrella Expert Group (UEG) for release N of the platform decides to propose that a particular feature be removed. The specification for that release documents the proposal.
2. The UEG for release N+1 decides whether to remove the feature from that release, retain it as a required component, or leave it in the "proposed removal" state for the next UEG to decide.

The result of successfully applying this policy to a feature is not the actual deletion of the feature but rather the conversion of the feature from a required component of the platform into an optional component. No actual removal from the specification occurs, although the feature may be removed from products at the choice of the product vendor.

Technologies that have been pruned as of Jakarta EE 8 are marked Optional in Jakarta EE Technologies. Technologies that may be pruned in a future release are marked Proposed Optional in Jakarta EE Technologies.

6.2. Java Platform, Standard Edition (Java SE) Requirements

6.2.1. Programming Restrictions

The Jakarta EE programming model divides responsibilities between Application Component Providers and Jakarta EE Product Providers: Application Component Providers focus on writing business logic and the Jakarta EE Product Providers focus on providing a managed system infrastructure in which the application components can be deployed.

This division leads to a restriction on the functionality that application components can contain. If application components contain the same functionality provided by Jakarta EE system infrastructure, there are clashes and mis-management of the functionality.

For example, if enterprise beans were allowed to manage threads, the Jakarta EE platform could not manage the life cycle of the enterprise beans, and it could not properly manage transactions.

Since we do not want to subset the Java SE platform, and we want Jakarta EE Product Providers to be

able to use Java SE products without modification in the Jakarta EE platform, we use the Java SE security permissions mechanism to express the programming restrictions imposed on Application Component Providers.

In this section, we specify the Java SE security permissions that the Jakarta EE Product Provider must provide for each application component type. We call these permissions the Jakarta EE security permissions set. The Jakarta EE security permissions set is a required part of the Jakarta EE API contract. We also specify the set of permissions that the Jakarta EE Product Provider must be able to restrict from being provided to application components. In addition, we specify the means by which application component providers may declare the need for specific permissions and how these declarations must be processed by Jakarta EE products.

The Java SE security permissions are fully described in *http://docs.oracle.com/javase/8/docs/technotes/guides/security/permissions.html* .

6.2.2. Jakarta EE Security Manager Related Requirements

Every Jakarta EE product must be capable of running with a Java security manager that enforces Java security permissions and that prevents application components from performing operations for which they have not been provided the required permissions.

6.2.2.1. Jakarta EE Product Provider's Responsibilities

A Jakarta EE product may allow application components to run without a security manager, but every Jakarta EE product must be capable of running application components with a security manager that enforces security permissions, as described below.

The set of security permissions provided to application components by a particular installation is a matter of policy outside the scope of this specification, however, every Jakarta EE product must be capable of running with a configuration that provides application classes and packaged libraries the permissions defined in Jakarta EE Security Permissions Set.

All Jakarta EE products must allow the set of permissions available to application classes in a module to be configurable, providing application components in some modules with different permissions than those described in Jakarta EE Security Permissions Set.

As defined in Declaring Permissions Required by Application Components," a component provider may declare the permissions required by the application classes and libraries packaged in a module. When a component provider has declared the permissions required by a module, on successful deployment of the module, at least the declared permissions must have been granted to the application classes and libraries packaged in the module. If security permissions are declared that conflict with the policy of the product installation, the Jakarta EE product must fail deployment of the application module. If an application module does not contain a declaration of required security permissions and deployment otherwise succeeds, the Jakarta EE product must grant the application classes and libraries the permissions established by the security policy of the installation. The Jakarta EE product must ensure that the system administrator for the installation be able to define the security policy for the

installation to include the permissions in Jakarta EE Security Permissions Set.

Note that, on some installations of Jakarta EE products, the security policy of the installation may be such that applications are granted fewer permissions than those defined in

Jakarta EE Security Permissions Set and, as a result, some applications that declare only the permissions defined in Jakarta EE Security Permissions Set may not be deployable. Other applications that require the same permissions but do not declare them may deploy but will encounter runtime failures when the missing permission is required by the application component.

Every Jakarta EE product must be capable of running with a Java security manager and with an installation policy that does not grant the permissions described in Restrictable Jakarta EE Security Permissions to Web, enterprise beans, and resource adapter components. That environment must otherwise fully support the requirements of this specification.

6.2.2.2. Application Component Provider's Responsibilities

To ensure that application deployment will only succeed if required permissions are compatible with security policy of the installation environment, application component providers should declare all Java security permissions required by their application components.

Declaring Permissions Required by Application Components," defines the mechanism(s) by which required permissions may be declared.

Note that, while FilePermissions or SocketPermissions for specific resources may be granted as a result of application components declaring them as required, the local operating system or network security policy may restrict access to the requested resources. This may result in a runtime failure to access these resources even though deployment of the application has succeeded.

6.2.2.3. System Administrator's Responsibilities

Security policy requirements differ from one installation environment to another. The system administrator is responsible for configuring the permissions available to application modules to meet the security policy requirements of the installation environment. For example, cloud environments may require greater restrictions on the system resources available to applications than on-premise enterprise installations. Note that restricting the permissions beyond those in Jakarta EE Security Permissions Set may prevent some applications from working correctly.

Care should be taken by the system administrator to ensure that resources that are expected to be available to application components are appropriately represented in the security policy of the operational environment.

In particular, the temporary file directory made available through the ServletContext attribute *javax.servlet.context.tempdir* should be available to deployed applications. The security policy of the operational environment should grant the application server process access to the corresponding part of the file system. The Jakarta EE Product must be capable of using the security manager to enforce that an application only has access to the part of the filesystem namespace named by the

javax.security.context.tempdir attribute, and that that part of the filesystem namespace is separate from the corresponding filesystem namespace available to other applications.

6.2.2.4. Listing of the Jakarta EE Security Permissions Set

Jakarta EE Security Permissions Set lists the Java permissions that Jakarta EE components (by type) can reliably be granted by a Jakarta EE product, given appropriate local installation configuration.

Table 3. Jakarta EE 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
Web, Enterprise Beans, and Resource Adapter Components		
java.lang.RuntimePermission	loadLibrary.*	
java.lang.RuntimePermission	queuePrintJob	
java.net.SocketPermission	*	connect
java.io.FilePermission	*	read,write [19]
java.io.FilePermission	file:${javax.servlet.context.tempdir}	read, write [20]
java.util.PropertyPermission	*	read

6.2.2.5. Restrictable Jakarta EE Security Permissions

Restrictable Jakarta EE Security Permissions lists the Java permissions that a Jakarta EE product must be capable of restricting when running a Web or Enterprise Beans application component. If the Target field is empty, a Jakarta EE product must be capable of deploying application modules such that no instances of that permission are granted to the components in the application module.

Table 4. Restrictable Jakarta EE Security Permissions

Security Permissions	Target	Action
Web, Enterprise Beans, and Resource Adapter Components		
java.security.AllPermission		
java.security.SecurityPermission		
java.security.UnresolvedPermission		
java.awt.AWTPermission		
java.io.SerializablePermission		
java.lang.reflect.ReflectPermission		
java.lang.RuntimePermission	<any except loadLibrary.* and queuePrintJob> [21]	
java.net.NetPermission		
java.sql.SQLPermission		
java.util.PropertyPermission	<any>	write [22]
java.util.logging.LoggingPermission		
javax.net.ssl.SSLPermission		
java.security.auth.AuthPermission		
java.security.auth.PrivateCredentialPermission		
java.security.auth.kerberos.DelegationPermission		
java.security.auth.kerberos.ServicePermission		
javax.sound.sampled.AudioPermission		

6.2.2.6. Declaring Permissions Required by Application Components

By declaring the permissions required by an application as described in this section, an application component provider is ensured, through the successful deployment of his or her application, that the Jakarta EE Product has granted at least the declared permissions to the classes and libraries packaged in the application module.

Since the specific set of permissions granted to a successfully deployed application is a function of the security policy for the installation and the permissions declared within the *permissions.xml* files, the application component provider is ensured that the effective permission set consists of at least those permissions that are declared within the application.

Permission declarations must be stored in *META-INF/permissions.xml* file within an enterprise beans, web, application client, or resource adapter archive in order for them to be located and subsequently processed by the deployment machinery of the Jakarta EE Product. The Jakarta EE Product is not required to support *permissions.xml* files that specify permission classes that are packaged in the application.

The permissions for a packaged library are the same as the permissions for the module. Thus, if a library is packaged in a *.war* file, it gets the permissions of the *.war* file.

For applications packaged in an *.ear* file, the declaration of permissions must be at *.ear* file level. This permission set is applied to all modules and libraries packaged within the *.ear* file or within its contained modules. Any *permissions.xml* files within such packaged modules are ignored, regardless of whether a *permissions.xml* file has been supplied for the *.ear* file itself.

The fact that these permission declarations are being made from within the context of a particular application implies the codeBase(s) to which the grant should be made. This simplifies the syntax that is needed to just the Permission class name and two String arguments. This aligns the declaration syntax with the default policy language and the constructor signature for permissions that is compliant with the default policy syntax.

```
permission <class> [<name> [, <action list>]];
```

The following is an example of a permission set declaration:

```
...
<permissions>
  <permission>
    <class-name>java.io.FilePermission</class-name>
    <name>/tmp/abc</name>
    <actions>read,write</actions>
  </permission>
  <permission>
    <class-name>java.lang.RuntimePermission</class-name>
    <name>createClassLoader</name>
  </permission>
</permissions>
...
```

The Jakarta EE permissions XML Schema is located at *http://xmlns.jcp.org/xml/ns/javaee/permissions_8.xsd* .

6.2.3. Additional Requirements

6.2.3.1. Networking

The Java SE platform includes a pluggable mechanism for supporting multiple URL protocols through the *java.net.URLStreamHandler* class and the *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* . File access is restricted according to the permissions described above.
- *http :* Version 1.1 of the HTTP protocol must be supported. An *http* URL must support both input and output.
- *https* : SSL version 3.0 and TLS version 1.2 must be supported by *https* URL objects. Both input and output must be supported.

The Java SE 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 can convert 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 *Java Type of Objects Returned When Using the getContent Method* must be returned.

Table 5. Java Type of Objects Returned When Using the getContent Method

MIME Type	Java Type
image/gif	java.awt.Image
image/jpeg	java.awt.Image
image/png	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 Java SE 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. 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, and RMI-IIOP, can pass through firewalls.

These considerations have implications on the use of various protocols to communicate between application components. This specification requires that HTTP access through firewalls be possible where local policy allows. Some Jakarta EE products may provide support for tunneling other communication through firewalls, but this is neither specified nor required. Application developers should consider the impact of these issues in the design of applications, particularly in view of cloud environments, where a cloud platform provider might only allow HTTP-based access.

6.2.3.2. JDBC™ API

The JDBC API, which is part of the Java SE platform, allows for access to a wide range of data storage systems. The Java SE platform, however, 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, the Jakarta EE specification does require that such a database be available and accessible from a Jakarta EE 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.

Jakarta EE 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 Resource Manager Connection Factory References.” The Jakarta EE 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 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, roll back 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 Jakarta Enterprise Beans specification contains the precise rules for *enterprise beans.*

Note that the same restrictions apply when a component creates a transaction using the Jakarta Transactions *UserTransaction* interface. The component should not attempt the operations listed above on the JDBC *Connection* object that would conflict with the transaction context.

Drivers supporting the JDBC API in a Jakarta EE environment must meet the JDBC API Compliance requirements as specified in the JDBC specification.

The JDBC API includes APIs for 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. Jakarta EE 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, and a full packaging and deployment functionality for resource adapters. A Jakarta EE product that supports the Connector architecture must support deploying and using a JDBC driver that has been written and packaged as a resource adapter using the Connector architecture.

The JDBC 4.2 specification is available at *https://jcp.org/en/jsr/detail?id=221* .

6.2.3.3. Jakarta API for XML Web Services (JAX-WS) Requirements

The JAX-WS specification provides support for web services that use the JAXB API for binding XML data to Java objects. The JAX-WS specification defines client APIs for accessing web services as well as techniques for implementing web service endpoints. The Web Services for Jakarta EE specification describes the deployment of JAX-WS-based services and clients. The Enterprise Beans and Servlet specifications also describe aspects of such deployment. It must be possible to deploy JAX-WS-based applications using any of these deployment models.

The JAX-WS specification describes the support for message handlers that can process message requests and responses. In general, these message handlers execute in the same container and with the same privileges and execution context as the JAX-WS client or endpoint component with which they are associated. These message handlers have access to the same JNDI *java:comp/env* namespace as their associated component. Custom serializers and deserializers, if supported, are treated in the same way as message handlers.

The JAX-WS specification is available at *http://jcp.org/en/jsr/summary?id=224* .

6.2.3.4. Java IDL (Proposed Optional)

The requirements in this section only apply to Jakarta EE products that support interoperability using CORBA.

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

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

The IIOP protocol supports the ability to multiplex calls over a single connection. All Jakarta EE products must support requests from clients that multiplex calls on a connection to either Java IDL server objects or RMI-IIOP server objects (such as enterprise beans). The server must allow replies to be sent in any order, to avoid deadlocks where one call would be blocked waiting for another call to complete. Jakarta EE clients are not required to multiplex calls, although such support is highly recommended.

A Jakarta EE product must provide support for a CORBA Portable Object Adapter (POA) to support portable stub, skeleton, and tie classes. A Jakarta EE application that defines or uses CORBA objects other than enterprise beans must include such portable stub, skeleton, and tie classes in the application package.

Jakarta EE applications need to use an instance of *org.omg.CORBA.ORB* to perform many Java IDL 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, enterprise bean, and application client 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, portable 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 Jakarta EE product must provide a COSNaming service to support the Jakarta Enterprise Beans interoperability requirements. It must be possible to access this COSNaming service using the Java IDL 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://www.omg.org/cgi-bin/doc?formal/2000-06-19* .

6.2.3.5. RMI-JRMP

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

6.2.3.6. RMI-IIOP (Proposed Optional)

The requirements in this section only apply to Jakarta EE products that include an Enterprise Beans container and support interoperability using 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 remote *enterprise bean accessible via* RMI-IIOP. Some Jakarta EE products will simply make all remote enterprise beans always (and only) accessible via RMI-IIOP; other products might control this via an administrative or deployment action. These and other approaches are allowed, provided that any remote enterprise bean (or by extension, all remote enterprise beans) can be made accessible using RMI-IIOP.

Components accessing remote *enterprise beans* may need to use the *narrow* method of the *javax.rmi.PortableRemoteObject* class, under circumstances described in the Jakarta Enterprise Beans specification. Because remote enterprise beans may be deployed using other RMI protocols, portable applications must not depend on the characteristics of RMI-IIOP objects (for example, the use of the *Stub* and *Tie* base classes) beyond what is specified in the Jakarta Enterprise Beans specification.

The Jakarta EE security restrictions typically prevent all application component types, except application clients, from creating and exporting an RMI-IIOP object. All Jakarta EE application component types can be clients of RMI-IIOP objects. Jakarta EE applications should also use JNDI to lookup non-Enterprise Beans RMI-IIOP objects. The JNDI names used for such non-Enterprise Beans RMI-IIOP objects should be configured at deployment time using the standard environment entries mechanism (see JNDI Naming Context"). The application should fetch a name from JNDI using an environment entry, and use the 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 Jakarta EE application clients can create non-Enterprise Beans RMI-IIOP server objects for use as callback objects, or to pass in calls to other RMI-IIOP objects.

Note that while RMI-IIOP doesn't specify how to propagate the current security context or transaction context, the Jakarta Enterprise Beans interoperability specification does define such context propagation. This specification only requires that the propagation of context information as defined in the Jakarta Enterprise Beans specification be supported in the use of RMI-IIOP to access enterprise beans. The propagation of context information is not required in the uses of RMI-IIOP to access objects other than enterprise beans.

The RMI-IIOP specification describes how portable Stub and *Tie* classes can be created. To be portable

to all implementations that use a CORBA Portable Object Adapter (POA), the *Tie* classes must extend the *org.omg.PortableServer.Servant* class. This is typically done by using the *-poa* option to the *rmic* command. A Jakarta EE product must provide support for these portable *Stub* and *Tie* classes, typically using the required CORBA POA. However, for portability to systems that do not use a POA to implement RMI-IIOP, applications should not depend on the fact that the *Tie* extends the *Servant* class. A Jakarta EE 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, however, must not be included with the application: they will be generated, if needed, by the Jakarta EE 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://www.omg.org/cgi-bin/doc?formal/99-10-07* , and by the Java™ Language To IDL Mapping Specification, available at *http://www.omg.org/cgi-bin/doc?ptc/2000-01-06* .

6.2.3.7. JNDI

A Jakarta EE product that supports the following types of objects must be able to make them available in the application's JNDI namespace: *EJBHome* objects, *EJBLocalHome* objects, Enterprise Beans business interface objects, Jakarta Transactions *UserTransaction* objects, JDBC API *DataSource* objects, JMS *ConnectionFactory* and *Destination* objects, JavaMail *Session* objects, *URL* objects, resource manager *ConnectionFactory* objects (as specified in the Connector specification), *ORB* objects, *EntityManagerFactory* objects, and other Java language objects as described in Resources, Naming, and Injection." The JNDI implementation in a Jakarta EE 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 Jakarta EE objects are application dependent. The application component's metadata annotations and/or deployment descriptor are used to list the names and types of objects expected. 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 Resources, Naming, and Injection" for details.

Particular names are defined by this specification for the cases when the Jakarta EE product includes the corresponding technology. For all application components that have access to the Jakarta Transaction *UserTransaction* interface, the appropriate *UserTransaction* object can be found using the name *java:comp/UserTransaction* . In all containers except the applet container, application components may lookup a CORBA *ORB* instance using the name *java:comp/ORB* . For all application components that have access to the CDI *BeanManager* interface, the appropriate *BeanManager* object can be found using the name *java:comp/BeanManager* . For all application components that have access to the Validation APIs, the appropriate *Validator* and *ValidatorFactory* objects can be found using the names *java:comp/Validator* and *java:comp/ValidatorFactory* respectively.

The name used to lookup a particular Jakarta EE 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 Jakarta EE product to provide the necessary JNDI service providers for accessing the various objects defined in this specification.

This specification requires that the Jakarta EE platform provide the ability to perform lookup operations as described above. 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.

A Jakarta EE product may be required to provide a COSNaming name service to meet the Jakarta Enterprise Beans interoperability requirements. In such a case, a COSNaming JNDI service provider must be available through the web, Enterprise Beans, 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 Java SE 8 SDK and JRE from Oracle, but is not a required component of the Java SE specification. The COSNaming JNDI service provider specification is available at *http://docs.oracle.com/javase/8/docs/technotes/guides/jndi/jndi-cos.html* .

See Resources, Naming, and Injection" for the complete naming requirements for the Jakarta EE platform. The JNDI specification is available at *http://docs.oracle.com/javase/8/docs/technotes/guides/jndi/index.html* .

6.2.3.8. Context Class Loader

This specification requires that Jakarta EE containers provide a per thread context class loader for the use of system or library classes in dynamically loading classes provided by the application. The Jakarta Enterprise Beans specification requires that all Jakarta Enterprise Beans client containers provide a per thread context class loader for dynamically loading system value classes. The per thread context class loader is accessed using the *Thread* method *getContextClassLoader* .

The classes used by an application will typically be loaded by a hierarchy of class loaders. There may be a top level application class loader, an extension class loader, and so on, down to a system class loader. The top level application class loader delegates to the lower class loaders as needed. Classes loaded by lower class loaders, such as portable Jakarta Enterprise Beans system value classes, need to be able to discover the top level application class loader used to dynamically load application classes.

This specification requires that containers provide a per thread context class loader that can be used to load top level application classes as described above. See Dynamic Class Loading" for recommendations for libraries that dynamically load classes.

6.2.3.9. Jakarta Authentication Requirements

All enterprise beans containers and all web containers must support the use of the Jakarta Authentication APIs as specified in the Connector specification. All application client containers must support use of the Jakarta Authentication APIs.

The Jakarta Authentication specification is available at *https://jakarta.ee/specifications/authentication* .

6.2.3.10. Logging API Requirements

The Logging API provides classes and interfaces in the *java.util.logging* package that are the Java™ platform's core logging facilities. This specification does not require any additional support for logging. A Jakarta EE application typically will not have the *LoggingPermission* necessary to control the logging configuration, but may use the logging API to produce log records. A future version of this specification may require that the Jakarta EE containers use the logging API to log certain events.

6.2.3.11. Preferences API Requirements

The Preferences API in the *java.util.prefs* package allows applications to store and retrieve user and system preference and configuration data. A Jakarta EE application typically will not have the *RuntimePermission("preferences")* necessary to use the Preferences API. This specification does not define any relationship between the principal used by a Jakarta EE application and the user preferences tree defined by the Preferences API. A future version of this specification may define the use of the Preferences API by Jakarta EE applications.

