

# Java™ 2 Platform Enterprise Edition Specification, v5.0

Please send comments to: [j2ee-spec-feedback@sun.com](mailto:j2ee-spec-feedback@sun.com)

Public  
Review

Public Review - 6/20/05 Bill Shannon



**We make the net work.**



Java™ 2 Platform, Enterprise Edition 5.0 (J2EE™ 5.0) Specification ("Specification")

Version: 2.0

Status: Pre-FCS, Public Review

Release: Monday, June 20, 2005

Copyright 2005 Sun Microsystems, Inc.

4150 Network Circle, Santa Clara, California 95054, U.S.A.

All rights reserved.

## LIMITED EVALUATION LICENSE

Sun hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense), under Sun's applicable intellectual property rights to view, download, use and reproduce the Specification only for the purpose of internal evaluation. This includes (i) developing applications intended to run on an implementation of the Specification, provided that such applications do not themselves implement any portion(s) of the Specification, and (ii) excerpting brief portions of the Specification in oral or written communications which discuss the Specification provided that such excerpts do not in the aggregate constitute a significant portion of the Technology. No license of any kind is granted hereunder for any other purpose including, for example, creating and distributing implementations of the Specification, modifying the Specification (other than to the extent of your fair use rights), or distributing the Specification to third parties. Also, no right, title, or interest in or to any trademarks, service marks, or trade names of Sun or Sun's licensors is granted hereunder. If you wish to create and distribute an implementation of the Specification, a license for that purpose is available at <http://www.jcp.org>. The foregoing license is expressly conditioned on your acting within its scope, and will terminate immediately without notice from Sun if you breach the Agreement or act outside the scope of the licenses granted above. Java, and Java-related logos, marks and names are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

## DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED "AS IS". SUN MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT (INCLUDING AS A CONSEQUENCE OF ANY PRACTICE OR IMPLEMENTATION OF THE SPECIFICATION), OR THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE. This document does not represent any commitment to release or implement any portion of the Specification in any product. In addition, the Specification could include technical inaccuracies or typographical errors.

## LIMITATION OF LIABILITY

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, RELATED IN ANY WAY TO YOUR HAVING OR USING THE SPECIFICATION, EVEN IF SUN AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

## RESTRICTED RIGHTS LEGEND

U.S. Government: If this Specification is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this

is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

#### REPORT

If you provide Sun with any comments or suggestions concerning the Specification ("Feedback"), you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Sun a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose.

#### GOVERNING LAW

Any action relating to or arising out of this Agreement will be governed by California law and controlling U.S. federal law. The U.N. Convention for the International Sale of Goods and the choice of law rules of any jurisdiction will not apply.

Rev. May 9 2005

---

# Contents

<b>Java™ 2 Platform</b>	
<b>Enterprise Edition Specification, v5.0</b>	<b>i</b>
<b>J2EE.1 Introduction</b>	<b>1</b>
J2EE.1.1 Acknowledgements	2
J2EE.1.2 Acknowledgements for Version 1.3	2
J2EE.1.3 Acknowledgements for Version 1.4	3
J2EE.1.4 Acknowledgements for Version 5.0	3
<b>J2EE.2 Platform Overview</b>	<b>5</b>
J2EE.2.1 Architecture	5
J2EE.2.2 Application Components	6
J2EE.2.2.1 J2EE Server Support for Application Components	7
J2EE.2.3 Containers	8
J2EE.2.3.1 Container Requirements	8
J2EE.2.3.2 J2EE Servers	9
J2EE.2.4 Resource Adapters	9
J2EE.2.5 Database	9
J2EE.2.6 J2EE Standard Services	9
J2EE.2.6.1 HTTP	10
J2EE.2.6.2 HTTPS	10
J2EE.2.6.3 Java™ Transaction API (JTA)	10
J2EE.2.6.4 RMI-IIOP	10
J2EE.2.6.5 Java IDL	11
J2EE.2.6.6 JDBC™ API	11
J2EE.2.6.7 Java™ Persistence API	11
J2EE.2.6.8 Java™ Message Service (JMS)	11

J2EE.2.6.9	Java Naming and Directory Interface™ (JNDI) . . . . .	12
J2EE.2.6.10	JavaMail™ . . . . .	12
J2EE.2.6.11	JavaBeans™ Activation Framework (JAF). . . . .	12
J2EE.2.6.12	XML Processing . . . . .	12
J2EE.2.6.13	J2EE™ Connector Architecture . . . . .	12
J2EE.2.6.14	Security Services . . . . .	13
J2EE.2.6.15	Web Services . . . . .	14
J2EE.2.6.16	Management . . . . .	14
J2EE.2.6.17	Deployment . . . . .	14
J2EE.2.7	Interoperability . . . . .	15
J2EE.2.8	Flexibility of Product Requirements . . . . .	16
J2EE.2.9	J2EE Product Extensions . . . . .	16
J2EE.2.10	Platform Roles . . . . .	17
J2EE.2.10.1	J2EE Product Provider . . . . .	17
J2EE.2.10.2	Application Component Provider . . . . .	18
J2EE.2.10.3	Application Assembler . . . . .	18
J2EE.2.10.4	Deployer . . . . .	18
J2EE.2.10.5	System Administrator . . . . .	19
J2EE.2.10.6	Tool Provider. . . . .	19
J2EE.2.10.7	System Component Provider. . . . .	19
J2EE.2.11	Platform Contracts . . . . .	20
J2EE.2.11.1	J2EE APIs . . . . .	20
J2EE.2.11.2	J2EE Service Provider Interfaces (SPIs) . . . . .	20
J2EE.2.11.3	Network Protocols . . . . .	20
J2EE.2.11.4	Deployment Descriptors and Annotations . . . . .	21
J2EE.2.12	Changes in J2EE 1.3. . . . .	21
J2EE.2.13	Changes in J2EE 1.4. . . . .	21
J2EE.2.14	Changes in J2EE 5.0. . . . .	22
<b>J2EE.3</b>	<b>Security . . . . .</b>	<b>25</b>
J2EE.3.1	Introduction . . . . .	25
J2EE.3.2	A Simple Example . . . . .	26
J2EE.3.3	Security Architecture . . . . .	30
J2EE.3.3.1	Goals . . . . .	30
J2EE.3.3.2	Non Goals . . . . .	31
J2EE.3.3.3	Terminology . . . . .	32
J2EE.3.3.4	Container Based Security . . . . .	33
J2EE.3.3.5	Distributed Security. . . . .	34
J2EE.3.3.6	Authorization Model . . . . .	35

J2EE.3.3.7	HTTP Login Gateways .....	36
J2EE.3.3.8	User Authentication .....	36
J2EE.3.3.9	Lazy Authentication .....	39
J2EE.3.4	User Authentication Requirements .....	39
J2EE.3.4.1	Login Sessions .....	39
J2EE.3.4.2	Required Login Mechanisms .....	40
J2EE.3.4.3	Unauthenticated Users .....	41
J2EE.3.4.4	Application Client User Authentication .....	42
J2EE.3.4.5	Resource Authentication Requirements .....	43
J2EE.3.5	Authorization Requirements .....	44
J2EE.3.5.1	Code Authorization .....	44
J2EE.3.5.2	Caller Authorization .....	45
J2EE.3.5.3	Propagated Caller Identities .....	45
J2EE.3.5.4	Run As Identities .....	45
J2EE.3.6	Deployment Requirements .....	46
J2EE.3.7	Future Directions .....	46
J2EE.3.7.1	Auditing .....	46
J2EE.3.7.2	Instance-based Access Control .....	47
J2EE.3.7.3	User Registration .....	47
<b>J2EE.4</b>	<b>Transaction Management .....</b>	<b>49</b>
J2EE.4.1	Overview .....	49
J2EE.4.2	Requirements .....	51
J2EE.4.2.1	Web Components .....	51
J2EE.4.2.2	Transactions in Web Component Life Cycles .....	52
J2EE.4.2.3	Transactions and Threads .....	53
J2EE.4.2.4	Enterprise JavaBeans™ Components .....	54
J2EE.4.2.5	Application Clients .....	54
J2EE.4.2.6	Applet Clients .....	54
J2EE.4.2.7	Transactional JDBC™ Technology Support .....	54
J2EE.4.2.8	Transactional JMS Support .....	54
J2EE.4.2.9	Transactional Resource Adapter (Connector) Support .....	55
J2EE.4.3	Transaction Interoperability .....	55
J2EE.4.3.1	Multiple J2EE Platform Interoperability .....	55
J2EE.4.3.2	Support for Transactional Resource Managers .....	55
J2EE.4.4	Local Transaction Optimization .....	56
J2EE.4.4.1	Requirements .....	56
J2EE.4.4.2	A Possible Design .....	56
J2EE.4.5	Connection Sharing .....	57

J2EE.4.6	JDBC and JMS Deployment Issues . . . . .	58
J2EE.4.7	Two-Phase Commit Support . . . . .	59
J2EE.4.8	System Administration Tools . . . . .	59
<b>J2EE.5</b>	<b>Resources, Naming, and Injection . . . . .</b>	<b>61</b>
J2EE.5.1	Overview . . . . .	61
J2EE.5.1.1	Chapter Organization . . . . .	62
J2EE.5.1.2	Required Access to the JNDI Naming Environment . . . . .	63
J2EE.5.2	JNDI Naming Context . . . . .	63
J2EE.5.2.1	The Application Component's Environment . . . . .	63
J2EE.5.2.2	Sharing of Environment Entries . . . . .	64
J2EE.5.2.3	Annotations and Resource Injection . . . . .	65
J2EE.5.2.4	Annotations and Deployment Descriptors . . . . .	67
J2EE.5.3	Responsibilities by J2EE Role . . . . .	68
J2EE.5.3.1	Application Component Provider's Responsibilities . . . . .	68
J2EE.5.3.2	Application Assembler's Responsibilities . . . . .	69
J2EE.5.3.3	Deployer's Responsibilities . . . . .	69
J2EE.5.3.4	J2EE Product Provider's Responsibilities . . . . .	69
J2EE.5.4	Simple Environment Entries . . . . .	70
J2EE.5.4.1	Application Component Provider's Responsibilities . . . . .	70
J2EE.5.5	Enterprise JavaBeans™ (EJB) References . . . . .	75
J2EE.5.5.1	Application Component Provider's Responsibilities . . . . .	76
J2EE.5.5.2	Application Assembler's Responsibilities . . . . .	79
J2EE.5.5.3	Deployer's Responsibilities . . . . .	81
J2EE.5.5.4	J2EE Product Provider's Responsibilities . . . . .	81
J2EE.5.6	Resource Manager Connection Factory References . . . . .	82
J2EE.5.6.1	Application Component Provider's Responsibilities . . . . .	83
J2EE.5.6.2	Deployer's Responsibilities . . . . .	87
J2EE.5.6.3	J2EE Product Provider's Responsibilities . . . . .	88
J2EE.5.6.4	System Administrator's Responsibilities . . . . .	89
J2EE.5.7	Resource Environment References . . . . .	90
J2EE.5.7.1	Application Component Provider's Responsibilities . . . . .	90
J2EE.5.7.2	Deployer's Responsibilities . . . . .	91
J2EE.5.7.3	J2EE Product Provider's Responsibilities . . . . .	92
J2EE.5.8	Message Destination References . . . . .	92
J2EE.5.8.1	Application Component Provider's Responsibilities . . . . .	92
J2EE.5.8.2	Application Assembler's Responsibilities . . . . .	95
J2EE.5.8.3	Deployer's Responsibilities . . . . .	97
J2EE.5.8.4	J2EE Product Provider's Responsibilities . . . . .	97



J2EE.5.9	UserTransaction References . . . . .	97
J2EE.5.9.1	Application Component Provider's Responsibilities . . . . .	99
J2EE.5.9.2	J2EE Product Provider's Responsibilities . . . . .	99
J2EE.5.10	ORB References . . . . .	99
J2EE.5.10.1	Application Component Provider's Responsibilities . . . . .	100
J2EE.5.10.2	J2EE Product Provider's Responsibilities . . . . .	101
<b>J2EE.6</b>	<b>Application Programming Interface . . . . .</b>	<b>103</b>
J2EE.6.1	Required APIs . . . . .	103
J2EE.6.1.1	Java Compatible APIs . . . . .	103
J2EE.6.1.2	Java Optional Packages . . . . .	104
J2EE.6.2	Java 2 Platform, Standard Edition (J2SE) Requirements . . . . .	106
J2EE.6.2.1	Programming Restrictions . . . . .	106
J2EE.6.2.2	The J2EE Security Permissions Set . . . . .	106
J2EE.6.2.3	Listing of the J2EE Security Permissions Set . . . . .	107
J2EE.6.2.4	Additional Requirements . . . . .	108
J2EE.6.3	Enterprise JavaBeans™ (EJB) 3.0 Requirements . . . . .	120
J2EE.6.4	Servlet 2.4 Requirements . . . . .	120
J2EE.6.5	JavaServer Pages™ (JSP) 2.1 Requirements . . . . .	122
J2EE.6.6	Java™ Message Service (JMS) 1.1 Requirements . . . . .	122
J2EE.6.7	Java™ Transaction API (JTA) 1.0 Requirements . . . . .	123
J2EE.6.8	JavaMail™ 1.3 Requirements . . . . .	124
J2EE.6.9	JavaBeans™ Activation Framework 1.1 Requirements . . . . .	125
J2EE.6.10	J2EE™ Connector Architecture 1.5 Requirements . . . . .	126
J2EE.6.11	Web Services for J2EE 1.1 Requirements . . . . .	126
J2EE.6.12	Java™ API for XML-based RPC (JAX-RPC) 1.1 Requirements . . . . .	127
J2EE.6.13	Java™ API for XML Web Services (JAX-WS) 2.0 Requirements . . . . .	127
J2EE.6.14	Java™ Architecture for XML Binding (JAXB) 2.0 Requirements . . . . .	128
J2EE.6.15	SOAP with Attachments API for Java™ (SAAJ) 1.3 . . . . .	128
J2EE.6.16	Java™ API for XML Registries (JAXR) 1.0 Requirements . . . . .	128
J2EE.6.17	Java™ 2 Platform, Enterprise Edition Management API 1.0 Requirements . . . . .	129
J2EE.6.18	Java™ 2 Platform, Enterprise Edition Deployment API 1.1 Requirements . . . . .	129
J2EE.6.19	Java™ Authorization Service Provider Contract for Containers . . . . .	

(JACC) 1.0 Requirements . . . . .	129
J2EE.6.20 Debugging Support for Other Languages (JSR-45) Requirements 130	
J2EE.6.21 Standard Tag Library for JavaServer Pages™ (JSTL) 1.1 Re- quirements . . . . .	130
J2EE.6.22 Web Services Metadata for the Java™ Platform 1.0 Require- ments. . . . .	130
J2EE.6.23 JavaServer Faces™ 1.2 Requirements . . . . .	130
J2EE.6.24 Common Annotations for the Java™ Platform 1.0 Requirements 131	
J2EE.6.25 Streaming API for XML (StAX) 1.0 Requirements. . . . .	132
J2EE.6.26 Java Persistence API 1.0. . . . .	132
<b>J2EE.7 Interoperability . . . . .</b>	<b>133</b>
J2EE.7.1 Introduction to Interoperability . . . . .	133
J2EE.7.2 Interoperability Protocols . . . . .	134
J2EE.7.2.1 Internet and Web Protocols . . . . .	134
J2EE.7.2.2 OMG Protocols . . . . .	135
J2EE.7.2.3 Java Technology Protocols . . . . .	136
J2EE.7.2.4 Data Formats . . . . .	136
<b>J2EE.8 Application Assembly and Deployment . . . . .</b>	<b>139</b>
J2EE.8.1 Application Development Life Cycle. . . . .	140
J2EE.8.1.1 Component Creation . . . . .	141
J2EE.8.1.2 Application Assembly . . . . .	142
J2EE.8.1.3 Deployment . . . . .	143
J2EE.8.2 Library Support . . . . .	143
J2EE.8.2.1 Bundled Libraries . . . . .	143
J2EE.8.2.2 Installed Libraries . . . . .	144
J2EE.8.2.3 Library Conflicts . . . . .	145
J2EE.8.2.4 Library Resources . . . . .	145
J2EE.8.2.5 Dynamic Class Loading . . . . .	146
J2EE.8.2.6 Examples . . . . .	147
J2EE.8.3 Application Assembly . . . . .	148
J2EE.8.3.1 Assembling a J2EE Application . . . . .	148
J2EE.8.3.2 Adding and Removing Modules . . . . .	151
J2EE.8.4 Deployment . . . . .	151
J2EE.8.4.1 Deploying a Stand-Alone J2EE Module . . . . .	153
J2EE.8.4.2 Deploying a J2EE Application . . . . .	154

J2EE.8.4.3	Deploying a Library . . . . .	156
J2EE.8.5	J2EE Application XML Schema . . . . .	156
J2EE.8.6	Common J2EE XML Schema Definitions . . . . .	165
<b>J2EE.9</b>	<b>Application Clients . . . . .</b>	<b>205</b>
J2EE.9.1	Overview . . . . .	205
J2EE.9.2	Security . . . . .	205
J2EE.9.3	Transactions . . . . .	206
J2EE.9.4	Resources, Naming, and Injection . . . . .	207
J2EE.9.5	Application Programming Interfaces . . . . .	207
J2EE.9.6	Packaging and Deployment . . . . .	207
J2EE.9.7	J2EE Application Client XML Schema . . . . .	209
<b>J2EE.10</b>	<b>Service Provider Interface . . . . .</b>	<b>219</b>
<b>J2EE.11</b>	<b>Compatibility and Migration . . . . .</b>	<b>221</b>
<b>J2EE.12</b>	<b>Future Directions . . . . .</b>	<b>223</b>
J2EE.12.1	JNLP (Java™ Web Start) . . . . .	223
J2EE.12.2	J2EE SPI . . . . .	224
J2EE.12.3	Security APIs . . . . .	224
<b>Appendix J2EE.A:</b>	<b>Previous Version Deployment Descriptors. . . .</b>	<b>225</b>
J2EE.A.1	J2EE 1.4 Application XML Schema . . . . .	225
J2EE.A.2	Common J2EE 1.4 XML Schema Definitions . . . . .	233
J2EE.A.3	J2EE:application 1.3 XML DTD . . . . .	270
J2EE.A.4	J2EE:application 1.2 XML DTD . . . . .	276
J2EE.A.5	J2EE 1.4 Application Client XML Schema . . . . .	281
J2EE.A.6	J2EE:application-client 1.3 XML DTD . . . . .	289
J2EE.A.7	J2EE:application-client 1.2 XML DTD . . . . .	299
<b>Appendix J2EE.B:</b>	<b>Revision History . . . . .</b>	<b>307</b>
J2EE.B.1	Changes in Expert Draft 1 . . . . .	307
J2EE.B.1.1	Additional Requirements . . . . .	307
J2EE.B.1.2	Removed Requirements . . . . .	307
J2EE.B.1.3	Editorial Changes . . . . .	307
J2EE.B.2	Changes in Expert Draft 2 . . . . .	308
J2EE.B.2.1	Additional Requirements . . . . .	308
J2EE.B.2.2	Removed Requirements . . . . .	308
J2EE.B.2.3	Editorial Changes . . . . .	308

J2EE.B.3	Changes in Early Draft Review 1 .....	309
J2EE.B.3.1	Additional Requirements .....	309
J2EE.B.3.2	Removed Requirements .....	309
J2EE.B.3.3	Editorial Changes .....	309
J2EE.B.4	Changes in Early Draft Review 2 .....	309
J2EE.B.4.1	Additional Requirements .....	309
J2EE.B.4.2	Removed Requirements .....	310
J2EE.B.4.3	Editorial Changes .....	310
J2EE.B.5	Changes in Public Review Draft .....	310
J2EE.B.5.1	Additional Requirements .....	310
J2EE.B.5.2	Removed Requirements .....	310
J2EE.B.5.3	Editorial Changes .....	310
<b>Appendix J2EE.C: Related Documents .....</b>		<b>313</b>

# CHAPTER J2EE.1

---

## Introduction

**E**nterprises 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 Java™ 2 Platform, Enterprise Edition (J2EE™) reduces the cost and complexity of developing multitier, enterprise services. J2EE applications can be rapidly deployed and easily enhanced as the enterprise responds to competitive pressures.

J2EE achieves these benefits by defining a standard architecture with the following elements:

- **J2EE Platform** - A standard platform for hosting J2EE applications.
- **J2EE Compatibility Test Suite** - A suite of compatibility tests for verifying that a J2EE platform product complies with the J2EE platform standard.

- **J2EE Reference Implementation** - A reference implementation for prototyping J2EE applications and for providing an operational definition of the J2EE platform.
- **J2EE BluePrints** - A set of best practices for developing multitier, thin-client services.

This document is the J2EE platform specification. It sets out the requirements that a J2EE platform product must meet.

## **J2EE.1.1 Acknowledgements**

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

## **J2EE.1.2 Acknowledgements for Version 1.3**

Version 1.3 of this specification grew out of discussions with our partners during the creation of version 1.2, as well as meetings with those partners subsequent to the final release of version 1.2. Version 1.3 was created under the Java Community Process as JSR-058. The JSR-058 Expert Group included representatives from the following companies and organizations: Allaire, BEA Systems, Bluestone Software, 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.

### **J2EE.1.3 Acknowledgements for 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).

### **J2EE.1.4 Acknowledgements for Version 5.0**

Version 5.0 (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), 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 Dudley (Individual), Francois Exertier (INRIA), Evan Ireland (Sybase, Inc.), Michael Keith (Oracle Corporation), 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.), Ricardo Morin (Intel Corporation), 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 Tiwari (Individual), Sivasundaram Umapathy (Individual), Steve Weston (Cap Gemini), and Umit Yalcinalp (SAP AG). Once again, my colleagues at Sun provided invaluable assistance: Roberto Chinnici provided draft proposals for many issues related to resource injection.



# CHAPTER J2EE.2

---

## Platform Overview

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

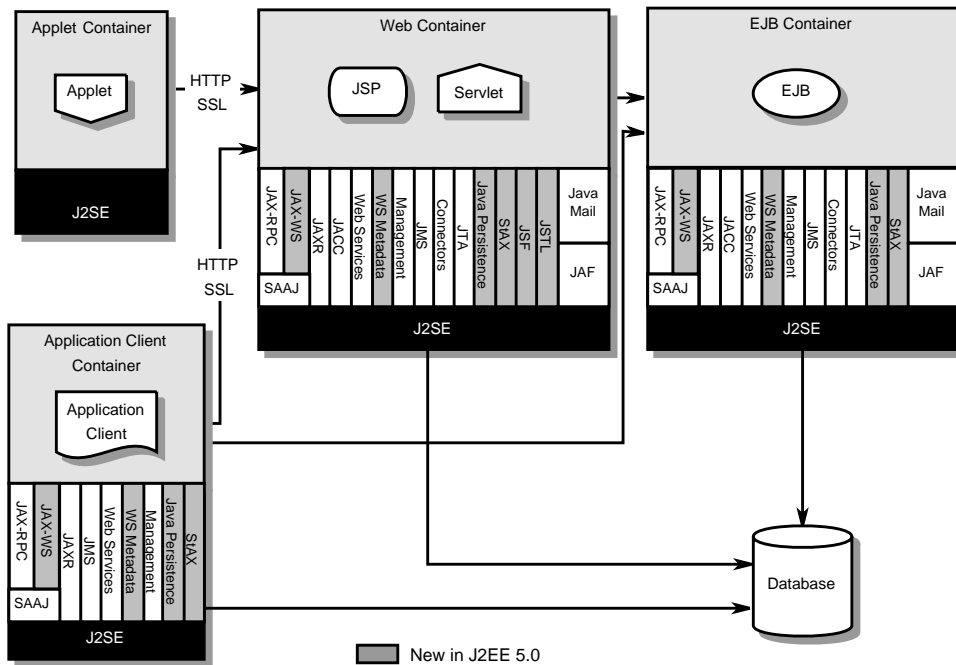
### J2EE.2.1 Architecture

The required relationships of architectural elements of the J2EE platform are shown in **Figure J2EE.2-1**. 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 J2EE 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 Java Message Service (JMS) APIs to Application Clients, as well as the other services represented. All these services are explained below. See Section J2EE.2.6, “J2EE Standard Services”.

The arrows represent required access to other parts of the J2EE platform. The Application Client Container provides Application Clients with direct access to the J2EE required Database through the Java API for connectivity with database systems, the JDBC™ API. Similar access to databases is provided to JSP pages and servlets by the Web Container, and to enterprise beans by the EJB Container.

As indicated, the APIs of the Java™ 2 Platform, Standard Edition (J2SE™), are supported by J2SE runtime environments for each type of application component.



**Figure J2EE.2-1** J2EE Architecture Diagram

The following sections describe the J2EE Platform requirements for each kind of J2EE platform element.

## J2EE.2.2 Application Components

The J2EE runtime environment defines four application component types that a J2EE product must support:

- Application clients are Java programming language programs that are typically GUI programs that execute on a desktop computer. Application clients offer a user experience similar to that of native applications, and have access to all of the facilities of the J2EE middle tier.
- Applets are GUI components that typically execute in a web browser, but can execute in a variety of other applications or devices that support the applet

programming model. Applets can be used to provide a powerful user interface for J2EE applications. (Simple HTML pages can also be used to provide a more limited user interface for J2EE applications.)

- Servlets, JSP pages, JSF applications, filters, and web event listeners typically execute in a web container and may respond to HTTP requests from web clients. Servlets, JSP pages, JSF 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 JavaServer Pages™ technology or JavaServer™ 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 J2EE specifications.
- Enterprise JavaBeans™ (EJB) components execute in a managed environment that supports transactions. Enterprise beans typically contain the business logic for a J2EE application. Enterprise beans may directly provide web services using the SOAP/HTTP protocol.

### **J2EE.2.2.1 J2EE Server Support for Application Components**

The J2EE 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 J2EE server:

- Components that are deployed, managed, and executed on a J2EE server. These components include web components and Enterprise JavaBeans components. See the separate specifications for these components.
- Components that are deployed and managed on a J2EE 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 Chapter J2EE.9, “Application Clients” for a descrip-

tion of Application Clients.

### **J2EE.2.3 Containers**

Containers provide the runtime support for J2EE application components. Containers provide a federated view of the underlying J2EE APIs to the application components. J2EE application components never interact directly with other J2EE 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 J2EE 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 J2EE product will provide a container for each application component type: application client container, applet container, web component container, and enterprise bean container.

#### **J2EE.2.3.1 Container Requirements**

This specification requires that containers provide a Java Compatible™ runtime environment, as defined by the Java 2 Platform, Standard Edition, v5.0 specification (J2SE). The applet container may use the Java Plugin product to provide this environment, or it may provide it natively. The use of applet containers providing 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 J2EE Product Provider. See the description of the Product Provider role in Section J2EE.2.10.1, “J2EE Product Provider“.

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

### **J2EE.2.3.2 J2EE Servers**

Underlying a J2EE container is the server of which it is a part. A J2EE Product Provider typically implements the J2EE server-side functionality using an existing transaction processing infrastructure in combination with Java 2 Platform, Standard Edition (J2SE) technology. The J2EE client functionality is typically built on J2SE technology.

### **J2EE.2.4 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 J2EE platform either by implementing one of the J2EE standard 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 J2EE platform through the J2EE service provider interfaces (J2EE SPI). A resource adapter that uses the J2EE SPIs to attach to the J2EE platform will be able to work with all J2EE products.

### **J2EE.2.5 Database**

The J2EE 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.

### **J2EE.2.6 J2EE Standard Services**

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

### **J2EE.2.6.1 HTTP**

The HTTP client-side API is defined by the `java.net` package. The HTTP server-side API is defined by the `servlet`, `JSP`, and `JSF` interfaces and by the web services support that is a part of the J2EE platform.

### **J2EE.2.6.2 HTTPS**

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

### **J2EE.2.6.3 Java™ Transaction API (JTA)**

The Java Transaction API consists of two parts:

- An application-level demarcation interface that is used by the container and application components to demarcate transaction boundaries.
- An interface between the transaction manager and a resource manager used at the J2EE SPI level (in a future release).

### **J2EE.2.6.4 RMI-IIOP**

The RMI-IIOP subsystem is composed of APIs that allow for the use of RMI-style programming that is independent of the underlying protocol, as well as an implementation of those APIs that supports both the J2SE native RMI protocol (JRMP) and the CORBA IIOP protocol. J2EE 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 spec for details). Such CORBA services would typically be defined by components that live outside of a J2EE product, usually in a legacy system. Only J2EE application clients are required to be able to define their own CORBA services directly, using the RMI-IIOP APIs. Typically such CORBA objects would be used for callbacks when accessing other CORBA objects.

J2EE applications are required to use the RMI-IIOP APIs, specifically the narrow method of `javax.rmi.PortableRemoteObject`, when accessing Enterprise JavaBeans components, as described in the EJB specification. This allows enterprise beans to be protocol independent. Note that the most common use of the narrow method is not needed when using resource injection instead of JNDI

lookups; the container will perform the lookup for the application before injecting the object reference. J2EE products must be capable of exporting enterprise beans using the IIOP protocol, and accessing enterprise beans using the IIOP protocol, as specified in the EJB specification. The ability to use the IIOP protocol is required to enable interoperability between J2EE products, however a J2EE product may also use other protocols.

#### **J2EE.2.6.5      Java IDL**

Java IDL allows J2EE application components to invoke external CORBA objects using the IIOP protocol. These CORBA objects may be written in any language and typically live outside a J2EE product. J2EE applications may use Java IDL to act as clients of CORBA services, but only J2EE application clients are required to be allowed to use Java IDL directly to present CORBA services themselves.

#### **J2EE.2.6.6      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 J2EE platform. Support for the service provider interface is not required in J2EE products. Instead, JDBC drivers should be packaged as resource adapters that use the facilities of the Connector API to interface with a J2EE product.