6.3. Jakarta Enterprise Beans 3.2 Requirements

This specification requires that a Jakarta EE product provide support for *enterprise beans* as specified in the Jakarta Enterprise Beans specification. The Jakarta Enterprise Beans specification is available at *https://jakarta.ee/specifications/enterprise-beans* .

This specification does not impose any additional requirements at this time. Note that the Jakarta Enterprise Beans specification includes the specification of the Jakarta Enterprise Beans interoperability protocol based on RMI-IIOP. Support for the Jakarta Enterprise Beans interoperability protocol is Proposed Optional in Jakarta EE 8. All containers that support Jakarta Enterprise Beans clients must be capable of using the Jakarta Enterprise Beans interoperability protocol to invoke enterprise beans. All Jakarta Enterprise Beans containers must support the invocation of enterprise beans using the Jakarta Enterprise Beans interoperability protocol. A Jakarta EE product may also support other protocols for the invocation of enterprise beans.

A Jakarta EE product may support multiple object systems (for example, RMI-IIOP and RMI-JRMP). 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 Java IDL objects as arguments to methods on such an enterprise bean, and to return such object references as return values 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 Java IDL object, or to return such an enterprise bean object reference as a return value from such an RMI-IIOP or Java IDL object.

In a Jakarta EE product that includes both an enterprise beans container and a web container, both containers are required to support access to local enterprise beans. No support is provided for access to local enterprise beans from the application client container or the applet container.

6.4. Jakarta Servlet 4.0 Requirements

The Jakarta Servlet specification defines the packaging and deployment of web applications, whether standalone or as part of a Jakarta EE application. The Servlet specification also addresses security, both standalone and within the Jakarta EE platform. These optional components of the Servlet specification are requirements of the Jakarta EE platform.

The Servlet specification includes additional requirements for web containers that are part of a Jakarta EE product and a Jakarta EE product must meet these requirements as well.

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

Web containers must support Jakarta EE distributable web applications placing objects of any of the following types (when supported by the Jakarta EE product) into a *javax.servlet.http.HttpSession* object using the *setAttribute* or *putValue* methods:

java.io.Serializable

- *javax.ejb.EJBObject*
- *javax.ejb.EJBHome*
- *javax.ejb.EJBLocalObject*
- *javax.ejb.EJBLocalHome*
- *javax.transaction.UserTransaction*
- a *javax.naming.Context* object for the *java:comp/env* context

a reference to an Enterprise Bean local or remote business interface or no-interface view

Web containers may support objects of other types as well. Web containers must throw a *java.lang.IllegalArgumentException* if an object that is not one of the above types, or another type supported by the container, is passed to the *setAttribute* or *putValue* methods of an *HttpSession* object corresponding to a Jakarta EE 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 supported types are accurately recreated on the target VM.

The Servlet specification defines access to local enterprise beans as an optional feature. This specification requires that all Jakarta EE products that include both a web container and an Enterprise Beans container provide support for access to local enterprise beans from the web container.

The Jakarta Servlet specification is available at *https://jakarta.ee/specifications/servlet* .

6.5. Jakarta Server Pages 2.3 Requirements

The Jakarta Server Pages specification depends on and builds on the servlet framework. A Jakarta EE product must support the entire Jakarta Server Pages specification.

The Jakarta Server Pages specification is available at *https://jakarta.ee/specifications/pages* .

6.6. Jakarta Expression Language (EL) 3.0 Requirements

The Jakarta Expression Language specification was formerly a part of the Jakarta Server Pages specification. It was split off into its own specification so that it could be used independently of Jakarta Server Pages. A Jakarta EE product must support the Expression Language.

The Jakarta Expression Language specification is available at *https://jakarta.ee/specifications/expression-language* .

6.7. Jakarta Messaging 2.0 Requirements

A Jakarta Messaging provider must be included in a Jakarta EE product that requires support for Jakarta Messaging. The Jakarta Messaging implementation must provide support for both Jakarta Messaging point-to-point and publish/subscribe messaging, and thus must make those facilities available using the *ConnectionFactory* and *Destination* APIs.

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

- *javax.jms.ServerSession*
- *javax.jms.ServerSessionPool*
- *javax.jms.ConnectionConsumer*

all *javax.jms* XA interfaces

The following methods may only be used by application components executing in the application client container:

- *javax.jms.MessageConsumer* method *getMessageListener*
- *javax.jms.MessageConsumer* method *setMessageListener*
- *javax.jms.JMSConsumer* method *getMessageListener*
- *javax.jms.JMSConsumer* method *setMessageListener*
- *javax.jms.Connection* method *setExceptionListener*
- *javax.jms.Connection* method *stop*

- *javax.jms.Connection* method *setClientID*
- *javax.jms.JMSContext* method *stop*
- *javax.jms.JMSContext* method *setClientID*
- *javax.jms.JMSContext* method *setExceptionListener*
- *javax.jms.JMSContext* method *createContext*
- *javax.jms.Producer* method *setAsync*
- *javax.jms.MessageProducer* method *send(Message message, CompletionListener completionListener)*
- *javax.jms.MessageProducer* method *send(Message message, int deliveryMode, int priority, long timeToLive, CompletionListener completionListener)*
- *javax.jms.MessageProducer* method *send(Destination destination, Message message, CompletionListener completionListener)*
- *javax.jms.MessageProducer* method *send(Destination destination, Message message, int deliveryMode, int priority, long timeToLive, CompletionListener completionListener)*

The following methods may only be used by application components executing in the application client container. Note, however, that these methods provide an expert facility not used by ordinary applications. See the JMS specification for further detail.

javax.jms.Session method *setMessageListener*

- *javax.jms.Session* method *getMessageListener*
- *javax.jms.Session* method *run*
- *javax.jms.Connection* method *createConnectionConsumer*
- *javax.jms.Connection* method *createSharedConnectionConsumer*
- *javax.jms.Connection* method *createDurableConnectionConsumer*

javax.jms.Connection method *createSharedDurableConnectionConsumer*

A Jakarta EE container may throw a *JMSException* (if allowed by the method) or a *JMSRuntimeException* (if throwing a *JMSException* is not allowed by the method) if the application component violates any of the above restrictions.

Application components in the web and enterprise bean containers must not attempt to create more than one active (not closed) *Session* object per connection. An attempt to use the *Connection* object's *createSession* method when an active *Session* object exists for that connection should be prohibited by the container. The container should throw a *JMSException* if the application component violates this restriction. An attempt to use the *JMSContext* object's *createContext* method should be prohibited by the container. The container should throw a *JMSRuntimeException* , since the first *JMSContext* already contains a connection and session and this method would create a second session on the same connection. Application client containers must support the creation of multiple sessions for each connection.

The Jakarta Messaging specification defines further restrictions on the use of Jakarta Messaging in the Enterprise Beans and web containers. In general, the behavior of a Jakarta Messaging provider should be the same in both the enterprise beans container and the web container.

The Jakarta Messaging specification is available at _ https://jakarta.ee/specifications/messaging_ .

6.8. Jakarta Transaction 1.3 Requirements

Jakarta Transaction defines the *UserTransaction* interface that is used by applications to start, and commit or abort transactions. Application components get a *UserTransaction* object through a JNDI lookup using the name *java:comp/UserTransaction* or by requesting injection of a *UserTransaction* object.

Jakarta Transaction also defines the *TransactionSynchronizationRegistry* interface that can be used by system level components such as persistence managers to interact with the transaction manager. These components get a *TransactionSynchronizationRegistry* object through a JNDI lookup using the name *java:comp/TransactionSynchronizationRegistry* or by requesting injection of a *TransactionSynchronizationRegistry* object.

A number of interfaces defined by Jakarta Transaction 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 transparently to the application by a Jakarta EE product.

The Jakarta Transaction specification is available at *https://jakarta.ee/specifications/transactions* .

6.9. Jakarta Mail 1.6 Requirements

The Jakarta Mail 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 Jakarta Mail API specification does not require any specific protocol providers, but the JavaMail reference implementation includes an IMAP message store provider, a POP3 message store provider, and an SMTP message transport provider.

Configuration of the Jakarta Mail 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 Jakarta EE 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, or by using a *Resource* annotation. A Jakarta Mail API *Session* object should be considered a resource factory, as described in Resource Manager Connection Factory References.” This specification requires that the Jakarta EE platform support *javax.mail.Session* objects as resource factories, as described in that section.

The Jakarta EE 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 Jakarta EE product support any message store protocols.

Note that the Jakarta Mail 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 Enterprise Beans containers, for instance, it is typically not possible to create threads.

The Jakarta Mail API uses the JavaBeans Activation Framework API to support various MIME data types. The Jakarta Mail API must include *javax.activation.DataContentHandlers* for the following MIME data types, corresponding to the Java programming language type indicated in *JavaMail API MIME Data Type to Java Type Mappings* .

Table 6. Jakarta Mail API MIME Data Type to Java Type Mappings

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

The Jakarta Mail API specification is available at *https://jakarta.ee/specifications/mail* .

6.10. Jakarta EE Connectors 1.7 Requirements

In full Jakarta EE products, all Jakarta Enterprise Beans containers and all web containers must support the full set of Connector APIs. All such containers must support Resource Adapters that use any of the specified transaction capabilities. The Jakarta EE 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 Jakarta EE Connectors specification is available at *https://jakarta.ee/specifications/connectors* .

6.11. Jakarta XML RPC 1.1 Requirements (Optional)

The Jakarta XML RPC specification defines client APIs for accessing web services as well as techniques for implementing web service endpoints. The Web Services for Jakarta EE specification describes the deployment of Jakarta XML RPC-based services and clients. The Jakarta Enterprise Beans and Servlet specifications also describe aspects of such deployment. In Jakarta EE products that support Jakarta XML RPC, it must be possible to deploy Jakarta XML RPC-based applications using any of these deployment models.

The Jakarta XML RPC specification describes the support for message handlers that can process message requests and responses. In general, these message handlers execute in the same container and with the same privileges and execution context as the Jakarta XML RPC client or endpoint component with which they are associated. These message handlers have access to the same JNDI *java:comp/env* namespace as their associated component. Custom serializers and deserializers, if supported, are treated in the same way as message handlers.

Note that neither web service annotations nor injection is supported for Jakarta XML RPC service endpoints and handlers. New applications are encouraged to use Jakarta XML Web Services to take advantage of these new facilities that make it easier to write web services.

The Jakarta XML RPC specification is available at *https://jakarta.ee/specifications/xml-rpc* .

6.12. Jakarta RESTful Web Services 2.1 Requirements

Jakarta RESTful Web Services defines APIs for the development of Web services built according to the Representational State Transfer (REST) architectural style.

In a full Jakarta EE product, all Jakarta EE web containers are required to support applications that use Jakarta RESTful Web Services technology.

The specification describes the deployment of services as a servlet. It must be possible to deploy Jakarta RESTful Web Services-based applications using this deployment model with the *servlet-class* element of the web.xml descriptor naming the application-supplied extension of the Jakarta RESTful Web Services *Application* abstract class.

The specification defines a set of optional container-managed facilities and resources that are intended to be available in a Jakarta EE container — all such features and resources must be made available.

The Jakarta RESTful Web Services specification is available at *https://jakarta.ee/specifications/restful-ws* .

6.13. Jakarta WebSocket 1.1 (WebSocket) Requirements

The Jakarta WebSocket (WebSocket) is a standard API for creating WebSocket applications. In a full Jakarta EE product, all Jakarta EE web containers are required to support the WebSocket API.

The Jakarta WebSocket specification can be found at *https://jakarta.ee/specifications/websocket* .

6.14. Jakarta JSON Processing 1.1 (JSON-P) Requirements

JSON (JavaScript Object Notation) is a lightweight data-interchange format used by many web services. The Jakarta JSON Processing (JSON-P) provides a convenient way to process (parse, generate, transform, and query) JSON text.

In a full Jakarta EE product, all Jakarta EE application client containers, web containers, and enterprise beans containers are required to support the JSON-P API.

The Jakarta JSON Processing specification can be found at *https://jakarta.ee/specifications/jsonp* .

6.15. Jakarta JSON Binding 1.0 (JSON-B) Requirements

The Jakarta JSON Binding API for JSON Binding (JSON-B) provides a convenient way to map between JSON text and Java objects.

In a full Jakarta EE product, all Jakarta EE application client containers, web containers, and enterprise beans containers are required to support the JSON-B API.

The Jakarta JSON Binding specification can be found at *https://jakarta.ee/specifications/jsonb*.

6.16. Jakarta Concurrency 1.1 (Concurrency Utilities) Requirements

Jakarta Concurrency Utilities for Jakarta EE is a standard API for providing asynchronous capabilities to Jakarta EE application components through the following types of objects: managed executor service, managed scheduled executor service, managed thread factory, and context service. In a full Jakarta EE product, all web containers and enterprise beans containers are required to support the Concurrency Utilities API. The Jakarta EE Product Provider must provide preconfigured default managed executor service, managed scheduled executor service, managed thread factory, and context service objects for use by the application in the containers in which the Concurrency Utilities API is required to be supported.

The Jakarta Concurrency specification can be found at *https://jakarta.ee/specifications/concurrency* .

6.17. Jakarta Batch 1.0 Specification Requirements

The Jakarta Batch provides a programming model for batch applications and a runtime for scheduling and executing jobs.

In a full Jakarta EE product, all Jakarta EE web containers and Jakarta Enterprise Beans containers are required to support the Batch API.

The Jakarta Batch specification can be found at *https://jakarta.ee/specifications/batch* .

6.18. Jakarta XML Registries 1.0 Requirements (Optional)

The Jakarta XML Registries specification defines APIs for client access to XML-based registries such as ebXML registries and UDDI registries. Jakarta EE products that support Jakarta XML Registries must include a Jakarta XML registry provider that meets at least the Jakarta XML Registries level 0 requirements.

The Jakarta XML Registries specification is available at *https://jakarta.ee/specifications/xml-registries* .

6.19. Jakarta Management 1.1 Requirements

Jakarta Management provides APIs for management tools to query a Jakarta EE application server to determine its current status, applications deployed, and so on. All Jakarta EE products must support this API as described in its specification.

The Jakarta Management specification is available at *https://jakarta.ee/specifications/management* .

6.20. Jakarta Deployment API 1.7 Requirements (Optional)

The Jakarta Deployment API defines the interfaces between the runtime environment of a deployment tool and plug-in components provided by a Jakarta EE application server. These plug-in components execute in the deployment tool and implement the Jakarta EE product-specific deployment mechanisms. Jakarta EE products that support the Jakarta Deployment API are required to supply these plug-in components for use in tools from other vendors.

Note that the Jakarta Deployment specification does not define new APIs for direct use by Jakarta EE applications. However, it would be possible to create a Jakarta EE application that acts as a deployment tool and provides the runtime environment required by the Jakarta Deployment The Jakarta EE Deployment API specification is available at *https://jakarta.ee/specifications/deployment* .

6.21. Jakarta Authorization 1.5 Requirements

The Jakarta Authorization specification defines a contract between a Jakarta EE application server and an authorization policy provider. In a full Jakarta EE product, all Jakarta EE web containers and enterprise bean containers are required to support this contract.

The Jakarta Authorization specification can be found at *https://jakarta.ee/specifications/authorization* .

6.22. Jakarta Authentication 1.1 Requirements

The Jakarta Authentication specification defines a service provider interface (SPI) by which authentication providers implementing message authentication mechanisms may be integrated in client or server message processing containers or runtimes. Authentication providers integrated through this interface operate on network messages provided to them by their calling container. They transform outgoing messages such that the source of the message may be authenticated by the receiving container, and the recipient of the message may be authenticated by the message sender. They authenticate incoming messages and return to their calling container the identity established as a result of the message authentication.

In a full Jakarta EE product, all Jakarta EE web containers and enterprise bean containers are required to support the baseline compatibility requirements as defined by the Jakarta Authentication specification. In a full Jakarta EE product, all web containers must also support the Servlet Container Profile as defined in the Jakarta Authentication specification. In a Jakarta EE profile product that includes Servlet and Jakarta Authentication, all web containers must also support the Servlet Container Profile as defined in the Jakarta Authentication specification. Support for the Jakarta Authentication SOAP Profile is not required.

The Jakarta Authentication specification can be found at *https://jakarta.ee/specifications/authentication* .

6.23. Jakarta Security 1.0 Requirements

Jakarta Security leverages Jakarta Authentication , but provides an easier to use SPI for authentication of users of web applications and defines identity store APIs for authentication and authorization.

In a full Jakarta EE product, all Jakarta EE web containers and enterprise bean containers are required to support the requirements defined by the Jakarta Security specification.

The Jakarta Security Specification can be found at *https://jakarta.ee/specifications/security* .

6.24. Jakarta Debugging Support for Other Languages Requirements 1.0

Jakarta Server Pages pages are usually translated into Java language pages and then compiled to create class files. The Jakarta Debugging Support for Other Languages specification describes information that can be included in a class file to relate class file data to data in the original source file. All Jakarta EE products are required to be able to include such information in class files that are generated from Jakarta Server Pages.

The Jakarta Debugging Support for Other Languages specification can be found at *https://jakarta.ee/specifications/debugging* .

6.25. Jakarta Standard Tag Library for Jakarta Server Pages 1.2 Requirements

Jakarta Standard Tag Library specification defines a standard tag library that makes it easier to develop Jakarta Server Pages Pages. All Jakarta EE products are required to provide a Jakarta Standard Tag Library for use by all Jakarta Server Pages.

The Jakarta Standard Tag Library for Jakarta Server Pages specification can be found at *https://jakarta.ee/specifications/tags* .

6.26. Jakarta Server Faces 2.3 Requirements

Jakarta Server Faces technology simplifies building user interfaces for Jakarta applications. Developers of various skill levels can quickly build web applications by: assembling reusable UI components in a page; connecting these components to an application data source; and wiring client-generated events to server-side event handlers. In a full Jakarta EE product, all Jakarta EE web containers are required to support applications that use the Jakarta Server Faces technology.

The Jakarta Server Faces specification can be found at *https://jakarta.ee/specifications/faces* .

6.27. Jakarta Annotations 1.3 Requirements

The Jakarta Annotations specification defines Java language annotations that are used by several other specifications, including this specification. The specifications that use these annotations fully define the requirements for these annotations. The applet container need not support any of these annotations. All other containers must provide definitions for all of these annotations, and must support the semantics of these annotations as described in the corresponding specifications and summarized in the following table.

Table 7. Common Annotations Support by Container

Annotation	App Client	Web	Enterprise Beans
Resource	Y	Y	Y
Resources	Y	Y	Y
PostConstruct	Y	Y	Y
PreDestroy	Y	Y	Y
Generated	N	N	N
RunAs	N	Y	Y
DeclareRoles	N	Y	Y
RolesAllowed	N	Y	Y
PermitAll	N	Y	Y

Annotation	App Client	Web	Enterprise Beans
DenyAll	N	Y	Y
ManagedBean	Y	Y	Y
DataSourceDefinition	Y	Y	Y
DataSourceDefinitions	Y	Y	Y
Priority	Y	Y	Y

The Jakarta Annotations specification can be found at *https://jakarta.ee/specifications/annotations* .

6.28. Jakarta Persistence 2.2 Requirements

Jakarta Persistence is the standard API for the management of persistence and object/relational mapping. The Jakarta Persistence specification provides an object/relational mapping facility for application developers using a Java domain model to manage a relational database.

As mandated by the Jakarta Persistence specification, in a Jakarta EE environment the classes of the persistence unit should not be loaded by the application class loader or any of its parent class loaders until after the entity manager factory for the persistence unit has been created.

The Jakarta Persistence specification can be found at *https://jakarta.ee/specifications/persistence* .

6.29. Bean Validation 2.0 Requirements

The Bean Validation specification defines a metadata model and API for JavaBean validation. The default metadata source is annotations, with the ability to override and extend the metadata through the use of XML validation descriptors.

The Jakarta EE platform requires that web containers make an instance of *ValidatorFactory* available to Jakarta Server Faces implementations by storing it in a servlet context attribute named *javax.faces.validator.beanValidator.ValidatorFactory.*

The Jakarta EE platform also requires that an instance of *ValidatorFactory* be made available to Jakarta Persistence providers as a property in the map that is passed as the second argument to the *createContainerEntityManagerFactory(PersistenceUnitInfo, Map)* method of the *PersistenceProvider* interface, under the name *javax.persistence.validation.factory* .

Additional requirements on Jakarta EE platform containers are specified in the Bean Validation specification, which can be found at *https://jakarta.ee/specifications/bean-validation* .

6.30. Managed Beans 1.0 Requirements

The Managed Beans specification defines a lightweight component model that supports the basic

lifecycle model, resource injection facility and interceptor service present in the Jakarta EE platform.

The Managed Beans specification can be found at *https://jakarta.ee/specifications/managed-beans* .

6.31. Interceptors 1.2 Requirements

The Interceptors specification makes more generally available the interceptor facility originally defined as part of the Jakarta Enterprise Beans 3.0 specification.

The Interceptors specification can be found at *https://jakarta.ee/specifications/interceptors* .