#### **J2EE.2.6.7      Java™ Persistence API**

---

**Note** – Need to fill this in with a description of the Java Persistence API being developed by the EJB 3.0 expert group.

---

#### **J2EE.2.6.8      Java™ Message Service (JMS)**

The Java Message Service is a standard API for messaging that supports reliable point-to-point messaging as well as the publish-subscribe model. This specification requires a JMS provider that implements both point-to-point messaging as well as publish-subscribe messaging.

### **J2EE.2.6.9      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.

### **J2EE.2.6.10     JavaMail™**

Many Internet applications require the ability to send email notifications, so the J2EE platform includes the JavaMail API along with a JavaMail service provider that allows an application component to send Internet mail. The JavaMail API has two parts: an application-level interface used by the application components to send mail, and a service provider interface used at the J2EE SPI level.

### **J2EE.2.6.11     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, so it must be included as well.

### **J2EE.2.6.12     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.

### **J2EE.2.6.13     J2EE™ Connector Architecture**

The Connector architecture is a J2EE SPI that allows resource adapters that support access to Enterprise Information Systems to be plugged in to any J2EE product. The Connector architecture defines a standard set of system-level contracts between a J2EE server and a resource adapter. The standard contracts include:

- A connection management contract that lets a J2EE server pool connections to an underlying EIS, and lets application components connect to an EIS. This



leads to a scalable application environment that can support a large number of clients requiring access to EIS systems.

- A transaction management contract between the transaction manager and an EIS that supports transactional access to EIS resource managers. This contract lets a J2EE server use a transaction manager to manage transactions across multiple resource managers. This contract also supports transactions that are managed internal to an EIS resource manager without the necessity of involving an external transaction manager.
- A security contract that enables secure access to an EIS. This contract provides support for a secure application environment, which reduces security threats to the EIS and protects valuable information resources managed by the EIS.
- 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 J2EE server via a resource adapter.
- A contract that allows a resource adapter to propagate an imported transaction context to the J2EE 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.

#### **J2EE.2.6.14    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. The Java™ Authorization Service Provider Contract for Containers (JACC) defines a contract between a J2EE

application server and an authorization service provider, allowing custom authorization service providers to be plugged into any J2EE product.

#### **J2EE.2.6.15    Web Services**

J2EE 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. The Java API for XML Web Services (JAX-WS) and the Java API for XML-based RPC (JAX-RPC) both provide support for web service calls using the SOAP/HTTP protocol. JAX-WS is the primary API for web services and is a follow-on to JAX-RPC. JAX-WS offers extensive web services functionality, with support for multiple bindings/protocols and RESTful web services. JAX-WS and JAX-RPC are fully interoperable when using the SOAP 1.1 over HTTP protocol as constrained by the WS-I Basic Profile specification.

JAX-WS and the Java Architecture for XML Binding (JAXB) define the mapping between Java classes and XML as used in SOAP calls, and provides support for 100% of XML Schema. The SOAP with Attachments API for Java (SAAJ) provides support for manipulating low level SOAP messages. The Web Services for J2EE specification fully defines the deployment of web service clients and web service endpoints in J2EE, 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 Java API for XML Registries (JAXR) provides client access to XML registry servers.

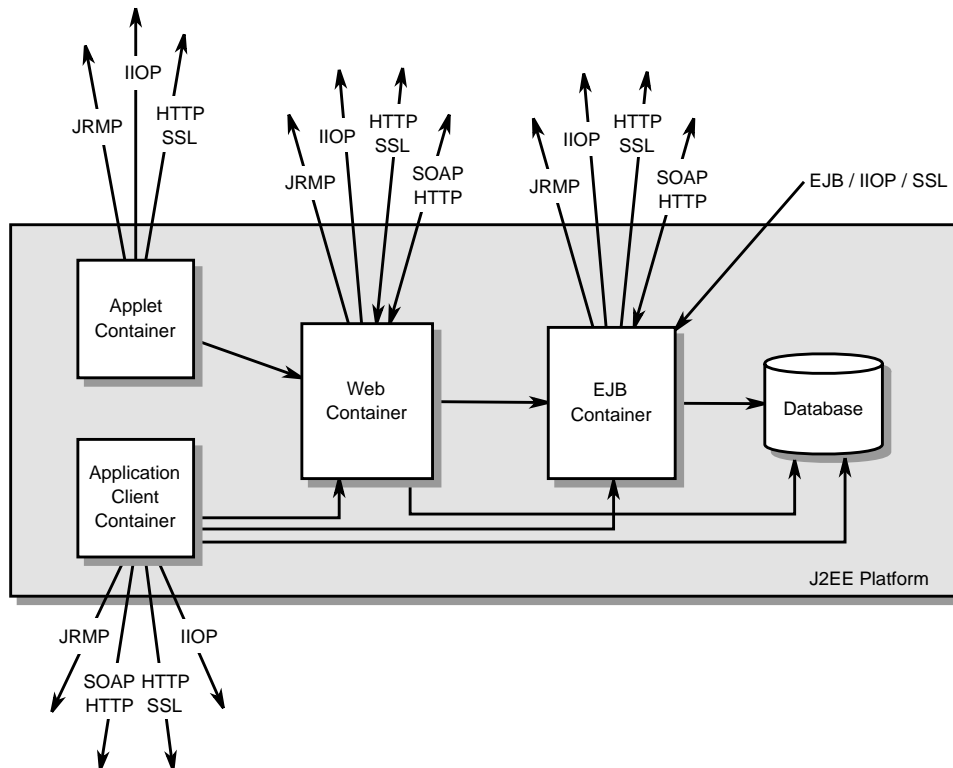
#### **J2EE.2.6.16    Management**

The Java 2 Platform, Enterprise Edition Management Specification defines APIs for managing J2EE servers using a special management enterprise bean. The Java™ Management Extensions (JMX) API is also used to provide some management support.

#### **J2EE.2.6.17    Deployment**

The Java 2 Platform, Enterprise Edition Deployment Specification defines a contract between deployment tools and J2EE products. The J2EE products provide plug-in components that run in the deployment tool and allow the deployment tool to deploy

applications into the J2EE product. The deployment tool provides services used by these plug-in components.



**Figure J2EE.2-2** J2EE Interoperability

## J2EE.2.7 Interoperability

Many of the APIs described above provide interoperability with components that are not a part of the J2EE platform, such as external web or CORBA services.

**Figure J2EE.2-2** illustrates the interoperability facilities of the J2EE platform. (The directions of the arrows indicate the client/server relationships of the components.)

## **J2EE.2.8      Flexibility of Product Requirements**

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

A typical low end J2EE 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 J2EE product might split the server components into multiple servers, each of which can be distributed and load-balanced across a collection of machines. This specification does not prescribe or preclude any of these configurations.

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

## **J2EE.2.9      J2EE Product Extensions**

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

However, many other extensions are allowed. A J2EE product may provide additional Java APIs, either other Java optional packages or other (appropriately named) packages. A J2EE product may include support for additional protocols or services not specified here. A J2EE product may support applications written in other languages, or may support connectivity to other platforms or applications.

Of course, portable applications will not make use of any platform extensions. Applications that do make use of facilities not required by this specification will

be less portable. Depending on the facility used, the loss of portability may be minor or it may be significant. The document *Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition* supplies information to help application developers construct portable applications, and contains advice on how best to manage the use of non-portable code when the use of such facilities is necessary.

We expect J2EE 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.

## **J2EE.2.10 Platform Roles**

This section describes typical Java 2 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. Relevant subsets of these roles are described in the EJB, JSP, and servlet specifications included herein as parts of the J2EE specification.

### **J2EE.2.10.1 J2EE Product Provider**

A J2EE Product Provider is the implementor and supplier of a J2EE product that includes the component containers, J2EE platform APIs, and other features defined in this specification. A J2EE Product Provider is typically an operating system vendor, a database system vendor, an application server vendor, or a web server vendor. A J2EE Product Provider must make available the J2EE APIs to the application components through containers. A Product Provider frequently bases their implementation on an existing infrastructure.

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

A J2EE Product Provider must provide application deployment and management tools. Deployment tools enable a Deployer (see Section J2EE.2.10.4, "Deployer") to deploy application components on the J2EE product. Management

tools allow a System Administrator (see Section J2EE.2.10.5, “System Administrator”) to manage the J2EE product and the applications deployed on the J2EE product. The form of these tools is not prescribed by this specification.

### **J2EE.2.10.2    Application Component Provider**

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

### **J2EE.2.10.3    Application Assembler**

The Application Assembler takes a set of components developed by Application Component Providers and assembles them into a complete J2EE application delivered in the form of 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.

### **J2EE.2.10.4    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 J2EE 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 J2EE 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 J2EE product, the Deployer may provide simple application code that wraps an enterprise bean's business methods, or customizes the appearance of a JSP 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 J2EE container.

#### **J2EE.2.10.5    System Administrator**

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

#### **J2EE.2.10.6    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 J2EE 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.

#### **J2EE.2.10.7    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 Java Authorization Service Provider Contract for Containers specification.

## **J2EE.2.11 Platform Contracts**

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

### **J2EE.2.11.1 J2EE APIs**

The J2EE APIs define the contract between the J2EE application components and the J2EE platform. The contract specifies both the runtime and deployment interfaces.

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

### **J2EE.2.11.2 J2EE Service Provider Interfaces (SPIs)**

The J2EE Service Provider Interfaces (SPIs) define the contract between the J2EE platform and service providers that may be plugged into a J2EE product. The Connector APIs define service provider interfaces for integrating resource adapters with a J2EE application server. Resource adapter components implementing the Connector APIs are called Connectors. The J2EE Authorization APIs define service provider interfaces for integrating security authorization mechanisms with a J2EE application server.

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

### **J2EE.2.11.3 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 J2EE product technology. See Chapter J2EE.7, “Interoperability” for details on the network protocol support required for interoperability.

The J2EE Product Provider is required to publish the installed application components on the industry-standard protocols. This specification defines the



mapping of servlets and JSP pages to the HTTP and HTTPS protocols, and the mapping of EJB components to IIOP and SOAP protocols.

#### **J2EE.2.11.4    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 J2EE Product Provider is required to provide a deployment tool that interprets the J2EE deployment descriptors and class file annotations and allows the Deployer to map the application component's requirements to the capabilities of a specific J2EE product and environment.

#### **J2EE.2.12    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.

#### **J2EE.2.13    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.

## **J2EE.2.14 Changes in J2EE 5.0**

The focus of J2EE 5.0 is ease of development. To simplify the development process for programmers just starting with J2EE, or developing small to medium applications, we've made extensive use of Java language annotations that were introduced by J2SE 5.0. Annotations reduce or eliminate the need to deal with J2EE 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 into J2EE components. Resource 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. Resource injection also works with deployment descriptors to allow the deployer to customize or override resource settings specified in the application's source code.

Annotations are made even more effective by providing better defaults. Better default behavior and better default configuration allows most applications to get what 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.

Major additions to J2EE 5.0 include the JSTL and JSF technologies that simplify development of web applications, and the Java Persistence API being developed by the EJB 3.0 expert group that greatly simplifies mapping Java objects to databases.

---

**Note** – This section needs to be expanded.

---



# CHAPTER J2EE.3

---

## Security

**T**his chapter describes the security requirements for the Java™ 2 Platform, Enterprise Edition (J2EE) that must be satisfied by J2EE products.

In addition to the J2EE requirements, each J2EE Product Provider will determine the level of security and security assurances that will be provided by their implementation.

### J2EE.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 J2EE platform requirements address security requirements, and identifies requirements that may be addressed by J2EE Product Providers. Finally, issues being considered for future versions of this specification are briefly mentioned in Section J2EE.3.7, “Future Directions”.

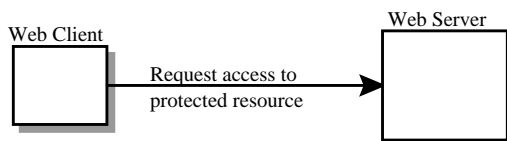
## J2EE.3.2 A Simple Example

The security behavior of a J2EE environment may be better understood by examining what happens in a simple application with a web client, 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.

### Step 1: Initial Request

The web client requests the main application URL, shown in **Figure J2EE.3-1**.

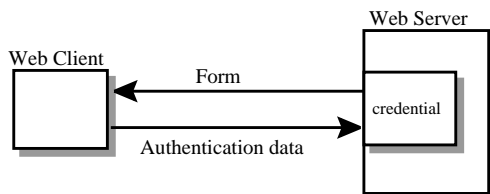


**Figure J2EE.3-1** Initial Request

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.

### Step 2: 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 **Figure J2EE.3-2**.

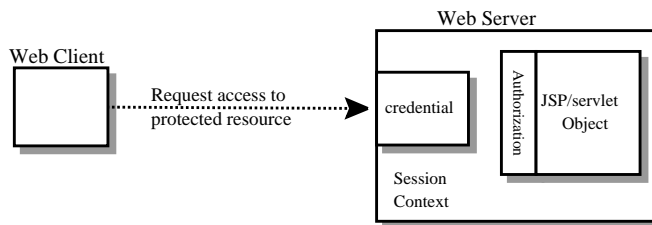


**Figure J2EE.3-2** Initial Authentication

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.

### Step 3: 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. **Figure J2EE.3-3** shows this process.



**Figure J2EE.3-3** URL Authorization

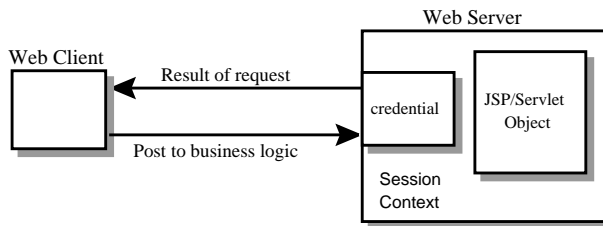
The web server's evaluation stops with an “is authorized” outcome when the web server is able to map the user to a role. A “not authorized” outcome is reached if the web server is unable to map the user to any of the permitted

roles.



#### Step 4: Fulfilling the Original Request

If the user is authorized, the web server returns the result of the original URL-request, as shown in **Figure J2EE.3-4**.

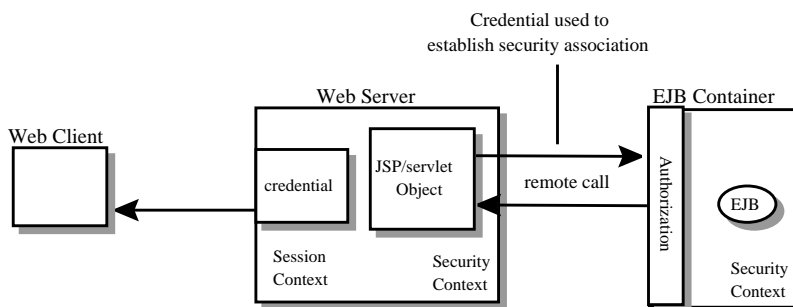


**Figure J2EE.3-4** Fulfilling the Original Request

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.

#### Step 5: 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 **Figure J2EE.3-5**). The association is implemented as two related security contexts, one in the web server and one in the EJB container.



**Figure J2EE.3-5** Invoking an Enterprise Bean Business Method

The EJB 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 EJB

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.

### **J2EE.3.3 Security Architecture**

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

#### **J2EE.3.3.1 Goals**

The following are goals for the J2EE security architecture:

1. Portability: The J2EE security architecture must support the Write Once, Run Anywhere™ application property.
2. Transparency: Application Component Providers should not have to know anything about security to write an application.
3. Isolation: The J2EE platform 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 J2EE applications.

4. 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.
5. Flexibility: The security mechanisms and declarations used by applications under this specification should not impose a particular security policy, but facil-

itate the implementation of security policies specific to the particular J2EE installation or application.

6. **Abstraction:** An application component's security requirements will be logically specified using deployment descriptors. 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 deployment descriptor should document which security properties can be modified and which cannot.
7. **Independence:** Required security behaviors and deployment contracts should be implementable using a variety of popular security technologies.
8. **Compatibility testing:** The J2EE security requirements architecture must be expressed in a manner that allows for an unambiguous determination of whether or not an implementation is compatible.
9. **Secure interoperability:** Application components executing in a J2EE product must be able to invoke services provided in a J2EE 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.

### **J2EE.3.3.2      Non Goals**

The following are not goals for the J2EE 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 J2EE 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 J2EE product.

### **J2EE.3.3.3 Terminology**

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

#### **Principal**

A *principal* is an entity that can be authenticated by an authentication protocol in a security service that is deployed in an enterprise. A principal is identified using a *principal name* and authenticated using *authentication data*. The content and format of the principal name and the authentication data 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 J2EE Product Provider.

The J2EE 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 J2EE product services. A principal acquires a credential upon authentication, or from another principal that allows its creden-

tial 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.

#### **J2EE.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 J2EE environment. A container provides two kinds of security (discussed in the following sections):

- Declarative security
- Programmatic security

##### ***J2EE.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 a form external to the application. The deployment descriptor is the primary vehicle for declarative security in the J2EE platform.

A deployment descriptor is a contract between an Application Component Provider and a Deployer or Application Assembler. It 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 deployment descriptor's 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 deployment descriptor.

At runtime, the container uses the security policy security structure derived from the deployment descriptor and configured by the Deployer to enforce authorization (see Section J2EE.3.3.6, "Authorization Model").

##### ***J2EE.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 required by this specification consists of two methods of the

EJB `EJBContext` interface and two methods of the servlet `HttpServletRequest` interface:

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

These methods allow components to make business logic decisions based on the security role of the caller or remote user. 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.)

### **J2EE.3.3.5      Distributed Security**

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

Such threats can be countered by using a *secure association* to secure communications. A secure association is shared security state information 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 J2EE, secure associations for a component are typically established by a container. Secure associations for web

access are specified here. Secure associations for access to enterprise beans are described in the EJB specification.

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 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.

### **J2EE.3.3.6 Authorization Model**

The J2EE authorization model is based on the concept of security roles. A security role is a logical grouping of users that is defined by an Application Component Provider or Assembler. 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 the enterprise bean deployment descriptor. An enterprise bean method can be associated with a `method-permission` element in the deployment descriptor. The `method-permission` element contains a list of methods that can be accessed by a given security role. If the calling principal is in one of the security roles allowed access to a method, the principal is allowed to execute the method. Conversely, if the calling principal is in none of the roles, the caller is not allowed to execute the method. Access to web resources can be protected in a similar manner.

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

#### ***J2EE.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. 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 J2EE 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 a trusted third party, such as a directory service or a security service.

### **J2EE.3.3.7 HTTP Login Gateways**

Secure interoperability between enterprise beans in different security policy domains is addressed in the EJB 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 Section J2EE.3.3.8.1, "Authentication by Web Clients."

### **J2EE.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 J2EE application components. An end user can authenticate using either of the two supported client types:

- Web client
- Application client



### ***J2EE.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, it is a required feature of the J2EE 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 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.

#### ***J2EE.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 they access.
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).

#### ***J2EE.3.3.8.3 Login Session***

In the J2EE 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.<sup>1</sup>

#### ***J2EE.3.3.8.4 Authentication by Application Clients***

Application clients (described in detail in Chapter J2EE.9, "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 J2EE application component types, execute in a managed environment that is provided by an appropriate container. Application clients are expected to have access to a graphical display and input device, and are expected to communicate with a human user.

Application clients are used to authenticate end users to the J2EE platform, when the users access protected web resources or enterprise beans.

### **J2EE.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 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.

## **J2EE.3.4 User Authentication Requirements**

The J2EE Product Provider must meet the following requirements concerning user authentication.

### **J2EE.3.4.1 Login Sessions**

All J2EE 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

---

<sup>1</sup> 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.

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 J2EE 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.

### **J2EE.3.4.2 Required Login Mechanisms**

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

#### ***J2EE.3.4.2.1 HTTP Basic Authentication***

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

#### ***J2EE.3.4.2.2 SSL Mutual Authentication***

SSL 3.0<sup>2</sup> and the means to perform mutual (client and server) certificate based authentication are required by this specification.

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

- 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

---

<sup>2</sup> The SSL 3.0 specification is available at: <http://home.netscape.com/eng/ss13>

- `SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA`

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

#### ***J2EE.3.4.2.3 Form Based Login***

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

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

#### **J2EE.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 `HttpServletRequest` method `getUserPrincipal`. This is different than the corresponding result for EJB containers. The EJB specification requires that the `EJBContext` method `getCallerPrincipal` always return a valid `Principal` object. The method can never return `null`.

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 J2EE product must provide a principal for use in the call.

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

- Always use a single distinguished principal.
- Use a different distinguished principal per server, or per session, or per application.
- 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 J2EE product should choose a principal to represent unauthenticated users, although future versions of this specification may add requirements in this area. Note that the EJB specification does include requirements in this area when using the EJB interoperability protocol. Applications are encouraged to use the Run As capability in cases where the web component may be unauthenticated and needs to call EJB components.

#### **J2EE.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 J2EE product's authentication system, to provide a single signon capability, or the container may authenticate the user when the application is started. The container may delay authentication until 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 Section J2EE.9.7, "J2EE 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 execute in an environment controlled by a J2SE security manager and are subject to the security permissions defined in Section J2EE.6.2, "Java 2 Platform, Standard Edition (J2SE) 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 J2EE product be able to relate

these identities. Additional application client requirements are described in Chapter J2EE.9.7 of this specification.

### J2EE.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 J2EE 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 J2EE 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 J2EE 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:

3. **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.
4. **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 mecha-

nism. For example, Kerberos supports a mechanism for the delegation of authentication. (Refer to the Kerberos v5 specification for more details.)

**5. Credentials Mapping.** This technique may be used when an application server and an EIS support different authentication domains. For example:

- a. The initiating principal may have been authenticated and have public key certificate-based credentials.
- b. 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.

## **J2EE.3.5 Authorization Requirements**

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

### **J2EE.3.5.1 Code Authorization**

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

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

A future version of the J2EE deployment descriptor definition (see Chapter J2EE.8, “Application Assembly and Deployment”) may make it possible to express additional permissions that a component needs for access.



### **J2EE.3.5.2      Caller Authorization**

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

### **J2EE.3.5.3      Propagated Caller Identities.**

It must be possible to configure a J2EE 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 J2EE product, the principal name returned by the `EJBContext` method `getCallerPrincipal` must be the same as that returned by the first enterprise bean in the call chain. If the first enterprise bean in the call chain is called by a servlet or JSP page, the principal name must be the same as that returned by the `HttpServletRequest` method `getUserPrincipal` in the calling servlet or JSP page. (However, if the `HttpServletRequest` method `getUserPrincipal` returns `null`, the principal used in calls to enterprise beans is not specified by this specification, although it must still be possible to configure enterprise beans to be callable by such components.)

Note that this does not require delegation of credentials, only identification of the caller. 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 J2EE product has been configured to propagate caller identity.

### **J2EE.3.5.4      Run As Identities**

J2EE products must also support the Run As capability that allows the Application Component Provider and the Deployer to specify an identity under which an enterprise bean or web component must run. In this case it is the Run As identity that is propagated to subsequent EJB 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 Java Authorization Contract for Containers specification does specify the relationship between the Run As identity and the access control context used by the J2SE security manager.

## **J2EE.3.6      Deployment Requirements**

All J2EE products must implement the access control semantics described in the EJB, JSP, and servlet specifications, and provide a means of mapping the deployment descriptor security roles to the actual roles exposed by a J2EE product.

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

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

Application developers will need to specify (in the application's deployment descriptors) the security requirements of an application in which some components may be accessed by unauthenticated users as well as authenticated users (as described above in Section J2EE.3.4.3, "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.

## **J2EE.3.7      Future Directions**

### **J2EE.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.

### **J2EE.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.

### **J2EE.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.



# CHAPTER J2EE.4

---

## Transaction Management

**T**his chapter describes the required Java™ 2 Platform, Enterprise Edition (J2EE) transaction management and runtime environment.

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

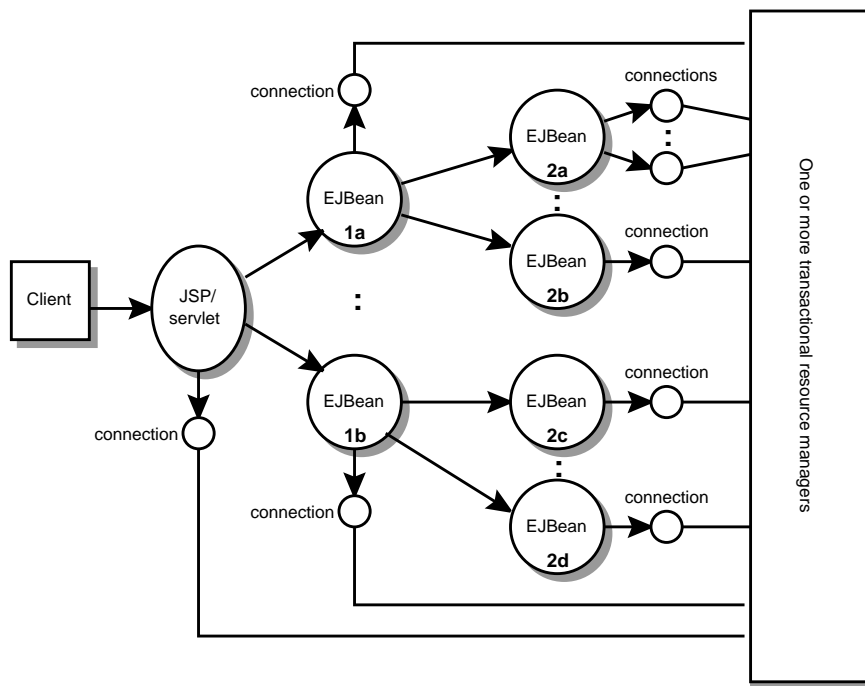
The following components are considered transactional resources and must behave as specified here:

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

### J2EE.4.1 Overview

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

For example, in **Figure J2EE.4-1**, the call tree starts from a servlet or JSP page accessing multiple enterprise beans, which in turn may access other enterprise beans. The components access resource managers via connections.



**Figure J2EE.4-1** Servlets/JSP Pages Accessing Enterprise Beans

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

For example, the application may require that all the components in **Figure J2EE.4-1** access resources as part of a single transaction. The Platform Provider must provide the transaction capabilities to support such a scenario.

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

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

## **J2EE.4.2      Requirements**

This section defines the transaction support requirements of J2EE Products that must be supported by Product Providers.

### **J2EE.4.2.1      Web Components**

Servlets and JSP pages demarcate a transaction using the `javax.transaction.UserTransaction` interface which is defined in the JTA 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).

Servlet filters and web application event listeners must not demarcate transactions using the `javax.transaction.UserTransaction` interface. Servlet filters may use transactional resources in a local transaction mode 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.

#### ***J2EE.4.2.1.1      Transaction Requirements***

The J2EE platform must meet the following requirements:

- The J2EE 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`.
- If a web component invokes an enterprise bean from a thread associated with a JTA transaction, the J2EE platform must propagate the transaction context with the enterprise bean invocation. Whether the target enterprise bean will be invoked in this transaction context or not is determined by the rules defined in the EJB specification.

Note that this transaction propagation requirement applies only to invocations of enterprise beans in the same J2EE product instance<sup>1</sup> as the invoking component. Invocations of enterprise beans in another J2EE product instance (for

example, using the EJB interoperability protocol) need not propagate the transaction context. See the EJB specification for details.

- If a web component accesses a transactional resource manager from a thread associated with a JTA transaction, the J2EE platform must ensure that the resource access is included as part of the JTA transaction.
- If a web component creates a thread, the J2EE platform must ensure that the newly created thread is not associated with any JTA transaction.

#### ***J2EE.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 JSP page.

#### **J2EE.4.2.2 Transactions in Web Component Life Cycles**

Transactions may not span web requests from a client. A web component starts a transaction in the `service` method of a servlet (or, for a JSP page, the `service` method of the equivalent JSP page Implementation Class) and it must be completed before the `service` method returns. Returning from the `service` method with an

---

<sup>1</sup>. A product instance corresponds to a single installation of a J2EE 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.



active transaction context is an error. The web container is required to detect this error and abort the transaction.

### **J2EE.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.

- JTA transactions 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 JTA transactions.
- Transactional resources may be acquired and released by a thread other than the service method thread, but should not be shared between threads.
- Transactional resource objects (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 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 EJB specification.

#### **J2EE.4.2.4      Enterprise JavaBeans™ Components**

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

#### **J2EE.4.2.5      Application Clients**

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

#### **J2EE.4.2.6      Applet Clients**

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

#### **J2EE.4.2.7      Transactional JDBC™ Technology Support**

A J2EE product must support a JDBC technology database as a transactional resource manager. The platform must enable transactional JDBC API access from 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 J2EE 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 Java Transaction API's XA interfaces (in the `javax.transaction.xa` package), then the J2EE product must be capable of using the XA interfaces provided by the JDBC driver to accomplish two-phase commit operations. The J2EE 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.

#### **J2EE.4.2.8      Transactional JMS Support**

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

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

#### **J2EE.4.2.9 Transactional Resource Adapter (Connector) Support**

A J2EE 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, JSP pages, and enterprise beans.

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

### **J2EE.4.3 Transaction Interoperability**

#### **J2EE.4.3.1 Multiple J2EE Platform Interoperability**

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

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

#### **J2EE.4.3.2 Support for Transactional Resource Managers**

This specification requires all J2EE products to support the `javax.transaction.xa.XAResource` interface, as specified in the Connector specification. This specification also requires all J2EE 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 JMS 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 JMS sessions, one or more enterprise beans, and multiple resource adapters supporting the `XATransaction` mode in a single JTA transaction.

## **J2EE.4.4     Local Transaction Optimization**

### **J2EE.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 J2EE application.

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

### **J2EE.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.

## **J2EE.4.5 Connection Sharing**

When multiple connections acquired by a J2EE 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 J2EE applications are considered potentially shared or shareable. A J2EE 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.

J2EE application components may use the optional deployment descriptor element `res-sharing-scope` to indicate whether a connection to a resource manager is shareable or unshareable. Containers must assume connections to be shareable if no deployment hint is provided. Section J2EE.9.7, “J2EE Application

Client XML Schema”, the EJB specification, and the servlet specification provide descriptions of the deployment descriptor element.

J2EE 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.

## **J2EE.4.6      JDBC and JMS Deployment Issues**

The JDBC transaction requirements in Section J2EE.4.2.7, “Transactional JDBC™ Technology Support” and the JMS transaction requirements in Section J2EE.4.2.8, “Transactional JMS Support” may impose some restrictions on a Deployer’s configuration of an application’s JDBC and JMS resources. J2EE 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 J2EE Product Provider may restrict all JDBC access within that transaction to that non-XA-capable JDBC resource manager. Otherwise, a J2EE Product Provider must support use of multiple XA-capable JDBC resource managers within a transaction. In addition, a J2EE Product Provider may restrict the security configuration of all JDBC connections within a transaction to a single user identity. A J2EE 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 J2EE Product Provider may make the same restrictions as above, resulting in a transaction being restricted to a single JMS resource manager and user identity.

In addition, when both a JDBC resource manager and a JMS resource manager are used in the same transaction, a J2EE 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 JMS access in a single global transaction must not mark the corresponding transactional resources as “unshareable”.

Although these restrictions are allowed, it is recommended that J2EE Product Providers support JDBC and JMS resource managers that provide full two-phase commit functionality and, as a result, do not impose these restrictions.

## **J2EE.4.7 Two-Phase Commit Support**

A J2EE 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 JMS provider may be provided as an XA-capable resource adapter. In such a case, it must be possible to include JMS 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 Section J2EE.4.2.7, “Transactional JDBC™ Technology Support.”

## **J2EE.4.8 System Administration Tools**

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

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





# CHAPTER J2EE.5

---

## Resources, Naming, and Injection

**T**his chapter describes how applications declare dependencies on external resources and configuration parameters, and how those items are represented in the J2EE naming system and can be injected into application components. These requirements are based on annotations defined in the Java Metadata specification (JSR-175) and features defined in the Java Naming and Directory Interface™ (JNDI) specification. The Resource annotation described here is defined in more detail in the Common Annotations specification (JSR-250). The EJB annotation described here is defined in more detail in the Enterprise JavaBeans specification (JSR-220).

### J2EE.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.

### **J2EE.5.1.1 Chapter Organization**

The following sections contain the J2EE platform solutions to the above issues:

- Section J2EE.5.2, “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.
- Section J2EE.5.3, “Responsibilities by J2EE Role” defines the general responsibilities for each of the J2EE roles such as Application Component Provider, Application Assembler, Deployer, and J2EE Product Provider.
- Section J2EE.5.4, “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.
- Section J2EE.5.5, “Enterprise JavaBeans™ (EJB) References” defines the interfaces for obtaining the home interface or an instance of an enterprise bean using an EJB reference. An EJB reference is a special entry in the application component’s environment.
- Section J2EE.5.6, “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.
- Section J2EE.5.7, “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.
- Section J2EE.5.8, “Message Destination References” defines the interfaces for declaring and using message destination references.
- Section J2EE.5.9, “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.
- Section J2EE.5.10, “ORB References” describes the use by eligible application components of references to a CORBA ORB object in the component’s environment.

### **J2EE.5.1.2 Required Access to the JNDI Naming Environment**

J2EE application clients, enterprise beans, and web components are required to have access to a JNDI naming environment. 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.

## **J2EE.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.

### **J2EE.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.
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.
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.

#### **J2EE.5.2.2      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 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 name `java:comp/ORB` is such a name.

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.

### **J2EE.5.2.3      Annotations and Resource 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. 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 J2SE 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 may 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 the JNDI name is always 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's 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; a development tool that processes such annotations must detect this 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 Table J2EE.5-1. They 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 J2EE.5-1    Component classes supporting injection**

Container	Spec	Classes supporting injection
web container	Servlet	servlets servlet filters event listeners
	JSP	tag handlers tag library event listeners
	JSF	managed beans
	JAX-RPC	service endpoints handlers
	JAX-WS	service endpoints handlers
EJB container	EJB	beans callback listeners interceptors
	JAX-RPC	service endpoints handlers
	JAX-WS	service endpoints handlers
application client container	J2EE platform	main class (static) login callback handler

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 field or method definition that overrides a field or method on a superclass defines the resource, if any, to be injected into that field or method. An overriding field or method may request injection even though the superclass field or 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 field or 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.

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 `InjectionComplete` annotation. This method will be called after all resource injections have occurred 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.

#### **J2EE.5.2.4 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 annotated 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, must name exactly the annotated field or property method.
- The `res-sharing-scope` element, if specified, overrides the `shareable` element of the annotation.
- The `res-auth` element, if specified, overrides the `authenticationType` element of the annotation.

The rules for how a deployment descriptor entry may override an EJB annotation are included in the EJB specification. The rules for how a deployment descriptor entry may override a `WebServiceRef` annotation are included in the Web Services for J2EE specification.

### **J2EE.5.3 Responsibilities by J2EE Role**

This section describes the responsibilities for each J2EE role that apply to all uses of the J2EE 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.

#### **J2EE.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.

### **J2EE.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.

### **J2EE.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.

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

### **J2EE.5.3.4      J2EE Product Provider's Responsibilities**

The J2EE Product Provider has the following responsibilities:

- Provide a deployment tool that allows the Deployer to set and modify the entries of the application component's naming context.
- Implement the `java:comp/env` environment naming context, and provide it to the application component instances at runtime. The naming context must include all the 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.

- 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.

## **J2EE.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`, and `Float`.

The following subsections describe the responsibilities of each J2EE Role.

### **J2EE.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.

#### ***J2EE.5.4.1.1     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();
    ...
}

```

#### ***J2EE.5.4.1.2 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");
...
}

```

### ***J2EE.5.4.1.3 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 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>
  ...

```

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.

## J2EE.5.5 Enterprise JavaBeans™ (EJB) References

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

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

specified by the Application Component Provider using annotations in the source code of the component.

### **J2EE.5.5.1      Application Component Provider's Responsibilities**

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

#### ***J2EE.5.5.1.1      Injection of EJB Entries***

A field or a method of an application component may be annotated with the EJB annotation. The EJB annotation represents a reference to an EJB session bean. The reference may be to the local or remote home interface of the session bean, or may be to the business interface of an EJB 3 bean. If the reference is to the EJB 3 business interface, a reference to an instance of the enterprise bean will be injected.

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.myCart` in the naming context. The target of the reference is not named and must be resolved by the Deployer.

```
@EJB private ShoppingCart myCart;
```

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

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

#### ***J2EE.5.5.1.2      Programming Interfaces for EJB References***

The Application Component Provider may use EJB references to locate the home interfaces or instances of enterprise beans as follows.



- Assign an entry in the application component's environment to the reference. (See subsection 5.5.1.3 for information on how EJB 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 home interface or instance of the referenced enterprise bean in the application component's environment using JNDI.

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

```
public void changePhoneNumber(...) {  
    ...  
    // Obtain the default initial JNDI context.  
    Context initCtx = new InitialContext();  
  
    // Look up the home interface of the EmployeeRecord  
    // enterprise bean in the environment.  
    Object result = initCtx.lookup("java:comp/env/ejb/EmplRecord");  
  
    // Convert the result to the proper type.  
    EmployeeRecordHome emplRecordHome = (EmployeeRecordHome)  
        javax.rmi.PortableRemoteObject.narrow(result,  
            EmployeeRecordHome.class);  
    ...  
}
```

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

### ***J2EE.5.5.1.3 Declaration of EJB References***

Although the EJB reference is an entry in the application component's environment, the Application Component Provider must not use a `env-entry` element to declare it. Instead, the Application Component Provider must declare all the EJB references using 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 EJB references used by the application component. Deployment descriptor entries may also be used to specify injection of an EJB 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 contains a description element and the ejb-ref-name, ejb-ref-type, home, and remote elements.

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

An EJB reference is scoped to the application component whose declaration contains the ejb-ref or ejb-local-ref element. This means that the EJB 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 following example illustrates the declaration of EJB references in the deployment descriptor.

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

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

```
</ejb-ref>

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

### J2EE.5.5.2 Application Assembler's Responsibilities

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

The 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. (It is the name defined in the `ejb-name` element of the target enterprise bean.) The target enterprise bean can be in any `ejb-jar` file in the same J2EE application as the referencing application component.
- Alternatively, to avoid the need to rename enterprise beans to have unique names within an entire J2EE application, the Application Assembler may use the following syntax in the `ejb-link` element of the referencing application component. The Application Assembler specifies the path name of the `ejb-jar` file containing the referenced enterprise bean and appends the `ejb-name` of the target bean separated from the path name by “#”. The path name is relative to the referencing application component JAR file. In this manner, multiple beans with the same `ejb-name` may be uniquely identified when the Application Assembler cannot change `ejb-names`.
- The Application Assembler must ensure that the target enterprise bean is type-compatible with the declared EJB reference. This means that the target enterprise bean must be of the type indicated in the `ejb-ref-type` element, and that the home and remote interfaces of the target enterprise bean must be Java type-compatible with the interfaces declared in the EJB reference.

The following example illustrates the use of the `ejb-link` element in the deployment descriptor. The enterprise bean reference should be satisfied by the

bean named `EmployeeRecord`. The `EmployeeRecord` enterprise bean may be packaged in the same module as the component making this reference, or it may be packaged in another module within the same J2EE application as the component making this reference.

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

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

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

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 J2EE 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>
...
```

### **J2EE.5.5.3     Deployer's Responsibilities**

The Deployer is responsible for the following:

- The Deployer must ensure that all the declared EJB references are bound to the homes or instances 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 EJB 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 home and remote interfaces of the target enterprise bean must be Java type-compatible with the home and remote interfaces declared in the EJB reference (if specified).
- If an EJB 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.

### **J2EE.5.5.4     J2EE Product Provider's Responsibilities**

The J2EE Product Provider must provide the deployment tools that allow the Deployer to perform the tasks described in the previous subsection. The deployment tools provided by the J2EE Product Provider must be able to process the information supplied in class file annotations and in the `ejb-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 an EJB reference to the home interface or instance of the specified target enterprise bean.
- Inform the Deployer of any unresolved EJB references, and allow him or her to resolve an EJB reference by binding it to a specified compatible target enterprise bean.

## **J2EE.5.6      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.

### **J2EE.5.6.1      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

#### ***J2EE.5.6.1.1    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();
    ...
}
```

#### ***J2EE.5.6.1.2    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 5.6.1.3 for in-

formation on how resource manager connection factory references are declared in the deployment descriptor.)

- This specification recommends, but does not require, that all resource manager connection factory references be organized in the subcontexts of the application component's environment, using a different subcontext for each resource manager type. For example, all JDBC™ DataSource references should be declared in the `java:comp/env/jdbc` subcontext, all JMS connection factories in the `java:comp/env/jms` subcontext, all JavaMail connection factories in the `java:comp/env/mail` subcontext, and all URL connection factories in the `java:comp/env/url` subcontext. 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 `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 `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();  
    ...  
}
```

### ***J2EE.5.6.1.3 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`, `res-type`, and `res-auth` elements, and the optional `res-sharing-scope` element. 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-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`. 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>
```

#### ***J2EE.5.6.1.4 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.QueueConnectionFactory`, the `javax.jms.TopicConnectionFactory`, or the `javax.jms.ConnectionFactory` for obtaining JMS connections.

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

The J2EE 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.

#### **J2EE.5.6.2 Deployer's Responsibilities**

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

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

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

### **J2EE.5.6.3      J2EE Product Provider's Responsibilities**

The J2EE Product Provider is responsible for the following:

- Provide the deployment tools that allow the Deployer to perform the tasks described in the previous subsection.
- Provide the implementation of the resource manager connection factory classes that are required by this specification.
- If the Application Component Provider set the `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<sup>1</sup>.

- 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 J2EE products will provide the following features:

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

#### **J2EE.5.6.4 System Administrator's Responsibilities**

The System Administrator is typically responsible for the following:

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

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

---

<sup>1</sup>. Connections obtained from the same resource manager connection factory through a different resource manager connection factory reference may be shareable.

## **J2EE.5.7     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.

### **J2EE.5.7.1     Application Component Provider’s Responsibilities**

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

#### ***J2EE.5.7.1.1     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 resource 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 references 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.

#### ***J2EE.5.7.1.2     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 5.7.1.3 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.

#### ***J2EE.5.7.1.3 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 an optional `description` element and the mandatory `resource-env-ref-name` and `resource-env-ref-type` elements.

The `resource-env-ref-name` element specifies the resource environment reference name. Its value is the environment entry name used in the application component code. The name of the 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.

#### **J2EE.5.7.2 Deployer's Responsibilities**

The Deployer is responsible for the following:

- The Deployer must ensure that all the declared resource environment references are bound to administered objects that exist in the operational environment.

The Deployer may use, for example, the JNDI `LinkRef` mechanism to create a symbolic link to the actual JNDI name of the target object.

- The Deployer must ensure that the target object is type-compatible with the type declared for the resource environment reference. This means that the target object must be of the type indicated in the `Resource` annotation or the `resource-env-ref-type` element.

### **J2EE.5.7.3      J2EE Product Provider's Responsibilities**

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

## **J2EE.5.8      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.

### **J2EE.5.8.1      Application Component Provider's Responsibilities**

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

#### ***J2EE.5.8.1.1      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;
```

#### ***J2EE.5.8.1.2 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 5.8.1.3 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 JMS 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 JMS Destination.

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

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

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

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

### ***J2EE.5.8.1.3 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 an optional `description` element and the mandatory `message-destination-ref-name`, `message-destination-type`, and `message-destination-usage` elements.

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. 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 JMS Destination, its value might be

`javax.jms.Queue`. The `message-destination-usage` element specifies whether messages are consumed from the message destination, produced for the destination, or both.

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 JMS 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>
...
```

### **J2EE.5.8.2      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 in an `ejb-jar` file, 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 in an `ejb-jar` deployment descriptor 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 EJB module in the same J2EE 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 J2EE 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 `ejb-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 com-

ponent 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.

### **J2EE.5.8.3      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 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.

### **J2EE.5.8.4      J2EE Product Provider's Responsibilities**

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

## **J2EE.5.9      UserTransaction References**

Certain J2EE application component types are allowed to use the JTA `UserTransaction` interface to start, commit, and abort transactions. Such

application components can find an appropriate object implementing the `UserTransaction` interface by looking up the JNDI name `java:comp/UserTransaction` 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.

### **J2EE.5.9.1      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 **Table J2EE.6-1** in this specification and the EJB specification for details.

### **J2EE.5.9.2      J2EE Product Provider's Responsibilities**

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

## **J2EE.5.10      ORB References**

Some J2EE 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.

### **J2EE.5.10.1 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.

#### **J2EE.5.10.2 J2EE Product Provider's Responsibilities**

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



# CHAPTER J2EE.6

---

## Application Programming Interface

**T**his Chapter describes API requirements for the Java™ 2 Platform, Enterprise Edition (J2EE). J2EE requires the provision of a number of APIs for use by J2EE applications, starting with the core Java APIs and including several Java optional packages<sup>1</sup>.

### J2EE.6.1 Required APIs

J2EE application components execute in runtime environments provided by the containers that are a part of the J2EE platform. The J2EE platform supports four types of containers corresponding to J2EE application component types: application client containers, applet containers, web containers for servlets and JSP pages, and enterprise bean containers.

#### J2EE.6.1.1 Java Compatible APIs

The containers provide all application components with the Java 2 Platform, Standard Edition, v5.0 (J2SE) APIs, which include the following enterprise APIs:

---

<sup>1</sup> Note that “optional packages” were previously called “standard extensions”. The packages described here are optional relative to J2SE, but *required* for J2EE.

- Java IDL API
- JDBC API
- RMI-IIOP API
- JNDI API
- JAXP API
- JAAS API
- JMX API

In particular, the applet execution environment must be J2SE 5.0 compatible. Since typical browsers don't yet provide such support, J2EE 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 J2SE 5.0 compatible applet execution environment.

The specifications for the J2SE APIs are available at <http://java.sun.com/j2se/5.0/docs/>.

### **J2EE.6.1.2      Java Optional Packages**

The J2EE platform also requires a number of Java optional packages. **Table J2EE.6-1** indicates the required optional packages with their required versions.

**Table J2EE.6-1      J2EE-Required Java Optional Packages**

<b>Optional Package</b>	<b>App Client</b>	<b>Applet</b>	<b>Web</b>	<b>EJB</b>
EJB 3.0	Y <sup>a</sup>	N	Y <sup>b</sup>	Y
Servlet 2.4	N	N	Y	N
JSP 2.1	N	N	Y	N
JMS 1.1	Y	N	Y	Y
JTA 1.0	N	N	Y	Y
JavaMail 1.3	Y	N	Y	Y
JAF 1.1	Y	N	Y	Y
Connector 1.5	N	N	Y	Y
Web Services 1.1	Y	N	Y	Y

**Table J2EE.6-1 J2EE-Required Java Optional Packages**

Optional Package	App Client	Applet	Web	EJB
JAX-RPC 1.1	Y	N	Y	Y
JAX-WS 2.0	Y	N	Y	Y
JAXB 2.0	Y	N	Y	Y
SAAJ 1.3	Y	N	Y	Y
JAXR 1.0	Y	N	Y	Y
J2EE Management 1.0	Y	N	Y	Y
J2EE Deployment 1.1 <sup>c</sup>	N	N	N	N
JACC 1.0	N	N	Y	Y
JSP Debugging 1.0	N	N	Y	N
JSTL 1.1	N	N	Y	N
WebSvc Metadata 1.0	Y	N	Y	Y
JSF 1.2	N	N	Y	N
Common Annotations 1.0	Y	N	Y	Y
StAX 1.0	Y	N	Y	Y
Java Persistence 1.0	Y	N	Y	Y

a. Client APIs only.

b. Client APIs only.

c. See section J2EE.6.18 on page 129 for details.

All classes and interfaces required by the specifications for the APIs must be provided by the J2EE containers. In some cases, a J2EE 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 J2EE platform.

**Note** – Several of the included APIs are likely to be revised before J2EE 5.0 is finalized. The table above will be updated at that time.

## **J2EE.6.2      Java 2 Platform, Standard Edition (J2SE) Requirements**

### **J2EE.6.2.1      Programming Restrictions**

The J2EE programming model divides responsibilities between Application Component Providers and J2EE Product Providers: Application Component Providers focus on writing business logic and the J2EE Product Providers focus on providing a managed system infrastructure in which the application components can be deployed.

This division leads to a restriction on the functionality that application components can contain. If application components contain the same functionality provided by J2EE system infrastructure, there are clashes and mis-management of the functionality.

For example, if enterprise beans were allowed to manage threads, the J2EE platform could not manage the life cycle of the enterprise beans, and it could not properly manage transactions.

Since we do not want to subset the J2SE platform, and we want J2EE Product Providers to be able to use J2SE products without modification in the J2EE platform, we use the J2SE security permissions mechanism to express the programming restrictions imposed on Application Component Providers.

In this section, we specify the J2SE security permissions that the J2EE Product Provider must provide for each application component type. We call these permissions the J2EE security permissions set. The J2EE security permissions set is a required part of the J2EE API contract. Portable applications will rely on only the set of permissions specified here.

### **J2EE.6.2.2      The J2EE Security Permissions Set**

The J2EE security permissions set defines the minimum set of permissions that application components can expect. All J2EE products must be capable of deploying application components that require the set of permissions described here. The Product Provider must ensure that the application components do not use functions that conflict with the J2EE security permission set.

The exact set of security permissions for application components in use at a particular installation is a matter of policy outside the scope of this specification. A J2EE product may allow applications to run with no security manager at all, or with a security manager that enforces any set of security permissions, as required by the enterprise environment. All J2EE products must be capable of running applications with at least the set of permissions described here. Some J2EE products will allow the set of permissions available to a component to be configurable, providing some components with more or fewer permissions than those described here. A future version of this specification will allow these security requirements to be specified in the deployment descriptor for application components. At the present time, application components that need permissions not in this minimal set should describe their requirements in their documentation. Note that it may not be possible to deploy applications that require more than this minimal set on some J2EE products.

The J2SE security permissions are fully described in <http://java.sun.com/j2se/5.0/docs/guide/security/permissions.html>.

### **J2EE.6.2.3 Listing of the J2EE Security Permissions Set**

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

**Table J2EE.6-2 J2EE Security Permissions Set**

<b>Security Permissions</b>	<b>Target</b>	<b>Action</b>
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

**Table J2EE.6-2 J2EE Security Permissions Set**

Security Permissions	Target	Action
java.net.SocketPermission	localhost:1024-	accept,listen
java.io.FilePermission	*	read,write
java.util.PropertyPermission	*	read
Applet Clients		
java.net.SocketPermission	<i>codebase</i>	connect
java.util.PropertyPermission	<i>limited</i>	read
Web Components and EJB Components		
java.lang.RuntimePermission	loadLibrary	
java.lang.RuntimePermission	queuePrintJob	
java.net.SocketPermission	*	connect
java.io.FilePermission	*	read,write
java.util.PropertyPermission	*	read

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

## **J2EE.6.2.4 Additional Requirements**

### ***J2EE.6.2.4.1 Networking***

The J2SE 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 1.0 must be supported by https URL objects. Both input and output must be supported.

The J2SE platform also includes a mechanism for converting a URL's byte stream to an appropriate object, using the `java.net.ContentHandler` class and `java.net.ContentHandlerFactory` interface. A `ContentHandler` object 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 **Table J2EE.6-3** must be returned.

**Table J2EE.6-3 Java Type of Objects Returned When Using the `getContent` Method**

MIME Type	Java Type
image/gif	<code>java.awt.Image</code>
image/jpeg	<code>java.awt.Image</code>
image/png	<code>java.awt.Image</code>

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

Most enterprise environments will include a firewall that limits access from the internal network (intranet) to the public Internet, and vice versa. 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 J2EE

products may provide support for tunneling other communication through firewalls, but this is neither specified nor required.

#### ***J2EE.6.2.4.2 JDBC™ API***

The JDBC API, which is part of the J2SE platform, allows for access to a wide range of data storage systems. The J2SE 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 J2EE specification does require that such a database be available and accessible from a J2EE product through the JDBC API. Such a database must be accessible from web components, enterprise beans, and application clients, but need not be accessible from applets. In addition, the driver for the database must meet the JDBC Compatible requirements in the JDBC specification.

J2EE applications should not attempt to load JDBC drivers directly. Instead, they should use the technique recommended in the JDBC specification and perform a JNDI lookup to locate a `DataSource` object. The JNDI name of the `DataSource` object should be chosen as described in Section J2EE.5.6, “Resource Manager Connection Factory References.” The J2EE platform must be able to supply a `DataSource` that does not require the application to supply any authentication information when obtaining a database connection. Of course, applications may also supply a user name and password 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 EJB specification contains the precise rules for enterprise beans.

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

Drivers supporting the JDBC API in a J2EE environment must meet the JDBC 3.0 API Compliance requirements as specified in the JDBC specification

and must meet a number of additional requirements in their implementation of JDBC APIs, as described below:

- Drivers are required to provide accurate and complete metadata through the `Connection.getMetaData` method. J2EE applications should examine the `DatabaseMetaData` object and adapt their behavior to the capabilities of the current database. How this information is used to create portable applications that are independent of the underlying database vendor and driver is beyond the scope of this specification.
- Drivers must support stored procedures. The `DatabaseMetaData` method `supportsStoredProcedures` must return `true`. The driver must also support the full JDBC API escape syntax for calling stored procedures with the following methods on the `Statement`, `PreparedStatement`, and `CallableStatement` classes:
  - `executeUpdate`
  - `executeQuery`

Support for calling stored procedures using the method `execute` on the `Statement`, `PreparedStatement`, and `CallableStatement` interfaces is not required because some databases don't support returning more than a single `ResultSet` from a stored procedure.

- Drivers must support all of the `CallableStatement` methods that apply to SQL92 types, including the following:
  - `getBigDecimal`
  - `getBoolean`
  - `getByte`
  - `getBytes`
  - `getDate`
  - `getDouble`
  - `getFloat`
  - `getInt`
  - `getLong`
  - `getObject`

- `getShort`
- `getString`
- `getTime`
- `getTimestamp`
- `registerOutParameter`
- `wasNull`

Support for the new BLOB, CLOB, ARRAY, REF, STRUCT, and JAVA\_OBJECT types is not required. All parameter types (IN, OUT, and INOUT) must be supported.

- Drivers must support all of the `PreparedStatement` methods that apply to SQL92 types, including the following:
  - `setAsciiStream`
  - `setBigDecimal`
  - `setBinaryStream`
  - `setBoolean`
  - `setByte`
  - `setBytes`
  - `setCharacterStream`
  - `setDate`
  - `setDouble`
  - `setFloat`
  - `setInt`
  - `setLong`
  - `setNull`
  - `setObject`
  - `setShort`
  - `setString`
  - `setTime`
  - `setTimestamp`

Support for the new BLOB, CLOB, ARRAY, REF, STRUCT, and JAVA\_OBJECT types is

not required. Support for the `PreparedStatement` method `getMetaData` is not required. This method must throw an `SQLException` if it is not supported. Support for the `PreparedStatement` method `getParameterMetaData` is required.

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

Drivers are free to implement these methods any way they choose (including a non-batching implementation) as long as the semantics are correct.

- Drivers must support the `ResultSet` type `TYPE_FORWARD_ONLY`, with a concurrency of `CONCUR_READ_ONLY`. Support for other `ResultSet` types `TYPE_SCROLL_INSENSITIVE` and `TYPE_SCROLL_SENSITIVE`, and concurrency `CONCUR_UPDATABLE`, is not required.
- A driver must provide full support for `DatabaseMetaData` and `ResultSetMetaData`. This implies that all of the methods in the `DatabaseMetaData` interface must be implemented and must behave as specified in the JDBC specification. None of the methods in `DatabaseMetaData` and `ResultSetMetaData` may throw an exception because they are not implemented.
- The JDBC API core specification requires that JDBC compliant drivers provide support for the SQL92, Transitional Level, `DROP TABLE` command, full support for the `CASCADE` and `RESTRICT` options is required. As many popular databases do not support `DROP TABLE` as specified in the SQL92 specification, the following clarification is required.

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

- A driver must support the `Statement` escape syntax for the following functions as specified by the JDBC specification:

- CONCAT
- SUBSTRING
- LOCATE (two argument version only)
- LENGTH
- ABS
- SQRT
- MOD

The JDBC API includes APIs for row sets, connection naming via JNDI, connection pooling, and distributed transaction support. The connection pooling and distributed transaction features are intended for use by JDBC drivers to coordinate with an application server. J2EE products are not required to support the application server facilities described by these APIs, although they may prove useful.

The Connector architecture defines an SPI that essentially extends the functionality of the JDBC SPI with additional security functionality, and a full packaging and deployment functionality for resource adapters. A J2EE product must support deploying and using a JDBC driver that has been written and packaged as a resource adapter using the Connector architecture.

The JDBC 3.0 specification is available at <http://java.sun.com/products/jdbc/download.html>.

### ***J2EE.6.2.4.3 Java IDL***

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

A J2EE product must support 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 J2EE 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. J2EE clients are not required to multiplex calls, although such support is highly recommended.

A J2EE product must provide support for a CORBA Portable Object Adapter (POA) to support portable stub, skeleton, and tie classes. A J2EE application that defines or uses CORBA objects other than enterprise beans must include such portable stub, skeleton, and tie classes in the application package.

J2EE 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 J2EE product must provide a COSNaming service to support the EJB 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>.

#### ***J2EE.6.2.4.4 RMI-JRMP***

JRMP is the Java technology-specific Remote Method Invocation (RMI) protocol. The J2EE security restrictions typically prevent all application component types

except application clients from creating and exporting an RMI object, but all J2EE application component types can be clients of RMI objects.

#### ***J2EE.6.2.4.5 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 J2EE 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.

All components accessing remote enterprise beans must use the narrow method of the `javax.rmi.PortableRemoteObject` class, as described in the EJB 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 EJB specification.

The J2EE security restrictions typically prevent all application component types, except application clients, from creating and exporting an RMI-IIOP object. All J2EE application component types can be clients of RMI-IIOP objects. J2EE applications should also use JNDI to lookup non-EJB RMI-IIOP objects. The JNDI names used for such non-EJB RMI-IIOP objects should be configured at deployment time using the standard environment entries mechanism (see Section J2EE.5.2, “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 J2EE application clients can create non-EJB RMI-IIOP server objects for use as callback objects, or to pass in calls to other RMI-IIOP objects.

Note that while RMI-IIOP doesn't specify how to propagate the current security context or transaction context, the EJB interoperability specification does define such context propagation. This specification only requires that the propagation of context information as defined in the EJB 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 J2EE 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 J2EE application that defines or uses RMI-IIOP objects other than enterprise beans must include such portable Stub and Tie classes in the application package. Stub and Tie objects for enterprise beans, however, must not be included with the application: they will be generated, if needed, by the J2EE product at deployment time or at run time.

RMI-IIOP is defined by chapters 5, 6, 13, 15, and section 10.6.2 of the CORBA 2.3.1 specification, available at <http://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>.

#### ***J2EE.6.2.4.6 JNDI***

A J2EE product must be able to make the following types of objects available in the application's JNDI namespace: `EJBHome` objects, `EJBLocalHome` objects, `JTA 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, `EntityManager` objects, and other Java language objects as described in Chapter J2EE.5, "Resources, Naming, and Injection." The JNDI implementation in a J2EE product must be capable of supporting all of these uses in a single application component using a single `JNDI InitialContext`. Application components will generally create a `JNDI InitialContext` using the default constructor with no arguments. The application component may then perform lookups on that `InitialContext` to find objects as specified above.

The names used to perform lookups for J2EE objects are application dependent. The application component's deployment descriptor is 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 Chapter J2EE.5, “Resources, Naming, and Injection” for details.