6.32. Contexts and Dependency Injection for the Jakarta EE Platform 2.0 Requirements

The Contexts and Dependency Injection (CDI) specification defines a set of contextual services, provided by Jakarta EE containers, aimed at simplifying the creation of applications that use both web tier and business tier technologies.

The CDI specification can be found at *https://jakarta.ee/specifications/cdi* .

6.33. Dependency Injection for Java 1.0 Requirements

The Dependency Injection for Java (DI) specification defines a standard set of annotations (and one interface) for use on injectable classes.

In the Jakarta EE platform, support for Dependency Injection is mediated by CDI. See Support for Dependency Injection" for more detail.

The DI specification can be found at *https://jakarta.ee/specifications/dependency-injection* .

6.34. Enterprise Web Services 1.4 Requirements

The Enterprise Web Services specification defines the integration between the various Web Service technologies in Jakarta EE, including XML-RPC, XMl Web Services and XML Web Service Metadata.

The Enterprise Web Services specification can be found at *https://jakarta.ee/specifications/enterprise-ws* .

---

[14] Client APIs only.
[15] Jakarta™ Enterprise Beans entity beans and associated query language only.
[16] IIOP interoperability, including Jakarta™ Enterprise Beans 2.x and 1.x client view.
[17] See Jakarta™ Enterprise Edition Deployment API 1.7 Requirements (Optional) for details.
[18] Note that a component specification is permitted to specify an exception to this in order to

accommodate interface type dependencies—for example, the Jakarta™ Enterprise Beans SessionContext dependency on the *javax.xml.rpc.handler.MessageContext* type.

[19] The FilePermission * specifically refers to all files under the current directory.

[20] (For Web components only.) It must be possible to grant FilePermission for the tempdir provided to web components through the ServletContext regardless of its physical location. In addition, it must be possible to grant FilePermission for the tempdir without granting it for all files under the current directory.

[21] It must be possible to deploy an application module such that no instances of java.lang.RuntimePermission are granted to the components in the application module except those with a target of loadlibrary.* for any specific library or a target of queuePrintJob. Ideally a container would be capable of restricting those as well, but that is not a requirement.

[22] It must be possible to deploy an application module such that no instances of java.util.PropertyPermission are granted that allow writing any property.

Chapter 7. Interoperability

This chapter describes the interoperability requirements for the Jakarta™ EE Platform.

7.1. Introduction to Interoperability

The Jakarta EE platform will be used by enterprise environments that support clients of many different types. The enterprise environments will add new services to existing Enterprise Information Systems (EISs). They will be using various hardware platforms and applications written in various languages.

In particular, the Jakarta EE platform may be used in enterprise environments to bring together any of the following kinds of applications:

- applications written in such languages as C++ and Visual Basic.
- applications running on a personal computer platform, or Unix® workstation.
- standalone Java™ technology-based applications that are not directly supported by the Jakarta EE platform.

It is the interoperability requirements of the Jakarta EE platform, set out in this chapter, that make it possible for it to provide indirect support for various types of clients, different hardware platforms, and a multitude of software applications. The interoperability features of the Jakarta EE platform permit the underlying disparate systems to work together seamlessly, while hiding much of the complexity required to join these pieces together.

The interoperability requirements for the current Jakarta EE platform release allow:

- Jakarta EE applications to connect to legacy systems using CORBA or low-level socket interfaces.
- Jakarta EE applications to connect to other Java Jakarta applications across multiple Jakarta EE products, whether from different Product Providers or from the same Provider, and multiple Jakarta EE platforms.

In this version of the specification, interoperability between Jakarta EE applications running in different platforms is accomplished through the HTTP protocol, possibly using SSL, or the Jakarta Enterprise Beans interoperability protocol based on IIOP.

7.2. Interoperability Protocols

This specification requires that a Jakarta EE product support a standard set of protocols and formats to ensure interoperability between Jakarta EE applications and with other applications that also implement these protocols and formats. The specification requires support for the following groups of protocols and formats:

- Internet and web protocols

- OMG protocols
- Java technology protocols
- Data formats

Most of these protocols and formats are supported by Java SE and by the underlying operating system.

7.2.1. Internet and Web Protocols

Standards based Internet protocols are the means by which different pieces of the platform communicate. The Jakarta EE platform typically supports the following Internet protocols, as described in the corresponding technology specifications:

- 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. TCP/IP is supported by Java SE and the underlying operating system.
- HTTP 1.1—This is the core protocol of web communication. As with TCP/IP, HTTP 1.1 is supported by Java SE and the underlying operating system. A Jakarta EE web container must be capable of advertising its HTTP services on the standard HTTP port, port 80.
- HTTP/2—Server-side support for the HTTP/2 protocol is required by the Servlet specification.
- SSL 3.0, TLS 1.2—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 Jakarta EE web container must be capable of advertising its HTTPS service on the standard HTTPS port, port 443. Note that SSL 3.0 and only TLS 1.0 are required as part of the Jakarta Enterprise Beans interoperability protocol in the Jakarta Enterprise Beans specification.
- SOAP 1.1—SOAP is a presentation layer protocol for the exchange of XML messages. Support for SOAP layered on HTTP is required, as described in the Jakarta XML-based RPC and Jakarta XML Web Services specifications.
- SOAP 1.2—SOAP 1.2 is the version of the SOAP protocol standardized through W3C and supported by Jakarta XML Web Services.
- WS-I Basic Profile 1.1—The WS-I Basic Profile, in combination with the Simple SOAP Binding Profile and Attachment Profile, describes interoperability requirements for the use of SOAP 1.1, WSDL 1.1, and MIME-based SOAP with Attachments. It is required by the Jakarta XML-based RPC and Jakarta XML Web Services specifications.
- WebSocket protocol—The WebSocket protocol enables two-way communication layered over TCP. It enables bi-directional communication over a single connection established by an initial HTTP handshake and upgrade request. The WebSocket protocol has been standardized by IETF under RFC 6455.

7.2.2. OMG Protocols (Proposed Optional)

This specification requires the a full Jakarta EE product to support the following Object Management Group (OMG) based protocols:

- IIOP (Internet Inter-ORB Protocol)—Supported by Java IDL and RMI-IIOP in Java SE. 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://omg.org/cgi-bin/doc?formal/99-10-07* .
- Jakarta Enterprise Beans interoperability protocol—The Jakarta Enterprise Beans interoperability protocol is based on IIOP (GIOP 1.2) and the CSIv2 CORBA Secure Interoperability specification. The Jakarta Enterprise Beans interoperability protocol is defined in the Jakarta Enterprise Beans specification.
- CORBA Interoperable Naming Service protocol—The COSNaming-based INS protocol is an IIOP-based protocol for accessing a name service. The Jakarta Enterprise Beans interoperability protocol requires the use of the INS protocol for lookup of Jakarta Enterprise Beans objects using the JNDI API. In addition, it must be possible to use the Java IDL COSNaming API to access the INS name service. All Jakarta EE products must provide a name service that meets the requirements of the Interoperable Naming Service specification, available at *http://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. Either approach is consistent with this specification.

7.2.3. Java Technology Protocols

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

JRMP is a distributed object model for the Java programming language. Distributed systems, running in different address spaces and often on different hosts, must 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://docs.oracle.com/javase/8/docs/technotes/guides/rmi* .

7.2.4. Data Formats

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

The following data formats must be supported:

- XML 1.0—The XML format can be used to construct documents, RPC messages, etc. The JAXP API provides support for processing XML format data. The Jakarta XML-based RPC API provides support for XML RPC messages, as well as a mapping between Java classes and XML.

- JSON—JSON is a language-neutral plain text format commonly used to transfer structured data between a server and web application. The Jakarta JSON Processing API provides support for the parsing, generation, transformation, and querying of JSON text. The Jakarta JSON Binding API provides support for mapping between JSON text and Java objects.
- HTML 4.01—This represents the minimum web browser standard document format. While all Jakarta EE APIs with the exception of Jakarta Server Faces are agnostic to the version of the browser document format, Jakarta EE web clients must be able to display HTML 4.01 documents.
- Image file formats—The Jakarta EE platform must support GIF, JPEG, and PNG images. Support for these formats is provided by the *java.awt.image* APIs (see the URL: *http://docs.oracle.com/javase/8/docs/api/java/awt/image/package-summary.html*) and by Jakarta EE 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 Jakarta EE 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 *http://docs.oracle.com/javase/8/docs/technotes/guides/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 *http://docs.oracle.com/javase/specs/* .

Chapter 8. Application Assembly and Deployment

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

Jakarta EE applications are composed of one or more Jakarta EE components and an optional Jakarta EE application deployment descriptor. The deployment descriptor, if present, lists the application's components as *module* s. If the deployment descriptor is not present, the application's modules are discovered using default naming rules. A Jakarta EE module represents the basic unit of composition of a Jakarta EE application. Jakarta EE modules consist of one or more Jakarta EE components and an optional module level deployment descriptor. The flexibility and extensibility of the Jakarta EE component model facilitates the packaging and deployment of Jakarta EE components as individual components, component libraries, or Jakarta EE applications.

A full Jakarta EE product must support all the facilities described in this chapter. A Jakarta EE profile may support only a subset of the Jakarta EE module types. Any requirements related to a module type not supported by a product based on a particular Jakarta EE profile should be understood to not apply to such a product.

Jakarta EE Deployment shows the composition model for Jakarta EE deployment units and includes the optional use of alternate deployment descriptors by the application package to preserve any digital signatures of the original Jakarta EE modules. An alternate deployment descriptor may also be provided external to the application package as described in Assembling a Jakarta EE Application."

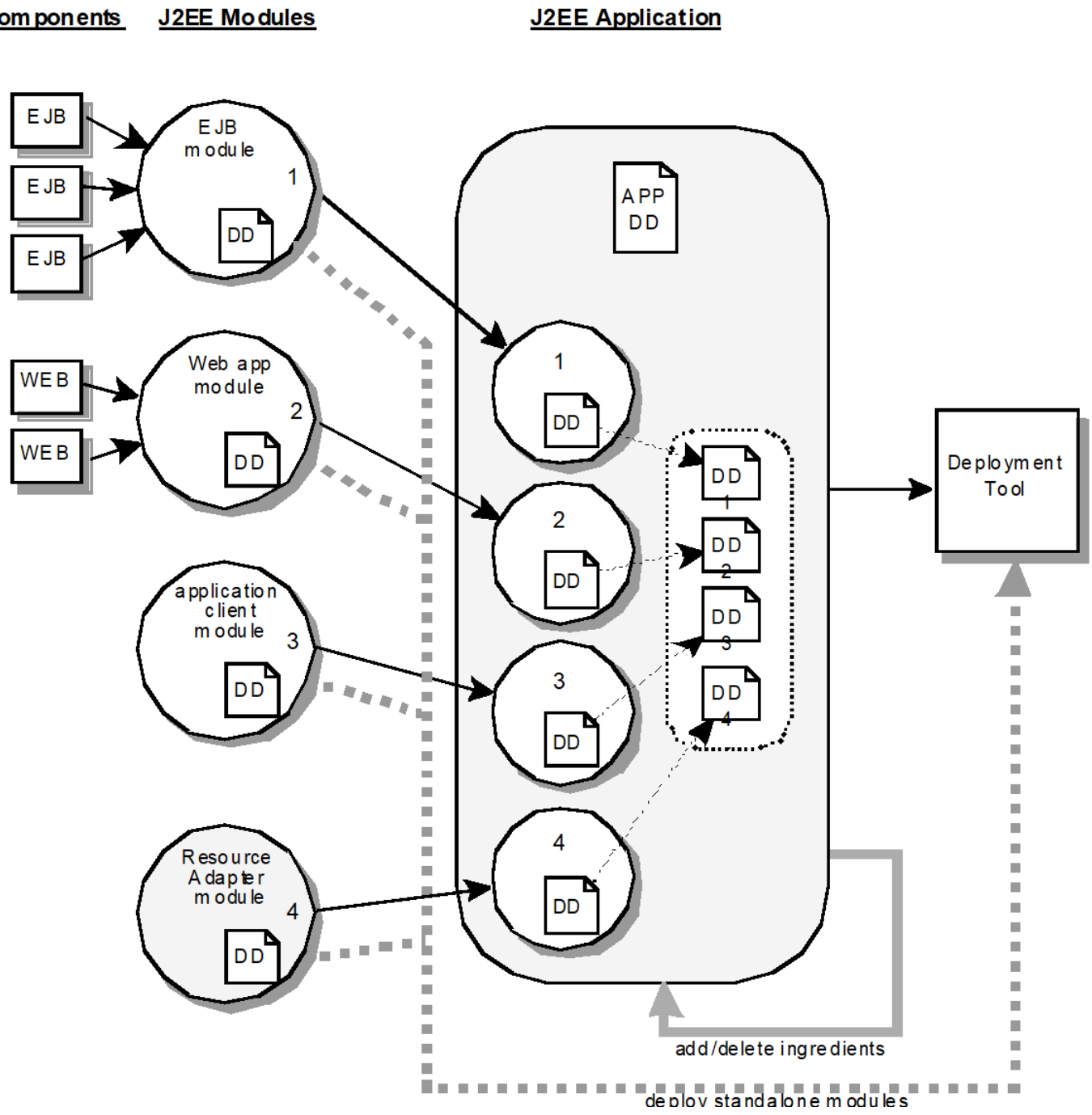

Figure 9. Jakarta EE Deployment

8.1. Application Development Life Cycle

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

Jakarta EE Application Life Cycle shows the life cycle of a Jakarta EE application.

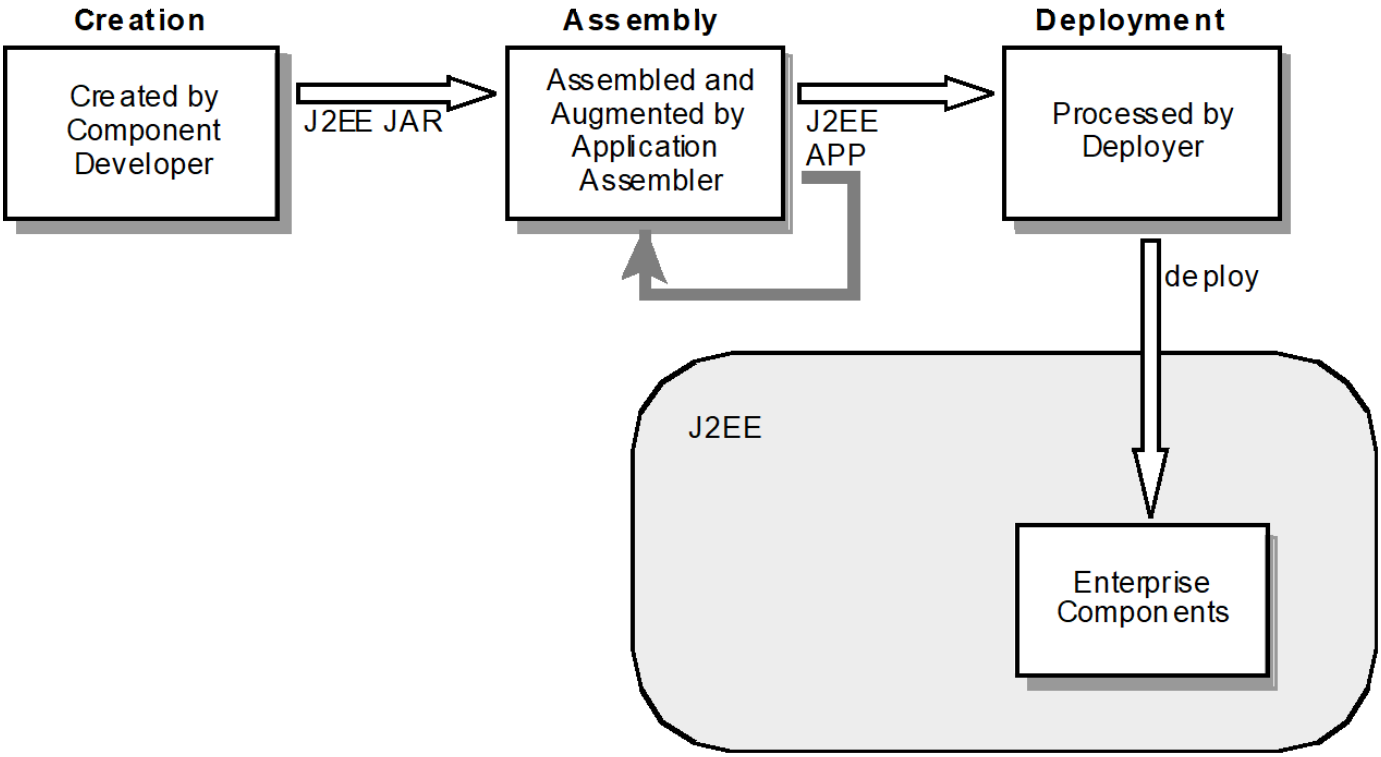

Figure 10. Jakarta EE Application Life Cycle

8.1.1. Component Creation

The Jakarta Enterprise Beans, Jakarta Servlet, application client, and Jakarta Connector specifications include the XML Schema definition of the associated module level deployment descriptors and component packaging architecture required to produce Jakarta EE modules. (The application client specification is found in CHApter of this document.)

A Jakarta EE module is a collection of one or more Jakarta EE components (web, Jakarta Enterprise Beans, application client, or Jakarta Connector) with an optional module deployment descriptor of that type. Any number of components of the same container type can be packaged together with a single Jakarta EE deployment descriptor appropriate to that container type to produce a Jakarta EE module. Components of different container types may not be mixed in a single Jakarta EE module, except for the packaging of Jakarta Enterprise Beans components within a web module.

- A Jakarta EE module represents the basic unit of composition of a Jakarta EE application. In some cases a single Jakarta EE module (not necessarily packaged into a Jakarta EE application package) will contain an entire application. In other cases an application will be composed of multiple Jakarta EE modules.
- The deployment descriptor for a Jakarta EE module contains declarative data required to deploy the components in the module. The deployment descriptor for a Jakarta EE module also contains assembly instructions that describe how the components are composed into an application.
- Starting with version 5 of the Java EE, a web application module, an enterprise bean module, or an application client module need not contain a deployment descriptor. Instead, the deployment information may be specified by annotations present in the class files of the module.

- Starting with version 5 of the Java EE, a Jakarta EE enterprise application archive need not contain a deployment descriptor. Instead, the deployment information may be determined using default naming rules for embedded modules.
- An individual Jakarta EE module can be deployed as a stand-alone Jakarta EE module without an application level deployment descriptor and represents a valid Jakarta EE application.
- Jakarta EE modules may express dependencies on libraries as described below in Library Support."

All Jakarta EE modules have a name. The name can be explicitly set in the deployment descriptor for the module. If not set, the name of the module is the pathname of the module in the ear file with any filename extension (.jar, .war, .rar) removed, but with any directory names included. The name of a module must be unique within an application. If and only if the name is not unique (e.g., because two names are identical after removing different filename extensions) the deployment tool may choose new unique names for any of the conflicting modules; module names that do not conflict must not be changed. The algorithm for choosing unique names in such a case is product specific. Applications that depend on the names of their modules must ensure that their module names are unique.

For example, an application with this structure:

```
myapp.ear
 inventory.jar
 ui.war
```

has a default application name of "myapp", and defines two modules with default names "inventory" and "ui".

An application with this structure:

```
bigapp.ear
 ejbs
   inventory.jar
   accounts.jar
 ui
   store.war
   admin.war
```

has a default application name of "bigapp", and defines four modules with default names "ejbs/inventory", "ejbs/accounts", "ui/store", and "ui/admin".

8.1.2. Application Assembly

A Jakarta EE application may consist of one or more Jakarta EE modules and one Jakarta EE application deployment descriptor. A Jakarta EE application is packaged using the Jakarta Archive (JAR) file format into a file with a *.ear* (Enterprise ARchive) filename extension. A minimal Jakarta EE

application package will only contain Jakarta EE modules and, optionally, the application deployment descriptor. A Jakarta EE application package may also include libraries referenced by Jakarta EE modules (using the *Class-Path* mechanism described below in Library Support"), help files, and documentation to aid the deployer.

The deployment of a portable Jakarta EE 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 Jakarta EE application must be possible using only the application deployment descriptor, if any, and the Jakarta EE modules (and their dependent libraries) and descriptors listed in it.

The Jakarta EE application deployment descriptor represents the top level view of a Jakarta EE application's contents. The Jakarta EE application deployment descriptor is specified by an XML schema or document type definition (see Jakarta EE Application XML Schema").

In certain cases, a Jakarta EE application will need customization before it can be deployed into the enterprise. New Jakarta EE modules may be added to the application. Existing modules may be removed from the application. Some Jakarta EE 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 Jakarta EE web application.