Two particular names are defined by this specification. For all application components that have access to the JTA `UserTransaction` interface, the appropriate `UserTransaction` object can be found using the name `java:comp/UserTransaction`. In all containers except the applet container, application components may lookup a CORBA ORB instance using the name `java:comp/ORB`.

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

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

This specification requires that the J2EE 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.

All J2EE products must provide a `COSNaming` name service to meet the EJB interoperability requirements. In addition, a `COSNaming` JNDI service provider must be available through the web, EJB, and application client containers. It will also typically be available in the applet container, but this is not required.

A `COSNaming` JNDI service provider is a part of the J2SE 5.0 SDK and JRE from Sun, but is not a required component of the J2SE specification. The `COSNaming` JNDI service provider specification is available at <http://java.sun.com/j2se/5.0/docs/guide/jndi/jndi-cos.html>.

See Chapter J2EE.5, “Resources, Naming, and Injection” for the complete naming requirements for the J2EE platform. The JNDI specification is available at <http://java.sun.com/products/jndi/docs.html>.

#### ***J2EE.6.2.4.7 Context Class Loader***

This specification requires that J2EE 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 EJB specification requires that all EJB 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 EJB 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 Section J2EE.8.2.5, “Dynamic Class Loading” for recommendations for libraries that dynamically load classes.

#### ***J2EE.6.2.4.8    Java™ Authentication and Authorization Service (JAAS) Requirements***

All EJB containers and all web containers must support the use of the JAAS APIs as specified in the Connector specification. All application client containers must support use of the JAAS APIs as specified in Chapter J2EE.9, “Application Clients.”

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

#### ***J2EE.6.2.4.9    Logging API Requirements***

The Logging API provides classes and interfaces in the `java.util.logging` package that are the Java™ 2 platform’s core logging facilities. This specification does not require any additional support for logging. A J2EE 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 J2EE containers use the logging API to log certain events.

#### ***J2EE.6.2.4.10   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 J2EE 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 J2EE 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 J2EE applications.

### **J2EE.6.3      Enterprise JavaBeans™ (EJB) 3.0 Requirements**

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

This specification does not impose any additional requirements at this time. Note that the EJB specification includes the specification of the EJB interoperability protocol based on RMI-IIOP. All containers that support EJB clients must be capable of using the EJB interoperability protocol to invoke enterprise beans. All EJB containers must support the invocation of enterprise beans using the EJB interoperability protocol. A J2EE product may also support other protocols for the invocation of enterprise beans.

A J2EE product may support multiple object systems (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.

The EJB container and the web container are both 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.

### **J2EE.6.4      Servlet 2.4 Requirements**

The servlet specification defines the packaging and deployment of web applications, whether standalone or as part of a J2EE application. The servlet specification also

addresses security, both standalone and within the J2EE platform. These optional components of the servlet specification are requirements of the J2EE platform.

The servlet specification includes additional requirements for web containers that are part of a J2EE product and a J2EE product must meet these requirements as well.

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

Web containers must support J2EE distributable web applications placing objects of any of the following types into a `javax.servlet.http.HttpSession` object using the `setAttribute` or `putValue` methods:

- `java.io.Serializable`
- `javax.ejb.EJBObject`
- `javax.ejb.EJBHome`
- `javax.ejb.EJBLocalObject`
- `javax.ejb.EJBLocalHome`
- `javax.transaction.UserTransaction`
- a `javax.naming.Context` object for the `java:comp/env` context
- a reference to an EJB 3 local or remote business interface

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 J2EE 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 J2EE products provide support for access to local enterprise beans from the web container.

The servlet specification is available at <http://java.sun.com/products/servlet>.

## **J2EE.6.5     JavaServer Pages™ (JSP) 2.1 Requirements**

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

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

## **J2EE.6.6     Java™ Message Service (JMS) 1.1 Requirements**

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

The JMS specification defines several interfaces intended for integration with an application server. A J2EE product need not provide objects that implement these interfaces, and portable J2EE applications must not use 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.Session` method `setMessageListener`
- `javax.jms.Session` method `getMessageListener`
- `javax.jms.Session` method `run`
- `javax.jms.QueueConnection` method `createConnectionConsumer`
- `javax.jms.TopicConnection` method `createConnectionConsumer`
- `javax.jms.TopicConnection` method `createDurableConnectionConsumer`
- `javax.jms.MessageConsumer` method `getMessageListener`
- `javax.jms.MessageConsumer` method `setMessageListener`
- `javax.jms.Connection` method `setExceptionListener`
- `javax.jms.Connection` method `stop`
- `javax.jms.Connection` method `setClientID`

A J2EE container may throw a `JMSEException` (if allowed by the method) if the application component violates these restrictions.

Application components in the web and EJB 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 may throw a `JMSEException` if the application component violates this restriction. Application client containers must support the creation of multiple sessions for each connection.

In general, the behavior of a JMS provider should be the same in both the EJB container and the web container. The EJB specification describes restrictions on the use of JMS in an EJB container, as well as the interaction of JMS with transactions in an EJB container. Applications running in the web container should follow the same restrictions.

The JMS specification is available at <http://java.sun.com/products/jms>.

## **J2EE.6.7      Java™ Transaction API (JTA) 1.0 Requirements**

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

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

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

## **J2EE.6.8      JavaMail™ 1.3 Requirements**

The JavaMail API allows for access to email messages contained in message stores, and for the creation and sending of email messages using a message transport. Specific support is included for Internet standard MIME messages. Access to message stores and transports is through protocol providers supporting specific store and transport protocols. The JavaMail API specification does not require any specific protocol providers, but the JavaMail reference implementation includes an IMAP message store provider, a POP3 message store provider, and an SMTP message transport provider.

Configuration of the JavaMail API is typically done by setting properties in a `Properties` object that is used to create a `javax.mail.Session` object using a static factory method. To allow the J2EE platform to configure and manage JavaMail API sessions, an application component that uses the JavaMail API should request a `Session` object using JNDI, and should list its need for a `Session` object in its deployment descriptor using a `resource-ref` element. A JavaMail API `Session` object should be considered a resource factory, as described in Section J2EE.5.6, “Resource Manager Connection Factory References.” This specification requires that the J2EE platform support `javax.mail.Session` objects as resource factories, as described in that section.

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



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

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

The JavaMail API uses the JavaBeans Activation Framework API to support various MIME data types. The JavaMail API must include `javax.activation.DataContentHandlers` for the following MIME data types, corresponding to the Java programming language type indicated in **Table J2EE.6-4**.

**Table J2EE.6-4    JavaMail API MIME Data Type to Java Type Mappings**

Mime Type	Java Type
text/plain	<code>java.lang.String</code>
text/html	<code>java.lang.String</code>
text/xml	<code>java.lang.String</code>
multipart/*	<code>javax.mail.internet.MimeMultipart</code>
message/rfc822	<code>javax.mail.internet.MimeMessage</code>

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

## **J2EE.6.9    JavaBeans™ Activation Framework 1.1 Requirements**

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

The JavaBeans Activation Framework is used by the JavaMail API to handle the data included in email messages. Typical J2EE applications will not need to

use the JavaBeans Activation Framework directly, although applications making sophisticated use of email may need it.

This specification requires that a J2EE product provide only the `DataContentHandlers` specified above for the JavaMail API. This includes requirement of a `javax.activation.MimetypesFileTypeMap` that supports the mappings listed in **Table J2EE.6-5**.

**Table J2EE.6-5    Filename Extension to MIME Type Mappings**

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

The JavaBeans Activation Framework 1.1 specification is available at <http://java.sun.com/beans/glasgow/jaf.html>.

## **J2EE.6.10    J2EE™ Connector Architecture 1.5 Requirements**

All EJB 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 J2EE deployment tools must support deployment of Resource Adapters, as defined in the Connector specification, and must support the deployment of applications that use Resource Adapters.

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

## **J2EE.6.11    Web Services for J2EE 1.1 Requirements**

The Web Services for J2EE specification defines the capabilities a J2EE application server must support for deployment of web service endpoints. A complete deployment model is defined, including several new deployment descriptors. All

J2EE products must support the deployment and execution of web services as specified by the Web Services for J2EE 1.1 specification (JSR-109).

The Web Services for J2EE specification is available at <http://jcp.org/en/jsr/detail?id=109> and <http://jcp.org/en/jsr/detail?id=921>.

### **J2EE.6.12    Java™ API for XML-based RPC (JAX-RPC) 1.1 Requirements**

The JAX-RPC specification defines client APIs for accessing web services as well as techniques for implementing web service endpoints. The Web Services for J2EE specification describes the deployment of JAX-RPC-based services and clients. The EJB and servlet specifications also describe aspects of such deployment. It must be possible to deploy JAX-RPC-based applications using any of these deployment models.

The JAX-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 JAX-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.

The JAX-RPC specification is available at <http://java.sun.com/xml/jaxrpc>.

### **J2EE.6.13    Java™ API for XML Web Services (JAX-WS) 2.0 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 J2EE specification describes the deployment of JAX-WS-based services and clients. The EJB 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://java.sun.com/xml/jaxws>.

## **J2EE.6.14    Java™ Architecture for XML Binding (JAXB) 2.0 Requirements**

The Java Architecture for XML Binding (JAXB) provides a convenient way to bind an XML schema to a representation in Java language programs. JAXB can be used independently or in combination with JAX-WS, where it provides a standard data binding for web service messages. All J2EE application client containers, web containers, and EJB containers are required to support the JAXB API.

The Java API for XML Data Binding specification can be found at <http://jcp.org/en/jsr/detail?id=222>.

## **J2EE.6.15    SOAP with Attachments API for Java™ (SAAJ) 1.3**

The SAAJ API is used to manipulate SOAP messages. The SAAJ API is used by the JAX-RPC API to represent XML fragments and to access the entire SOAP message in a JAX-RPC message handler. As described in the SAAJ specification, implementations of the `SOAPConnectionFactory` method `newInstance` may, and typically will, throw an exception indicating that this functionality is not implemented.

The SAAJ specification is available at <http://java.sun.com/xml/saaaj>.

## **J2EE.6.16    Java™ API for XML Registries (JAXR) 1.0 Requirements**

The JAXR specification defines APIs for client access to XML-based registries such as ebXML registries and UDDI registries. J2EE products must include a JAXR registry provider that meets at least the JAXR level 0 requirements, as well as a registry implementation that can be accessed using that provider.

The JAXR specification is available at <http://java.sun.com/xml/jaxr>.

### **J2EE.6.17    Java™ 2 Platform, Enterprise Edition Management API 1.0 Requirements**

The J2EE Management API provides APIs for management tools to query a J2EE application server to determine its current status, applications deployed, and so on. All J2EE products must support this API as described in its specification.

The J2EE Management API specification is available at <http://jcp.org/jsr/detail/77.jsp>.

### **J2EE.6.18    Java™ 2 Platform, Enterprise Edition Deployment API 1.1 Requirements**

The J2EE Deployment API defines the interfaces between the runtime environment of a deployment tool and plug-in components provided by a J2EE application server. These plug-in components execute in the deployment tool and implement the J2EE product-specific deployment mechanisms. All J2EE products are required to supply these plug-in components for use in tools from other vendors.

Note that the J2EE Deployment specification does not define new APIs for direct use by J2EE applications. However, it would be possible to create a J2EE application that acts as a deployment tool and provides the runtime environment required by the J2EE Deployment specification.

The J2EE Deployment API specification is available at <http://java.sun.com/j2ee/tools/deployment>.

### **J2EE.6.19    Java™ Authorization Service Provider Contract for Containers (JACC) 1.0 Requirements**

The JACC specification defines a contract between a J2EE application server and an authorization policy provider. All J2EE application containers, web containers, and enterprise bean containers are required to support this contract.

The JACC specification can be found at <http://jcp.org/jsr/detail/115.jsp>.

## **J2EE.6.20    Debugging Support for Other Languages (JSR-45) Requirements**

JSP pages are usually translated into Java language pages and then compiled to create class files. The 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 J2EE products are required to be able to include such information in class files that are generated from JSP pages.

The Debugging Support for Other Languages specification can be found at <http://jcp.org/en/jsr/detail?id=45>.

## **J2EE.6.21    Standard Tag Library for JavaServer Pages™ (JSTL) 1.1 Requirements**

JSTL defines a standard tag library that makes it easier to develop JSP pages. All J2EE products are required to provide JSTL for use by all JSP pages.

The Standard Tag Library for JavaServer Pages specification can be found at <http://jcp.org/en/jsr/detail?id=52>.

## **J2EE.6.22    Web Services Metadata for the Java™ Platform 1.0 Requirements**

The Web Services Metadata for the Java Platform specification defines Java language annotations that can be used to simplify the development of web services.

---

**Note** – Will need to say more about packaging and deployment.

---

The Web Services Metadata for the Java Platform specification can be found at <http://jcp.org/en/jsr/detail?id=181>.

## **J2EE.6.23    JavaServer Faces™ 1.2 Requirements**

JavaServer Faces technology simplifies building user interfaces for JavaServer 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. All J2EE web containers are required to support applications that use the JavaServer Faces technology.

The JavaServer Faces specification can be found at <http://jcp.org/en/jsr/detail?id=252>.

## **J2EE.6.24 Common Annotations for the Java™ Platform 1.0 Requirements**

The Common 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 J2EE.6-6 Common Annotations Support by Container**

<b>Annotation</b>	<b>App Client</b>	<b>Web</b>	<b>EJB</b>
PropertySet	Y	Y	Y
Resource	Y	Y	Y
Resources	Y	Y	Y
InjectionComplete	Y	Y	Y
Generated	N	N	N
RunAs	N	N	Y
RolesReferenced	N	N	Y
RolesAllowed	N	N	Y
PermitAll	N	N	Y
DenyAll	N	N	Y

The Common Annotations for the Java Platform specification can be found at <http://jcp.org/en/jsr/detail?id=250>.

## **J2EE.6.25    Streaming API for XML (StAX) 1.0 Requirements**

The Streaming API for XML (StAX) specification defines a pull-parsing API for XML. The streaming API gives parsing control to the programmer by exposing a simple iterator based API. This allows the programmer to ask for the next event (pull the event) and allows state to be stored in a procedural fashion. All J2EE application client containers, web containers, and EJB containers are required to support the StAX API.

The Streaming API for XML specification can be found at <http://jcp.org/en/jsr/detail?id=173>.

## **J2EE.6.26    Java Persistence API 1.0**

---

**Note** – The inclusion of the Java Persistence API that's being developed by the EJB 3.0 expert group in J2EE 5.0 is currently a contentious issue in the J2EE expert group. Some experts believe this new technology will not be mature and should not be included in the J2EE platform until J2EE 6.0. Other experts believe that the advantages this technology brings to the J2EE platform are significant and it should be included in J2EE 5.0. Sun strongly supports inclusion of the Java Persistence API. Your feedback on this issue is encouraged.

---

---

**Note** – This section is still incomplete.

---



# CHAPTER J2EE.7

---

## Interoperability

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

### **J2EE.7.1 Introduction to Interoperability**

The J2EE 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 J2EE platform in enterprise environments 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 J2EE platform.

It is the interoperability requirements of the J2EE 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 J2EE 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 J2EE platform release allow:

- J2EE applications to connect to legacy systems using CORBA or low-level socket interfaces.
- J2EE applications to connect to other J2EE applications across multiple J2EE products, whether from different Product Providers or from the same Provider, and multiple J2EE platforms.

In this version of the specification, interoperability between J2EE applications running in different platforms is accomplished through the HTTP protocol, possibly using SSL, or the EJB interoperability protocol based on IIOP.

## **J2EE.7.2      Interoperability Protocols**

This specification requires that a J2EE product support a standard set of protocols and formats to ensure interoperability between J2EE 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 J2SE and by the underlying operating system.

### **J2EE.7.2.1      Internet and Web Protocols**

Standards based Internet protocols are the means by which different pieces of the platform communicate. The J2EE platform requires support for the following Internet protocols:

- TCP/IP protocol family—This is the core component of Internet communication. TCP/IP and UDP/IP are the standard transport protocols for the Internet. TCP/IP is supported by J2SE 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 J2SE and the underlying operating system. A J2EE

web container must be capable of advertising its HTTP services on the standard HTTP port, port 80.

- **SSL 3.0, TLS 1.0**—SSL 3.0 (Secure Socket Layer) represents the security layer for Web communication. It is available indirectly when using the `https` URL as opposed to the `http` URL. A J2EE web container must be capable of advertising its HTTPS service on the standard HTTPS port, port 443. SSL 3.0 and TLS 1.0 are also required as part of the EJB interoperability protocol in the EJB 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 JAX-RPC and JAX-WS specifications.
- **SOAP 1.2**—SOAP 1.2 is the version of the SOAP protocol standardized through W3C and supported by JAX-WS.
- **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 JAX-RPC specification.

### **J2EE.7.2.2      OMG Protocols**

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

- **IIOP (Internet Inter-ORB Protocol)**—Supported by Java IDL and RMI-IIOP in J2SE. Java IDL provides standards-based interoperability and connectivity through the Common Object Request Broker Architecture (CORBA). CORBA specifies the Object Request Broker (ORB) which allows applications to communicate with each other regardless of location. This interoperability is delivered through IIOP, and is typically found in an intranet setting. IIOP can be used as an RMI protocol using the RMI-IIOP technology. IIOP is defined in Chapters 13 and 15 of the CORBA 2.3.1 specification, available at <http://cgi.omg.org/cgi-bin/doc?formal/99-10-07>.
- **EJB interoperability protocol**—The EJB interoperability protocol is based on IIOP (GIOP 1.2) and the CSIv2 CORBA Secure Interoperability specification. The EJB interoperability protocol is defined in the EJB specification.
- **CORBA Interoperable Naming Service protocol**—The COSNaming-based INS protocol is an IIOP-based protocol for accessing a name service. The EJB

interoperability protocol requires the use of the INS protocol for lookup of EJB 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 J2EE products must provide a name service that meets the requirements of the Interoperable Naming Service specification, available at <http://cgi.omg.org/cgi-bin/doc?formal/2000-06-19>. This name service may be provided as a separate name server or as a protocol bridge or gateway to another name service. Either approach is consistent with this specification.

### **J2EE.7.2.3      Java Technology Protocols**

This specification requires the J2EE platform to support the JRMP protocol, which is the Java technology-specific Remote Method Invocation (RMI) protocol. JRMP is a required component of J2SE and is one of 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://java.sun.com/j2se/1.4/docs/guide/rmi>.

### **J2EE.7.2.4      Data Formats**

In addition to the protocols that allow communication between components, this specification requires J2EE 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 JAX-RPC API provides support for XML RPC messages, as well as a mapping between Java classes and XML.
- HTML 3.2—This represents the minimum web browser standard document format. While not directly supported by J2EE APIs, J2EE web clients must be able to display HTML 3.2 documents.

- Image file formats—The J2EE platform must support GIF, JPEG, and PNG images. Support for these formats is provided by the `java.awt.image` APIs (see the URL: <http://java.sun.com/j2se/5.0/docs/api/java/awt/image/package-summary.html>) and by J2EE web clients.
- JAR files—JAR (Java Archive) files are the standard packaging format for Java technology-based application components, including the `ejb-jar` specialized format, the Web application archive (WAR) format, the Resource Adapter archive (RAR), and the J2EE enterprise application archive (EAR) format. JAR is a platform-independent file format that permits many files to be aggregated into one file. This allows multiple Java components to be bundled into one JAR file and downloaded to a browser in a single HTTP transaction. JAR file formats are supported by the `java.util.jar` and `java.util.zip` packages. For complete information on the JAR specification, see <http://java.sun.com/j2se/5.0/docs/guide/jar>.
- Class file format—The class file format is specified in the Java Virtual Machine specification. Each class file contains one Java programming language type—either a class or an interface—and consists of a stream of 8-bit bytes. For complete information on the class file format, see <http://java.sun.com/docs/books/vmspec>.



# CHAPTER J2EE.8

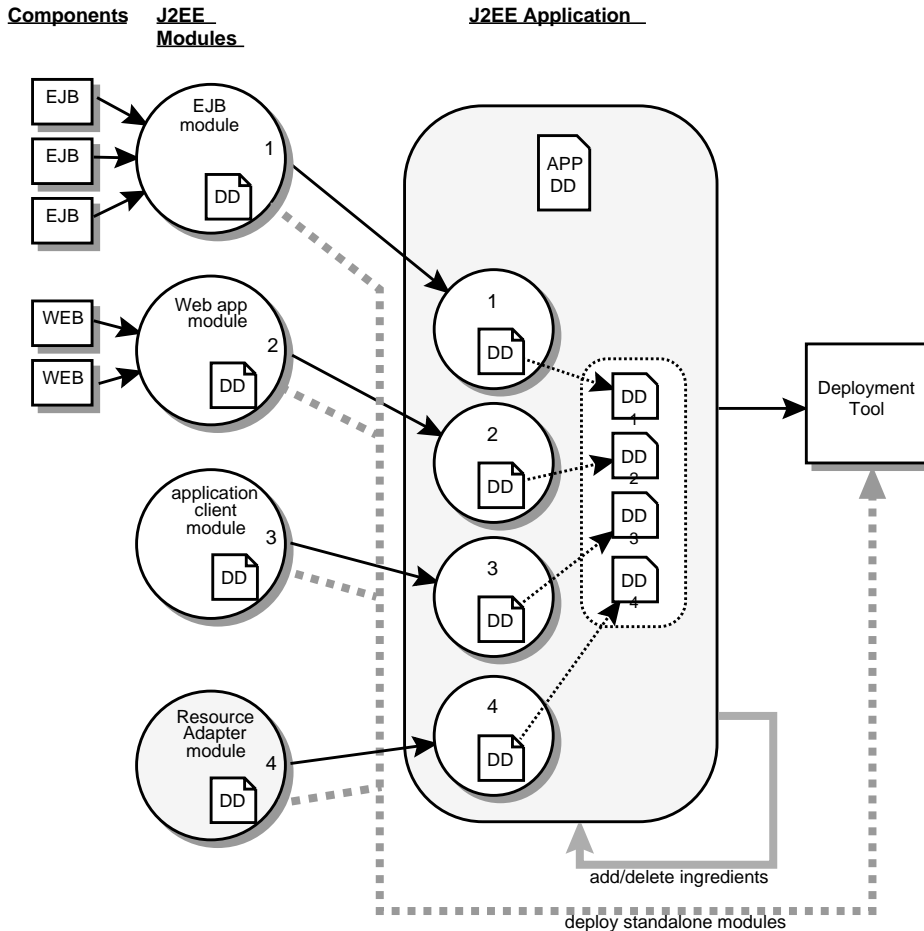
---

## Application Assembly and Deployment

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

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

**Figure J2EE.8-1** shows the composition model for J2EE deployment units and includes the optional use of alternate deployment descriptors by the application package to preserve any digital signatures of the original J2EE modules.



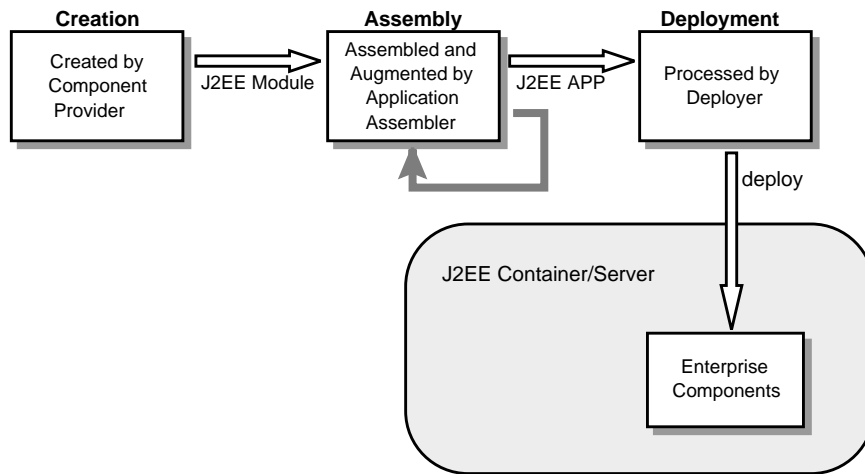
**Figure J2EE.8-1** J2EE Deployment

## J2EE.8.1 Application Development Life Cycle

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

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





**Figure J2EE.8-2** J2EE Application Life Cycle

### J2EE.8.1.1 Component Creation

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

A J2EE module is a collection of one or more J2EE components of the same component type (web, EJB, application client, or 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 J2EE deployment descriptor appropriate to that container type to produce a J2EE module. Components of different container types may not be mixed in a single J2EE module.

- A J2EE module represents the basic unit of composition of a J2EE application. In some cases a single J2EE module (not necessarily packaged into a J2EE application package) will contain an entire application. In other cases an application will be composed of multiple J2EE modules.
- The deployment descriptor for a J2EE module contains declarative data required to deploy the components in the module. The deployment descriptor

for a J2EE module also contains assembly instructions that describe how the components are composed into an application.

- Starting with version 5.0 of the J2EE platform, 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.
- An individual J2EE module can be deployed as a stand-alone J2EE module without an application level deployment descriptor and represents a valid J2EE application.
- J2EE modules may express dependencies on libraries as described below in Section J2EE.8.2, “Library Support.”

### **J2EE.8.1.2      Application Assembly**

A J2EE application may consist of one or more J2EE modules and one J2EE application deployment descriptor. A J2EE application is packaged using the Java Archive (JAR) file format into a file with a `.ear` (Enterprise ARchive) filename extension. A minimal J2EE application package will only contain J2EE modules and the application deployment descriptor. A J2EE application package may also include libraries referenced by J2EE modules (using the `Class-Path` mechanism described below in Section J2EE.8.2, “Library Support”), help files, and documentation to aid the deployer.

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

The J2EE application deployment descriptor represents the top level view of a J2EE application’s contents. The J2EE application deployment descriptor is specified by an XML schema or document type definition (see Section J2EE.8.5, “J2EE Application XML Schema”).

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

### J2EE.8.1.3 Deployment

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

## J2EE.8.2 Library Support

The J2EE platform 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.

J2EE 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://java.sun.com/j2se/5.0/docs/guide/extensions>) and the *JAR File Specification* (available at <http://java.sun.com/j2se/5.0/docs/guide/jar/jar.html>). Using this mechanism a J2EE 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 J2EE application package, or that has been previously installed in the J2EE containers.

### J2EE.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*

The J2EE deployment tools must process all such referenced files and directories when processing a J2EE 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 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.

2. 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.

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.

3. A web application may include libraries in the WEB-INF/lib directory. See the 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.

### **J2EE.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, EJB `.jar` files, application `.jar` files, and resource adapter `.rar` files within the `.ear` file.

A J2EE product is not required to support downloading of libraries (using the `<extension>-Implementation-URL` header) at deployment time or runtime. A J2EE product is also not required to support more than a single version of an installed library at once. A J2EE 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 J2EE product support multiple versions of an installed library, and only allow applications to access the installed libraries for which they have expressed a dependency.

### **J2EE.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 J2EE platform version on which the application is being deployed, the platform version may (and typically will) take precedence.

### **J2EE.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.

### **J2EE.8.2.5      Dynamic Class Loading**

Libraries that dynamically load classes must consider the class loading environment of a J2EE 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. 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. 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 J2SE and J2EE.

```
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)
    clazz = cl.loadClass(name);
else
    clazz = Class.forName(name);
```

### J2EE.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 servlet1.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 EJB 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 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/servlet1.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-Extension-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
```

## **J2EE.8.3     Application Assembly**

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

### **J2EE.8.3.1     Assembling a J2EE Application**

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

The directory structure of an application is arbitrary, 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.

3. Reconcile J2EE module deployment descriptors.

The deployment descriptors for the J2EE modules must be edited to link inter-



nally satisfied dependencies and eliminate any redundant security role names. An optional element `alt-dd` (described in Section J2EE.8.5, “J2EE 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.

- a. Link the internally satisfied dependencies of all components in every module contained in the application. For each component dependency, there must only be one corresponding component that fulfills that dependency in the scope of the application.
  - i. For each `ejb-link`, there must be only one matching `ejb-name` in the scope of the entire application (see Section J2EE.5.5, “Enterprise JavaBeans™ (EJB) References”).
  - ii. Dependencies that are not linked to internal components must be handled by the Deployer as external dependencies that must be met by resources previously installed on the platform. External dependencies must be linked to the resources on the platform during deployment.
- b. Synchronize security role-names across the application. Rename unique role-names with redundant meaning to a common name. Rename role-names with common names but different meanings to unique names. Descriptions of role-names that are used by many components of the application can be included in the application-level deployment descriptor.
- c. Assign a context root for each web module included in the J2EE application. The context root is a relative name in the web namespace for the application. Each web module must be given a distinct 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 J2EE application, the context root may be the empty string. If no deployment descriptor is included in the application package, the context root of the web module will be the name of the web module file relative to the root of the application package, with the `.war` extension removed. See the servlet specification for detailed requirements of context root naming.

- d. Make sure that each component in the application properly describes any dependencies it may have on other components in the application. A J2EE application should not assume that all components in the application will be available on the 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.
- e. 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. A J2EE application should not assume that each component is loaded in a separate class loader and has a separate namespace. All components in a single application may be loaded in a single class loader and share a single namespace. Note, 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.

4. (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 J2EE:application XML document. (Alternatively, the deployment descriptor may meet the requirements of previous versions of J2EE.)

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.

5. Package the application.

- a. Place the J2EE modules and the deployment descriptor in the appropriate directories.

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

### **J2EE.8.3.2 Adding and Removing Modules**

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

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

## **J2EE.8.4 Deployment**

The J2EE platform supports three types of deployment units:

- Stand-alone J2EE modules.
- J2EE applications, consisting of one or more J2EE modules. A J2EE application must include one J2EE application deployment descriptor.
- Class libraries packaged as `.jar` files according to the *Extension Mechanism Architecture*. These class libraries then become installed libraries.

Any J2EE product must be able to accept a J2EE application delivered as a `.ear` file or a stand-alone J2EE module delivered as a `.jar`, `.war`, or `.rar` file (as appropriate to its type). 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 Java classes in the application are in a separate namespace from classes in other Java 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 J2EE product.

In all cases, the deployment of a J2EE application must be complete before the container delivers requests to any of the application's components. When an application is started, the container must deliver requests to enterprise bean components immediately. Containers must deliver requests to web components and resource adapters only after initialization of the component has completed.

The J2EE Deployment API describes how a product-independent deployment tool accepts plugins for a specific J2EE product, and how the tool and those plugins cooperate to deploy J2EE 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 J2EE 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 J2EE 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 J2EE containers must validate the deployment descriptors against the J2EE 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.

### **J2EE.8.4.1      Deploying a Stand-Alone J2EE Module**

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

1. The deployment tool must first read the J2EE module deployment descriptor 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 J2EE 5.0 version descriptor and the `full` 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 J2EE 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. The deployment tool must deploy all of the components listed in the J2EE module deployment descriptor, or marked via annotations and discovered as described in the previous requirement, according to the deployment requirements of the respective J2EE 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.
4. 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.
5. 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.

### **J2EE.8.4.2      Deploying a J2EE Application**

This section specifies the requirements for deploying a J2EE application.

1. The deployment tool must first read the J2EE application deployment descriptor from the application `.ear` file (`META-INF/application.xml`). If no deployment descriptor is present, the deployment tool uses the following rules to determine the components included in the application.
  - a. 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 name of the file relative to the root of the application package, but with the `.war` extension removed.
  - b. All files in the application package with a filename extension of `.rar` are considered resource adapters.
  - c. A directory named `lib` is considered to be the library directory, as described in Section J2EE.8.2.1, “Bundled Libraries.”
  - d. For all files in the application package with a filename extension of `.jar`, but not contained in the `lib` directory, do the following:
    - i. 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.
    - ii. If the JAR file contains a `META-INF/ejb-jar.xml` file, or contains any class with an EJB component annotation (`Stateless`, etc.), consider the JAR file to be an EJB module.
    - iii. 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.
2. The deployment tool must open each of the J2EE modules listed in the J2EE application deployment descriptor or discovered using the rules above and read the J2EE module deployment descriptor, if present in the package. See the Enterprise JavaBeans, servlet, J2EE Connector and application client specifications for the required location and name of the deployment descriptor for each component type. (The application client specification is Chapter J2EE.9, “Application Clients”.)

3. If the module deployment descriptor is absent, or is present and is a J2EE 5.0 version descriptor and the `full` attribute is not set to `true`, the deployment tool must examine all the class files in the application package [XXX - and referenced jar files?]. 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 J2EE 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.
4. 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 J2EE component specification. All classes in `.jar` files referenced from other JAR files using the `Class-Path` manifest header must be included in the deployment. 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.
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 3.
6. The deployment tool must allow the Deployer to deploy the same J2EE application multiple times, as multiple independent applications, possibly with different configurations. For example, the enterprise beans in an `ejb-jar` file might be deployed multiple times under different JNDI names and with different configurations of their resources.
7. When presenting security role descriptions to the Deployer, the deployment tool must use the descriptions in the J2EE application deployment descriptor rather than the descriptions in any 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.

### J2EE.8.4.3 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 J2EE 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 J2EE products at all times. Libraries that conflict with the operation of the J2EE product may not be deployable. For example, an attempt to deploy an older version of a library that has subsequently been included in the J2EE platform specification may be rejected. Similarly, deployment of a library that is also used in the implementation of the J2EE product may be rejected. Deployment of a library that is in active use by an application may be rejected.

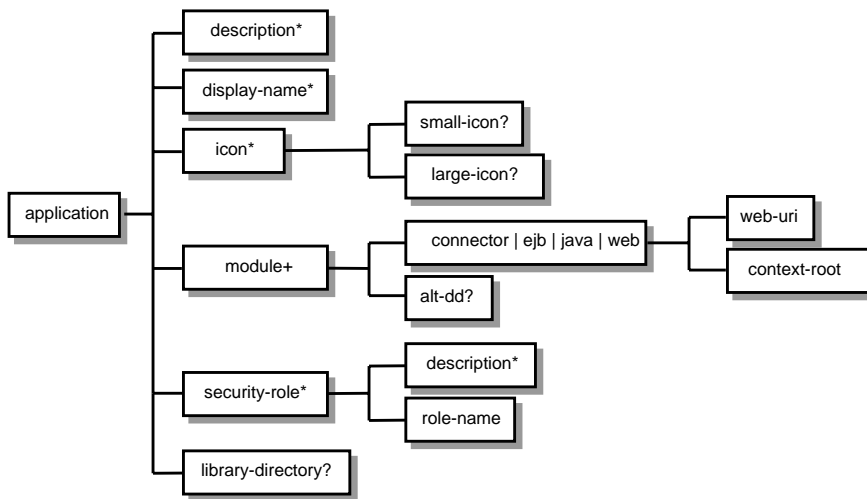
## J2EE.8.5 J2EE Application XML Schema

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>`.

All valid J2EE application deployment descriptors must conform to the XML Schema definition below, or the DTD or schema definition from a previous version of this specification. (See Appendix J2EE.A, “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.



**Figure J2EE.8-3** shows a graphic representation of the structure of the J2EE application XML Schema.



**Figure J2EE.8-3** J2EE Application XML Schema Structure

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

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://java.sun.com/xml/ns/j2ee"
  xmlns:j2ee="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
  version="5.0">

  <xsd:annotation>
    <xsd:documentation>

      @(#)application_5_0.xsds 1.15 01/04/05

    </xsd:documentation>
  </xsd:annotation>

```

```
<xsd:annotation>
  <xsd:documentation>
```

*This is the XML Schema for the application 5.0 deployment descriptor. The deployment descriptor must be named "META-INF/application.xml" in the application's ear file. All application deployment descriptors must indicate the application schema by using the J2EE namespace:*

*http://java.sun.com/xml/ns/j2ee*

*and indicate the version of the schema by using the version element as shown below:*

```
<application xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/application_5_0.xsd"
  version="5.0">
  ...
</application>
```

*The instance documents may indicate the published version of the schema using the xsi:schemaLocation attribute for J2EE namespace with the following location:*

*http://java.sun.com/xml/ns/j2ee/application\_5\_0.xsd*

```
</xsd:documentation>
</xsd:annotation>
```

```
<xsd:annotation>
  <xsd:documentation>
```

*The following conventions apply to all J2EE deployment descriptor elements unless indicated otherwise.*

- *In elements that specify a pathname to a file within the same JAR file, relative filenames (i.e., those not starting with "/") are considered relative to the root of the JAR file's namespace. Absolute filenames (i.e., those starting with "/") also specify names in the root of the JAR file's namespace. In general, relative names are*

*preferred. The exception is .war files where absolute names are preferred for consistency with the Servlet API.*

```

    </xsd:documentation>
  </xsd:annotation>

  <xsd:include schemaLocation="j2ee_5_0.xsd"/>

<!-- ***** -->
  <xsd:element name="application" type="j2ee:applicationType">

    <xsd:annotation>
      <xsd:documentation>

        The application element is the root element of a J2EE
        application deployment descriptor.

      </xsd:documentation>
    </xsd:annotation>

    <xsd:unique name="context-root-uniqueness">

      <xsd:annotation>
        <xsd:documentation>

          The context-root element content must be unique
          in the ear.

        </xsd:documentation>
      </xsd:annotation>

      <xsd:selector xpath="j2ee:module/j2ee:web"/>
      <xsd:field xpath="j2ee:context-root"/>
    </xsd:unique>
    <xsd:unique name="security-role-uniqueness">

      <xsd:annotation>
        <xsd:documentation>

          The security-role-name element content
          must be unique in the ear.

```

```

        </xsd:documentation>
    </xsd:annotation>

    <xsd:selector xpath="j2ee:security-role"/>
    <xsd:field xpath="j2ee:role-name"/>
</xsd:unique>
</xsd:element>

<!-- ***** -->
<xsd:complexType name="applicationType">

    <xsd:annotation>
        <xsd:documentation>

            The applicationType defines the structure of the
            application.

        </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>
        <xsd:group ref="j2ee:descriptionGroup"/>
        <xsd:element name="module"
            type="j2ee:moduleType"
            maxOccurs="unbounded">

            <xsd:annotation>
                <xsd:documentation>

                    The application deployment descriptor must have one
                    module element for each J2EE module in the
                    application package. A module element is defined
                    by moduleType definition.

                </xsd:documentation>
            </xsd:annotation>

        </xsd:element>
        <xsd:element name="security-role"
            type="j2ee:security-roleType"
            minOccurs="0"
            maxOccurs="unbounded"/>
        <xsd:element name="library-directory"

```

```

        type="j2ee:pathType"
        minOccurs="0"
        maxOccurs="1">

<xsd:annotation>
  <xsd:documentation>

    The library-directory element specifies the pathname
    of a directory within the application package, relative
    to the top level of the application package. All files
    named "*.jar" in this directory must be made available
    in the class path of all components included in this
    application package.

  </xsd:documentation>
</xsd:annotation>

</xsd:element>
</xsd:sequence>
<xsd:attribute name="version"
  type="j2ee:dewey-versionType"
  fixed="5.0"
  use="required">

<xsd:annotation>
  <xsd:documentation>

    The required value for the version is 5.0.

  </xsd:documentation>
</xsd:annotation>

</xsd:attribute>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="moduleType">

  <xsd:annotation>
    <xsd:documentation>

      The moduleType defines a single J2EE module and contains a

```

*connector, ejb, java, or web element, which indicates the module type and contains a path to the module file, and an optional alt-dd element, which specifies an optional URI to the post-assembly version of the deployment descriptor.*

```

</xsd:documentation>
</xsd:annotation>

<xsd:sequence>
  <xsd:choice>
    <xsd:element name="connector"
      type="j2ee:pathType">

      <xsd:annotation>
        <xsd:documentation>

          The connector element specifies the URI of a
            resource adapter archive file, relative to the
            top level of the application package.

        </xsd:documentation>
      </xsd:annotation>

    </xsd:element>
    <xsd:element name="ejb"
      type="j2ee:pathType">

      <xsd:annotation>
        <xsd:documentation>

          The ejb element specifies the URI of an ejb-jar,
            relative to the top level of the application
            package.

        </xsd:documentation>
      </xsd:annotation>

    </xsd:element>
    <xsd:element name="java"
      type="j2ee:pathType">

      <xsd:annotation>
        <xsd:documentation>

```

*The java element specifies the URI of a java application client module, relative to the top level of the application package.*

```
</xsd:documentation>
</xsd:annotation>
```

```
</xsd:element>
<xsd:element name="web"
              type="j2ee:webType"/>
</xsd:choice>
<xsd:element name="alt-dd"
              type="j2ee:pathType"
              minOccurs="0">
```

```
<xsd:annotation>
  <xsd:documentation>
```

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

```
</xsd:documentation>
</xsd:annotation>
```

```
</xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
```

```
<!-- ***** -->
```

```
<xsd:complexType name="webType">
```

```
<xsd:annotation>
  <xsd:documentation>
```

*The webType defines the web-uri and context-root of*

*a web application module.*

```

    </xsd:documentation>
  </xsd:annotation>

  <xsd:sequence>
    <xsd:element name="web-uri"
      type="j2ee:pathType">

      <xsd:annotation>
        <xsd:documentation>

          The web-uri element specifies the URI of a web
          application file, relative to the top level of the
          application package.

        </xsd:documentation>
      </xsd:annotation>

    </xsd:element>
    <xsd:element name="context-root"
      type="j2ee:string">

      <xsd:annotation>
        <xsd:documentation>

          The context-root element specifies the context root
          of a web application.

        </xsd:documentation>
      </xsd:annotation>

    </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

</xsd:schema>

```



## J2EE.8.6 Common J2EE XML Schema Definitions

The following XML Schema defines types that are used by many other J2EE deployment descriptor schemas, both in this specification and in other specifications.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
  targetNamespace="http://java.sun.com/xml/ns/j2ee"
  xmlns:j2ee="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
  version="5.0">

  <xsd:annotation>
    <xsd:documentation>

      @(#)j2ee_5_0.xsds 1.50 05/06/08

    </xsd:documentation>
  </xsd:annotation>

  <xsd:annotation>
    <xsd:documentation>
```

*The following definitions that appear in the common shareable schema(s) of J2EE deployment descriptors should be interpreted with respect to the context they are included:*

*Deployment Component may indicate one of the following:*

```
j2ee application;
application client;
web application;
enterprise bean;
resource adapter;
```

*Deployment File may indicate one of the following:*

```
ear file;
war file;
jar file;
rar file;
```

```

</xsd:documentation>
</xsd:annotation>

<xsd:import namespace="http://www.w3.org/XML/1998/namespace"
            schemaLocation="http://www.w3.org/2001/xml.xsd"/>
<xsd:include schemaLocation=
            "http://www.ibm.com/webservices/xsd/
j2ee_web_services_client_1_1.xsd"/>

<!-- ***** -->
<xsd:group name="descriptionGroup">

    <xsd:annotation>
        <xsd:documentation>

            This group keeps the usage of the contained description related
            elements consistent across J2EE deployment descriptors.

            All elements may occur multiple times with different languages,
            to support localization of the content.

        </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>
        <xsd:element name="description"
                    type="j2ee:descriptionType"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
        <xsd:element name="display-name"
                    type="j2ee:display-nameType"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
        <xsd:element name="icon"
                    type="j2ee:iconType"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:group>

<!-- ***** -->
<xsd:complexType name="descriptionType">

```

```
<xsd:annotation>
  <xsd:documentation>
```

*The description type is used by a description element to provide text describing the parent element. The elements that use this type should include any information that the Deployment Component's Deployment File file producer wants to provide to the consumer of the Deployment Component's Deployment File (i.e., to the Deployer). Typically, the tools used by such a Deployment File consumer will display the description when processing the parent element that contains the description.*

*The lang attribute defines the language that the description is provided in. The default value is "en" (English).*

```
  </xsd:documentation>
</xsd:annotation>
```

```
<xsd:simpleContent>
  <xsd:extension base="j2ee:xsdStringType">
    <xsd:attribute ref="xml:lang"/>
  </xsd:extension>
</xsd:simpleContent>
</xsd:complexType>
```

```
<!-- ***** -->
```

```
<xsd:simpleType name="dewey-versionType">
```

```
  <xsd:annotation>
    <xsd:documentation>
```

*This type defines a dewey decimal that is used to describe versions of documents.*

```
  </xsd:documentation>
</xsd:annotation>
```

```
<xsd:restriction base="xsd:token">
  <xsd:pattern value=".[0-9]+(. [0-9]+)*"/>
</xsd:restriction>
</xsd:simpleType>
```

```

<!-- ***** -->
<xsd:complexType name="display-nameType">

  <xsd:annotation>
    <xsd:documentation>

      The display-name type contains a short name that is intended
      to be displayed by tools. It is used by display-name
      elements. The display name need not be unique.

      Example:

      ...
      <display-name xml:lang="en">
        Employee Self Service
      </display-name>

      The value of the xml:lang attribute is "en" (English) by default.

    </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleContent>
    <xsd:extension base="j2ee:string">
      <xsd:attribute ref="xml:lang"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="ejb-linkType">

  <xsd:annotation>
    <xsd:documentation>

      The ejb-linkType is used by ejb-link
      elements in the ejb-ref or ejb-local-ref elements to specify
      that an EJB reference is linked to enterprise bean.

      The value of the ejb-link element must be the ejb-name of an
      enterprise bean in the same ejb-jar file or in another ejb-jar
      file in the same J2EE application unit.

```

*Alternatively, the name in the `ejb-link` element may be composed of a path name specifying the `ejb-jar` containing the referenced enterprise bean with the `ejb-name` of the target bean appended and separated from the path name by "#". The path name is relative to the Deployment File containing Deployment Component that is referencing the enterprise bean. This allows multiple enterprise beans with the same `ejb-name` to be uniquely identified.*

*Examples:*

```
<ejb-link>EmployeeRecord</ejb-link>
```

```
<ejb-link>../products/product.jar#ProductEJB</ejb-link>
```

```
</xsd:documentation>
```

```
</xsd:annotation>
```

```
<xsd:simpleContent>
```

```
<xsd:restriction base="j2ee:string"/>
```

```
</xsd:simpleContent>
```

```
</xsd:complexType>
```

```
<!-- ***** -->
```

```
<xsd:complexType name="ejb-local-refType">
```

```
<xsd:annotation>
```

```
<xsd:documentation>
```

*The `ejb-local-refType` is used by `ejb-local-ref` elements for the declaration of a reference to an enterprise bean's local home. The declaration consists of:*

- an optional description*
- the EJB reference name used in the code of the Deployment Component that's referencing the enterprise bean*
- the expected type of the referenced enterprise bean*
- the expected local home and local interfaces of the referenced enterprise bean*
- optional `ejb-link` information, used to specify the referenced enterprise bean*

```

    </xsd:documentation>
  </xsd:annotation>

  <xsd:sequence>
    <xsd:element name="description"
      type="j2ee:descriptionType"
      minOccurs="0"
      maxOccurs="unbounded"/>
    <xsd:element name="ejb-ref-name"
      type="j2ee:ejb-ref-nameType"/>
    <xsd:element name="ejb-ref-type"
      type="j2ee:ejb-ref-typeType"/>
    <xsd:element name="local-home"
      type="j2ee:local-homeType"/>
    <xsd:element name="local"
      type="j2ee:localType"/>
    <xsd:element name="ejb-link"
      type="j2ee:ejb-linkType"
      minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="ejb-ref-nameType">

  <xsd:annotation>
    <xsd:documentation>

      The ejb-ref-name element contains the name of an EJB
      reference. The EJB reference is an entry in the
      Deployment Component's environment and is relative to the
      java:comp/env context. The name must be unique within the
      Deployment Component.

      It is recommended that name is prefixed with "ejb/".

      Example:

      <ejb-ref-name>ejb/Payroll</ejb-ref-name>

    </xsd:documentation>
  </xsd:annotation>

```

```

    <xsd:simpleContent>
      <xsd:restriction base="j2ee:jndi-nameType"/>
    </xsd:simpleContent>
  </xsd:complexType>

<!-- ***** -->
  <xsd:complexType name="ejb-ref-typeType">

    <xsd:annotation>
      <xsd:documentation>

        The ejb-ref-typeType contains the expected type of the
        referenced enterprise bean.

        The ejb-ref-type designates a value
        that must be one of the following:

          Entity
          Session

      </xsd:documentation>
    </xsd:annotation>

    <xsd:simpleContent>
      <xsd:restriction base="j2ee:string">
        <xsd:enumeration value="Entity"/>
        <xsd:enumeration value="Session"/>
      </xsd:restriction>
    </xsd:simpleContent>
  </xsd:complexType>

<!-- ***** -->
  <xsd:complexType name="ejb-refType">

    <xsd:annotation>
      <xsd:documentation>

        The ejb-refType is used by ejb-ref elements for the
        declaration of a reference to an enterprise bean's home. The
        declaration consists of:

          - an optional description
          - the EJB reference name used in the code of

```

*the Deployment Component that's referencing the enterprise bean*

- *the expected type of the referenced enterprise bean*
- *the expected home and remote interfaces of the referenced enterprise bean*
- *optional ejb-link information, used to specify the referenced enterprise bean*

```

    </xsd:documentation>
</xsd:annotation>

<xsd:sequence>
  <xsd:element name="description"
    type="j2ee:descriptionType"
    minOccurs="0"
    maxOccurs="unbounded"/>
  <xsd:element name="ejb-ref-name"
    type="j2ee:ejb-ref-nameType"/>
  <xsd:element name="ejb-ref-type"
    type="j2ee:ejb-ref-typeType"/>
  <xsd:element name="home"
    type="j2ee:homeType"/>
  <xsd:element name="remote"
    type="j2ee:remoteType"/>
  <xsd:element name="ejb-link"
    type="j2ee:ejb-linkType"
    minOccurs="0"/>
</xsd:sequence>
  <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="emptyType">

  <xsd:annotation>
    <xsd:documentation>

      This type is used to designate an empty
      element when used.

    </xsd:documentation>
  </xsd:annotation>

```



```

    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

<!-- ***** -->
<xsd:complexType name="env-entry-type-valuesType">

  <xsd:annotation>
    <xsd:documentation>

      This type contains the fully-qualified Java type of the
      environment entry value that is expected by the
      application's code.

      The following are the legal values of env-entry-type-valuesType:

        java.lang.Boolean
        java.lang.Byte
        java.lang.Character
        java.lang.String
        java.lang.Short
        java.lang.Integer
        java.lang.Long
        java.lang.Float
        java.lang.Double

      Example:

        <env-entry-type>java.lang.Boolean</env-entry-type>

    </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleContent>
    <xsd:restriction base="j2ee:string">
      <xsd:enumeration value="java.lang.Boolean"/>
      <xsd:enumeration value="java.lang.Byte"/>
      <xsd:enumeration value="java.lang.Character"/>
      <xsd:enumeration value="java.lang.String"/>
      <xsd:enumeration value="java.lang.Short"/>
      <xsd:enumeration value="java.lang.Integer"/>
      <xsd:enumeration value="java.lang.Long"/>
      <xsd:enumeration value="java.lang.Float"/>
      <xsd:enumeration value="java.lang.Double"/>
    </xsd:restriction>
  </xsd:simpleContent>

```

```

        </xsd:restriction>
    </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="env-entryType">

    <xsd:annotation>
        <xsd:documentation>

            The env-entryType is used to declare an application's
            environment entry. The declaration consists of an optional
            description, the name of the environment entry, a type
            (optional if the value is injected, otherwise required), and
            an optional value.

            It also includes optional elements to define injection of
            the named resource into fields or JavaBeans properties.

            If a value is not specified and injection is requested,
            no injection will occur and no entry of the specified name
            will be created. This allows an initial value to be
            specified in the source code without being incorrectly
            changed when no override has been specified.

            If a value is not specified and no injection is requested,
            a value must be supplied during deployment.

            This type is used by env-entry elements.

        </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>
        <xsd:element name="description"
            type="j2ee:descriptionType"
            minOccurs="0"
            maxOccurs="unbounded"/>
        <xsd:element name="env-entry-name"
            type="j2ee:jndi-nameType">

            <xsd:annotation>
                <xsd:documentation>

```

*The env-entry-name element contains the name of a Deployment Component's environment entry. The name is a JNDI name relative to the java:comp/env context. The name must be unique within a Deployment Component. The uniqueness constraints must be defined within the declared context.*

*Example:*

```
<env-entry-name>minAmount</env-entry-name>
```

```
</xsd:documentation>
```

```
</xsd:annotation>
```

```
</xsd:element>
```

```
<xsd:element name="env-entry-type"
              type="j2ee:env-entry-type-valuesType"
              minOccurs="0">
```

```
<xsd:annotation>
```

```
<xsd:documentation>
```

*The env-entry-type element contains the Java language type of the environment entry. If an injection target is specified for the environment entry, the type may be omitted, or must match the type of the injection target. If no injection target is specified, the type is required.*

*Example:*

```
<env-entry-type>java.lang.Integer</env-entry-type>
```

```
</xsd:documentation>
```

```
</xsd:annotation>
```

```
</xsd:element>
```

```
<xsd:element name="env-entry-value"
              type="j2ee:xsdStringType"
              minOccurs="0">
```

```

<xsd:annotation>
  <xsd:documentation>

    The env-entry-value designates the value of a
    Deployment Component's environment entry. The value
    must be a String that is valid for the
    constructor of the specified type that takes a
    single String parameter, or for java.lang.Character,
    a single character.

    Example:

    <env-entry-value>100.00</env-entry-value>

  </xsd:documentation>
</xsd:annotation>

</xsd:element>
<xsd:element name="injection-target"
  type="j2ee:injection-targetType"
  minOccurs="0"
  maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="fully-qualified-classType">

  <xsd:annotation>
    <xsd:documentation>

      The elements that use this type designate the name of a
      Java class or interface. The name is in the form of a
      "binary name", as defined in the JLS. This is the form
      of name used in Class.forName(). Tools that need the
      canonical name (the name used in source code) will need
      to convert this binary name to the canonical name.

    </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleContent>

```

```

        <xsd:restriction base="j2ee:string"/>
    </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="generic-booleanType">

    <xsd:annotation>
        <xsd:documentation>

            This type defines four different values which can designate
            boolean values. This includes values yes and no which are
            not designated by xsd:boolean

        </xsd:documentation>
    </xsd:annotation>

    <xsd:simpleContent>
        <xsd:restriction base="j2ee:string">
            <xsd:enumeration value="true"/>
            <xsd:enumeration value="false"/>
            <xsd:enumeration value="yes"/>
            <xsd:enumeration value="no"/>
        </xsd:restriction>
    </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="homeType">

    <xsd:annotation>
        <xsd:documentation>

            The homeType defines the fully-qualified name of
            an enterprise bean's home interface.

            Example:

                <home>com.aardvark.payroll.PayrollHome</home>

        </xsd:documentation>
    </xsd:annotation>

```

```

<xsd:simpleContent>
  <xsd:restriction base="j2ee:fully-qualified-classType"/>
</xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="iconType">

  <xsd:annotation>
    <xsd:documentation>

      The icon type contains small-icon and large-icon elements
      that specify the file names for small and large GIF, JPEG,
      or PNG icon images used to represent the parent element in a
      GUI tool.

      The xml:lang attribute defines the language that the
      icon file names are provided in. Its value is "en" (English)
      by default.

    </xsd:documentation>
  </xsd:annotation>

  <xsd:sequence>
    <xsd:element name="small-icon" type="j2ee:pathType"
      minOccurs="0">

      <xsd:annotation>
        <xsd:documentation>

          The small-icon element contains the name of a file
          containing a small (16 x 16) icon image. The file
          name is a relative path within the Deployment
          Component's Deployment File.

          The image may be in the GIF, JPEG, or PNG format.
          The icon can be used by tools.

          Example:

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

```

```

        </xsd:documentation>
    </xsd:annotation>

</xsd:element>
<xsd:element name="large-icon" type="j2ee:pathType"
    minOccurs="0">

    <xsd:annotation>
        <xsd:documentation>

            The large-icon element contains the name of a file
            containing a large
            (32 x 32) icon image. The file name is a relative
            path within the Deployment Component's Deployment
            File.

            The image may be in the GIF, JPEG, or PNG format.
            The icon can be used by tools.

            Example:

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

        </xsd:documentation>
    </xsd:annotation>

</xsd:element>
</xsd:sequence>
<xsd:attribute ref="xml:lang"/>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="injection-targetType">

    <xsd:annotation>
        <xsd:documentation>

            An injection target specifies a class and a name within
            that class into which a resource should be injected.

            The injection target class specifies the fully qualified
            class name that is the target of the injection. The

```

*J2EE specifications describe which classes can be an injection target.*

*The injection target name specifies the target within the specified class. The target is first looked for as a JavaBeans property name. If not found, the target is looked for as a field name.*

*The specified resource will be injected into the target during initialization of the class by either calling the set method for the target property or by setting a value into the named field.*

```

</xsd:documentation>
</xsd:annotation>

<xsd:sequence>
  <xsd:element name="injection-target-class"
    type="j2ee:fully-qualified-classType"/>
  <xsd:element name="injection-target-name"
    type="j2ee:java-identifierType"/>
</xsd:sequence>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="java-identifierType">

  <xsd:annotation>
    <xsd:documentation>

      The java-identifierType defines a Java identifier.
      The users of this type should further verify that
      the content does not contain Java reserved keywords.

    </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleContent>
    <xsd:restriction base="j2ee:string">
      <xsd:pattern value="($|_|p{L})(p{L}|p{Nd}|_|$)*"/>
    </xsd:restriction>
  </xsd:simpleContent>
</xsd:complexType>

```



```

<!-- ***** -->
<xsd:complexType name="java-typeType">

  <xsd:annotation>
    <xsd:documentation>

      This is a generic type that designates a Java primitive
      type or a fully qualified name of a Java interface/type,
      or an array of such types.

    </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleContent>
    <xsd:restriction base="j2ee:string">
      <xsd:pattern value="^[p{Z}]*"/>
    </xsd:restriction>
  </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="jndi-nameType">

  <xsd:annotation>
    <xsd:documentation>

      The jndi-nameType type designates a JNDI name in the
      Deployment Component's environment and is relative to the
      java:comp/env context. A JNDI name must be unique within the
      Deployment Component.

    </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleContent>
    <xsd:restriction base="j2ee:string"/>
  </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:group name="jndiEnvironmentRefsGroup">

```

```

<xsd:annotation>
  <xsd:documentation>

    This group keeps the usage of the contained JNDI environment
    reference elements consistent across J2EE deployment descriptors.

  </xsd:documentation>
</xsd:annotation>

<xsd:sequence>
  <xsd:element name="env-entry"
    type="j2ee:env-entryType"
    minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="ejb-ref"
    type="j2ee:ejb-refType"
    minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="ejb-local-ref"
    type="j2ee:ejb-local-refType"
    minOccurs="0" maxOccurs="unbounded"/>
  <xsd:group ref="j2ee:service-refGroup"/>
  <xsd:element name="resource-ref"
    type="j2ee:resource-refType"
    minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="resource-env-ref"
    type="j2ee:resource-env-refType"
    minOccurs="0" maxOccurs="unbounded"/>
  <xsd:element name="message-destination-ref"
    type="j2ee:message-destination-refType"
    minOccurs="0" maxOccurs="unbounded"/>
</xsd:sequence>
</xsd:group>

<!-- ***** -->
<xsd:complexType name="listenerType">

  <xsd:annotation>
    <xsd:documentation>

      The listenerType indicates the deployment properties for a web
      application listener bean.

    </xsd:documentation>
  </xsd:annotation>

```

```

<xsd:sequence>
  <xsd:group ref="j2ee:descriptionGroup"/>
  <xsd:element name="listener-class"
    type="j2ee:fully-qualified-classType">

    <xsd:annotation>
      <xsd:documentation>

        The listener-class element declares a class in the
        application must be registered as a web
        application listener bean. The value is the fully
        qualified classname of the listener class.