All Jakarta EE applications have a name. The name can be explicitly set in the application deployment descriptor. If not set, the name of the application is the base name of the ear file with any *.ear* extension removed and with any directory names removed. The name of an application must be unique in an application server instance. If an attempt is made to deploy an application with a name that conflicts with an already deployed application, the deployment tool may choose a new unique name for the application. The deployment tool may also allow a different name to be specified at deployment time. A deployment tool may use product-specific means to decide whether a deployment operation is a deployment of a new application, in which case the name must be unique, or a redeployment of an existing application, in which case the name may match the existing application.

Similarly, when a stand-alone module is deployed, the module name is used as the application name, and obeys the same rules as described above for application names. The module name can be explicitly set in the module deployment descriptor. If not set, the name of the module is the base name of the module file with any extension (*.war* , *.jar* , *.rar*) removed and with any directory names removed.

8.1.3. Deployment

During the deployment phase of an application's life cycle, the application is installed on the Jakarta EE platform and then is configured and integrated into the existing infrastructure. Each Jakarta EE module listed in the application deployment descriptor (or discovered using the default rules described below) must be deployed according to the requirements of the specification for the respective Jakarta EE 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.

Every resource reference should be bound to a resource of the required type.

Some resources have default mapping rules specified; see sections Default Data Source, Default JMS Connection Factory, and Default Concurrency Utilities Objects. By default, a product must map otherwise unmapped resources using these default rules. A product may include an option to disable or override these default mapping rules.

Once a resource reference is bound to a resource in the target operational environment, and deployment succeeds, that binding is not expected to change. A product may provide administrative operations that change the resource bindings that are used by applications. A product may notify applications of changes to their resource bindings using JNDI events, but this is not required.

If deployment succeeds, in addition to binding resource references as specified above, every resource definition (see section Resource Definition and Configuration) specified by the application or specified or overridden by the Deployer must be present in the target operational environment.

8.2. Library Support

The Jakarta EE provides several mechanisms for applications to use optional packages and shared libraries (hereafter referred to as *libraries*). Libraries may be bundled with an application or may be installed separately for use by any application.

Jakarta EE products are required to support the use of bundled and installed libraries as specified in the *Extension Mechanism Architecture* and *Optional Package Versioning* specifications (available at *http://docs.oracle.com/javase/8/docs/technotes/guides/extensions/*) and the JAR File Specification (available at *http://docs.oracle.com/javase/8/docs/technotes/guides/jar/jar.html*). Using this mechanism a Jakarta EE JAR file can reference utility classes or other shared classes or resources packaged in a separate *.jar* file or directory that is included in the same Jakarta EE application package, or that has been previously installed in the Jakarta EE containers.

8.2.1. Bundled Libraries

Libraries bundled with an application may be referenced in the following ways:

1. A JAR format file (such as a *.jar* file, *.war* file, or *.rar* file) may reference a *.jar* file or directory by naming the referenced *.jar* file or directory in a *Class-Path* header in the referencing JAR file's Manifest file. The referenced *.jar* file or directory is named using a URL relative to the URL of the referencing JAR file. The Manifest file is named *META-INF/MANIFEST.MF* in the JAR file. The *Class-Path* entry in the Manifest file is of the form

Class-Path: list-of-jar-files-or-directories-separated-by-spaces

(See the JAR File Specification for important details and limitations of the syntax of *Class-Path* headers.) The Jakarta EE deployment tools must process all such referenced files and directories when processing a Jakarta EE module. Any deployment descriptors in referenced *.jar* files must be ignored when processing the referencing *.jar* file. The deployment tool must install the *.jar* files and directories

in a way that preserves the relative references between the files. Typically this is done by installing the *.jar* files into a directory hierarchy that matches the original application directory hierarchy. All referenced *.jar* files or directories must appear in the logical class path of the referencing JAR files at runtime.

Only JAR format files or directories containing class files or resources to be loaded directly by a standard class loader should be the target of a *Class-Path* reference; such files are always named with a *.jar* extension. Top level JAR files that are processed by a deployment tool should not contain *Class-Path* entries; such entries would, by definition, reference other files external to the deployment unit. A deployment tool is not required to process such external references.

1. A *.ear* file may contain a directory that contains libraries packaged in JAR files. The *library-directory* element of the *.ear* file's deployment descriptor contains the name of this directory. If a *library-directory* element isn't specified, or if the *.ear* file does not contain a deployment descriptor, the directory named *lib* is used. An empty *library-directory* element may be used to specify that there is no library directory.

All files in this directory (but not subdirectories) with a *.jar* extension must be made available to all components packaged in the EAR file, including application clients. These libraries may reference other libraries, either bundled with the application or installed separately, using any of the techniques described herein.

1. A web application may include libraries in the *WEB-INF/lib* directory. See the Jakarta Servlet specification for details. These libraries may reference other libraries, either bundled with the application or installed separately, using any of the techniques described herein.

8.2.2. Installed Libraries

Libraries that have been installed separately may be referenced in the following way:

1. JAR format files of all types may contain an *Extension-List* attribute in their Manifest file, indicating a dependency on an installed library. The JAR File Specification defines the semantics of such attributes for use by applets; this specification requires support for such attributes for all component types and corresponding JAR format files. The deployment tool is required to check such dependency information and reject the deployment of any component for which the dependency can not be met. Portable applications should not assume that any installed libraries will be available to a component unless the component's JAR format file, or one of the containing JAR format files, expresses a dependency on the library using the *Extension-List* and related attributes.

The referenced libraries must be made available to all components contained within the referencing file, including any components contained within other JAR format files within the referencing file. For example, if a *.ear* file references an installed library, the library must be made available to all components in all *.war* files, Jakarta Enterprise Beans *.jar* files, application *.jar* files, and resource adapter *.rar* files within the *.ear* file.

A Jakarta EE product is not required to support downloading of libraries (using the *<extension>-Implementation-URL* header) at deployment time or runtime. A Jakarta EE product is also not required to support more than a single version of an installed library at once. A Jakarta EE product is not required to limit access to installed libraries to only those for which the application has expressed a dependency; the application may be given access to more installed libraries than it has requested. In all of these cases, such support is highly recommended and may be required in a future version of this specification. In particular, we recommend that a Jakarta EE product support multiple versions of an installed library, and by default only allow applications to access the installed libraries for which they have expressed a dependency.

8.2.3. Library Conflicts

If an application includes a bundled version of a library, and the same library exists as an installed library, the instance of the library bundled with the application should be used in preference to any installed version of the library. This allows an application to bundle exactly the version of a library it requires without being influenced by any installed libraries. Note that if the library is also a required component of the Jakarta EE version on which the application is being deployed, this version may (and typically will) take precedence.

8.2.4. Library Resources

In addition to allowing access to referenced classes, as described above, any resources contained in the referenced JAR files must also be accessible using the *Class* and *ClassLoader getResource* methods, as allowed by the security permissions of the application. An application will typically have the security permissions required to access resources in any of the JAR files packaged with the application.

8.2.5. Dynamic Class Loading

Libraries that dynamically load classes must consider the class loading environment of a Jakarta EE application. Libraries will often be loaded by a class loader that is a parent class loader of the class loader that is used to load application classes and thus will not have direct visibility to classes of the application modules. A library that only needs to dynamically load classes provided by the library itself can safely use the *Class* method *forName* . However, libraries that need to dynamically load classes that have been provided as a part of the application need to use the context class loader to load the classes. Note that the context class loader may be different in each module of an application. Access to the context class loader requires *RuntimePermission* (“ *getClassLoader”)* , which is not normally granted to applications, but should be granted to libraries that need to dynamically load classes. Libraries can use a method such as the following to assert their privilege when accessing the context class loader. This technique will work in both Java SE and Jakarta EE.

```
public ClassLoader getContextClassLoader() {
  return AccessController.doPrivileged(
    new PrivilegedAction<ClassLoader>() {
      public ClassLoader run() {
        ClassLoader cl = null;
        try {
          cl = Thread.currentThread().
                        getContextClassLoader();
        } catch (SecurityException ex) { }
        return cl;
      }
    });
}
```

Libraries should then use the following technique to load classes.

```
ClassLoader cl = getContextClassLoader();
if (cl != null) {
  try {
    clazz = Class.forName(name, false, cl);
  } catch (ClassNotFoundException ex) {
    clazz = Class.forName(name);
  }
 } else
  clazz = Class.forName(name);
```

8.2.6. Examples

The following example illustrates a simple use of the bundled library mechanism to reference a library of utility classes that are shared between enterprise beans in two separate ejb-jar files.

```
app1.ear:
 META-INF/application.xml
 ejb1.jar      Class-Path: util.jar
 ejb2.jar      Class-Path: util.jar
 util.jar
```

The next example illustrates a more complex use of the *Class-Path* mechanism. In this example the Developer has chosen to package the enterprise bean client view classes in a separate JAR file and reference that JAR file from the other JAR files that need those classes. Those classes are needed both by *ejb2.jar* , packaged in the same application as *ejb1.jar* , and by *ejb3.jar* and *jakartaservlet1.jar* , packaged in a different application. Those classes are also needed by *ejb1.jar* itself because they define the remote interface of the enterprise beans in *ejb1.jar* , and the developer has chosen the by reference

model of making these classes available, as described in the Jakarta Enterprise Beans spec. The deployment descriptor for *ejb1.jar* names the client view JAR file in the *ejb-client-jar* element.

The *Class-Path* mechanism must be used by components in *app3.ear* to reference the client view JAR file that corresponds to the enterprise beans packaged in *ejb1.jar* of *app2.ear* . These enterprise beans are referenced by enterprise beans in *ejb3.jar* and by the Jakarta Servlets packaged in *webapp.war* .

```
app2.ear:
  META-INF/application.xml
  ejb1.jar      Class-Path: ejb1_client.jar
    deployment descriptor contains:
        <ejb-client-jar>ejb1_client.jar</ejb-client-jar>
  ejb1_client.jar
  ejb2.jar      Class-Path: ejb1_client.jar

app3.ear:
  META-INF/application.xml
  ejb1_client.jar
  ejb3.jar      Class-Path: ejb1_client.jar
  webapp.war    Class-Path: ejb1_client.jar
     WEB-INF/web.xml
     WEB-INF/lib/jakartaservlet1.jar
```

The following example illustrates a simple use of the installed library mechanism to reference a library of utility classes that is installed separately.

```
app1.ear:
  META-INF/application.xml
  ejb1.jar :
    META-INF/MANIFEST.MF:
      Extension-List: util
      util-Extension-Name: com/example/util
      util-Specification-Version: 1.4
    META-INF/ejb-jar.xml

util.jar:
  META-INF/MANIFEST.MF:
  Extension-Name: com/example/util
  Specification-Title: example.com's util package
  Specification-Version: 1.4
  Specification-Vendor: example.com
  Implementation-Version: build96
```

8.3. Class Loading Requirements

The Jakarta EE specification purposely does not define the exact types and arrangements of class loaders that must be used by a Jakarta EE product. Instead, the specification defines requirements in terms of what classes must or must not be visible to components. A Jakarta EE product is free to use whatever class loaders it chooses to meet these requirements. Portable applications must not depend on the

types of class loaders used or the hierarchical arrangement of class loaders, if any. Portable applications must not depend on the order in which classes and resources are loaded. Applications should use the techniques described in Dynamic Class Loading" if they need to load classes dynamically.

In addition to the required classes specified below, a Jakarta EE product must provide a way to allow an application to access a class library installed in the application server, even if it has not expressed a dependency on that library. This supports the use of old applications and extension libraries that do not use the defined extension dependency mechanism.

The following sections describe the requirements for each container type. In all cases, access to classes is governed by the rules of the Java language and the Java virtual machine. In all cases, access to classes and resources is governed by the rules of the Java security model.

Note that while libraries must be accessible to application classes as described below, it may be necessary to use the techniques described in Dynamic Class Loading" if libraries need to access classes packaged in the application modules.

8.3.1. Web Container Class Loading Requirements

Components in the web container must have access to the following classes and resources. Note that as of Java EE 6, Java Enterprise Beans components may be packaged in a web component module. Such Java Enterprise Beans components have the same access as other components in the web container. See the Jakarta Enterprise Beans specification for further detail.

- The content of the *WEB-INF/classes* directory of the containing war file.
- The content of all jar files in the *WEB-INF/lib* directory of the containing war file, but not any subdirectories.
- The transitive closure of any libraries referenced by the above jar files (as specified in Library Support").
- The transitive closure of any libraries referenced by the war file itself (as specified in Library Support").
- The transitive closure of any libraries specified by or referenced by the containing ear file (as specified in Library Support").
- The contents of all jar files included in any resource adapter archives (rar files) included in the same ear file.

- The contents of all jar files included in each resource adapter archive (rar file) deployed separately to the application server, if that resource adapter is used to satisfy any resource references in the module.
- The contents of all jar files included in each resource adapter archive (rar file) deployed separately to the application server, if any jar file in that rar file is used to satisfy any reference from the module using the Extension Mechanism Architecture (as specified in Library Support").
- The transitive closure of any libraries referenced by the jar files in the rar files above (as specified in Library Support").
- The transitive closure of any libraries referenced by the rar files themselves (as specified in Library Support").
- The Jakarta EE API classes specified in Jakarta EE Technologies for the web container.
- All Java SE 8 API classes.

Components in the web container may have access to the following classes and resources. Portable applications must not depend on having or not having access to these classes or resources.

- The classes and resources accessible to any other web modules included in the same ear file, as described above.
- The content of any Jakarta Enterprise Beans jar files included in the same ear file.
- The content of any client jar files specified by the above Jakarta Enterprise Beans jar files.
- The transitive closure of any libraries referenced by the above Jakarta Enterprise Beans jar files and client jar files (as specified in Library Support").
- The contents of any jar files included in any resource adapter archives (rar files) deployed separately to the application server.
- The transitive closure of any libraries referenced by the jar files in the rar files above (as specified in Library Support").
- The transitive closure of any libraries referenced by the rar files above themselves (as specified in Library Support").
- The Jakarta EE API classes specified in Jakarta EE Technologies for the containers other than the web container.
- Any installed libraries available in the application server.
- Other classes or resources contained in the application package, and specified by an explicit use of an extension not defined by this specification.
- Other classes and resources that are part of the implementation of the application server.

Components in the web container must not have access to the following classes and resources, unless such classes or resources are covered by one of the rules above.

- Other classes or resources in the application package. For example, the application should not have access to the classes in application client jar files.

8.3.2. Jakarta Enterprise Beans Container Class Loading Requirements

Components in the Jakarta Enterprise Beans container must have access to the following classes and resources.

- The content of the Jakarta Enterprise Beans jar file.
- The transitive closure of any libraries referenced by the Jakarta Enterprise Beans jar file (as specified in Library Support").
- The transitive closure of any libraries specified by or referenced by the containing ear file (as specified in Library Support").
- The contents of all jar files included in any resource adapter archives (rar files) included in the same ear file.
- The contents of all jar files included in each resource adapter archive (rar file) deployed separately to the application server, if that resource adapter is used to satisfy any resource references in the module.
- The contents of all jar files included in each resource adapter archive (rar file) deployed separately to the application server, if any jar file in that rar file is used to satisfy any reference from the module using the Extension Mechanism Architecture (as specified in Library Support").
- The transitive closure of any libraries referenced by the jar files in the rar files above (as specified in Library Support".
- The transitive closure of any libraries referenced by the rar files themselves (as specified in Library Support").
- The Jakarta EE API classes specified in Jakarta EE Technologies for the Jakarta Enterprise Beans container.
- All Java SE 8 API classes.

Components in the Jakarta Enterprise Beans container may have access to the following classes and resources. Portable applications must not depend on having or not having access to these classes or resources.

- The classes and resources accessible to any web modules included in the same ear file, as described in Web Container Class Loading Requirements" above.
- The content of any Jakarta Enterprise Beans jar files included in the same ear file.
- The content of any client jar files specified by the above Jakarta Enterprise Beans jar files.
- The transitive closure of any libraries referenced by the above Jakarta Enterprise Beans jar files and client jar files (as specified in Library Support").
- The contents of any jar files included in any resource adapter archives (rar files) deployed separately to the application server.
- The transitive closure of any libraries referenced by the jar files in the rar files above (as specified in Library Support").

- The transitive closure of any libraries referenced by the rar files above themselves (as specified in Library Support").
- The Jakarta EE API classes specified in Jakarta EE Technologies for the containers other than the Jakarta Enterprise Beans container.
- Any installed libraries available in the application server.
- Other classes or resources contained in the application package, and specified by an explicit use of an extension not defined by this specification.
- Other classes and resources that are part of the implementation of the application server.

Components in the Jakarta Enterprise Beans container must not have access to the following classes and resources, unless such classes or resources are covered by one of the

rules above.

- Other classes or resources in the application package. For example, the application should not have access to the classes in application client jar files.

8.3.3. Application Client Container Class Loading Requirements

Components in the application client container must have access to the following classes and resources.

- The content of the application client jar file.
- The transitive closure of any libraries referenced by the above jar file (as specified in Library Support").
- The transitive closure of any libraries specified by or referenced by the containing ear file (as specified in Library Support").
- The Jakarta EE API classes specified in Jakarta EE Technologies for the application client container.
- All Java SE 8 API classes.

Components in the application client container may have access to the following classes and resources. Portable applications must not depend on having or not having access to these classes or resources.

- The Jakarta EE API classes specified in Jakarta EE Technologies for the containers other than the application client container.
- Any installed libraries available in the application server.
- Other classes or resources contained in the application package, and specified by an explicit use of an extension not defined by this specification.
- Other classes and resources that are part of the implementation of the application server.

Components in the application client container must not have access to the following classes and resources, unless such classes or resources are covered by one of the rules above.

- Other classes or resources in the application package. For example, the application client should not have access to the classes in other application client jar files in the same ear file, nor should it have access to the classes in web applications or Jakarta Enterprise Beans jar files in the same ear file.

8.3.4. Applet Container Class Loading Requirements

The requirements for the applet container are completely specified by the Java SE 8 specification. This specification adds no new requirements for the applet container.

8.4. Application Assembly

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

8.4.1. Assembling a Jakarta EE Application

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

The directory structure of an application is arbitrary, but by following some simple conventions a deployment descriptor may not be needed. The structure should be designed around the requirements of the contained components.

1. Reconcile Jakarta EE module deployment descriptors.

The deployment descriptors for the Jakarta EE modules must be edited to link internally satisfied dependencies and eliminate any redundant security role names. An optional element *alt-dd* (described in Jakarta EE Application XML Schema") 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 module package.

1. Choose unique names for the modules contained in the application. If two modules specify conflicting names in their deployment descriptors, create an alternate deployment descriptor for at least one of the modules and change its name. If two modules in the same directory of the ear file have the same base name (e.g., *foo.jar* and *foo.war*), rename one of the modules or create an alternate deployment descriptor to specify a unique name for one of the modules.
2. 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.
3. For each *ejb-link* , there must be only one matching *ejb-name* in the scope of the entire application (see Enterprise JavaBeans™ (EJB) References").

4. 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.
5. 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 application-level deployment descriptor.
6. Assign a context root for each web module included in the Jakarta EE application. The context root is a relative name in the web namespace for the application. Each web module must be given a distinct and non-overlapping name for its context root. The web modules will be assigned a complete name in the namespace of the web server at deployment time. If there is only one web module in the Jakarta EE application, the context root may be the empty string. If no deployment descriptor is included in the application package, it will use the default-context-path in the web module. Otherwise, it will use the module name as the context root of the web module. See the Jakarta Servlet specification for detailed requirements of context root naming.
7. Make sure that each component in the application properly describes any dependencies it may have on other components in the application. A Jakarta EE application should not assume that all components in the application will be available on the class path 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 should reference the second JAR file using the *Class-Path* mechanism. A notable exception to this rule is JAR files located in the *WEB-INF/lib* directory of a web application. All such JAR files are included in the class path of the web application at runtime; explicit references to them using the *Class-Path* mechanism are not needed. Another exception to this rule is JAR files located in the library directory (usually named *lib*) in the application package. Note that the presence of component-declaring annotations in shared artifacts, such as libraries in the library directory and libraries referenced by more than one module through *Class-Path* references, can have unintended and undesirable consequences and is not recommended.
8. There must be only one version of each class in an application. If one component depends on one version of a library, and another component depends on another version, it may not be possible to deploy an application containing both components. With the exception of application clients, a Jakarta EE 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, however, that it must be possible to deploy an application such that all components of the application are in a namespace (or namespaces) separate from that of other applications. Typically, this will be the normal method of deployment. By default, application clients are each deployed into their own Java virtual machine instance, and thus each application client has its own class namespace, and the classes from application clients are not visible in the class namespace of other components.
9. (Optional) 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 XML schema for a Jakarta EE application XML document. (Alternatively, the deployment descriptor may meet the requirements of previous versions of Jakarta EE.)

Many applications that follow the conventions described below will not need a deployment descriptor for the application. The deployment tool will determine the components of the application using some simple rules.

1. Package the application.
2. Place the Jakarta EE modules and the deployment descriptor in the appropriate directories.
3. Package the application directory hierarchy in a file using the JAR file format. The file should be named with a *.ear* filename extension.
4. (Optional) Create an alternate deployment descriptor ("alt-dd") for the application, external to the packaged application.