      </xsd:documentation>
    </xsd:annotation>

  </xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="local-homeType">

  <xsd:annotation>
    <xsd:documentation>

      The local-homeType defines the fully-qualified
      name of an enterprise bean's local home interface.

    </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleContent>
    <xsd:restriction base="j2ee:fully-qualified-classType"/>
  </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="localType">

  <xsd:annotation>
    <xsd:documentation>

```

*The localType defines the fully-qualified name of an enterprise bean's local interface.*

```

    </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleContent>
    <xsd:restriction base="j2ee:fully-qualified-classType"/>
  </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="message-destination-linkType">

  <xsd:annotation>
    <xsd:documentation>

      The message-destination-linkType is used to link a message
      destination reference or message-driven bean to a message
      destination.

      The Assembler sets the value to reflect the flow of messages
      between producers and consumers in the application.

      The value must be the message-destination-name of a message
      destination in the same Deployment File or in another
      Deployment File in the same J2EE application unit.

      Alternatively, the value may be composed of a path name
      specifying a Deployment File containing the referenced
      message destination with the message-destination-name of the
      destination appended and separated from the path name by
      "#". The path name is relative to the Deployment File
      containing Deployment Component that is referencing the
      message destination. This allows multiple message
      destinations with the same name to be uniquely identified.

    </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleContent>
    <xsd:restriction base="j2ee:string"/>

```

```

    </xsd:simpleContent>
  </xsd:complexType>

<!-- ***** -->
<xsd:complexType name="message-destination-refType">

  <xsd:annotation>
    <xsd:documentation>

```

*The message-destination-ref element contains a declaration of Deployment Component's reference to a message destination associated with a resource in Deployment Component's environment. It consists of:*

- *an optional description*
- *the message destination reference name*
- *an optional message destination type*
- *an optional specification as to whether the destination is used for consuming or producing messages, or both. if not specified, "both" is assumed.*
- *an optional link to the message destination*
- *optional injection targets*

*The message destination type must be supplied unless an injection target is specified, in which case the type of the target is used. If both are specified, the type must be assignment compatible with the type of the injection target.*

*Examples:*

```

<message-destination-ref>
  <message-destination-ref-name>jms/StockQueue
</message-destination-ref-name>
  <message-destination-type>javax.jms.Queue
</message-destination-type>
  <message-destination-usage>Consumes
</message-destination-usage>
  <message-destination-link>CorporateStocks
</message-destination-link>
</message-destination-ref>

```

```

    </xsd:documentation>
  </xsd:annotation>

  <xsd:sequence>
    <xsd:element name="description"
      type="j2ee:descriptionType"
      minOccurs="0"
      maxOccurs="unbounded"/>
    <xsd:element name="message-destination-ref-name"
      type="j2ee:jndi-nameType">

      <xsd:annotation>
        <xsd:documentation>

          The message-destination-ref-name element specifies
            the name of a message destination reference; its
            value is the environment entry name used in
            Deployment Component code. The name is a JNDI name
            relative to the java:comp/env context and must be
            unique within an ejb-jar (for enterprise beans) or a
            Deployment File (for others).

        </xsd:documentation>
      </xsd:annotation>

    </xsd:element>
    <xsd:element name="message-destination-type"
      type="j2ee:message-destination-typeType"
      minOccurs="0"/>
    <xsd:element name="message-destination-usage"
      type="j2ee:message-destination-usageType"
      minOccurs="0"/>
    <xsd:element name="message-destination-link"
      type="j2ee:message-destination-linkType"
      minOccurs="0"/>
    <xsd:element name="injection-target"
      type="j2ee:injection-targetType"
      minOccurs="0"
      maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

```

```

<!-- ***** -->
<xsd:complexType name="message-destination-typeType">

  <xsd:annotation>
    <xsd:documentation>

      The message-destination-typeType specifies the type of
      the destination. The type is specified by the Java interface
      expected to be implemented by the destination.

      Example:

      <message-destination-type>javax.jms.Queue
      </message-destination-type>

    </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleContent>
    <xsd:restriction base="j2ee:fully-qualified-classType"/>
  </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="message-destination-usageType">

  <xsd:annotation>
    <xsd:documentation>

      The message-destination-usageType specifies the use of the
      message destination indicated by the reference. The value
      indicates whether messages are consumed from the message
      destination, produced for the destination, or both. The
      Assembler makes use of this information in linking producers
      of a destination with its consumers.

      The value of the message-destination-usage element must be
      one of the following:
        Consumes
        Produces
        ConsumesProduces

```

```

    </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleContent>
    <xsd:restriction base="j2ee:string">
      <xsd:enumeration value="Consumes"/>
      <xsd:enumeration value="Produces"/>
      <xsd:enumeration value="ConsumesProduces"/>
    </xsd:restriction>
  </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="message-destinationType">

```

```

  <xsd:annotation>
    <xsd:documentation>

```

*The message-destinationType specifies a message destination. The logical destination described by this element is mapped to a physical destination by the Deployer.*

*The message destination element contains:*

- an optional description
- an optional display-name
- an optional icon
- a message destination name which must be unique among message destination names within the same Deployment File.

*Example:*

```

    <message-destination>
      <message-destination-name>CorporateStocks
    </message-destination-name>
    </message-destination>

  </xsd:documentation>
</xsd:annotation>

<xsd:sequence>
  <xsd:group ref="j2ee:descriptionGroup"/>

```



```

<xsd:element name="message-destination-name"
              type="j2ee:string">

  <xsd:annotation>
    <xsd:documentation>

      The message-destination-name element specifies a
      name for a message destination. This name must be
      unique among the names of message destinations
      within the Deployment File.

    </xsd:documentation>
  </xsd:annotation>

</xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="param-valueType">

  <xsd:annotation>
    <xsd:documentation>

      This type is a general type that can be used to declare
      parameter/value lists.

    </xsd:documentation>
  </xsd:annotation>

  <xsd:sequence>
    <xsd:element name="description"
                  type="j2ee:descriptionType"
                  minOccurs="0"
                  maxOccurs="unbounded"/>
    <xsd:element name="param-name"
                  type="j2ee:string">

      <xsd:annotation>
        <xsd:documentation>

          The param-name element contains the name of a

```

```

        parameter.

    </xsd:documentation>
</xsd:annotation>

</xsd:element>
<xsd:element name="param-value"
             type="j2ee:xsdStringType">

    <xsd:annotation>
        <xsd:documentation>

            The param-value element contains the value of a
            parameter.

        </xsd:documentation>
    </xsd:annotation>

</xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="pathType">

    <xsd:annotation>
        <xsd:documentation>

            The elements that use this type designate either a relative
            path or an absolute path starting with a "/".

            In elements that specify a pathname to a file within the
            same Deployment File, relative filenames (i.e., those not
            starting with "/") are considered relative to the root of
            the Deployment File's namespace. Absolute filenames (i.e.,
            those starting with "/") also specify names in the root of
            the Deployment File's namespace. In general, relative names
            are preferred. The exception is .war files where absolute
            names are preferred for consistency with the Servlet API.

        </xsd:documentation>
    </xsd:annotation>

```

```

    <xsd:simpleContent>
      <xsd:restriction base="j2ee:string"/>
    </xsd:simpleContent>
  </xsd:complexType>

<!-- ***** -->
  <xsd:complexType name="remoteType">

    <xsd:annotation>
      <xsd:documentation>

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

        Example:

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

      </xsd:documentation>
    </xsd:annotation>

    <xsd:simpleContent>
      <xsd:restriction base="j2ee:fully-qualified-classType"/>
    </xsd:simpleContent>
  </xsd:complexType>

<!-- ***** -->
  <xsd:complexType name="res-authType">

    <xsd:annotation>
      <xsd:documentation>

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

        The value must be one of the two following:

          Application
          Container

```

```

    </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleContent>
    <xsd:restriction base="j2ee:string">
      <xsd:enumeration value="Application"/>
      <xsd:enumeration value="Container"/>
    </xsd:restriction>
  </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="res-sharing-scopeType">

  <xsd:annotation>
    <xsd:documentation>

      The res-sharing-scope type specifies whether connections
      obtained through the given resource manager connection
      factory reference can be shared. The value, if specified,
      must be one of the two following:

        Shareable
        Unshareable

      The default value is Shareable.

    </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleContent>
    <xsd:restriction base="j2ee:string">
      <xsd:enumeration value="Shareable"/>
      <xsd:enumeration value="Unshareable"/>
    </xsd:restriction>
  </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="resource-env-refType">

  <xsd:annotation>
    <xsd:documentation>

```

*The resource-env-refType is used to define resource-env-type elements. It contains a declaration of a Deployment Component's reference to an administered object associated with a resource in the Deployment Component's environment. It consists of an optional description, the resource environment reference name, and an optional indication of the resource environment reference type expected by the Deployment Component code.*

*It also includes optional elements to define injection of the named resource into fields or JavaBeans properties.*

*The resource environment type must be supplied unless an injection target is specified, in which case the type of the target is used. If both are specified, the type must be assignment compatible with the type of the injection target.*

*Example:*

```
<resource-env-ref>
  <resource-env-ref-name>jms/StockQueue
</resource-env-ref-name>
  <resource-env-ref-type>javax.jms.Queue
</resource-env-ref-type>
</resource-env-ref>
```

```
</xsd:documentation>
</xsd:annotation>
```

```
<xsd:sequence>
  <xsd:element name="description"
    type="j2ee:descriptionType"
    minOccurs="0"
    maxOccurs="unbounded"/>
  <xsd:element name="resource-env-ref-name"
    type="j2ee:jndi-nameType">
```

```
<xsd:annotation>
  <xsd:documentation>
```

*The resource-env-ref-name element specifies the name of a resource environment reference; its value is*

*the environment entry name used in the Deployment Component code. The name is a JNDI name relative to the java:comp/env context and must be unique within a Deployment Component.*

```
</xsd:documentation>
</xsd:annotation>
```

```
</xsd:element>
<xsd:element name="resource-env-ref-type"
              type="j2ee:fully-qualified-classType"
              minOccurs="0">
```

```
<xsd:annotation>
  <xsd:documentation>
```

*The resource-env-ref-type element specifies the type of a resource environment reference. It is the fully qualified name of a Java language class or interface.*

```
</xsd:documentation>
</xsd:annotation>
```

```
</xsd:element>
<xsd:element name="injection-target"
              type="j2ee:injection-targetType"
              minOccurs="0"
              maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
```

```
<!-- ***** -->
```

```
<xsd:complexType name="resource-refType">
```

```
<xsd:annotation>
  <xsd:documentation>
```

*The resource-refType contains a declaration of a Deployment Component's reference to an external resource. It consists of an optional description, the resource manager connection factory reference name, an optional indication of*

*the resource manager connection factory type expected by the Deployment Component code, an optional type of authentication (Application or Container), and an optional specification of the shareability of connections obtained from the resource (Shareable or Unshareable).*

*It also includes optional elements to define injection of the named resource into fields or JavaBeans properties.*

*The connection factory type must be supplied unless an injection target is specified, in which case the type of the target is used. If both are specified, the type must be assignment compatible with the type of the injection target.*

*Example:*

```
<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>
</resource-ref>

</xsd:documentation>
</xsd:annotation>

<xsd:sequence>
  <xsd:element name="description"
    type="j2ee:descriptionType"
    minOccurs="0"
    maxOccurs="unbounded"/>
  <xsd:element name="res-ref-name"
    type="j2ee:jndi-nameType">

<xsd:annotation>
  <xsd:documentation>
```

*The res-ref-name element specifies the name of a resource manager connection factory reference. The name is a JNDI name relative to the java:comp/env context. The name must be unique within a Deployment File.*

```

        </xsd:documentation>
    </xsd:annotation>

</xsd:element>
<xsd:element name="res-type"
              type="j2ee:fully-qualified-classType"
              minOccurs="0">

    <xsd:annotation>
        <xsd:documentation>

            The res-type element specifies the type of the data
            source. The type is specified by the fully qualified
            Java language class or interface
            expected to be implemented by the data source.

        </xsd:documentation>
    </xsd:annotation>

</xsd:element>
<xsd:element name="res-auth"
              type="j2ee:res-authType"
              minOccurs="0"/>
<xsd:element name="res-sharing-scope"
              type="j2ee:res-sharing-scopeType"
              minOccurs="0"/>
<xsd:element name="injection-target"
              type="j2ee:injection-targetType"
              minOccurs="0"
              maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="role-nameType">

    <xsd:annotation>
        <xsd:documentation>

            The role-nameType designates the name of a security role.

            The name must conform to the lexical rules for a token.

```



```

        </xsd:documentation>
    </xsd:annotation>

    <xsd:simpleContent>
        <xsd:restriction base="j2ee:string"/>
    </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="run-asType">

    <xsd:annotation>
        <xsd:documentation>

            The run-asType specifies the run-as identity to be
            used for the execution of a component. It contains an
            optional description, and the name of a security role.

        </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>
        <xsd:element name="description"
            type="j2ee:descriptionType"
            minOccurs="0"
            maxOccurs="unbounded"/>
        <xsd:element name="role-name"
            type="j2ee:role-nameType"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="security-role-refType">

    <xsd:annotation>
        <xsd:documentation>

            The security-role-refType contains the declaration of a
            security role reference in a component's or a
            Deployment Component's code. The declaration consists of an
            optional description, the security role name used in the
            code, and an optional link to a security role. If the


```

*security role is not specified, the Deployer must choose an appropriate security role.*

```
</xsd:documentation>
</xsd:annotation>
```

```
<xsd:sequence>
  <xsd:element name="description"
    type="j2ee:descriptionType"
    minOccurs="0"
    maxOccurs="unbounded"/>
  <xsd:element name="role-name"
    type="j2ee:role-nameType">
```

```
<xsd:annotation>
  <xsd:documentation>
```

*The value of the role-name element must be the String used as the parameter to the `EJBContext.isCallerInRole(String roleName)` method or the `HttpServletRequest.isUserInRole(String role)` method.*

```
</xsd:documentation>
</xsd:annotation>
```

```
</xsd:element>
<xsd:element name="role-link"
  type="j2ee:role-nameType"
  minOccurs="0">
```

```
<xsd:annotation>
  <xsd:documentation>
```

*The role-link element is a reference to a defined security role. The role-link element must contain the name of one of the security roles defined in the security-role elements.*

```
</xsd:documentation>
</xsd:annotation>
```

```
</xsd:element>
</xsd:sequence>
```

```

    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

<!-- ***** -->
<xsd:complexType name="security-roleType">

  <xsd:annotation>
    <xsd:documentation>

      The security-roleType contains the definition of a security
      role. The definition consists of an optional description of
      the security role, and the security role name.

      Example:

        <security-role>
          <description>
            This role includes all employees who are authorized
            to access the employee service application.
          </description>
          <role-name>employee</role-name>
        </security-role>

      </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>
      <xsd:element name="description"
        type="j2ee:descriptionType"
        minOccurs="0"
        maxOccurs="unbounded"/>
      <xsd:element name="role-name"
        type="j2ee:role-nameType"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

<!-- ***** -->
<xsd:complexType name="string">

  <xsd:annotation>
    <xsd:documentation>

```

*This is a special string datatype that is defined by J2EE as a base type for defining collapsed strings. When schemas require trailing/leading space elimination as well as collapsing the existing whitespace, this base type may be used.*

```

    </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleContent>
    <xsd:extension base="xsd:token">
      <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="true-falseType">

  <xsd:annotation>
    <xsd:documentation>

      This simple type designates a boolean with only two
      permissible values

      - true
      - false

    </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleContent>
    <xsd:restriction base="j2ee:xsdBooleanType">
      <xsd:pattern value="(true|false)"/>
    </xsd:restriction>
  </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="url-patternType">

  <xsd:annotation>
    <xsd:documentation>

```

*The url-patternType contains the url pattern of the mapping. It must follow the rules specified in Section 11.2 of the Servlet API Specification. This pattern is assumed to be in URL-decoded form and must not contain CR(#xD) or LF(#xA). If it contains those characters, the container must inform the developer with a descriptive error message. The container must preserve all characters including whitespaces.*

```

    </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleContent>
    <xsd:extension base="xsd:string"/>
  </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="xsdAnyURIType">

  <xsd:annotation>
    <xsd:documentation>

      This type adds an "id" attribute to xsd:anyURI.

    </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleContent>
    <xsd:extension base="xsd:anyURI">
      <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="xsdBooleanType">

  <xsd:annotation>
    <xsd:documentation>

      This type adds an "id" attribute to xsd:boolean.

```

```

        </xsd:documentation>
    </xsd:annotation>

    <xsd:simpleContent>
        <xsd:extension base="xsd:boolean">
            <xsd:attribute name="id" type="xsd:ID"/>
        </xsd:extension>
    </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="xsdIntegerType">

    <xsd:annotation>
        <xsd:documentation>

            This type adds an "id" attribute to xsd:integer.

        </xsd:documentation>
    </xsd:annotation>

    <xsd:simpleContent>
        <xsd:extension base="xsd:integer">
            <xsd:attribute name="id" type="xsd:ID"/>
        </xsd:extension>
    </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="xsdNMTOKENType">

    <xsd:annotation>
        <xsd:documentation>

            This type adds an "id" attribute to xsd:NMTOKEN.

        </xsd:documentation>
    </xsd:annotation>

    <xsd:simpleContent>
        <xsd:extension base="xsd:NMTOKEN">
            <xsd:attribute name="id" type="xsd:ID"/>
        </xsd:extension>

```

```

        </xsd:simpleContent>
    </xsd:complexType>

<!-- ***** -->
    <xsd:complexType name="xsdNonNegativeIntegerType">

        <xsd:annotation>
            <xsd:documentation>

                This type adds an "id" attribute to xsd:nonNegativeInteger.

            </xsd:documentation>
        </xsd:annotation>

        <xsd:simpleContent>
            <xsd:extension base="xsd:nonNegativeInteger">
                <xsd:attribute name="id" type="xsd:ID"/>
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>

<!-- ***** -->
    <xsd:complexType name="xsdPositiveIntegerType">

        <xsd:annotation>
            <xsd:documentation>

                This type adds an "id" attribute to xsd:positiveInteger.

            </xsd:documentation>
        </xsd:annotation>

        <xsd:simpleContent>
            <xsd:extension base="xsd:positiveInteger">
                <xsd:attribute name="id" type="xsd:ID"/>
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>

<!-- ***** -->
    <xsd:complexType name="xsdQNameType">

```

```

<xsd:annotation>
  <xsd:documentation>

    This type adds an "id" attribute to xsd:QName.

  </xsd:documentation>
</xsd:annotation>

<xsd:simpleContent>
  <xsd:extension base="xsd:QName">
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:extension>
</xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="xsdStringType">

  <xsd:annotation>
    <xsd:documentation>

      This type adds an "id" attribute to xsd:string.

    </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleContent>
    <xsd:extension base="xsd:string">
      <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

</xsd:schema>

```



# CHAPTER J2EE.9

---

## Application Clients

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

### **J2EE.9.1 Overview**

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

### **J2EE.9.2 Security**

The J2EE authentication requirements for application clients are the same as for other J2EE components, and the same authentication techniques may be used as for other J2EE 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 EJB

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 Section J2EE.9.7, "J2EE 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 execute in an environment with a `SecurityManager` installed, and have similar security permission requirements as servlets. The security permission requirements are described fully in Section J2EE.6.2, "Java 2 Platform, Standard Edition (J2SE) Requirements."

### **J2EE.9.3 Transactions**

Application clients are not required to have direct access to the transaction facilities of the J2EE platform. A J2EE product is not required to provide a `JTA UserTransaction` object for use by application clients. 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 EJB server.

## **J2EE.9.4      Resources, Naming, and Injection**

As with all J2EE 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 Chapter J2EE.5, “Resources, Naming, and Injection” for details).

Resource 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, resource injection into the application client class uses `static` fields and methods, unlike other J2EE components. Resource injection occurs before the `main` method is called.

## **J2EE.9.5      Application Programming Interfaces**

Application clients have all the facilities of the Java™ 2 Platform, Standard Edition (subject to security restrictions), as well as various standard extensions, as described in Chapter J2EE.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).

## **J2EE.9.6      Packaging and Deployment**

Application clients are packaged in JAR format files with a `.jar` extension and may include a deployment descriptor similar to other J2EE 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 `full`, annotations on the main class of

the application client may also be used to describe the resources needed by the application. As with other J2EE application components, access to resources must be configured at deployment time, names assigned for enterprise beans and resources, 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.

**Table J2EE.9-1    Deployment Descriptor Processing Requirements**

Deployment descriptor	full?	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_0	Yes	No
application-client_5_0	No	Yes
none	N/A	Yes

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 J2EE products may allow the application client to be deployed on a J2EE server and automatically made available to some set of (usually intranet) clients. Other J2EE products may require the J2EE application bundle containing the application client to be manually deployed and installed on each client machine. And yet another approach would be for the deployment tool on the J2EE server to produce an installation package that could be 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.

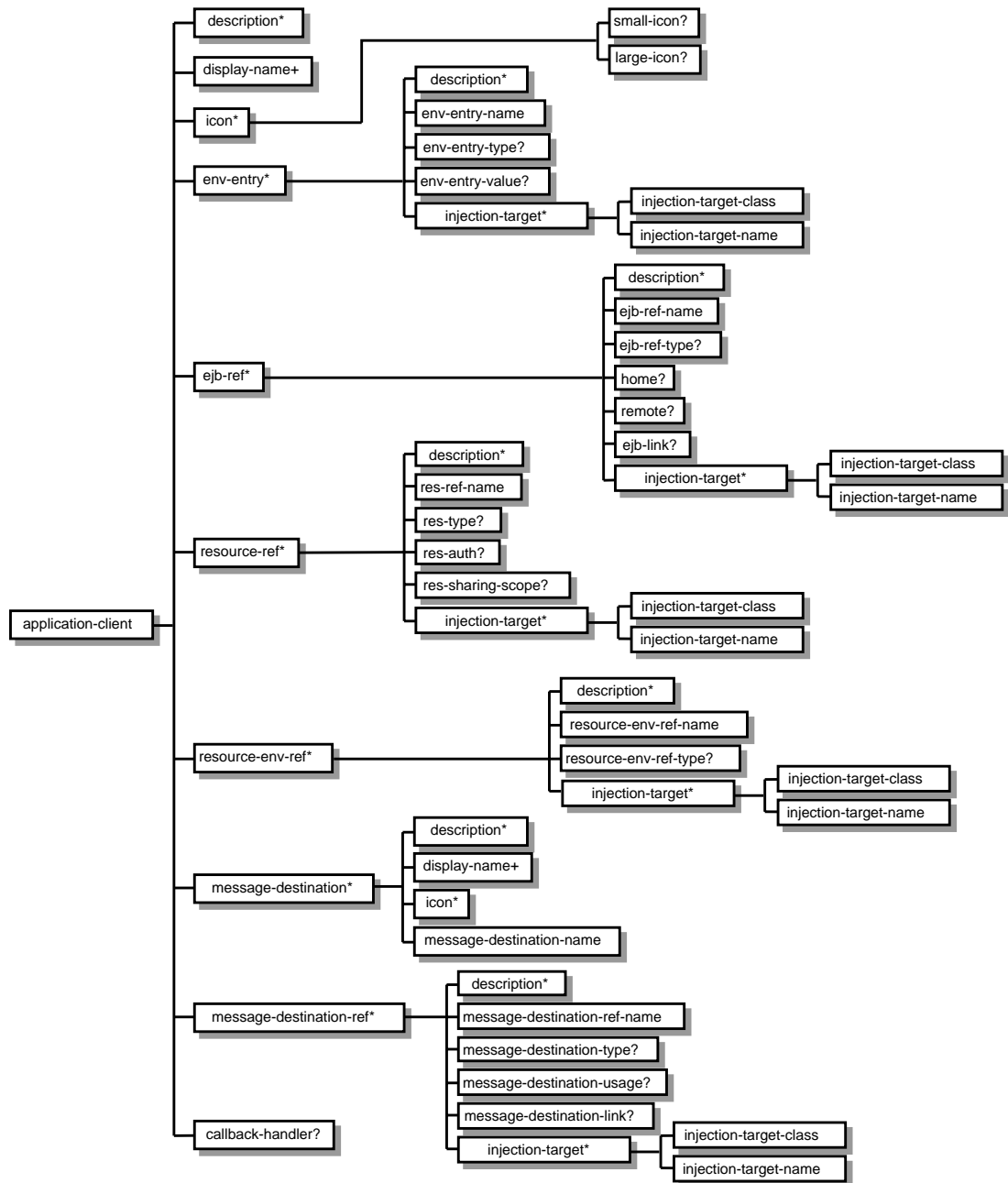
How an application client is invoked by an end user is unspecified. Typically a J2EE Product Provider will provide an application launcher that integrates with the application client machine's native operating system, but the level of such integration is unspecified.

## J2EE.9.7 J2EE 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>`.

All valid `application-client` deployment descriptors must conform to the following XML Schema definition, or to a DTD or schema definition from a previous version of this specification. (See Appendix J2EE.A, “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.

**Figure J2EE.9-1** shows the structure of the J2EE application-client XML Schema.



**Figure J2EE.9-1** J2EE Application Client XML Schema Structure

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://java.sun.com/xml/ns/j2ee"
  xmlns:j2ee="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
  version="5.0">

  <xsd:annotation>
    <xsd:documentation>

      @(#)application-client_5_0.xsds    1.19 03/15/05

    </xsd:documentation>
  </xsd:annotation>

  <xsd:annotation>
    <xsd:documentation>

```

*This is the XML Schema for the application client 5.0 deployment descriptor. The deployment descriptor must be named "META-INF/application-client.xml" in the application client's jar file. All application client deployment descriptors must indicate the application client schema by using the J2EE namespace:*

*http://java.sun.com/xml/ns/j2ee*

*and indicate the version of the schema by using the version element as shown below:*

```

  <application-client xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
      http://java.sun.com/xml/ns/j2ee/application-
client_5_0.xsd"
    version="5.0">
    ...
  </application-client>

```

*The instance documents may indicate the published version of the schema using the xsi:schemaLocation attribute for J2EE*

*namespace with the following location:*

*[http://java.sun.com/xml/ns/j2ee/application-client\\_5\\_0.xsd](http://java.sun.com/xml/ns/j2ee/application-client_5_0.xsd)*

`</xsd:documentation>`

`</xsd:annotation>`

`<xsd:annotation>`

`<xsd:documentation>`

*The following conventions apply to all J2EE deployment descriptor elements unless indicated otherwise.*

- *In elements that specify a pathname to a file within the same JAR file, relative filenames (i.e., those not starting with "/") are considered relative to the root of the JAR file's namespace. Absolute filenames (i.e., those starting with "/") also specify names in the root of the JAR file's namespace. In general, relative names are preferred. The exception is .war files where absolute names are preferred for consistency with the Servlet API.*

`</xsd:documentation>`

`</xsd:annotation>`

`<xsd:include schemaLocation="j2ee_5_0.xsd"/>`

`<!-- ***** -->`

`<xsd:element name="application-client" type="j2ee:application-clientType">`

`<xsd:annotation>`

`<xsd:documentation>`

*The application-client element is the root element of an application client deployment descriptor. The application client deployment descriptor describes the EJB components and external resources referenced by the application client.*

`</xsd:documentation>`

`</xsd:annotation>`



```
<xsd:unique name="env-entry-name-uniqueness">

  <xsd:annotation>
    <xsd:documentation>

      The env-entry-name element contains the name of an
      application client's environment entry. The name is a JNDI
      name relative to the java:comp/env context. The name must
      be unique within an application client.

    </xsd:documentation>
  </xsd:annotation>

  <xsd:selector xpath="j2ee:env-entry"/>
  <xsd:field    xpath="j2ee:env-entry-name"/>
</xsd:unique>
<xsd:unique name="ejb-ref-name-uniqueness">

  <xsd:annotation>
    <xsd:documentation>

      The ejb-ref-name element contains the name of an EJB
      reference. The EJB reference is an entry in the application
      client's environment and is relative to the
      java:comp/env context. The name must be unique within the
      application client.

      It is recommended that name is prefixed with "ejb/".

    </xsd:documentation>
  </xsd:annotation>

  <xsd:selector xpath="j2ee:ejb-ref"/>
  <xsd:field    xpath="j2ee:ejb-ref-name"/>
</xsd:unique>
<xsd:unique name="res-ref-name-uniqueness">

  <xsd:annotation>
    <xsd:documentation>

      The res-ref-name element specifies the name of a
      resource manager connection factory reference. The name
      is a JNDI name relative to the java:comp/env context.