8.4.2. Adding and Removing Modules

After the application is created, Jakarta EE 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 Jakarta EE modules that are being added to the application.
2. Ensure that the name of the new module does not conflict with any of the existing modules, either by choosing an appropriate default filename for the module or by explicitly specifying the module name in the module's deployment descriptor or in an alternate deployment descriptor.
3. Copy the new Jakarta EE modules to the desired location in the application package. The packaged modules are inserted directly in the desired location; the modules are not unpackaged.
4. Edit the deployment descriptors for the Jakarta EE modules to link the dependencies which are internally satisfied by the Jakarta EE modules included in the application.
5. Edit the Jakarta EE application deployment descriptor (if included) to meet the content requirements of the Jakarta EE and the validity requirements of the Jakarta EE application XML DTD or schema.

8.5. Deployment

The Jakarta EE supports three types of deployment units:

- Stand-alone Jakarta EE modules.
- Jakarta EE applications, consisting of one or more Jakarta EE modules.
- Class libraries packaged as *.jar* files according to the *Extension Mechanism Architecture* . These class libraries then become installed libraries.

Any Jakarta EE product must be able to accept a Jakarta EE application delivered as a *.ear* file or a stand-alone Jakarta EE module delivered as a *.jar* , *.war* , or *.rar* file (as appropriate to its type), together with an optional alternate deployment descriptor external to the application or standalone Jakarta EE module. If the application is delivered as a *.ear* , an enterprise bean module delivered as a *.jar* file, a web application delivered as a *.war* file, or an application client delivered as a *.jar* file, the deployment tool must be able to deploy the application such that the Jakarta classes in the application are in a separate namespace from classes in other Jakarta applications. Typically this will require the use of a separate class loader for each application. Standalone resource adapters delivered in *.rar* files and standalone class libraries delivered in *.jar* files that become installed libraries will of necessity appear in the class namespaces of applications that use them, and may appear in the class namespace of any application depending on the level of isolation supported by the Jakarta EE product.

As described in Jakarta EE Product Packaging", the Jakarta EE product might depend on external services to meet the requirements of this specification. While the Jakarta EE product is not required to assure the availability of these services, it is required to ensure that these services have been configured for use. Deployment of applications must fail if such required services have not been configured for use.

Deployment may provide an option that controls whether or not an application is attempted to be started during deployment. If no such option is provided or if the option to start the application is specified, and if deployment is successful, the application modules must be initialized as specified in section Module Initialization and the application must be started.

If the application is attempted to be started during deployment, the Jakarta Servlet and Jakarta Enterprise Beans containers must be initialized during deployment. Such initialization must include CDI initialization. If initialization fails, deployment must fail.

If the application is not attempted to be started during deployment, these containers must not be initialized during deployment.

In all cases, the deployment and initialization of a Jakarta EE application must be complete before the container delivers client requests to any of the application's components. The container must first initialize all startup-time singleton session bean components before delivering any requests to enterprise bean components. Containers must deliver requests to web components and resource adapters only after initialization of the component has completed.

The optional Jakarta EE Deployment API describes how a product-independent deployment tool accepts plugins for a specific Jakarta EE product, and how the tool and those plugins cooperate to deploy Jakarta EE applications. The requirements in this specification that refer to a deployment tool are meant to refer to the combination of any vendor-provided product-independent deployment tool and the vendor-specific deployment plugin for this tool, as well as any other vendor-specific deployment tools provided with the Jakarta EE product.

Typically a deployment tool will copy the deployed application or module to a product-specific location, along with the configuration settings and customizations specified by the Deployer. In some cases a deployment tool might include Application Assembly functionality as well, allowing the

Deployer to construct, modify, or customize the application before deployment. Still, it must be possible to deploy a portable Jakarta EE application, module, or library containing no product-specific deployment information without modifying the original files or artifacts that the Deployer specified to the deployment tool.

The deployment tools for Jakarta EE containers must validate the deployment descriptors against the Jakarta EE deployment descriptor schemas or DTDs that correspond to the deployment descriptors being processed. The appropriate schema or DTD is chosen by analyzing the deployment descriptor to determine which version it claims to conform to. Validation errors must cause an error to be reported to the Deployer. The deployment tool may allow the Deployer to correct the error and continue deployment. Note that the deployment descriptor version refers only to the version of the XML schema or DTD against which the descriptor is to be validated. It does not provide any information as to what version of the Jakarta EE the application is written to.

Some deployment descriptors are optional. The required deployment information is determined by using default rules or by annotations present on application class files. Some deployment descriptors that are included in an application may exist in either complete or incomplete form. A complete deployment descriptor provides a complete description of the deployment information; a deployment tool must not examine class files for this deployment information. An incomplete deployment descriptor provides only a subset of the required deployment information; a deployment tool must examine the application class files for annotations that specify deployment information.

If annotations are being processed (as required by Deployment Descriptor Processing Requirements, Jakarta Servlet Table 8-1, and Jakarta Enterprise Beans Tables 16 and 17), *at least* all of the classes specified in Component classes supporting injection must be scanned for annotations that specify deployment information. As specified in section Deploying a Jakarta EE Application, all classes that can be used by the application may optionally be scanned for these annotations. (These are the annotations that specify information equivalent to what can be specified in a deployment descriptor. This requirement says nothing about the processing of annotations that were defined for other purposes.) These annotations may appear on classes, methods, and fields. All resources specified by resource definition annotations must be created. All resource reference annotations must result in JNDI entries in the corresponding namespace. If the corresponding namespace is not available to the class declaring or inheriting the reference, the resulting behavior is undefined. Future versions of this specification may alter this behavior.

Any deployment information specified in a deployment descriptor overrides any deployment information specified in an application's class files. The Jakarta EE component specifications, including this specification, describe when deployment descriptors are optional and which deployment descriptors may exist in either complete or incomplete form. The attribute *metadata-complete* is used in the deployment descriptor to specify whether the descriptor is complete. The *metadata-complete* attribute in the standard deployment descriptors effects *only* the scanning of annotations that specify deployment information, including web services deployment information. It has no impact on the scanning of other annotations.

The scope of the *metadata-complete* attribute is the descriptor it appears in. For historical reasons, the *webservices.xml* deployment descriptor does not have its own *metadata-complete* attribute; instead, it

defers to the value of the *metadata-complete* attribute in the module's deployment descriptor. Specifications that define their own additional deployment descriptors should provide a *metadata-complete* attribute of their own, if deemed useful, with the appropriate semantics.

8.5.1. Deploying a Stand-Alone Jakarta EE Module

This section specifies the requirements for deploying a stand-alone Jakarta EE module.

1. The deployment tool must first read the Jakarta EE module deployment descriptor if provided externally to the package or if present in the package. See the component specifications for the required location and name of the deployment descriptor for each component type.
2. If the deployment descriptor is absent, or is present and is a Java EE 5 or later version descriptor and the *metadata-complete* attribute is not set to *true* , the deployment tool must examine all the class files in the application package. Any annotations that specify deployment information must be logically merged with the information in the deployment descriptor (if present). The correspondence of annotation information with deployment descriptor information, as well as the overriding rules, are described in this and other Jakarta EE specifications. The result of this logical merge process provides the deployment information used in subsequent deployment steps. Note that there is no requirement for the merge process to produce a new deployment descriptor, although that might be a common implementation technique.
3. When deploying a standalone module, the module name is used as the application name. The deployment tool must ensure that the application name is unique in the application server instance. If the name is not unique, the deployment tool may automatically choose a unique name or allow the Deployer to choose a unique name, but must not fail the deployment. This ensures that existing modules continue to be deployable.
4. The deployment tool must deploy all of the components listed in the Jakarta EE module deployment descriptor, or marked via annotations and discovered as described in the previous requirement, according to the deployment requirements of the respective Jakarta EE component specification. If the module is a type that contains JAR format 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. If the module, or any JAR format files within the module, declares a dependency on an installed library, that dependency must be satisfied.
5. The deployment tool must allow the Deployer to configure the container to provide the resources and configuration values needed for each component. The required resources and configuration parameters are specified in the deployment descriptor or via annotations discovered in requirement 2.
6. 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.

8.5.2. Deploying a Jakarta EE Application

This section specifies the requirements for deploying a Jakarta EE application.

1. The deployment tool must first read the Jakarta EE application deployment descriptor provided externally to the application *.ear* file or from within the application *.ear* file (*META-INF/application.xml*). If the deployment descriptor is present, it fully specifies the modules included in the application. If no deployment descriptor is present, the deployment tool uses the following rules to determine the modules included in the application.
2. All files in the application package with a filename extension of *.war* are considered web modules. The context root of the web module is the module name (see Component Creation").
3. All files in the application package with a filename extension of *.rar* are considered resource adapters.
4. A directory named *lib* is considered to be the library directory, as described in Bundled Libraries."
5. For all files in the application package with a filename extension of *.jar* , but not contained in the *lib* directory, do the following:
6. If the *.jar* file contains a *META-INF/MANIFEST.MF* file with a *Main-Class* attribute, or contains a *META-INF/application-client.xml* file, consider the .jar file to be an application client module.
7. If the *.jar* file contains a *META-INF/ejb-jar.xml* file, or contains any class with an Jakarta Enterprise Beans component-defining annotation (*Stateless* , etc.), consider the .jar file to be an Jakarta Enterprise Beans module.
8. All other *.jar* files are ignored unless referenced by a JAR file discovered above using one of the JAR file reference mechanisms such as the *Class-Path* header in a manifest file.
9. The deployment tool must ensure that the application name is unique in the application server instance. If the name is not unique, the deployment tool may automatically choose a unique name or allow the Deployer to choose a unique name, but must not fail the deployment. This ensures that existing applications continue to be deployable.
10. The deployment tool must open each of the Jakarta EE modules listed in the Jakarta EE application deployment descriptor or discovered using the rules above and read the Jakarta EE module deployment descriptor, if present in the package. See the Enterprise Jakarta Beans, Jakarta Servlet, Jakarta Connector and application client specifications for the required location and name of the deployment descriptor for each component type. Deployment descriptors are optional for all module types. (The application client specification is Application Clients".)
11. If the module deployment descriptor is absent, or is present and is a Java EE 5 or later version descriptor and the *metadata-complete* attribute is not set to *true* , the deployment tool must examine all the class files in the application package that can be used by the module (that is, all class files that are included in the *.ear* file and can be referenced by the module, such as the class files included in the module itself, class files referenced from the module by use of a *Class-Path* reference, class files included in the library directory, etc.). Any annotations that specify deployment information must be logically merged with the information in the deployment descriptor (if present). Note that the presence of component-declaring annotations in shared

artifacts, such as libraries in the library directory and libraries referenced by more than one module through *Class-Path* references, can have unintended and undesirable consequences and is not recommended. The correspondence of annotation information with deployment descriptor information, as well as the overriding rules, are described in this and other Jakarta EE specifications. The result of this logical merge process provides the deployment information used in subsequent deployment steps. Note that there is no requirement for the merge process to produce a new deployment descriptor, although that might be a common implementation technique.

12. The deployment tool must install all of the components described by each module deployment descriptor, or marked via annotations and discovered as described in the previous requirement, into the appropriate container according to the deployment requirements of the respective Jakarta EE component specification. All classes in *.jar* files or directories referenced from other JAR files using the *Class-Path* manifest header must be included in the deployment. If the *.ear* file, or any JAR format files within the *.ear* file, declares a dependency on an installed library, that dependency must be satisfied.
13. The deployment tool must allow the Deployer to configure the container to provide the resources and configuration values needed for each component. The required resources and configuration parameters are specified in the deployment descriptor or via annotations discovered in requirement 3.
14. The deployment tool must allow the Deployer to deploy the same Jakarta EE 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.
15. When presenting security role descriptions to the Deployer, the deployment tool must use the descriptions in the Jakarta EE application deployment descriptor rather than the descriptions in any module deployment descriptors for security roles with the same name. However, for security roles that appear in a module deployment descriptor but do not appear in the application deployment descriptor, the deployment tool must use the description provided in the module deployment descriptor.

8.5.3. Deploying a Library

This section specifies the requirements for deploying a library.

1. The deployment tool must record the extension name and version information from the manifest file of the library JAR file. The deployment tool must make the library available to other Jakarta EE deployment units that request it according to the version matching rules described in the Optional Package Versioning specification. Note that the library itself may include dependencies on other libraries and these dependencies must also be satisfied.
2. The deployment tool must make the library available with at least the same security permissions as any application or module that uses it. The library may be installed with the full security permissions of the container.
3. Not all libraries will be deployable on all Jakarta EE products at all times. Libraries that conflict

with the operation of the Jakarta EE product may not be deployable. For example, an attempt to deploy an older version of a library that has subsequently been included in the Jakarta EE specification may be rejected. Similarly, deployment of a library that is also used in the implementation of the Jakarta EE product may be rejected. Deployment of a library that is in active use by an application may be rejected.

8.5.4. Module Initialization

After a successful deployment, all the modules of an application other than application client modules are initialized. The specifications for the different module types describe the steps required to initialize a module. By default, the order of initialization of modules in an application is unspecified. In rare cases it may be important that modules are initialized in a certain order, for example, if a component in one modules uses a component in another module during its initialization. An application can declare that modules must be initialized in the order they're listed in the application deployment descriptor by including the *<initialize-in-order>true</initialize-in-order>* element in the application deployment descriptor. If the application deployment descriptor specifies a module initialization order that conflicts with the initialization order specified by any of the modules (for example, by the use of the Jakarta Enterprise Beans *DependsOn* annotation), the deployment tool must report an error. Application client modules are initialized on their own schedule, typically when an end user invokes them; as such, they are excluded from any initialization ordering requirements.

8.6. Jakarta EE Application XML Schema

The XML grammar for a Jakarta EE application deployment descriptor is defined by the Jakarta EE application schema. The root element of the deployment descriptor for a Jaav EE application is *application* . The granularity of composition for Jakarta EE application assembly is the Jakarta EE module. A Jakarta EE 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 Jakarta EE 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 Jakarta EE application deployment descriptors must conform to the XML Schema definition, or the DTD or schema definition from a previous version of this specification. (See Previous Version Deployment Descriptors.) The deployment descriptor must be named *META-INF/application.xml* in the *.ear* file. Note that this name is case-sensitive. The XML Schema located at *http://xmlns.jcp.org/xml/ns/javaee/application_8.xsd* defines the XML grammar for a Jakarta EE application deployment descriptor.

Jakarta EE Application XML Schema Structure shows a graphic representation of the structure of the Jakarta EE application XML schema.

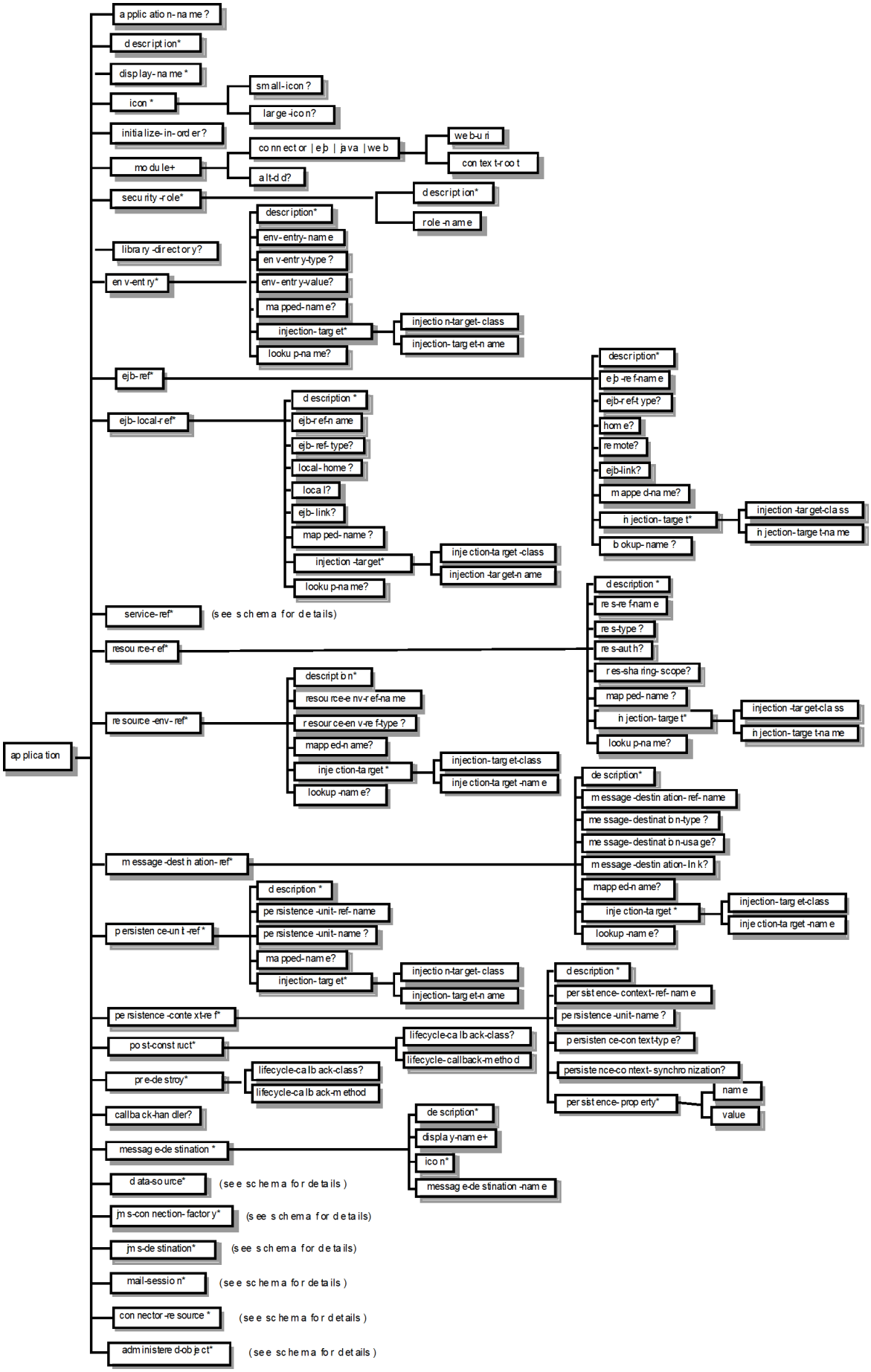

Figure 11. Jakarta EE Application XML Schema Structure

8.7. Common Jakarta EE XML Schema Definitions

The XML Schema located at *http://xmlns.jcp.org/xml/ns/javaee/javaee_8.xsd* defines types that are used by many other Jakarta EE deployment descriptor schemas, both in this specification and in other specifications.

Chapter 9. Profiles

This chapter describes the requirements common to all Jakarta™ EE profiles. It does not define any concrete profiles, delegating this task to separate specifications.

The Jakarta EE Web Profile Specification, published in conjunction with the present specification, defines the first Jakarta EE profile, the Web Profile.

The definition of other profiles is left to future specifications.

9.1. Introduction

A Jakarta EE profile (from now on, simply “a profile”) represents a configuration of the platform suited to a particular class of applications.

A profile may contain a proper subset of the technologies contained in the platform. By doing so, a profile can effectively drop technologies which the platform supports but which are not generally useful in a particular domain.

A profile may also add one or more technologies which are not present in the platform itself. For example, a hypothetical Jakarta EE Portal Profile would likely include the Portlet API (JSR-362).

Additionally, a profile may tag certain technologies as optional. In this case, products implementing the profile may or may not include the technology in question. Naturally, if they do, they need to obey all the relevant requirements mandated by the profile specification.

A product may implement two or more Jakarta EE profiles, or the full platform and one or more Jakarta EE profiles, as long as their combined requirements do not give rise to conflicts.

9.2. Profile Definition

A profile is defined in accordance with the rules of the Jakarta EE Specification Process. Typically, a proposal to create a new profile, or to revise an existing one, will be submitted as a Jakarta Specification Project. Once the Project is approved, a group of interested parties will be formed and conduct work as dictated by the process. The Project proposal for a profile must mention the version of the Jakarta EE Platform that it builds on. Additionally, if it builds on an existing profile, it must mention this fact as well.

Although profiles can be created and evolved independently of the Jakarta EE platform, modulo the rules contained in this specification, it is envisioned that profiles will maintain a reasonable level of alignment with the platform itself, in order to avoid fragmenting the development space into progressively incompatible islands. To this end, a profile must build on the most recent version of the Jakarta EE platform available at the time the Project for the profile is approved. It is also recommended that profile community groups go beyond this requirement and, as much as it is practical, ensure that their profile builds on the most recent version of the Jakarta EE platform at the time the profile is

finalized.

9.3. General Rules for Profiles

A profile must include all technologies that are required components of the Jakarta EE platform or of any profiles on which it builds. These technologies will be listed as required in the profile.

A profile may promote to required status any technologies that are optional components of the Jakarta EE platform or of any profile on which it builds.

Unless otherwise mandated by a profile, any technologies that are optional components of the Jakarta EE platform, or of any profile on which the profile in question builds, must be optional components of the profile itself.

A profile may include as a required or as an optional component any technology outside of those included in the Jakarta EE platform or any profile on which it builds, as long as the corresponding compatibility requirements are satisfied.

A profile must preserve any requirements defined in the Jakarta EE platform specification, or in the specification of any profile on which it builds, as long as the preconditions for those requirements are satisfied. Typically, the preconditions will involve the presence of one or more technologies among those included in the profile. Unconditional requirements must be obeyed unconditionally.

A profile may add any requirements that pertain to one or more technologies whose inclusion it allows or requires. Such requirements must not conflict with those set by the Jakarta EE platform or by any profile on which the present one builds.

The specification for individual technologies may allow for certain features of the technology in question to be optional. In this case, a profile may promote one or more of these features to required status, assuming the Jakarta EE platform or any profile on which it builds hasn't done so already.

A profile must not conflict with the specifications for any technologies it includes either as required or optional components. Therefore, unless the specification for an individual technology explicitly allows for certain features or sets of requirements to be optionally implementable, a profile must not itself attempt to redefine any such features or requirements. For example, a profile may not allow omitting a package or type or method from an API specified elsewhere, unless the specification for that API explicitly allows for this to happen.

Although this specification does not define any APIs, a profile may do so. Since such an API would be available only in profiles that build on the one that defines it, this approach limits the reusability of the API and thus is discouraged.

9.4. Expression of Requirements

The present specification uses the following conventions when expressing requirements that pertain to one or more technologies included in the platform:

- Chapters or sections which are conditional on the presence of a specific technology are marked as such at the very beginning. The condition is then intended to stay in force until the next textual unit at the same logical level (e.g. the following chapter, or section, etc.).
- Individual paragraphs and sentences are deemed to be conditional on any technologies they mention, unless otherwise indicated.
- Section or paragraphs which discuss examples, or are otherwise non-normative, do not contain any requirements.

9.5. Requirements for All Jakarta EE Profiles

The Java Platform, Standard Edition 8 is the required foundation for any Jakarta EE 8 profile.

The following technologies are required to be present in all Jakarta EE profiles:

- Resource and component lifecycle annotations defined by the Common Annotations specification (*Resource* , *Resources* , *PostConstruct* , *PreDestroy*)

The following functionality is required to be supported in all Jakarta EE profiles:

- JNDI “java:” naming context (see JNDI Naming Context”)
- Jakarta Transactions

9.6. Optional Features for Jakarta EE Profiles

All the technologies listed in Required APIs”, and not designated as required in Requirements for All Jakarta EE Profiles”, are designated as optional for use in Jakarta EE profiles.

The following functionality is designated as optional for use in Jakarta EE profiles:

- RMI/IIOP interoperability requirements (see OMG Protocols (Proposed Optional)”)
- Support for *java:comp/ORB* (see ORB References”)

9.7. Full Jakarta™ EE Product Requirements

This section defines the requirements for full Jakarta EE platform products. These requirements correspond to the full set of requirements in previous versions of the Jakarta EE platform specification and update those requirements for this new version of the platform.

Please note that, due to the effects of the pruning process, future versions of the Jakarta EE specification will likely relax the requirements given here, specifically by marking as optional technologies that have been subject to pruning and that are required by the present specification. The set of technologies that have been made optional and/or identified as candidates for pruning is given in Pruned Java Technologies”.

The following technologies are required:

- Jakarta Enterprise Beans 3.2 (except for Jakarta Enterprise Beans entity beans and associated Jakarta Enterprise Beans QL, which have been made optional)
- Jakarta Servlet 4.0
- Jakarta Server Pages 2.3
- Jakarta Expression Language 3.0
- Jakarta Messaging 2.0
- Jakarta Transactions 1.3
- Jakarta Mail 1.6
- Jakarta Connectors 1.7
- Jakarta Enterprise Web Services 1.4
- Jakarta RESTful Web Services 2.1
- Jakarta WebSocket 1.1
- Jakarta JSON Processing 1.1
- Jakarta JSON Binding 1.0
- Jakarta Concurrency 1.0
- Jakarta Batch 1.0
- Jakarta Management 1.1
- Jakarta Authorization 1.5
- Jakarta Authentication 1.1
- Jakarta Security 1.0
- Jakarta Debugging Support for Other Languages 1.0
- Jakarta Standard Tag Library 1.2
- Jakarta Web Services Metadata 1.1
- Jakarta Server Faces 2.3
- Jakarta Annotations 1.3
- Jakarta Persistence 2.2
- Jakarta Bean Validation 2.0
- Jakarta Managed Beans 1.0
- Jakarta Interceptors 1.2
- Jakarta Contexts and Dependency Injection 2.0
- Jakarta Dependency Injection 1.0

The following technologies are optional:

- Jakarta Enterprise Beans 3.2 and earlier entity beans and associated Jakarta Enterprise Beans QL
- Jakarta XML RPC 1.1
- Jakarta XML Registries 1.0
- Jakarta Deployment 1.2

Chapter 10. Application Clients

This chapter describes application clients in the Jakarta™ Enterprise Edition (Jakarta EE).

A full Jakarta EE product must support the application client container as described in this chapter. A Jakarta EE profile may or may not require support for the application client container.

10.1. 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 Jakarta EE application components, application clients depend on a container to provide system services. The application client container may be very light-weight compared to other Jakarta EE containers, providing only the security and deployment services described below.

10.2. Security

The Jakarta EE authentication requirements for application clients are the same as for other Jakarta EE components, and the same authentication techniques may be used as for other Jakarta EE application components.

No authentication is necessary when accessing unprotected web resources. When accessing protected web resources, the usual varieties of authentication may be used, namely HTTP Basic authentication, SSL client authentication, or HTTP Login Form authentication. Lazy authentication may be used.

Authentication is required when accessing protected enterprise beans. The authentication mechanisms for enterprise beans include those required in the Jakarta Enterprise Beans specification for enterprise bean interoperability. Lazy authentication may be used.

An application client makes use of an authentication service provided by the application client container for authenticating its users. The container's service may be integrated with the native platform's authentication system, so that a single signon capability is employed. The container may authenticate the user when the application is started, or it may use lazy authentication, authenticating the user when a protected resource is accessed. This specification does not describe the technique used to authenticate the user, although a later version may do so.

If the container interacts 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 Jakarta EE Application Client XML Schema" for details). The Deployer may override the callback handler specified by the application and use the container's default authentication user interface instead.

If a callback handler is 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 fully support *Callback* objects specified in the *javax.security.auth.callback* package.

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

Application clients typically execute in an environment with a SecurityManager installed, and have similar security permission requirements as servlets. The security permission requirements are described fully in Java Platform, Standard Edition (Java SE) Requirements."

10.3. Transactions

Application clients are not required to have direct access to the transaction facilities of the Jakarta EE platform. A Jakarta EE product is not required to provide a Jakarta Transactions *UserTransaction* object for use by application clients. Application clients can invoke enterprise beans that start transactions, 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 Jakarta Enterprise Beans server.

10.4. Resources, Naming, and Injection

As with all Jakarta EE components, application clients use JNDI to look up enterprise beans, get access to resource managers, reference configurable parameters set at deployment time, and so on. Application clients use the *java:* JNDI namespace to access these items (see Resources, Naming, and Injection" for details).

Injection is also supported for the application client main class. Because the application client container does not create instances of the application client main class, but merely loads the class and invokes the static *main* method, injection into the application client class uses *static* fields and methods, unlike other Jakarta EE components. Injection occurs before the *main* method is called.

10.5. Application Programming Interfaces

Application clients have all the facilities of the Java™ Platform, Standard Edition (subject to security restrictions), as well as various standard extensions, as described in Chapter EE.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, and so on).

10.6. Packaging and Deployment

Application clients are packaged in JAR format files with a *.jar* extension and may include a deployment descriptor similar to other Jakarta EE application components. The deployment descriptor describes the enterprise beans, web services, and other types of external resources referenced by the application. If the deployment descriptor is not included, or is included but not marked *metadata-complete* , annotations on the main class of the application client may also be used to describe the resources needed by the application. As with other Jakarta EE application components, access to resources must be configured at deployment time, names assigned for enterprise beans and resources, and so on.

The following table describes the cases the deployment tool must consider when deciding whether or not to process annotations on the application client main class. Whether or not to process annotations depends on the presence and version of the deployment descriptor and the setting of the *metadata-complete* attribute.

Table 8. Deployment Descriptor Processing Requirements

Deployment descriptor	metadata-complete?	process annotations?
application-client_1_2	N/A	No
application-client_1_3	N/A	No
application-client_1_4	N/A	No
application-client_5	Yes	No
application-client_5	No	Yes
application-client_6	Yes	No
application-client_6	No	Yes
application-client_7	Yes	No
application-client_7	No	Yes
application-client_8	Yes	No
application-client_8	No	Yes
none	N/A	Yes

The *metadata-complete* attribute defines whether the application client deployment descriptor is complete, or whether the class files available to the application client module should be examined for annotations that specify deployment information. Deployment information, in this sense, refers to any information that could have been specified by the application client deployment descriptor for the module.

If the value of the *metadata-complete* attribute is specified as *“true”* , the deployment tool must ignore any annotations that specify such deployment information in the class files packaged in the application client jar file. Such annotations must also be ignored when processing the class files that are available

to the application client module for the deployment of this module according to Deploying a Jakarta EE Application".

Note that a *"true"* value for the *metadata-complete* attribute does *not* preempt the processing of *all* annotations, only those that specify deployment information.

The list of annotations to which the *metadata-complete* attribute applies currently includes the following:

- *javax.annotation.PostConstruct*
- *javax.annotation.PreDestroy*
- *javax.annotation.Resource*
- *javax.annotation.Resources*
- *javax.annotation.sql.DataSourceDefinition*
- *javax.annotation.sql.DataSourceDefinitions*
- *javax.ejb.EJB*
- *javax.ejb.EJBs*
- *javax.jms.JMSConnectionFactoryDefinition*
- *javax.jms.JMSConnectionFactoryDefinitions*
- *javax.jms.JMSDestinationDefinition*
- *javax.jms.JMSDestinationDefinitions*
- *javax.mail.MailSessionDefinition*
- *javax.mail.MailSessionDefinitions*
- *javax.persistence.PersistenceUnit*
- *javax.persistence.PersistenceUnits*
- *javax.resource.AdministeredObjectDefinition*
- *javax.resource.AdministeredObjectDefinitions*
- *javax.resource.ConnectionFactoryDefinition*
- *javax.resource.ConnectionFactoryDefinitions*
- All annotations in the following packages:
  - *javax.jws*
  - *javax.jws.soap*
  - *javax.xml.ws*
  - *javax.xml.ws.soap*
  - *javax.xml.ws.spi*

If the *metadata-complete* attribute is not specified or its value is *"false"* , the deployment tool must examine the class files for all such annotations.

The tool used to deploy an application client to the client machine, and the mechanism used to install the application client, is not specified. Very sophisticated Jakarta EE products may allow the application client to be deployed on a Jakarta EE server and automatically made available to some set of (usually intranet) clients. Other Jakarta EE products may require the Jakarta EE 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 Jakarta EE server to produce an installation package that could be used by 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 Jakarta EE 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.

10.7. Jakarta EE Application Client XML Schema

The XML grammar for a Jakarta EE application client deployment descriptor is defined by the Jakarta EE application-client schema. 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 conform to the XML Schema definition, or to a DTD or schema definition from a previous version of this specification. (See Previous Version Deployment Descriptors.") The deployment descriptor must be named *META-INF/application-client.xml* in the application client's *.jar* file. Note that this name is case-sensitive.

Jakarta EE Application Client XML Schema Structure shows the structure of the Jakarta EE application-client XML Schema. The Jakarta EE application-client XML Schema is located at *http://xmlns.jcp.org/xml/ns/javaee/application-client_8.xsd* .

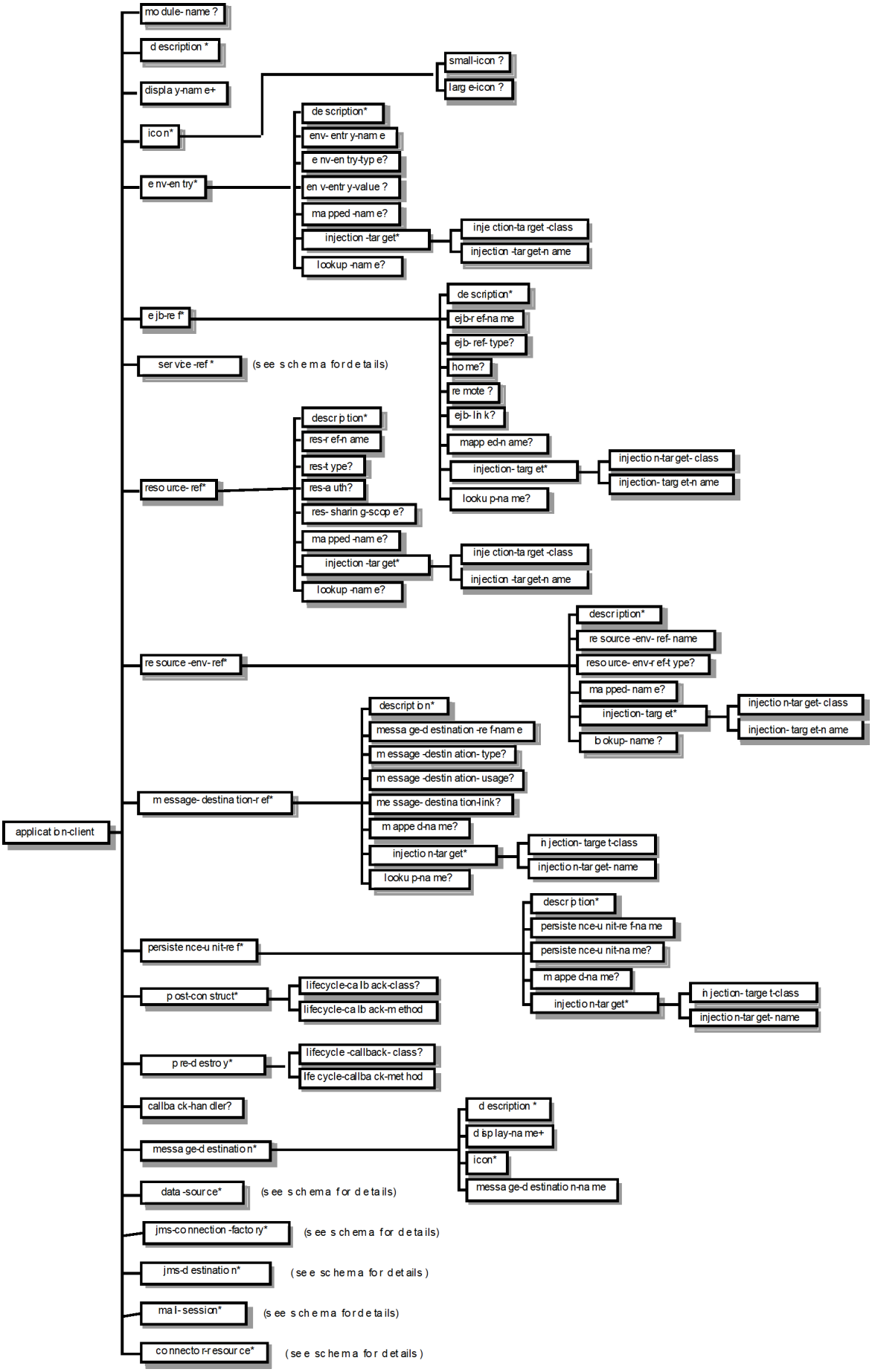

Figure 12. Jakarta EE Application Client XML Schema Structure

Chapter 11. Service Provider Interface

The Jakarta™ EE Platform includes several technologies that are primarily intended to be used to extend the capabilities of the Jakarta EE containers. In addition, some Jakarta EE technologies include service provider interfaces along with their application programming interfaces. A Jakarta EE profile may include some or all of these facilities, as described in Profiles".

11.1. Jakarta™ Connectors

The Connector API defines how resource adapters are packaged and integrated with any Jakarta EE product. Many types of service providers can be provided using the Connector API and packaging, including JDBC drivers, Jakarta Messaging providers, and Jakarta XML Registries providers. All Jakarta EE products must support the Connector APIs, as specified in the Connector specification.

The Jakarta EE Connectors specification is available at *https://jakarta.ee/specifications/connectors* .

11.2. Jakarta™ Authorization

The Jakarta Authorization specification defines the contract between a Jakarta EE container and an authorization policy provider.

The Jakarta Authorization specification can be found at *https://jakarta.ee/specifications/authorization* .

11.3. Jakarta™ Transactions

The Jakarta Transactions defines the *TransactionSynchronizationRegistry* interface that is intended for use by system level application server components such as persistence managers, resource adapters, as well as Jakarta Enterprise Beans and Web application components. This provides the ability to register synchronization objects with special ordering semantics, associate resource objects with the current transaction, get the transaction context of the current transaction, get current transaction status, and mark the current transaction for rollback.

The Jakarta Transaction specification is available at *https://jakarta.ee/specifications/transactions* .

11.4. Jakarta™ Persistence

Jakarta Persistence provides interfaces in the *javax.persistence.spi* package that allow a persistence provider to be plugged into the Jakarta Persistence framework.

The Jakarta Persistence specification can be found at *https://jakarta.ee/specifications/persistence* .

11.5. Jakarta™ Mail

The Jakarta Mail specification describes how Jakarta Mail protocol providers can be packaged and distributed so that they can be discovered and used through the Jakarta Mail API. This allows the Jakarta Mail API to be extended with support for new mail protocols and mailbox formats.

The Jakarta Mail API specification is available at *https://jakarta.ee/specifications/mail* .

Chapter 12. Compatibility and Migration

This chapter summarizes compatibility and migration issues for the Jakarta™ EE platform. The specifications for each of the component technologies included in Jakarta EE also describe compatibility and migration issues for that technology in much more detail.

12.1. Compatibility

The word compatibility covers many different concepts. Jakarta EE products are compatible with the Jakarta EE specification if they implement the APIs and behavior required by the specification. Applications are compatible with a release of the Jakarta EE platform if they only depend on APIs and behavior defined by that release of the platform. A new release of the Jakarta EE platform is compatible with previous releases of the platform if all portable applications written to the previous release of the platform will also run unchanged and with identical behavior on the new release of the platform.

Compatibility is a core value of the Jakarta EE platform. A Jakarta EE product is required to support portable applications written to previous versions of the platform. Compatibility and portability work together to provide the Write Once, Run Anywhere value of the Jakarta EE platform. Jakarta EE products conform to the Jakarta EE specifications by providing APIs and behavior as required by the specifications. Portable applications depend only on the APIs and behavior required by the Jakarta EE specifications. In general, portable applications written to a previous version of the platform will continue to work without change and with identical behavior on the current version of the platform.

12.2. Migration

Migration is the act of converting an application to use new facilities introduced in this release of the platform. Given the strong level of compatibility in this release of the Jakarta EE platform, migration is largely an optional exercise. Still, an application may be improved (better performance, simpler to develop, more flexible, etc.) by converting it to use newer facilities of the Jakarta EE platform.

12.2.1. Jakarta Persistence

Jakarta Persistence provides a much richer set of modeling capabilities and object/relational mapping capabilities than EJB CMP entity beans and is significantly easier to use.

Support for EJB CMP and BMP entity beans has been made optional with the Java EE 7 release. Support for EJB CMP 1.1 entity beans has been deprecated since Java EE 5. Applications are strongly encouraged to migrate applications using EJB entity beans to Jakarta Persistence.

12.2.2. Jakarta XML Web Services

Jakarta XML Web Services, along with Jakarta XML Binding and the Metadata for Web Services specification, provides simpler and more complete support for web services than is available using the

JAX-RPC technology. Support for JAX-RPC has been made optional with the Java EE 7 release. Applications that provide web services using JAX-RPC should consider migrating to the Jakarta XML Web Services API. Note that because both technologies support the same web service interoperability standards, clients and services can be migrated to the new API independently.

Chapter 13. Future Directions

This version of the Jakarta™ EE Platform 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 Jakarta EE platform will continue to evolve on their own and we will include the latest version of each API.

13.1. Jakarta EE SPI

Many of the APIs that make up the Jakarta EE platform include an SPI layer that allows 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 Jakarta Connectors Specification does define the requirements for certain types of service providers called resource adapters, and the Jakarta Authorization specification defines requirements for security service providers. Future versions of this specification will more fully define the Jakarta EE SPI.

Appendix A: Previous Version Deployment Descriptors

This appendix describes Document Type Definitions and XML schemas for Deployment Descriptors from previous versions of the Java EE specification. All Jakarta EE products are required to support these DTDs and schemas as well as the schemas 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. Java EE 7 Application XML Schema

The XML grammar for a Java EE application deployment descriptor is defined by the Java EE application schema. The root element of the deployment descriptor for a Jaav EE application is *application* . The granularity of composition for Java EE application assembly is the Java EE module. A Java EE 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 Java EE 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 Java EE application deployment descriptors must conform to the XML Schema definition, or the DTD or schema definition from a previous version of this specification. The deployment descriptor must be named *META-INF/application.xml* in the *.ear* file. Note that this name is case-sensitive. Java EE 7 Application XML Schema Structure shows a graphic representation of the structure of the Java EE application 7 XML Schema.