```

*The name must be unique within an application client.*

```

</xsd:documentation>
</xsd:annotation>

<xsd:selector xpath="j2ee:resource-ref"/>
<xsd:field    xpath="j2ee:res-ref-name"/>
</xsd:unique>
<xsd:unique name="resource-env-ref-uniqueness">

```

```

<xsd:annotation>
  <xsd:documentation>

```

*The resource-env-ref-name element specifies the name of a resource environment reference; its value is the environment entry name used in the application client code. The name is a JNDI name relative to the java:comp/env context and must be unique within an application client.*

```

</xsd:documentation>
</xsd:annotation>

```

```

<xsd:selector xpath="j2ee:resource-env-ref"/>
<xsd:field    xpath="j2ee:resource-env-ref-name"/>
</xsd:unique>
<xsd:unique name="message-destination-ref-uniqueness">

```

```

<xsd:annotation>
  <xsd:documentation>

```

*The message-destination-ref-name element specifies the name of a message destination reference; its value is the message destination reference name used in the application client code. The name is a JNDI name relative to the java:comp/env context and must be unique within an application client.*

```

</xsd:documentation>
</xsd:annotation>

```

```

<xsd:selector xpath="j2ee:message-destination-ref"/>
<xsd:field    xpath="j2ee:message-destination-ref-name"/>

```

```

    </xsd:unique>
  </xsd:element>

<!-- ***** -->
<xsd:complexType name="application-clientType">
  <xsd:sequence>
    <xsd:group ref="j2ee:descriptionGroup"/>
    <xsd:element name="env-entry"
      type="j2ee:env-entryType"
      minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="ejb-ref"
      type="j2ee:ejb-refType"
      minOccurs="0" maxOccurs="unbounded"/>
    <xsd:group ref="j2ee:service-refGroup"/>
    <xsd:element name="resource-ref"
      type="j2ee:resource-refType"
      minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="resource-env-ref"
      type="j2ee:resource-env-refType"
      minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="message-destination-ref"
      type="j2ee:message-destination-refType"
      minOccurs="0"
      maxOccurs="unbounded"/>
    <xsd:element name="callback-handler"
      type="j2ee:fully-qualified-classType"
      minOccurs="0">

    <xsd:annotation>
      <xsd:documentation>

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

      </xsd:documentation>
    </xsd:annotation>

  </xsd:element>

```

```

    <xsd:element name="message-destination"
        type="j2ee:message-destinationType"
        minOccurs="0"
        maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="version"
    type="j2ee:dewey-versionType"
    fixed="5.0"
    use="required">

    <xsd:annotation>
        <xsd:documentation>

            The required value for the version is 5.0.

        </xsd:documentation>
    </xsd:annotation>

</xsd:attribute>
<xsd:attribute name="full" type="xsd:boolean">

    <xsd:annotation>
        <xsd:documentation>

            The full attribute defines whether this deployment
            descriptor is complete, or whether the class files
            of the application should be examined for annotations
            that specify deployment information.

            If full is set to "true", the J2EE deployment tool
            must ignore any J2EE annotations present in the
            class files of the application.

            If full is not specified or is set to "false", the J2EE
            deployment tool must examine the class files of the
            application for annotations, as specified by the J2EE
            specifications.

        </xsd:documentation>
    </xsd:annotation>

</xsd:attribute>

```

```
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>
</xsd:schema>
```



# CHAPTER J2EE.10

---

## Service Provider Interface

The Java™ 2 Platform, Enterprise Edition (J2EE) includes the J2EE Connector Architecture and the Java Authorization Service Provider Contract for Containers as its service provider interfaces. The Connector API defines how resource adapters are packaged and integrated with any J2EE product. All J2EE products must support the Connector APIs, as specified in the Connector specification. The JACC specification defines the contract between a J2EE container and an authorization policy provider.

The Connector specification is available at <http://java.sun.com/j2ee/connector>. The JACC specification is available at <http://jcp.org/jsr/detail/115.jsp>.





# CHAPTER J2EE.11

---

## Compatibility and Migration

This chapter is a placeholder. In future drafts this chapter will discuss compatibility with, and migration from, previous versions of J2EE.



# CHAPTER J2EE.12

---

## Future Directions

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

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

### **J2EE.12.1 JNLP (Java™ Web Start)**

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

More information on JNLP is available at <http://jcp.org/en/jsr/detail?id=056>; more information on Java Web Start is available at <http://java.sun.com/products/javawebstart>.

## **J2EE.12.2 J2EE SPI**

Many of the APIs that make up the J2EE 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 J2EE Connector Architecture does define the requirements for certain types of service providers called resource adapters, and the Java Authorization Contract for Containers defines requirements for security service providers. Future versions of this specification will more fully define the J2EE SPI.

## **J2EE.12.3 Security APIs**

It is a goal of the J2EE platform to separate security from business logic, providing declarative security controls for application components. However, some applications need more control over security than can be provided by this approach. A future version of this specification may expand the set of APIs available to control authentication and authorization, and to allow the integration of new security technologies. In particular, we expect that the Java™ Authentication Service Provider Interface for Containers (JSR-196) will be required in the next version of this specification. More information on JSR-196 is available at <http://jcp.org/en/jsr/detail?id=196>.

# APPENDIX J2EE.A

---

## Previous Version Deployment Descriptors

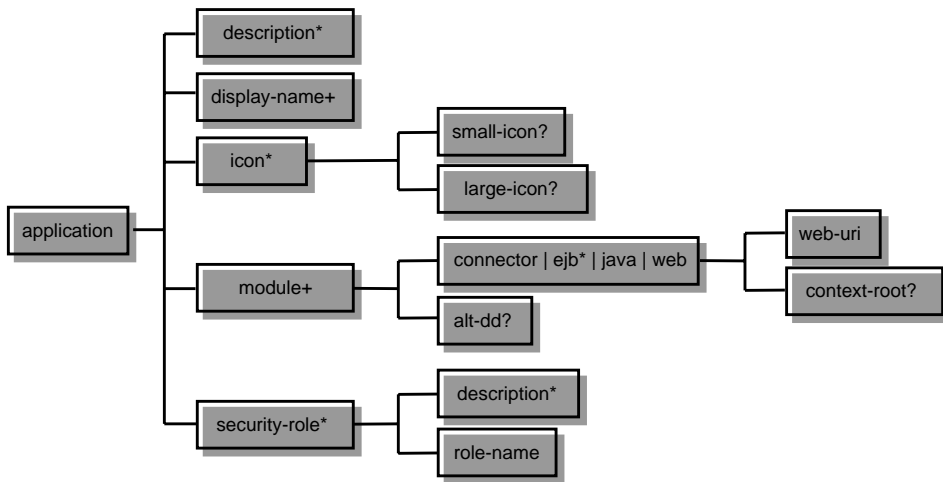
This appendix contains Document Type Definitions and XML schemas for Deployment Descriptors from previous versions of the J2EE specification. All J2EE products are required to support these DTDs and schemas as well as the schemas specified in this version of the specification. This ensures that applications 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.

### **J2EE.A.1 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.

**Figure J2EE.0-1** shows a graphic representation of the structure of the J2EE application XML Schema.



**Figure J2EE.0-1** J2EE Application XML Schema Structure

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

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://java.sun.com/xml/ns/j2ee"
  xmlns:j2ee="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
  version="1.4">

  <xsd:annotation>
    <xsd:documentation>

      @(#)application_1_4.xsds 1.13 02/11/03

    </xsd:documentation>
  </xsd:annotation>

```

```
<xsd:annotation>
  <xsd:documentation>
```

*This is the XML Schema for the application 1.4 deployment descriptor. The deployment descriptor must be named "META-INF/application.xml" in the application's ear file. All application deployment descriptors must indicate the application schema by using the J2EE namespace:*

*<http://java.sun.com/xml/ns/j2ee>*

*and indicate the version of the schema by using the version element as shown below:*

```
<application xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/application_1_4.xsd"
  version="1.4">
  ...
</application>
```

*The instance documents may indicate the published version of the schema using the xsi:schemaLocation attribute for J2EE namespace with the following location:*

*[http://java.sun.com/xml/ns/j2ee/application\\_1\\_4.xsd](http://java.sun.com/xml/ns/j2ee/application_1_4.xsd)*

```
</xsd:documentation>
</xsd:annotation>
```

```
<xsd:annotation>
  <xsd:documentation>
```

*The following conventions apply to all J2EE deployment descriptor elements unless indicated otherwise.*

- *In elements that specify a pathname to a file within the same JAR file, relative filenames (i.e., those not starting with "/" ) are considered relative to the root of the JAR file's namespace. Absolute filenames (i.e., those starting with "/" ) also specify names in the root of the JAR file's namespace. In general, relative names are*

*preferred. The exception is .war files where absolute names are preferred for consistency with the Servlet API.*

```

    </xsd:documentation>
  </xsd:annotation>

  <xsd:include schemaLocation="j2ee_1_4.xsd"/>

<!-- ***** -->
<xsd:element name="application" type="j2ee:applicationType">

  <xsd:annotation>
    <xsd:documentation>

      The application element is the root element of a J2EE
      application deployment descriptor.

    </xsd:documentation>
  </xsd:annotation>

  <xsd:unique name="context-root-uniqueness">

    <xsd:annotation>
      <xsd:documentation>

        The context-root element content must be unique
        in the ear.

      </xsd:documentation>
    </xsd:annotation>

    <xsd:selector xpath="j2ee:module/j2ee:web"/>
    <xsd:field xpath="j2ee:context-root"/>
  </xsd:unique>
  <xsd:unique name="security-role-uniqueness">

    <xsd:annotation>
      <xsd:documentation>

        The security-role-name element content
        must be unique in the ear.

```



```

        </xsd:documentation>
    </xsd:annotation>

    <xsd:selector xpath="j2ee:security-role"/>
    <xsd:field xpath="j2ee:role-name"/>
</xsd:unique>
</xsd:element>

<!-- ***** -->
<xsd:complexType name="applicationType">

    <xsd:annotation>
        <xsd:documentation>

            The applicationType defines the structure of the
            application.

        </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>
        <xsd:group ref="j2ee:descriptionGroup"/>
        <xsd:element name="module"
            type="j2ee:moduleType"
            maxOccurs="unbounded">

            <xsd:annotation>
                <xsd:documentation>

                    The application deployment descriptor must have one
                    module element for each J2EE module in the
                    application package. A module element is defined
                    by moduleType definition.

                </xsd:documentation>
            </xsd:annotation>

        </xsd:element>
        <xsd:element name="security-role"
            type="j2ee:security-roleType"
            minOccurs="0"
            maxOccurs="unbounded"/>
    </xsd:sequence>

```

```

<xsd:attribute name="version"
               type="j2ee:dewey-versionType"
               fixed="1.4"
               use="required">

<xsd:annotation>
  <xsd:documentation>

    The required value for the version is 1.4.

  </xsd:documentation>
</xsd:annotation>

</xsd:attribute>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="moduleType">

  <xsd:annotation>
    <xsd:documentation>

      The moduleType defines a single J2EE module and contains a
      connector, ejb, java, or web element, which indicates the
      module type and contains a path to the module file, and an
      optional alt-dd element, which specifies an optional URI to
      the post-assembly version of the deployment descriptor.

    </xsd:documentation>
  </xsd:annotation>

  <xsd:sequence>
    <xsd:choice>
      <xsd:element name="connector"
                   type="j2ee:pathType">

        <xsd:annotation>
          <xsd:documentation>

            The connector element specifies the URI of a
            resource adapter archive file, relative to the
            top level of the application package.

```

```

        </xsd:documentation>
    </xsd:annotation>

</xsd:element>
<xsd:element name="ejb"
              type="j2ee:pathType">

    <xsd:annotation>
        <xsd:documentation>

            The ejb element specifies the URI of an ejb-jar,
            relative to the top level of the application
            package.

        </xsd:documentation>
    </xsd:annotation>

</xsd:element>
<xsd:element name="java"
              type="j2ee:pathType">

    <xsd:annotation>
        <xsd:documentation>

            The java element specifies the URI of a java
            application client module, relative to the top
            level of the application package.

        </xsd:documentation>
    </xsd:annotation>

</xsd:element>
<xsd:element name="web"
              type="j2ee:webType"/>
</xsd:choice>
<xsd:element name="alt-dd"
              type="j2ee:pathType"
              minOccurs="0">

    <xsd:annotation>
        <xsd:documentation>

            The alt-dd element specifies an optional URI to the

```

*post-assembly version of the deployment descriptor file for a particular J2EE module. The URI must specify the full pathname of the deployment descriptor file relative to the application's root directory. If alt-dd is not specified, the deployer must read the deployment descriptor from the default location and file name required by the respective component specification.*

```

        </xsd:documentation>
    </xsd:annotation>

    </xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="webType">

    <xsd:annotation>
        <xsd:documentation>

            The webType defines the web-uri and context-root of
            a web application module.

        </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>
        <xsd:element name="web-uri"
            type="j2ee:pathType">

            <xsd:annotation>
                <xsd:documentation>

                    The web-uri element specifies the URI of a web
                    application file, relative to the top level of the
                    application package.

                </xsd:documentation>
            </xsd:annotation>

```

```

</xsd:element>
<xsd:element name="context-root"
              type="j2ee:string">

  <xsd:annotation>
    <xsd:documentation>

      The context-root element specifies the context root
      of a web application.

    </xsd:documentation>
  </xsd:annotation>

</xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

</xsd:schema>

```

## J2EE.A.2 Common J2EE 1.4 XML Schema Definitions

The following XML Schema defines types that are used by many other J2EE 1.4 deployment descriptor schemas, both in this specification and in other specifications.

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
  targetNamespace="http://java.sun.com/xml/ns/j2ee"
  xmlns:j2ee="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  attributeFormDefault="unqualified"
  version="1.4">

  <xsd:annotation>
    <xsd:documentation>

      @(#)j2ee_1_4.xsds 1.43 03/09/16
    
```

```

    </xsd:documentation>
  </xsd:annotation>

```

```

<xsd:annotation>
<xsd:documentation>

```

*The following definitions that appear in the common shareable schema(s) of J2EE deployment descriptors should be interpreted with respect to the context they are included:*

*Deployment Component may indicate one of the following:*

```

    j2ee application;
    application client;
    web application;
    enterprise bean;
    resource adapter;

```

*Deployment File may indicate one of the following:*

```

    ear file;
    war file;
    jar file;
    rar file;

```

```

</xsd:documentation>
</xsd:annotation>

```

```

    <xsd:import namespace="http://www.w3.org/XML/1998/namespace"
               schemaLocation="http://www.w3.org/2001/xml.xsd"/>
    <xsd:include schemaLocation=
               "http://www.ibm.com/webservices/xsd/
j2ee_web_services_client_1_1.xsd"/>

```

```

<!-- ***** -->

```

```

  <xsd:group name="descriptionGroup">

```

```

    <xsd:annotation>
    <xsd:documentation>

```

*This group keeps the usage of the contained description related elements consistent across J2EE deployment descriptors.*

*All elements may occur multiple times with different languages, to support localization of the content.*

```

    </xsd:documentation>
</xsd:annotation>

<xsd:sequence>
  <xsd:element name="description"
    type="j2ee:descriptionType"
    minOccurs="0"
    maxOccurs="unbounded"/>
  <xsd:element name="display-name"
    type="j2ee:display-nameType"
    minOccurs="0"
    maxOccurs="unbounded"/>
  <xsd:element name="icon"
    type="j2ee:iconType"
    minOccurs="0"
    maxOccurs="unbounded"/>
</xsd:sequence>
</xsd:group>

<!-- ***** -->
<xsd:complexType name="descriptionType">

  <xsd:annotation>
    <xsd:documentation>

      The description type is used by a description element to
      provide text describing the parent element. The elements
      that use this type should include any information that the
      Deployment Component's Deployment File file producer wants
      to provide to the consumer of the Deployment Component's
      Deployment File (i.e., to the Deployer). Typically, the
      tools used by such a Deployment File consumer will display
      the description when processing the parent element that
      contains the description.

      The lang attribute defines the language that the
      description is provided in. The default value is "en" (English).

    </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleContent>
    <xsd:extension base="j2ee:xsdStringType">

```

```

        <xsd:attribute ref="xml:lang"/>
    </xsd:extension>
</xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:simpleType name="dewey-versionType">

    <xsd:annotation>
        <xsd:documentation>

            This type defines a dewey decimal which is used
            to describe versions of documents.

        </xsd:documentation>
    </xsd:annotation>

    <xsd:restriction base="xsd:decimal">
        <xsd:whiteSpace value="collapse"/>
    </xsd:restriction>
</xsd:simpleType>

<!-- ***** -->
<xsd:complexType name="display-nameType">

    <xsd:annotation>
        <xsd:documentation>

            The display-name type contains a short name that is intended
            to be displayed by tools. It is used by display-name
            elements. The display name need not be unique.

            Example:

            ...
            <display-name xml:lang="en">Employee Self Service</display-
name>

            The value of the xml:lang attribute is "en" (English) by default.

        </xsd:documentation>
    </xsd:annotation>

```



```

    <xsd:simpleContent>
      <xsd:extension base="j2ee:string">
        <xsd:attribute ref="xml:lang"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>

<!-- ***** -->
<xsd:complexType name="ejb-linkType">

  <xsd:annotation>
    <xsd:documentation>

      The ejb-linkType is used by ejb-link
      elements in the ejb-ref or ejb-local-ref elements to specify
      that an EJB reference is linked to enterprise bean.

      The value of the ejb-link element must be the ejb-name of an
      enterprise bean in the same ejb-jar file or in another ejb-jar
      file in the same J2EE application unit.

      Alternatively, the name in the ejb-link element may be
      composed of a path name specifying the ejb-jar containing the
      referenced enterprise bean with the ejb-name of the target
      bean appended and separated from the path name by "#". The
      path name is relative to the Deployment File containing
      Deployment Component that is referencing the enterprise
      bean. This allows multiple enterprise beans with the same
      ejb-name to be uniquely identified.

      Examples:

      <ejb-link>EmployeeRecord</ejb-link>

      <ejb-link>../products/product.jar#ProductEJB</ejb-link>

    </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleContent>
    <xsd:restriction base="j2ee:string"/>
  </xsd:simpleContent>
</xsd:complexType>

```

```

<!-- ***** -->
<xsd:complexType name="ejb-local-refType">

  <xsd:annotation>
    <xsd:documentation>

      The ejb-local-refType is used by ejb-local-ref elements for
      the declaration of a reference to an enterprise bean's local
      home. The declaration consists of:

        - an optional description
        - the EJB reference name used in the code of the Deployment
           Component that's referencing the enterprise bean
        - the expected type of the referenced enterprise bean
        - the expected local home and local interfaces of the
           referenced enterprise bean
        - optional ejb-link information, used to specify the
           referenced enterprise bean

    </xsd:documentation>
  </xsd:annotation>

  <xsd:sequence>
    <xsd:element name="description"
      type="j2ee:descriptionType"
      minOccurs="0"
      maxOccurs="unbounded"/>
    <xsd:element name="ejb-ref-name"
      type="j2ee:ejb-ref-nameType"/>
    <xsd:element name="ejb-ref-type"
      type="j2ee:ejb-ref-typeType"/>
    <xsd:element name="local-home"
      type="j2ee:local-homeType"/>
    <xsd:element name="local"
      type="j2ee:localType"/>
    <xsd:element name="ejb-link"
      type="j2ee:ejb-linkType"
      minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

```

```

<!-- ***** -->
<xsd:complexType name="ejb-ref-nameType">

  <xsd:annotation>
    <xsd:documentation>

      The ejb-ref-name element contains the name of an EJB
      reference. The EJB reference is an entry in the
      Deployment Component's environment and is relative to the
      java:comp/env context. The name must be unique within the
      Deployment Component.

      It is recommended that name is prefixed with "ejb/".

      Example:

      <ejb-ref-name>ejb/Payroll</ejb-ref-name>

    </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleContent>
    <xsd:restriction base="j2ee:jndi-nameType"/>
  </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="ejb-ref-typeType">

  <xsd:annotation>
    <xsd:documentation>

      The ejb-ref-typeType contains the expected type of the
      referenced enterprise bean.

      The ejb-ref-type designates a value
      that must be one of the following:

      Entity
      Session

    </xsd:documentation>
  </xsd:annotation>

```

```

<xsd:simpleContent>
  <xsd:restriction base="j2ee:string">
    <xsd:enumeration value="Entity"/>
    <xsd:enumeration value="Session"/>
  </xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="ejb-refType">

  <xsd:annotation>
    <xsd:documentation>

      The ejb-refType is used by ejb-ref elements for the
      declaration of a reference to an enterprise bean's home. The
      declaration consists of:

        - an optional description
        - the EJB reference name used in the code of
        the Deployment Component that's referencing the enterprise
        bean
        - the expected type of the referenced enterprise bean
        - the expected home and remote interfaces of the referenced
        enterprise bean
        - optional ejb-link information, used to specify the
        referenced enterprise bean

    </xsd:documentation>
  </xsd:annotation>

  <xsd:sequence>
    <xsd:element name="description"
      type="j2ee:descriptionType"
      minOccurs="0"
      maxOccurs="unbounded"/>
    <xsd:element name="ejb-ref-name"
      type="j2ee:ejb-ref-nameType"/>
    <xsd:element name="ejb-ref-type"
      type="j2ee:ejb-ref-typeType"/>
    <xsd:element name="home"
      type="j2ee:homeType"/>
    <xsd:element name="remote"

```

```

        type="j2ee:remoteType"/>
    <xsd:element name="ejb-link"
        type="j2ee:ejb-linkType"
        minOccurs="0"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="emptyType">

    <xsd:annotation>
        <xsd:documentation>

            This type is used to designate an empty
            element when used.

        </xsd:documentation>
    </xsd:annotation>

    <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="env-entry-type-valuesType">

    <xsd:annotation>
        <xsd:documentation>

            This type contains the fully-qualified Java type of the
            environment entry value that is expected by the
            application's code.

            The following are the legal values of env-entry-type-valuesType:

            java.lang.Boolean
            java.lang.Byte
            java.lang.Character
            java.lang.String
            java.lang.Short
            java.lang.Integer
            java.lang.Long
            java.lang.Float

```

*java.lang.Double*

*Example:*

```

    <env-entry-type>java.lang.Boolean</env-entry-type>

</xsd:documentation>
</xsd:annotation>

<xsd:simpleContent>
  <xsd:restriction base="j2ee:string">
    <xsd:enumeration value="java.lang.Boolean"/>
    <xsd:enumeration value="java.lang.Byte"/>
    <xsd:enumeration value="java.lang.Character"/>
    <xsd:enumeration value="java.lang.String"/>
    <xsd:enumeration value="java.lang.Short"/>
    <xsd:enumeration value="java.lang.Integer"/>
    <xsd:enumeration value="java.lang.Long"/>
    <xsd:enumeration value="java.lang.Float"/>
    <xsd:enumeration value="java.lang.Double"/>
  </xsd:restriction>
</xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="env-entryType">

  <xsd:annotation>
    <xsd:documentation>

      The env-entryType is used to declare an application's
      environment entry. The declaration consists of an optional
      description, the name of the environment entry, and an
      optional value. If a value is not specified, one must be
      supplied during deployment.

      It is used by env-entry elements.

    </xsd:documentation>
  </xsd:annotation>

  <xsd:sequence>
    <xsd:element name="description"

```

```

        type="j2ee:descriptionType"
        minOccurs="0"
        maxOccurs="unbounded"/>
<xsd:element name="env-entry-name"
        type="j2ee:jndi-nameType">

```

```

<xsd:annotation>
  <xsd:documentation>

```

*The env-entry-name element contains the name of a Deployment Component's environment entry. The name is a JNDI name relative to the java:comp/env context. The name must be unique within a Deployment Component. The uniqueness constraints must be defined within the declared context.*

*Example:*

```

    <env-entry-name>minAmount</env-entry-name>

```

```

  </xsd:documentation>
</xsd:annotation>

```

```

</xsd:element>
<xsd:element name="env-entry-type"
        type="j2ee:env-entry-type-valuesType"/>
<xsd:element name="env-entry-value"
        type="j2ee:xsdStringType"
        minOccurs="0">

```

```

<xsd:annotation>
  <xsd:documentation>

```

*The env-entry-value designates the value of a Deployment Component's environment entry. The value must be a String that is valid for the constructor of the specified type that takes a single String parameter, or for java.lang.Character, a single character.*

*Example:*

```

        <env-entry-value>100.00</env-entry-value>

    </xsd:documentation>
</xsd:annotation>

</xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="fully-qualified-classType">

    <xsd:annotation>
        <xsd:documentation>

            The elements that use this type designate the name of a
            Java class or interface. The name is in the form of a
            "binary name", as defined in the JLS. This is the form
            of name used in Class.forName(). Tools that need the
            canonical name (the name used in source code) will need
            to convert this binary name to the canonical name.

        </xsd:documentation>
    </xsd:annotation>

    <xsd:simpleContent>
        <xsd:restriction base="j2ee:string"/>
    </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="generic-booleanType">

    <xsd:annotation>
        <xsd:documentation>

            This type defines four different values which can designate
            boolean values. This includes values yes and no which are
            not designated by xsd:boolean

        </xsd:documentation>
    </xsd:annotation>

```



```

    <xsd:simpleContent>
      <xsd:restriction base="j2ee:string">
        <xsd:enumeration value="true"/>
        <xsd:enumeration value="false"/>
        <xsd:enumeration value="yes"/>
        <xsd:enumeration value="no"/>
      </xsd:restriction>
    </xsd:simpleContent>
  </xsd:complexType>

<!-- ***** -->
  <xsd:complexType name="homeType">

    <xsd:annotation>
      <xsd:documentation>

        The homeType defines the fully-qualified name of
        an enterprise bean's home interface.

        Example:

          <home>com.aardvark.payroll.PayrollHome</home>

      </xsd:documentation>
    </xsd:annotation>

    <xsd:simpleContent>
      <xsd:restriction base="j2ee:fully-qualified-classType"/>
    </xsd:simpleContent>
  </xsd:complexType>

<!-- ***** -->
  <xsd:complexType name="iconType">

    <xsd:annotation>
      <xsd:documentation>

        The icon type contains small-icon and large-icon elements
        that specify the file names for small and large GIF or
        JPEG icon images used to represent the parent element in a
        GUI tool.

        The xml:lang attribute defines the language that the

```

*icon file names are provided in. Its value is "en" (English) by default.*

```
</xsd:documentation>
</xsd:annotation>
```

```
<xsd:sequence>
  <xsd:element name="small-icon" type="j2ee:pathType"
    minOccurs="0">
```

```
  <xsd:annotation>
    <xsd:documentation>
```

*The small-icon element contains the name of a file containing a small (16 x 16) icon image. The file name is a relative path within the Deployment Component's Deployment File.*

*The image may be either in the JPEG or GIF format. The icon can be used by tools.*

*Example:*

```
<small-icon>employee-service-icon16x16.jpg</small-icon>
```

```
</xsd:documentation>
</xsd:annotation>
```

```
</xsd:element>
<xsd:element name="large-icon" type="j2ee:pathType"
  minOccurs="0">
```

```
<xsd:annotation>
  <xsd:documentation>
```

*The large-icon element contains the name of a file containing a large (32 x 32) icon image. The file name is a relative path within the Deployment Component's Deployment File.*

*The image may be either in the JPEG or GIF format. The icon can be used by tools.*

*Example:*

```

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

</xsd:documentation>
</xsd:annotation>

</xsd:element>
</xsd:sequence>
<xsd:attribute ref="xml:lang"/>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="java-identifierType">

  <xsd:annotation>
    <xsd:documentation>

      The java-identifierType defines a Java identifier.
      The users of this type should further verify that
      the content does not contain Java reserved keywords.

    </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleContent>
    <xsd:restriction base="j2ee:string">
      <xsd:pattern value="($|_|p{L})(p{L}|p{Nd}|_|$)*"/>
    </xsd:restriction>
  </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="java-typeType">

  <xsd:annotation>
    <xsd:documentation>

      This is a generic type that designates a Java primitive
      type or a fully qualified name of a Java interface/type,
      or an array of such types.