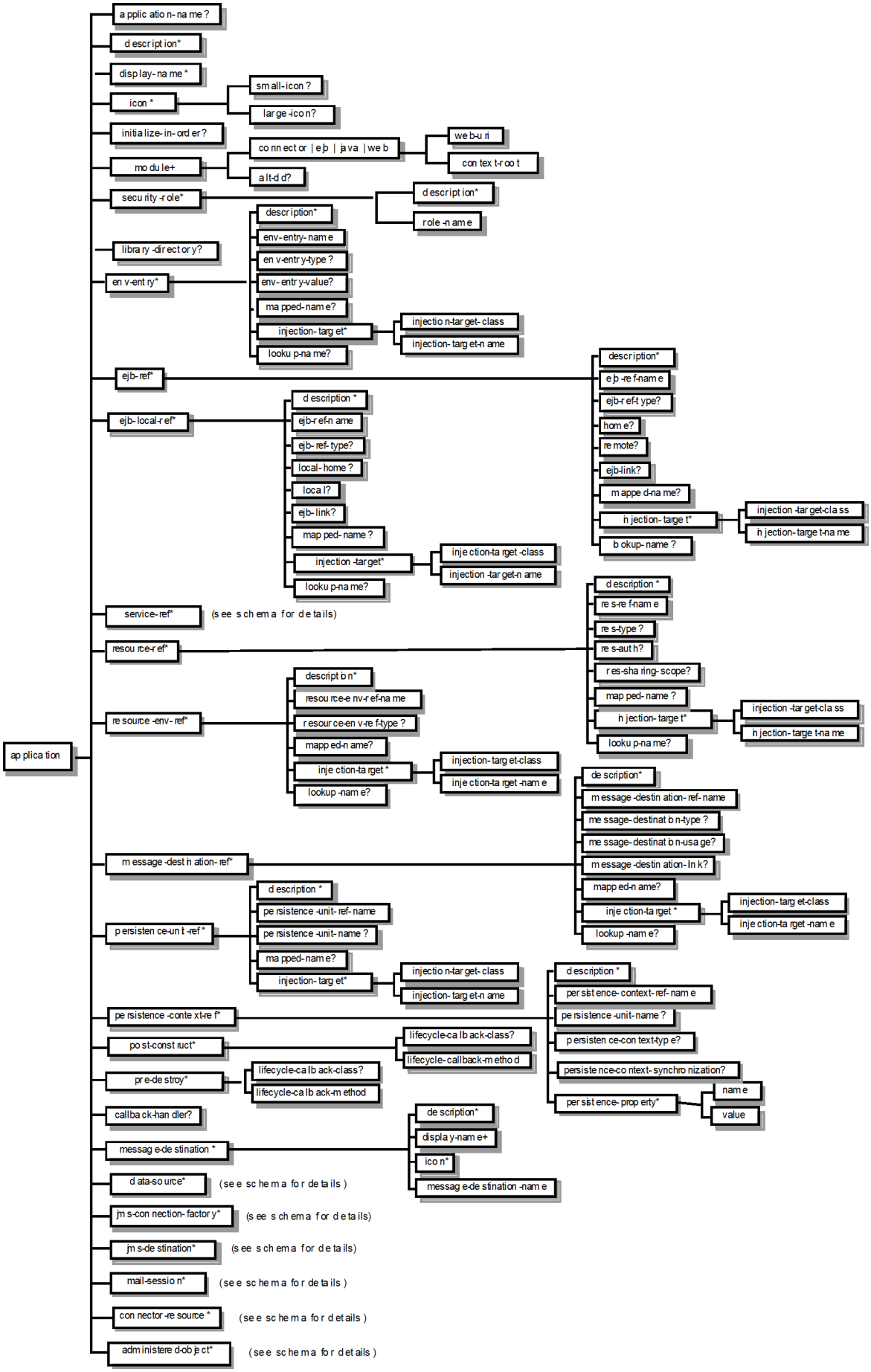

Figure 13. Java EE 7 Application XML Schema Structure

Java EE Application XML Schema Structure The XML Schema located at *http://xmlns.jcp.org/xml/ns/javaee/application_7.xsd* defines the XML grammar for a Java EE 7 application deployment descriptor.

A.2. Common Java EE XML Schema Definitions

The XML Schema located at *http://xmlns.jcp.org/xml/ns/javaee/javaee_7.xsd* defines types that are used by many other Java EE deployment descriptor schemas, both in this specification and in other specifications.

A.3. Java EE 7 Application Client XML Schema

The XML grammar for a Java EE application client deployment descriptor is defined by the Java EE application-client schema. 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 conform to the XML Schema definition, or to a DTD or schema definition from a previous version of this specification. The deployment descriptor must be named *META-INF/application-client.xml* in the application client's *.jar* file. Note that this name is case-sensitive.

Java EE 7 Application Client XML Schema Structure shows the structure of the Java EE application-client XML Schema. The Java EE application-client XML Schema is located at *http://xmlns.jcp.org/xml/ns/javaee/application-client_7.xsd* .

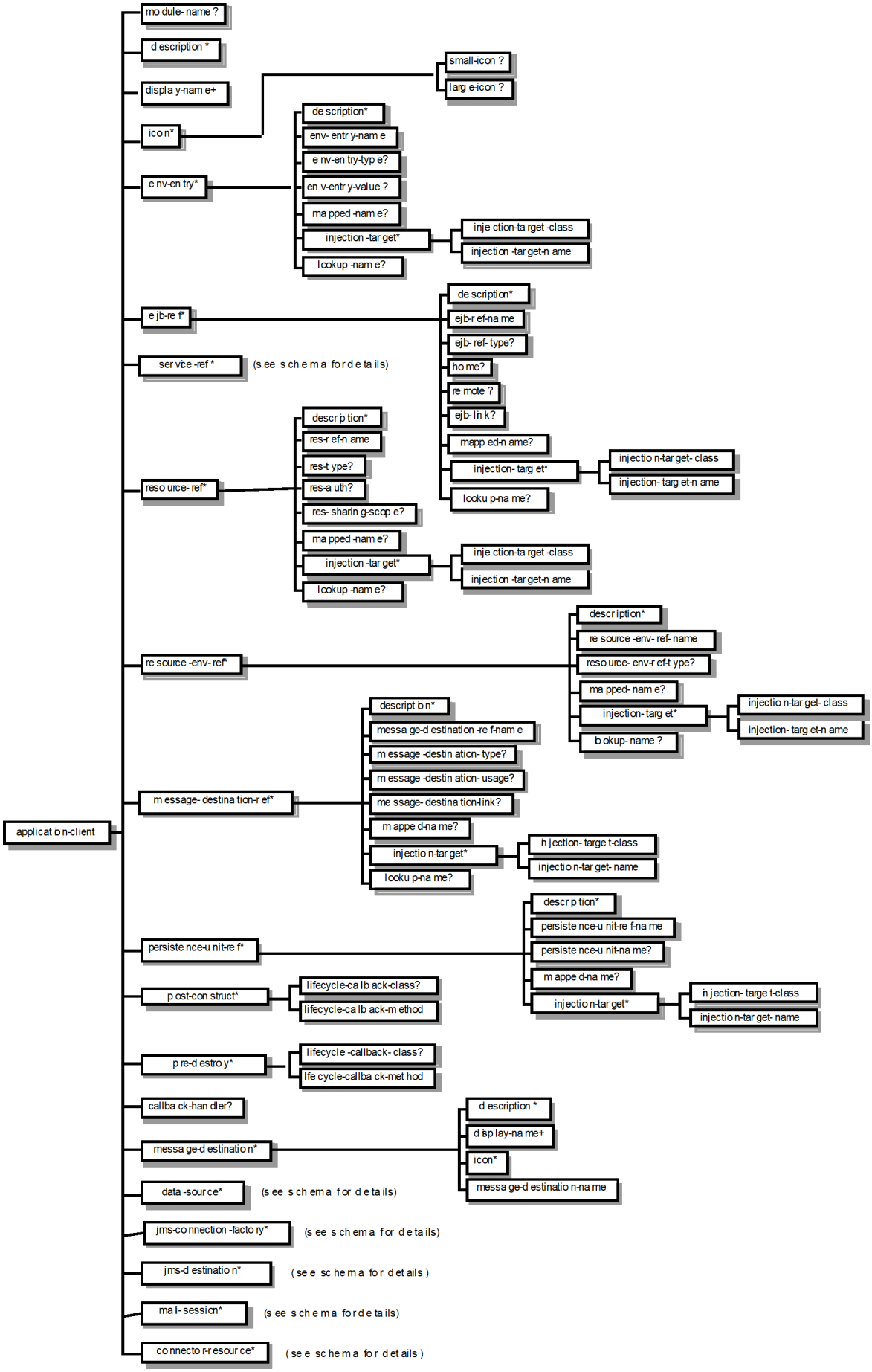

Figure 14. Java EE 7 Application Client XML Schema Structure

A.4. Java EE 6 Application XML Schema

The XML grammar for a Java EE application deployment descriptor is defined by the Java EE application schema. The root element of the deployment descriptor for a Jaav EE application is *application* . The granularity of composition for Java EE application assembly is the Java EE module. A Java EE 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 Java EE 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 Java EE application deployment descriptors must conform to the XML Schema definition, or the DTD or schema definition from a previous version of this specification. The deployment descriptor must be named *META-INF/application.xml* in the *.ear* file. Note that this name is case-sensitive. Java EE Application XML Schema Structure shows a graphic representation of the structure of the Java EE application XML Schema.

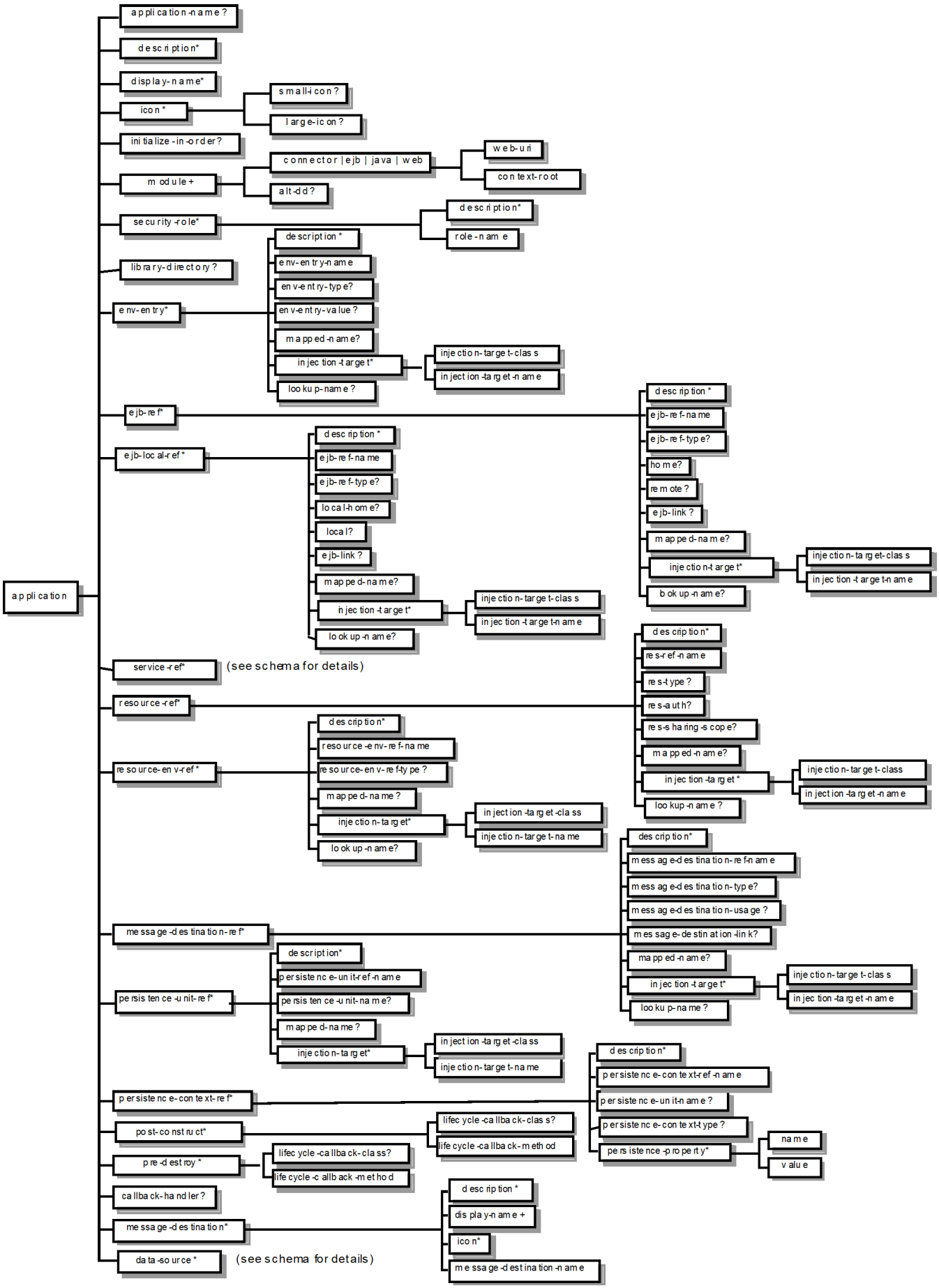

Figure 15. Java EE Application XML Schema Structure

Java EE Application XML Schema Structure The XML Schema located at *http://java.sun.com/xml/ns/javaee/application_6.xsd* defines the XML grammar for a Java EE application deployment descriptor.

A.5. Common Java EE XML Schema Definitions

The XML Schema located at *http://java.sun.com/xml/ns/javaee/javaee_6.xsd* defines types that are used by many other Java EE deployment descriptor schemas, both in this specification and in other specifications.

A.6. Java EE 6 Application Client XML Schema

The XML grammar for a Java EE application client deployment descriptor is defined by the Java EE application-client schema. 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 conform to the XML Schema definition, or to a DTD or schema definition from a previous version of this specification. The deployment descriptor must be named *META-INF/application-client.xml* in the application client's *.jar* file. Note that this name is case-sensitive.

Java EE Application Client XML Schema Structure shows the structure of the Java EE application-client XML Schema. The Java EE application-client XML Schema is located at *http://java.sun.com/xml/ns/javaee/application-client_6.xsd* .

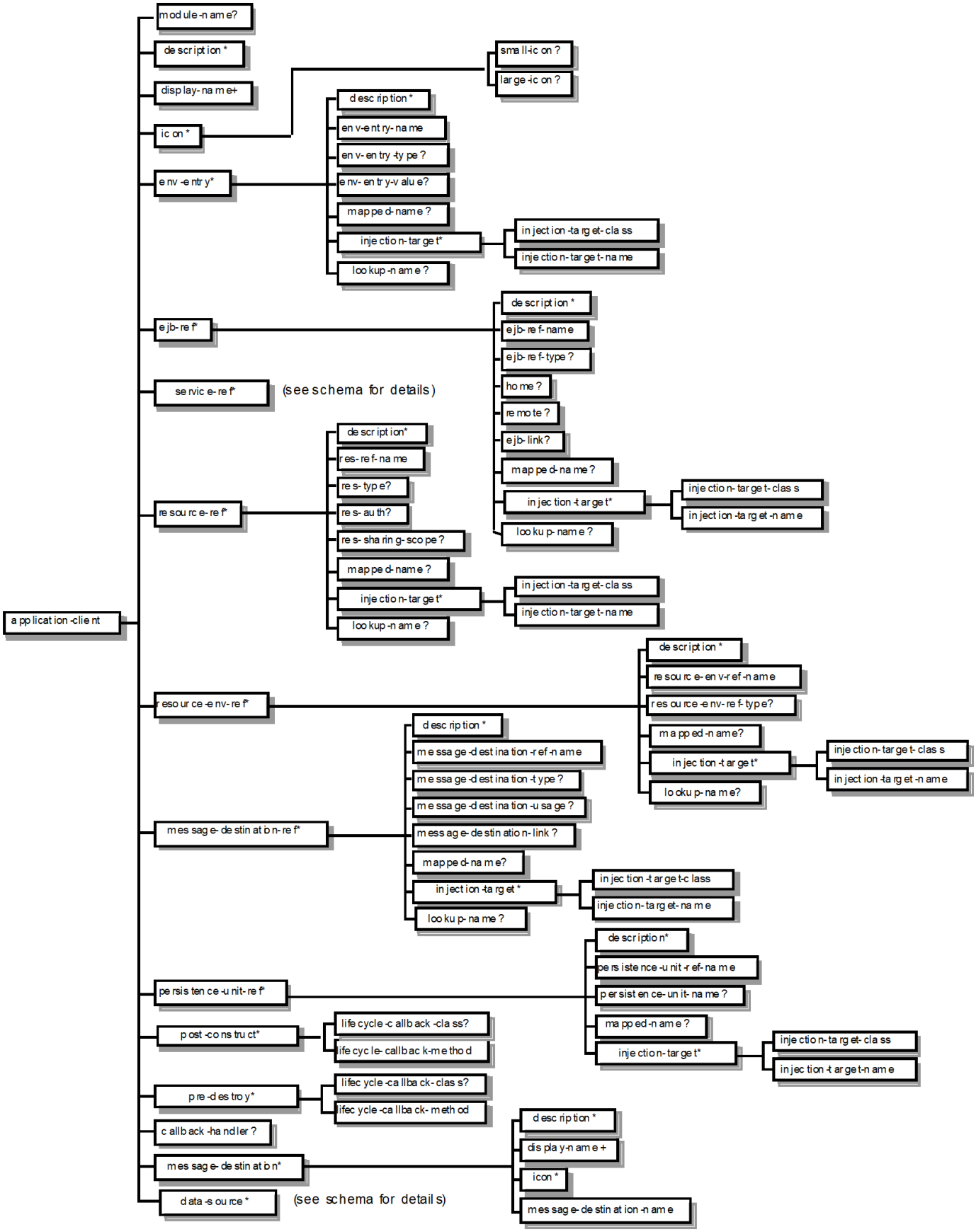

Figure 16. Java EE Application Client XML Schema Structure

A.7. Java EE 5 Application XML Schema

The XML grammar for a Java EE application deployment descriptor is defined by the Java EE application schema. The root element of the deployment descriptor for a Jaav EE application is *application* . The granularity of composition for Java EE application assembly is the Java EE module. A Java EE 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 Java EE 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>* .

A valid Java EE 5 application deployment descriptors must conform to this XML Schema definition.

The deployment descriptor must be named *META-INF/application.xml* in the *.ear* file. Note that this name is case-sensitive.

Java EE Application XML Schema Structure shows a graphic representation of the structure of the Java EE application XML Schema.

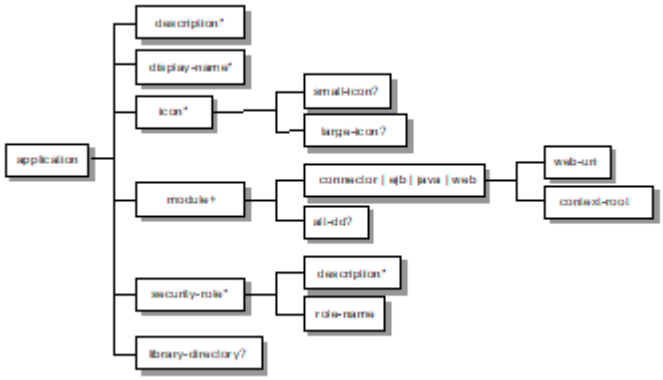

Figure 17. Java EE Application XML Schema Structure

The XML Schema located at *http://java.sun.com/xml/ns/javaee/application_5.xsd* defines the XML grammar for a Java EE application deployment descriptor.

A.8. Common Java EE 5 XML Schema Definitions

The XML Schema located at *http://java.sun.com/xml/ns/javaee/javaee_5.xsd* defines types that are used by many other Java EE deployment descriptor schemas, both in this specification and in other specifications.

A.9. Java EE 5 Application Client XML Schema

The XML grammar for a Java EE application client deployment descriptor is defined by the Java EE application-client schema. 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 conform to the XML Schema definition, or to a DTD or schema definition from a previous version of this specification. The deployment descriptor must be named *META-INF/application-client.xml* in the application client's *.jar* file. Note that this name is case-sensitive.

Java EE Application Client XML Schema Structure shows the structure of the Java EE application-client XML Schema. The Java EE application-client XML Schema is located at *http://java.sun.com/xml/ns/javaee/application-client_5.xsd* .

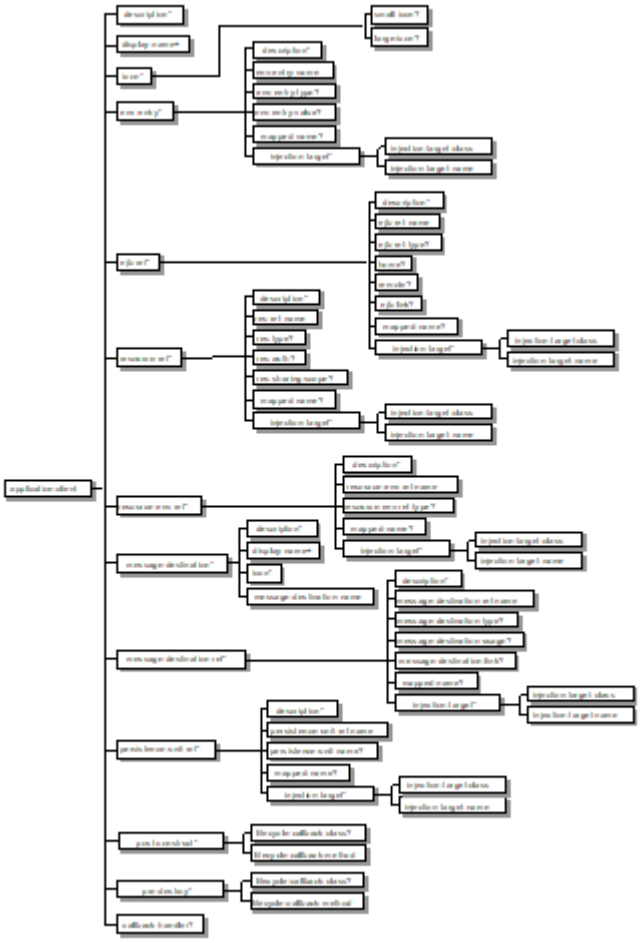

Figure 18. Java EE Application Client XML Schema Structure

A.10. J2EE 1.4 Application XML Schema

This section provides the XML Schema for the J2EE application deployment descriptor. The XML grammar for a J2EE application deployment descriptor is defined by the *J2EE:application* schema. 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*> .

A valid J2EE application deployment descriptors may conform to the XML Schema definition below. The deployment descriptor must be named *META-INF/application.xml* in the *.ear* file. Note that this name is case-sensitive.

J2EE:application XML DTD Structure J2EE Application XML Schema Structure shows a graphic representation of the structure of the J2EE application XML Schema.

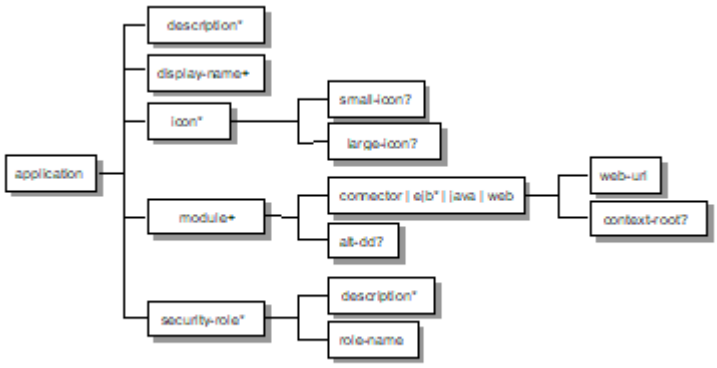

Figure 19. J2EE Application XML Schema Structure

The XML Schema that defines the XML grammar for a J2EE 1.4 application deployment descriptor is located at *http://java.sun.com/xml/ns/j2ee/application_1_4.xsd* .

A.11. Common J2EE 1.4 XML Schema Definitions

The XML Schema that defines types that are used by many other J2EE 1.4 deployment descriptor schemas, both in this specification and in other specifications, is located at *http://java.sun.com/xml/ns/j2ee/j2ee_1_4.xsd* .

A.12. J2EE:application 1.3 XML DTD

This section provides the XML DTD for the J2EE 1.3 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 as 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*> .

A valid J2EE 1.3 application deployment descriptor may 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">
```

The deployment descriptor must be named *META-INF/application.xml* in the *.ear* file.

J2EE:application XML DTD Structure shows a graphic representation of the structure of the *J2EE:application* XML DTD.

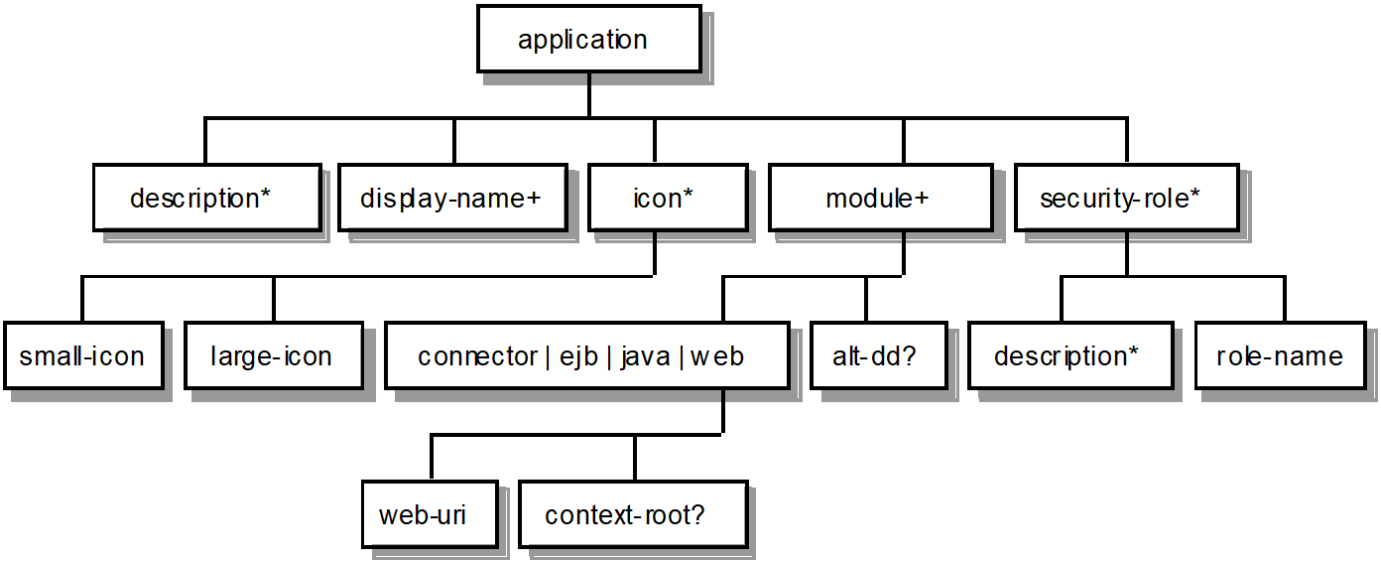

Figure 20. J2EE:application XML DTD Structure

The DTD that defines the XML grammar for a J2EE 1.3 application deployment descriptor is available at http://java.sun.com/dtd/application_1_3.dtd.

A.13. J2EE:application 1.2 XML DTD

This section provides the XML DTD for the J2EE 1.2 version of the application deployment descriptor. A valid J2EE 1.2 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">

J2EE:application XML DTD Structure shows a graphic representation of the structure of the *J2EE:application* XML DTD.

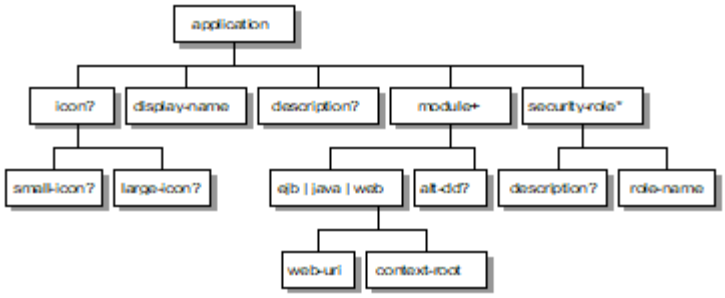

Figure 21. J2EE.application XML DTD Structure

The DTD that defines the XML grammar for a J2EE 1.2 application deployment descriptor is available at http://java.sun.com/j2ee/dtds/application_1_2.dtd.

A.14. J2EE 1.4 Application Client XML Schema

The XML grammar for a J2EE application client deployment descriptor is defined by the J2EE application-client schema. 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>* .

A valid *application-client* deployment descriptors may conform to the following XML Schema definition. The deployment descriptor must be named *META-INF/application-client.xml* in the application client's *.jar* file. Note that this name is case-sensitive.

J2EE Application Client XML Schema Structure shows the structure of the J2EE 1.4 application-client XML Schema, which is available at *http://java.sun.com/xml/ns/j2ee/application-client_1_4.xsd* .

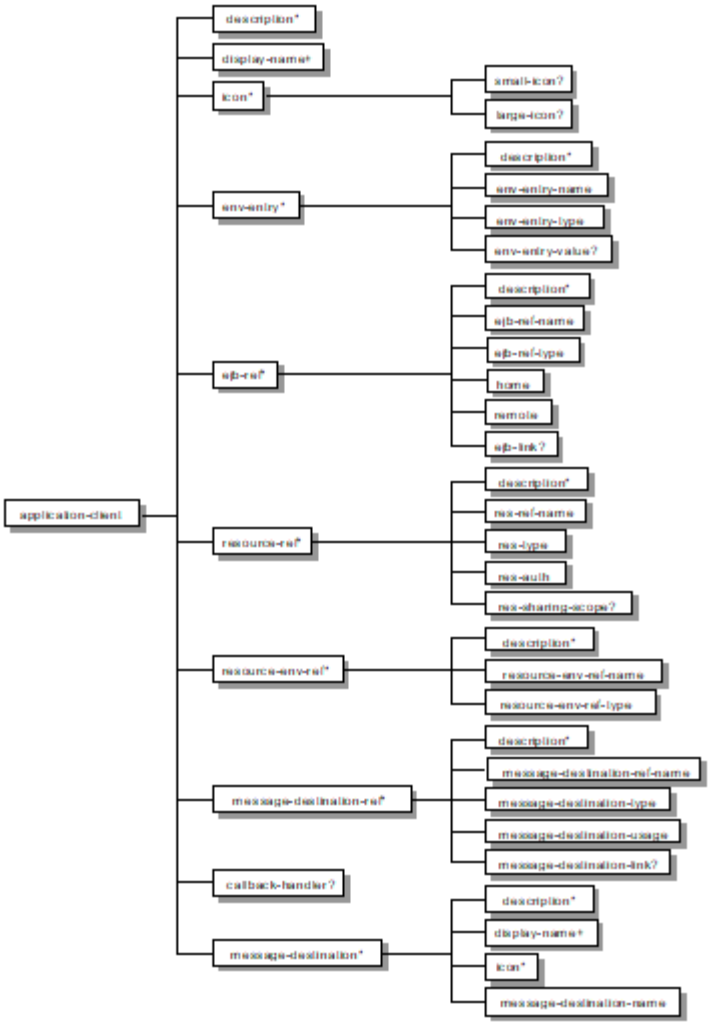

Figure 22. J2EE Application Client XML Schema Structure

A.15. J2EE:application-client 1.3 XML DTD

This section describes the XML DTD for the J2EE 1.3 version of the application client deployment descriptor. 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>* .

A valid *application-client* deployment descriptor may 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* ">

The deployment descriptor must be named *META-INF/application-client.xml* in the application client's *.jar* file.

J2EE:application-client XML DTD Structure shows the structure of the *J2EE:application-client* XML DTD,

which is available at http://java.sun.com/dtd/application-client_1_3.dtd.

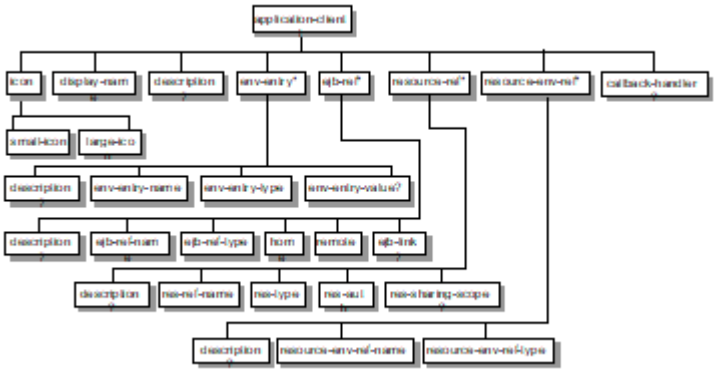

Figure 23. J2EE:application-client XML DTD Structure

A.16. J2EE:application-client 1.2 XML DTD

This section describes the XML DTD for the J2EE 1.2 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">

J2EE:application-client XML DTD Structure shows the structure of the *J2EE:application-client* XML DTD, which is available at http://java.sun.com/j2ee/dtds/application-client_1_2.dtd.

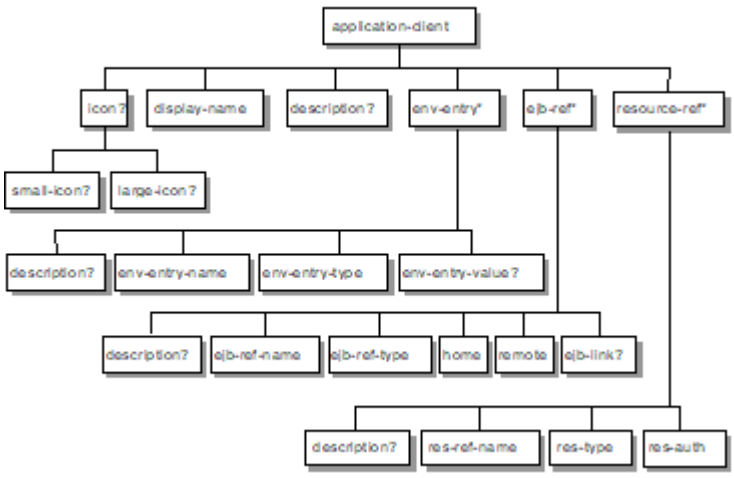

Figure 24. J2EE:application-client XML DTD Structure

Appendix B: Revision History

B.1. Changes in Final Release Draft

B.1.1. Editorial Changes

- Changed most instances of "Java EE" to "Jakarta EE", except where the previous technology was clearly the target.
- Modified references to Oracle™-related items per the documented guidelines.
- Updated “Related Documents".

Appendix C: Related Documents

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

Jakarta™ EE Platform Specification Version 8. Available at: *https://jakarta.ee/specifications/full-platform/8.0*

Java™ Platform, Standard Edition, v8 API Specification (Java SE specification). Available at: *https://docs.oracle.com/javase/8/docs/*

Jakarta™ Enterprise Beans Specification, Version 3.2. Available at: *https://jakarta.ee/specifications/enterprise-beans/3.2*

Jakarta™ Server Pages Specification, Version 2.3. Available at: *https://jakarta.ee/specifications/pages/2.3*

Jakarta™ Expression Language Specification, Version 3.0. Available at: *https://jakarta.ee/specifications/expression-language/3.0*

Jakarta™ Annotations Specification, Version 1.3. Available at: *https://jakarta.ee/specifications/annotations/1.3*

Jakarta™ Servlet Specification, Version 4.0. Available at: *https://jakarta.ee/specifications/servlet/4.0*

JDBC™ 4.2 API (JDBC specification). Available at: *https://jcp.org/en/jsr/detail?id=221*

Java™ Naming and Directory Interface 1.2 Specification (JNDI specification). Available at: *https://docs.oracle.com/javase/8/docs/technotes/guides/jndi/index.html*

Jakarta™ Messaging Specification, Version 2.0. Available at: *https://jakarta.ee/specifications/messaging/2.0*

Jakarta™ Transaction Specification, Version 1.3. Available at: *https://jakarta.ee/specifications/transactions/1.3*

Java™ Transaction Service, Version 1.0 (JTS specification). *Available at: _https://www.oracle.com/technetwork/java/javaee/jts-spec095-1508547.pdf*

Jakarta™ Mail Specification, Version 1.6. Available at: *https://jakarta.ee/specifications/mail/1.6*

JavaBeans™ Activation Framework Specification Version 1.2 (JAF specification). Available at: *https://www.oracle.com/technetwork/java/javase/tech/index-jsp-138795.html*

Jakarta™ Connectors Specification, Version 1.7. Available at: *https://jakarta.ee/specifications/connectors/1.7*

Jakarta™ XML Web Services Specification, Version 2.3. Available at: *https://jakarta.ee/specifications/xml-web-services/2.3*

Jakarta™ XML RPC Specification, Version 1.1. Available at: *https://jakarta.ee/specifications/xml-rpc/1.1*

SOAP with Attachments API for Java™ 1.4 (SAAJ specification). Available at: *https://download.oracle.com/otn-pub/jcp/saaj-1_4-mrel4-spec/saaj-1.4.pdf*

Jakarta™ XML Registries Specification, Version 1.0. Available at: *https://jakarta.ee/specifications/xml-registries/1.0*

Jakarta™ RESTful Web Services Specification, Version 2.1. Available at: *https://jakarta.ee/specifications/restful-web-services/2.1*

Jakarta™ Management Specification, Version 1.1. Available at: *https://jakarta.ee/specifications/management/1.1*

Jakarta™ Deployment Specification, Version 1.7. Available at: *https://jakarta.ee/specifications/deployment/1.7*

Jakarta™ Authorization Specification, Version 1.5. Available at: *https://jakarta.ee/specifications/authorization/1.5*

Jakarta™ Authentication Specification, Version 1.1. Available at: *https://jakarta.ee/specifications/authentication/1.1*

Jakarta™ Security Specification, Version 1.0. Available at: *https://jakarta.ee/specifications/security/1.0*

Jakarta™ Debugging Support for Other Languages Specification, Version 1.0. Available at: *https://jakarta.ee/specifications/debugging/1.0*

Jakarta™ Standard Tag Library Specification, Version 1.2. Available at: *https://jakarta.ee/specifications/tags/1.2*

Jakarta™ Web Services Metadata Specification, Version 1.1. Available at: *https://jakarta.ee/specifications/web-services-metadata/1.1*

Jakarta™ Server Faces Specification, Version 2.3. Available at: *https://jakarta.ee/specifications/faces/2.3*

Jakarta™ Persistence Specification, Version 2.2. Available at: *https://jakarta.ee/specifications/persistence/2.2*

Jakarta™ Bean Validation Specification, Version 2.0. Available at: *https://jakarta.ee/specifications/bean-validation/2.0*

Jakarta™ Managed Beans Specification, Version 1.0. Available at: *https://jakarta.ee/specifications/managed-beans/1.0*

Jakarta™ Interceptors Specification, Version 1.2. Available at: *https://jakarta.ee/specifications/interceptors/1.2*

Jakarta™ Contexts and Dependency Injection Specification, Version 2.0. Available at: *https://jakarta.ee/specifications/cdi/2.0*

Jakarta™ Dependency Injection Specification, Version 1.0. Available at: *https://jakarta.ee/specifications/dependency-injection/1.0*

Jakarta™ WebSocket Specification, Version 1.1. Available at: *https://jakarta.ee/specifications/websocket/1.1*

Jakarta™ JSON Processing Specification, Version 1.1. Available at: *https://jakarta.ee/specifications/jsonp/1.1*

Jakarta™ JSON Binding Specification, Version 1.0. Available at: *https://jakarta.ee/specifications/jsonb/1.0*

Jakarta™ Concurrency Specification, Version 1.1. Available at: *https://jakarta.ee/specifications/concurrency/1.1*

Jakarta™ Batch Specification, Version 1.0. Available at: *https://jakarta.ee/specifications/batch/1.0*

Extension Mechanism Architecture, Available at *https://docs.oracle.com/javase/8/docs/technotes/guides/extensions/index.html*.

Optional Package Versioning, Available at *https://docs.oracle.com/javase/8/docs/technotes/guides/extensions/index.html*.

JAR File Specification, Available at *https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jar.html* .

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

CORBA 2.6 - Chapter 26 - Secure Interoperability, Available at *https://www.omg.org/cgi-bin/doc?formal/01-12-30* .

IDL To Java™ Language Mapping Specification , Available at *https://www.omg.org/cgi-bin/doc?ptc/2000-01-08* .

Java™ Language To IDL Mapping Specification , Available at *https://www.omg.org/cgi-bin/doc?ptc/2000-01-06* .

Interoperable Naming Service, Available at *https://www.omg.org/cgi-bin/doc?ptc/00-08-07* .

Transaction Service Specification (OTS specification), Available at *https://www.omg.org/cgi-bin/doc?formal/2001-11-03* .

The SSL Protocol, Version 3.0. Available at *https://tools.ietf.org/html/rfc6101* .

Architectural Styles and the Design of Network-based Software Architectures (REST), R. Fielding, Ph.d dissertation, University of California, Irvine, 2000. Available at *https://roy.gbiv.com/pubs/dissertation/top.html* .

Java™ Community Process *SM* 2: Process Document, Version 2.10 (March 21, 2016). Available at *https://jcp.org/en/procedures/jcp2* .