```

```

        </xsd:documentation>
    </xsd:annotation>

    <xsd:simpleContent>
        <xsd:restriction base="j2ee:string">
            <xsd:pattern value="[\p{Z}]*"/>
        </xsd:restriction>
    </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="jndi-nameType">

    <xsd:annotation>
        <xsd:documentation>

            The jndi-nameType type designates a JNDI name in the
            Deployment Component's environment and is relative to the
            java:comp/env context. A JNDI name must be unique within the
            Deployment Component.

        </xsd:documentation>
    </xsd:annotation>

    <xsd:simpleContent>
        <xsd:restriction base="j2ee:string"/>
    </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:group name="jndiEnvironmentRefsGroup">

    <xsd:annotation>
        <xsd:documentation>

            This group keeps the usage of the contained JNDI environment
            reference elements consistent across J2EE deployment descriptors.

        </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>
        <xsd:element name="env-entry"

```

```

        type="j2ee:env-entryType"
        minOccurs="0" maxOccurs="unbounded"/>
<xsd:element name="ejb-ref"
        type="j2ee:ejb-refType"
        minOccurs="0" maxOccurs="unbounded"/>
<xsd:element name="ejb-local-ref"
        type="j2ee:ejb-local-refType"
        minOccurs="0" maxOccurs="unbounded"/>
<xsd:group ref="j2ee:service-refGroup"/>
<xsd:element name="resource-ref"
        type="j2ee:resource-refType"
        minOccurs="0" maxOccurs="unbounded"/>
<xsd:element name="resource-env-ref"
        type="j2ee:resource-env-refType"
        minOccurs="0" maxOccurs="unbounded"/>
<xsd:element name="message-destination-ref"
        type="j2ee:message-destination-refType"
        minOccurs="0" maxOccurs="unbounded"/>
</xsd:sequence>
</xsd:group>

<!-- ***** -->
<xsd:complexType name="listenerType">

    <xsd:annotation>
        <xsd:documentation>

            The listenerType indicates the deployment properties for a web
            application listener bean.

        </xsd:documentation>
    </xsd:annotation>

    <xsd:sequence>
        <xsd:group ref="j2ee:descriptionGroup"/>
        <xsd:element name="listener-class"
            type="j2ee:fully-qualified-classType">

            <xsd:annotation>
                <xsd:documentation>

                    The listener-class element declares a class in the
                    application must be registered as a web

```

*application listener bean. The value is the fully qualified classname of the listener class.*

```

        </xsd:documentation>
    </xsd:annotation>

    </xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="local-homeType">

    <xsd:annotation>
        <xsd:documentation>

            The local-homeType defines the fully-qualified
            name of an enterprise bean's local home interface.

        </xsd:documentation>
    </xsd:annotation>

    <xsd:simpleContent>
        <xsd:restriction base="j2ee:fully-qualified-classType"/>
    </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="localType">

    <xsd:annotation>
        <xsd:documentation>

            The localType defines the fully-qualified name of an
            enterprise bean's local interface.

        </xsd:documentation>
    </xsd:annotation>

    <xsd:simpleContent>
        <xsd:restriction base="j2ee:fully-qualified-classType"/>

```

```

    </xsd:simpleContent>
  </xsd:complexType>

<!-- ***** -->
<xsd:complexType name="message-destination-linkType">

  <xsd:annotation>
    <xsd:documentation>

      The message-destination-linkType is used to link a message
      destination reference or message-driven bean to a message
      destination.

      The Assembler sets the value to reflect the flow of messages
      between producers and consumers in the application.

      The value must be the message-destination-name of a message
      destination in the same Deployment File or in another
      Deployment File in the same J2EE application unit.

      Alternatively, the value may be composed of a path name
      specifying a Deployment File containing the referenced
      message destination with the message-destination-name of the
      destination appended and separated from the path name by
      "#". The path name is relative to the Deployment File
      containing Deployment Component that is referencing the
      message destination. This allows multiple message
      destinations with the same name to be uniquely identified.

    </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleContent>
    <xsd:restriction base="j2ee:string"/>
  </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="message-destination-refType">

  <xsd:annotation>
    <xsd:documentation>

```

*The message-destination-ref element contains a declaration of Deployment Component's reference to a message destination associated with a resource in Deployment Component's environment. It consists of:*

- *an optional description*
- *the message destination reference name*
- *the message destination type*
- *a specification as to whether the destination is used for consuming or producing messages, or both*
- *a link to the message destination*

*Examples:*

```
<message-destination-ref>
  <message-destination-ref-name>jms/StockQueue
</message-destination-ref-name>
  <message-destination-type>javax.jms.Queue
</message-destination-type>
  <message-destination-usage>Consumes
</message-destination-usage>
  <message-destination-link>CorporateStocks
</message-destination-link>
</message-destination-ref>
```

```
</xsd:documentation>
</xsd:annotation>
```

```
<xsd:sequence>
  <xsd:element name="description"
    type="j2ee:descriptionType"
    minOccurs="0"
    maxOccurs="unbounded"/>
  <xsd:element name="message-destination-ref-name"
    type="j2ee:jndi-nameType">
```

```
<xsd:annotation>
  <xsd:documentation>
```

*The message-destination-ref-name element specifies the name of a message destination reference; its value is the environment entry name used in*



*Deployment Component code. The name is a JNDI name relative to the java:comp/env context and must be unique within an ejb-jar (for enterprise beans) or a Deployment File (for others).*

```

        </xsd:documentation>
    </xsd:annotation>

</xsd:element>
<xsd:element name="message-destination-type"
              type="j2ee:message-destination-typeType"/>
<xsd:element name="message-destination-usage"
              type="j2ee:message-destination-usageType"/>
<xsd:element name="message-destination-link"
              type="j2ee:message-destination-linkType"
              minOccurs="0"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="message-destination-typeType">

    <xsd:annotation>
        <xsd:documentation>

            The message-destination-typeType specifies the type of
            the destination. The type is specified by the Java interface
            expected to be implemented by the destination.

            Example:

                <message-destination-type>javax.jms.Queue
                </message-destination-type>

        </xsd:documentation>
    </xsd:annotation>

    <xsd:simpleContent>
        <xsd:restriction base="j2ee:fully-qualified-classType"/>
    </xsd:simpleContent>
</xsd:complexType>

```

```

<!-- ***** -->
<xsd:complexType name="message-destination-usageType">

  <xsd:annotation>
    <xsd:documentation>

      The message-destination-usageType specifies the use of the
      message destination indicated by the reference. The value
      indicates whether messages are consumed from the message
      destination, produced for the destination, or both. The
      Assembler makes use of this information in linking producers
      of a destination with its consumers.

      The value of the message-destination-usage element must be
      one of the following:
        Consumes
        Produces
        ConsumesProduces

    </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleContent>
    <xsd:restriction base="j2ee:string">
      <xsd:enumeration value="Consumes"/>
      <xsd:enumeration value="Produces"/>
      <xsd:enumeration value="ConsumesProduces"/>
    </xsd:restriction>
  </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="message-destinationType">

  <xsd:annotation>
    <xsd:documentation>

      The message-destinationType specifies a message
      destination. The logical destination described by this
      element is mapped to a physical destination by the Deployer.

      The message destination element contains:

```

- *an optional description*
- *an optional display-name*
- *an optional icon*
- *a message destination name which must be unique among message destination names within the same Deployment File.*

*Example:*

```
<message-destination>
  <message-destination-name>CorporateStocks
</message-destination-name>
</message-destination>
```

```
</xsd:documentation>
</xsd:annotation>
```

```
<xsd:sequence>
  <xsd:group ref="j2ee:descriptionGroup"/>
  <xsd:element name="message-destination-name"
    type="j2ee:string">
```

```
<xsd:annotation>
  <xsd:documentation>
```

*The message-destination-name element specifies a name for a message destination. This name must be unique among the names of message destinations within the Deployment File.*

```
</xsd:documentation>
</xsd:annotation>
```

```
</xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>
```

```
<!-- ***** -->
```

```
<xsd:complexType name="param-valueType">
```

```
<xsd:annotation>
  <xsd:documentation>
```

*This type is a general type that can be used to declare parameter/value lists.*

```

    </xsd:documentation>
  </xsd:annotation>

  <xsd:sequence>
    <xsd:element name="description"
      type="j2ee:descriptionType"
      minOccurs="0"
      maxOccurs="unbounded"/>
    <xsd:element name="param-name"
      type="j2ee:string">

      <xsd:annotation>
        <xsd:documentation>

          The param-name element contains the name of a
            parameter.

        </xsd:documentation>
      </xsd:annotation>

    </xsd:element>
    <xsd:element name="param-value"
      type="j2ee:xsdStringType">

      <xsd:annotation>
        <xsd:documentation>

          The param-value element contains the value of a
            parameter.

        </xsd:documentation>
      </xsd:annotation>

    </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="pathType">

```

```
<xsd:annotation>
  <xsd:documentation>
```

*The elements that use this type designate either a relative path or an absolute path starting with a "/".*

*In elements that specify a pathname to a file within the same Deployment File, relative filenames (i.e., those not starting with "/") are considered relative to the root of the Deployment File's namespace. Absolute filenames (i.e., those starting with "/") also specify names in the root of the Deployment File's namespace. In general, relative names are preferred. The exception is .war files where absolute names are preferred for consistency with the Servlet API.*

```
  </xsd:documentation>
</xsd:annotation>
```

```
<xsd:simpleContent>
  <xsd:restriction base="j2ee:string"/>
</xsd:simpleContent>
</xsd:complexType>
```

```
<!-- ***** -->
```

```
<xsd:complexType name="remoteType">
```

```
  <xsd:annotation>
    <xsd:documentation>
```

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

*Example:*

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

```
  </xsd:documentation>
</xsd:annotation>
```

```
<xsd:simpleContent>
  <xsd:restriction base="j2ee:fully-qualified-classType"/>
</xsd:simpleContent>
</xsd:complexType>
```

```

<!-- ***** -->
<xsd:complexType name="res-authType">

  <xsd:annotation>
    <xsd:documentation>

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

      The value must be one of the two following:

          Application
          Container

    </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleContent>
    <xsd:restriction base="j2ee:string">
      <xsd:enumeration value="Application"/>
      <xsd:enumeration value="Container"/>
    </xsd:restriction>
  </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="res-sharing-scopeType">

  <xsd:annotation>
    <xsd:documentation>

      The res-sharing-scope type specifies whether connections
      obtained through the given resource manager connection
      factory reference can be shared. The value, if specified,
      must be one of the two following:

          Shareable
          Unshareable

```

*The default value is Shareable.*

```

    </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleContent>
    <xsd:restriction base="j2ee:string">
      <xsd:enumeration value="Shareable"/>
      <xsd:enumeration value="Unshareable"/>
    </xsd:restriction>
  </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="resource-env-refType">

  <xsd:annotation>
    <xsd:documentation>

```

*The resource-env-refType is used to define resource-env-type elements. It contains a declaration of a Deployment Component's reference to an administered object associated with a resource in the Deployment Component's environment. It consists of an optional description, the resource environment reference name, and an indication of the resource environment reference type expected by the Deployment Component code.*

*Example:*

```

    <resource-env-ref>
      <resource-env-ref-name>jms/StockQueue
    </resource-env-ref-name>
      <resource-env-ref-type>javax.jms.Queue
    </resource-env-ref-type>
    </resource-env-ref>

  </xsd:documentation>
</xsd:annotation>

<xsd:sequence>
  <xsd:element name="description"
    type="j2ee:descriptionType"

```

```

        minOccurs="0"
        maxOccurs="unbounded"/>
<xsd:element name="resource-env-ref-name"
            type="j2ee:jndi-nameType">

    <xsd:annotation>
        <xsd:documentation>

            The resource-env-ref-name element specifies the name
            of a resource environment reference; its value is
            the environment entry name used in
            the Deployment Component code. The name is a JNDI
            name relative to the java:comp/env context and must
            be unique within a Deployment Component.

        </xsd:documentation>
    </xsd:annotation>

</xsd:element>
<xsd:element name="resource-env-ref-type"
            type="j2ee:fully-qualified-classType">

    <xsd:annotation>
        <xsd:documentation>

            The resource-env-ref-type element specifies the type
            of a resource environment reference. It is the
            fully qualified name of a Java language class or
            interface.

        </xsd:documentation>
    </xsd:annotation>

    </xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="resource-refType">

    <xsd:annotation>
        <xsd:documentation>

```



*The resource-refType contains a declaration of a Deployment Component's reference to an external resource. It consists of an optional description, the resource manager connection factory reference name, the indication of the resource manager connection factory type expected by the Deployment Component code, the type of authentication (Application or Container), and an optional specification of the shareability of connections obtained from the resource (Shareable or Unshareable).*

*Example:*

```

<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>
</resource-ref>

</xsd:documentation>
</xsd:annotation>

<xsd:sequence>
  <xsd:element name="description"
    type="j2ee:descriptionType"
    minOccurs="0"
    maxOccurs="unbounded"/>
  <xsd:element name="res-ref-name"
    type="j2ee:jndi-nameType">

    <xsd:annotation>
      <xsd:documentation>

        The res-ref-name element specifies the name of a
        resource manager connection factory reference.
        The name is a JNDI name relative to the
        java:comp/env context.
        The name must be unique within a Deployment File.

      </xsd:documentation>
    </xsd:annotation>

  </xsd:element>

```

```

<xsd:element name="res-type"
              type="j2ee:fully-qualified-classType">

  <xsd:annotation>
    <xsd:documentation>

      The res-type element specifies the type of the data
      source. The type is specified by the fully qualified
      Java language class or interface
      expected to be implemented by the data source.

    </xsd:documentation>
  </xsd:annotation>

</xsd:element>
<xsd:element name="res-auth"
              type="j2ee:res-authType"/>
<xsd:element name="res-sharing-scope"
              type="j2ee:res-sharing-scopeType"
              minOccurs="0"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="role-nameType">

  <xsd:annotation>
    <xsd:documentation>

      The role-nameType designates the name of a security role.

      The name must conform to the lexical rules for a token.

    </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleContent>
    <xsd:restriction base="j2ee:string"/>
  </xsd:simpleContent>
</xsd:complexType>

```

```

<!-- ***** -->
<xsd:complexType name="run-asType">

  <xsd:annotation>
    <xsd:documentation>

      The run-asType specifies the run-as identity to be
      used for the execution of a component. It contains an
      optional description, and the name of a security role.

    </xsd:documentation>
  </xsd:annotation>

  <xsd:sequence>
    <xsd:element name="description"
      type="j2ee:descriptionType"
      minOccurs="0"
      maxOccurs="unbounded"/>
    <xsd:element name="role-name"
      type="j2ee:role-nameType"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="security-role-refType">

  <xsd:annotation>
    <xsd:documentation>

      The security-role-refType contains the declaration of a
      security role reference in a component's or a
      Deployment Component's code. The declaration consists of an
      optional description, the security role name used in the
      code, and an optional link to a security role. If the
      security role is not specified, the Deployer must choose an
      appropriate security role.

    </xsd:documentation>
  </xsd:annotation>

  <xsd:sequence>
    <xsd:element name="description"

```

```

        type="j2ee:descriptionType"
        minOccurs="0"
        maxOccurs="unbounded"/>
<xsd:element name="role-name"
        type="j2ee:role-nameType">

    <xsd:annotation>
        <xsd:documentation>

            The value of the role-name element must be the String used
            as the parameter to the
            EJBContext.isCallerInRole(String roleName) method or the
            HttpServletRequest.isUserInRole(String role) method.

        </xsd:documentation>
    </xsd:annotation>

</xsd:element>
<xsd:element name="role-link"
        type="j2ee:role-nameType"
        minOccurs="0">

    <xsd:annotation>
        <xsd:documentation>

            The role-link element is a reference to a defined
            security role. The role-link element must contain
            the name of one of the security roles defined in the
            security-role elements.

        </xsd:documentation>
    </xsd:annotation>

</xsd:element>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="security-roleType">

    <xsd:annotation>
        <xsd:documentation>

```

*The security-roleType contains the definition of a security role. The definition consists of an optional description of the security role, and the security role name.*

*Example:*

```

        <security-role>
        <description>
            This role includes all employees who are authorized
            to access the employee service application.
        </description>
        <role-name>employee</role-name>
        </security-role>

    </xsd:documentation>
</xsd:annotation>

<xsd:sequence>
    <xsd:element name="description"
        type="j2ee:descriptionType"
        minOccurs="0"
        maxOccurs="unbounded"/>
    <xsd:element name="role-name"
        type="j2ee:role-nameType"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="string">

    <xsd:annotation>
        <xsd:documentation>

            This is a special string datatype that is defined by J2EE as
            a base type for defining collapsed strings. When schemas
            require trailing/leading space elimination as well as
            collapsing the existing whitespace, this base type may be
            used.

        </xsd:documentation>
    </xsd:annotation>

```

```

    <xsd:simpleContent>
      <xsd:extension base="xsd:token">
        <xsd:attribute name="id" type="xsd:ID"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>

<!-- ***** -->
  <xsd:complexType name="true-falseType">

    <xsd:annotation>
      <xsd:documentation>

        This simple type designates a boolean with only two
        permissible values

        - true
        - false

      </xsd:documentation>
    </xsd:annotation>

    <xsd:simpleContent>
      <xsd:restriction base="j2ee:xsdBooleanType">
        <xsd:pattern value="(true|false)"/>
      </xsd:restriction>
    </xsd:simpleContent>
  </xsd:complexType>

<!-- ***** -->
  <xsd:complexType name="url-patternType">

    <xsd:annotation>
      <xsd:documentation>

        The url-patternType contains the url pattern of the mapping.
        It must follow the rules specified in Section 11.2 of the
        Servlet API Specification. This pattern is assumed to be in
        URL-decoded form and must not contain CR(#xD) or LF(#xA).
        If it contains those characters, the container must inform
        the developer with a descriptive error message.
        The container must preserve all characters including whitespaces.


```

```

        </xsd:documentation>
    </xsd:annotation>

    <xsd:simpleContent>
        <xsd:extension base="xsd:string"/>
    </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="xsdAnyURIType">

    <xsd:annotation>
        <xsd:documentation>

            This type adds an "id" attribute to xsd:anyURI.

        </xsd:documentation>
    </xsd:annotation>

    <xsd:simpleContent>
        <xsd:extension base="xsd:anyURI">
            <xsd:attribute name="id" type="xsd:ID"/>
        </xsd:extension>
    </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="xsdBooleanType">

    <xsd:annotation>
        <xsd:documentation>

            This type adds an "id" attribute to xsd:boolean.

        </xsd:documentation>
    </xsd:annotation>

    <xsd:simpleContent>
        <xsd:extension base="xsd:boolean">
            <xsd:attribute name="id" type="xsd:ID"/>
        </xsd:extension>
    </xsd:simpleContent>
</xsd:complexType>

```

```

<!-- ***** -->
<xsd:complexType name="xsdIntegerType">

  <xsd:annotation>
    <xsd:documentation>

      This type adds an "id" attribute to xsd:integer.

    </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleContent>
    <xsd:extension base="xsd:integer">
      <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="xsdNMTOKENType">

  <xsd:annotation>
    <xsd:documentation>

      This type adds an "id" attribute to xsd:NMTOKEN.

    </xsd:documentation>
  </xsd:annotation>

  <xsd:simpleContent>
    <xsd:extension base="xsd:NMTOKEN">
      <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="xsdNonNegativeIntegerType">

  <xsd:annotation>
    <xsd:documentation>

      This type adds an "id" attribute to xsd:nonNegativeInteger.

```



```

        </xsd:documentation>
    </xsd:annotation>

    <xsd:simpleContent>
        <xsd:extension base="xsd:nonNegativeInteger">
            <xsd:attribute name="id" type="xsd:ID"/>
        </xsd:extension>
    </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="xsdPositiveIntegerType">

    <xsd:annotation>
        <xsd:documentation>

            This type adds an "id" attribute to xsd:positiveInteger.

        </xsd:documentation>
    </xsd:annotation>

    <xsd:simpleContent>
        <xsd:extension base="xsd:positiveInteger">
            <xsd:attribute name="id" type="xsd:ID"/>
        </xsd:extension>
    </xsd:simpleContent>
</xsd:complexType>

<!-- ***** -->
<xsd:complexType name="xsdQNameType">

    <xsd:annotation>
        <xsd:documentation>

            This type adds an "id" attribute to xsd:QName.

        </xsd:documentation>
    </xsd:annotation>

    <xsd:simpleContent>
        <xsd:extension base="xsd:QName">
            <xsd:attribute name="id" type="xsd:ID"/>
        </xsd:extension>

```

```

        </xsd:simpleContent>
    </xsd:complexType>

<!-- ***** -->
    <xsd:complexType name="xsdStringType">

        <xsd:annotation>
            <xsd:documentation>

                This type adds an "id" attribute to xsd:string.

            </xsd:documentation>
        </xsd:annotation>

        <xsd:simpleContent>
            <xsd:extension base="xsd:string">
                <xsd:attribute name="id" type="xsd:ID"/>
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>

</xsd:schema>

```

### J2EE.A.3 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 application deployment descriptor may contain the following DOCTYPE declaration:

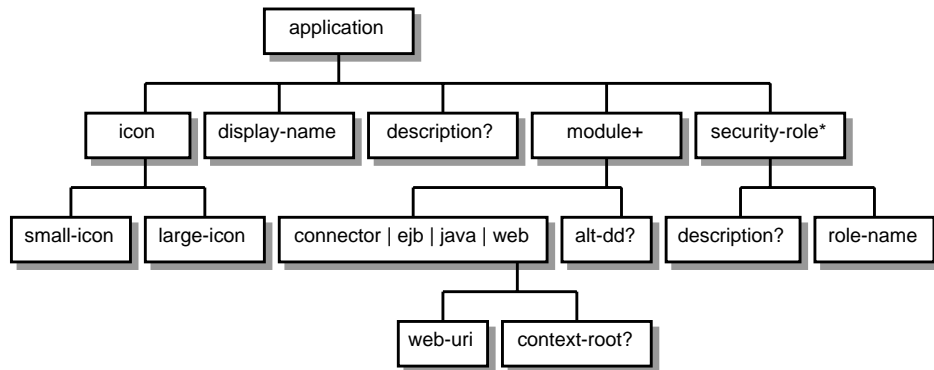
```

<!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.

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



**Figure J2EE.A-1** J2EE:application XML DTD Structure

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

```

<!--
This is the XML DTD for the J2EE 1.3 application deployment
descriptor. All J2EE 1.3 application deployment descriptors must
include a DOCTYPE of the following form:
  <!DOCTYPE application PUBLIC
    "-//Sun Microsystems, Inc.//DTD J2EE Application 1.3//EN"
    "http://java.sun.com/dtd/application_1_3.dtd">
-->
  
```

```

<!--
The following conventions apply to all J2EE deployment descriptor
elements unless indicated otherwise.
- In elements that contain PCDATA, leading and trailing whitespace
  in the data may be ignored.
- In elements whose value is an "enumerated type", the value is
  case sensitive.
- In elements that specify a pathname to a file within the same
  JAR file, relative filenames (i.e., those not starting with "/" )
  are considered relative to the root of the JAR file's namespace.
  Absolute filenames (i.e., those starting with "/" ) also specify
  
```

names in the root of the JAR file's namespace. In general, relative names are preferred. The exception is .war files where absolute names are preferred for consistency with the servlet API.

-->

<!--

The application element is the root element of a J2EE application deployment descriptor.

-->

**<!ELEMENT application (icon?, display-name, description?, module+, security-role\*)>**

<!--

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

Used in: module

-->

**<!ELEMENT alt-dd (#PCDATA)>**

<!--

The connector element specifies the URI of a resource adapter archive file, relative to the top level of the application package.

Used in: module

-->

**<!ELEMENT connector (#PCDATA)>**

<!--

The context-root element specifies the context root of a web application.

Used in: web

-->

**<!ELEMENT context-root (#PCDATA)>**

<!--

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

Used in: application, security-role

-->

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

<!--

The display-name element contains a short name that is intended to be displayed by tools. The display name need not be unique.

Used in: application

Example:

<display-name>Employee Self Service</display-name>

-->

**<!ELEMENT display-name (#PCDATA)>**

<!--

The ejb element specifies the URI of an ejb-jar, relative to the top level of the application package.

Used in: module

-->

**<!ELEMENT ejb (#PCDATA)>**

<!--

The icon element contains small-icon and large-icon elements that specify the file names for small and a large GIF or JPEG icon images used to represent the parent element in a GUI tool.

Used in: application

-->

**<!ELEMENT icon (small-icon?, large-icon?)>**

<!--

The java element specifies the URI of a java application client module, relative to the top level of the application package.

Used in: module

-->

**<!ELEMENT java (#PCDATA)>**

<!--

The large-icon element contains the name of a file containing a large (32 x 32) icon image. The file name is a relative path within the application's ear file.

The image may be either in the JPEG or GIF format. The icon can be used by tools.

Used in: icon

Example:

`<large-icon>employee-service-icon32x32.jpg</large-icon>`

-->

**<!ELEMENT large-icon (#PCDATA)>**

<!--

The module element represents a single J2EE module and contains a connector, ejb, java, or web element, which indicates the module type and contains a path to the module file, and an optional alt-dd element, which specifies an optional URI to the post-assembly version of the deployment descriptor.

The application deployment descriptor must have one module element for each J2EE module in the application package.

Used in: application

-->

**<!ELEMENT module ((connector | ejb | java | web), alt-dd?)>**

<!--

The role-name element contains the name of a security role.

The name must conform to the lexical rules for an NMTOKEN.

Used in: security-role

-->

**<!ELEMENT role-name (#PCDATA)>**

<!--

The security-role element contains the definition of a security role. The definition consists of an optional description of the security role, and the security role name.

Used in: application

Example:

```
<security-role>
  <description>
    This role includes all employees who are authorized
    to access the employee service application.
  </description>
  <role-name>employee</role-name>
</security-role>
-->
```

**<!ELEMENT security-role (description?, role-name)>**

<!--

The small-icon element contains the name of a file containing a small (16 x 16) icon image. The file name is a relative path within the application's ear file.

The image may be either in the JPEG or GIF format. The icon can be used by tools.

Used in: icon

Example:

```
<small-icon>employee-service-icon16x16.jpg</small-icon>
-->
```

**<!ELEMENT small-icon (#PCDATA)>**

<!--

The web element contains the web-uri and context-root of a web application module.

Used in: module

-->

**<!ELEMENT web (web-uri, context-root)>**

<!--

The web-uri element specifies the URI of a web application file, relative to the top level of the application package.

Used in: web

-->

**<!ELEMENT web-uri (#PCDATA)>**

```

<!--
The ID mechanism is to allow tools that produce additional deployment
information (i.e., information beyond the standard deployment
descriptor information) to store the non-standard information in a
separate file, and easily refer from these tool-specific files to
the information in the standard deployment descriptor.
Tools are not allowed to add the non-standard information into the
standard deployment descriptor.
-->

<!--ATTLIST alt-dd id ID #IMPLIED>
<!--ATTLIST application id ID #IMPLIED>
<!--ATTLIST connector id ID #IMPLIED>
<!--ATTLIST context-root id ID #IMPLIED>
<!--ATTLIST description id ID #IMPLIED>
<!--ATTLIST display-name id ID #IMPLIED>
<!--ATTLIST ejb id ID #IMPLIED>
<!--ATTLIST icon id ID #IMPLIED>
<!--ATTLIST java id ID #IMPLIED>
<!--ATTLIST large-icon id ID #IMPLIED>
<!--ATTLIST module id ID #IMPLIED>
<!--ATTLIST role-name id ID #IMPLIED>
<!--ATTLIST security-role id ID #IMPLIED>
<!--ATTLIST small-icon id ID #IMPLIED>
<!--ATTLIST web id ID #IMPLIED>
<!--ATTLIST web-uri id ID #IMPLIED>

```

## **J2EE.A.4 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 application deployment descriptor may contain the following DOCTYPE declaration:

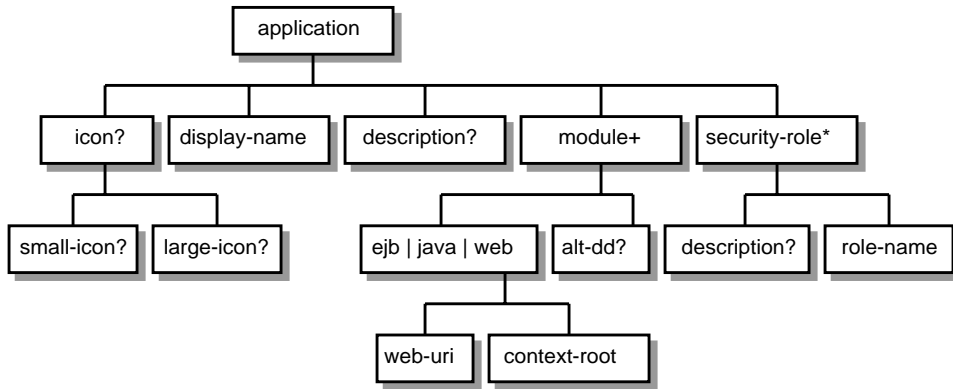
```

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

```



**Figure J2EE.A-2** shows a graphic representation of the structure of the J2EE:application XML DTD.



**Figure J2EE.A-2** J2EE:application XML DTD Structure

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

```
<!--
```

The alt-dd element specifies an optional URI to the post-assembly version of the deployment descriptor file for a particular J2EE module.

The URI must specify the full pathname of the deployment descriptor file relative to the application's root directory. If alt-dd is not specified, the deployer must read the deployment descriptor from the default location and file name required by the respective component specification.

```
-->
```

```
<!ELEMENT alt-dd (#PCDATA)>
```

```
<!--
```

The application element is the root element of a J2EE application deployment descriptor.

```
-->
```

```
<!ELEMENT application (icon?, display-name, description?, module+,
    security-role*)>
```

```
<!--
```

```
The context-root element specifies the context root of a web
application
```

```
-->
```

```
<!ELEMENT context-root (#PCDATA)>
```

```
<!--
```

```
The description element provides a human readable description of the
application.
```

```
The description element should include any information that the
application assembler wants to provide the deployer.
```

```
-->
```

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

```
<!--
```

```
The display-name element specifies an application name.
```

```
The application name is assigned to the application by the
application assembler and is used to identify the application to the
deployer at deployment time.
```

```
-->
```

```
<!ELEMENT display-name (#PCDATA)>
```

```
<!--
```

```
The ejb element specifies the URI of a ejb-jar, relative to the top
level of the application package.
```

```
-->
```

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

```
<!--
```

```
The icon element contains a small-icon and large-icon element which
specify the URIs for a small and a large GIF or JPEG icon image to
represent the application in a GUI.
```

```
-->
```

```
<!ELEMENT icon (small-icon?, large-icon?)>
```

<!--

The java element specifies the URI of a java application client module, relative to the top level of the application package.

-->

**<!ELEMENT java (#PCDATA)>**

<!--

The large-icon element specifies the URI for a large GIF or JPEG icon image to represent the application in a GUI.

-->

**<!ELEMENT large-icon (#PCDATA)>**

<!--

The module element represents a single J2EE module and contains an ejb, java, or web element, which indicates the module type and contains a path to the module file, and an optional alt-dd element, which specifies an optional URI to the post-assembly version of the deployment descriptor.

The application deployment descriptor must have one module element for each J2EE module in the application package.

-->

**<!ELEMENT module ((ejb | java | web), alt-dd?)>**

<!--

The role-name element contains the name of a security role.

-->

**<!ELEMENT role-name (#PCDATA)>**

<!--

The security-role element contains the definition of a security role which is global to the application.

The definition consists of a description of the security role, and the security role name.

The descriptions at this level override those in the component level security-role definitions and must be the descriptions tool display to the deployer.

-->

**<!ELEMENT security-role (description?, role-name)>**

**<!--**

The small-icon element specifies the URI for a small GIF or JPEG icon image to represent the application in a GUI.

**-->**

**<!ELEMENT small-icon (#PCDATA)>**

**<!--**

The web element contains the web-uri and context-root of a web application module.

**-->**

**<!ELEMENT web (web-uri, context-root)>**

**<!--**

The web-uri element specifies the URI of a web application file, relative to the top level of the application package.

**-->**

**<!ELEMENT web-uri (#PCDATA)>**

**<!--**

The ID mechanism is to allow tools to easily make tool-specific references to the elements of the deployment descriptor.

**-->**

**<!ATTLIST alt-dd id ID #IMPLIED>**

**<!ATTLIST application id ID #IMPLIED>**

**<!ATTLIST context-root id ID #IMPLIED>**

**<!ATTLIST description id ID #IMPLIED>**

**<!ATTLIST display-name id ID #IMPLIED>**

**<!ATTLIST ejb id ID #IMPLIED>**

**<!ATTLIST icon id ID #IMPLIED>**

**<!ATTLIST java id ID #IMPLIED>**

**<!ATTLIST large-icon id ID #IMPLIED>**

**<!ATTLIST module id ID #IMPLIED>**

**<!ATTLIST role-name id ID #IMPLIED>**

**<!ATTLIST security-role id ID #IMPLIED>**

**<!ATTLIST small-icon id ID #IMPLIED>**

**<!ATTLIST web id ID #IMPLIED>**

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

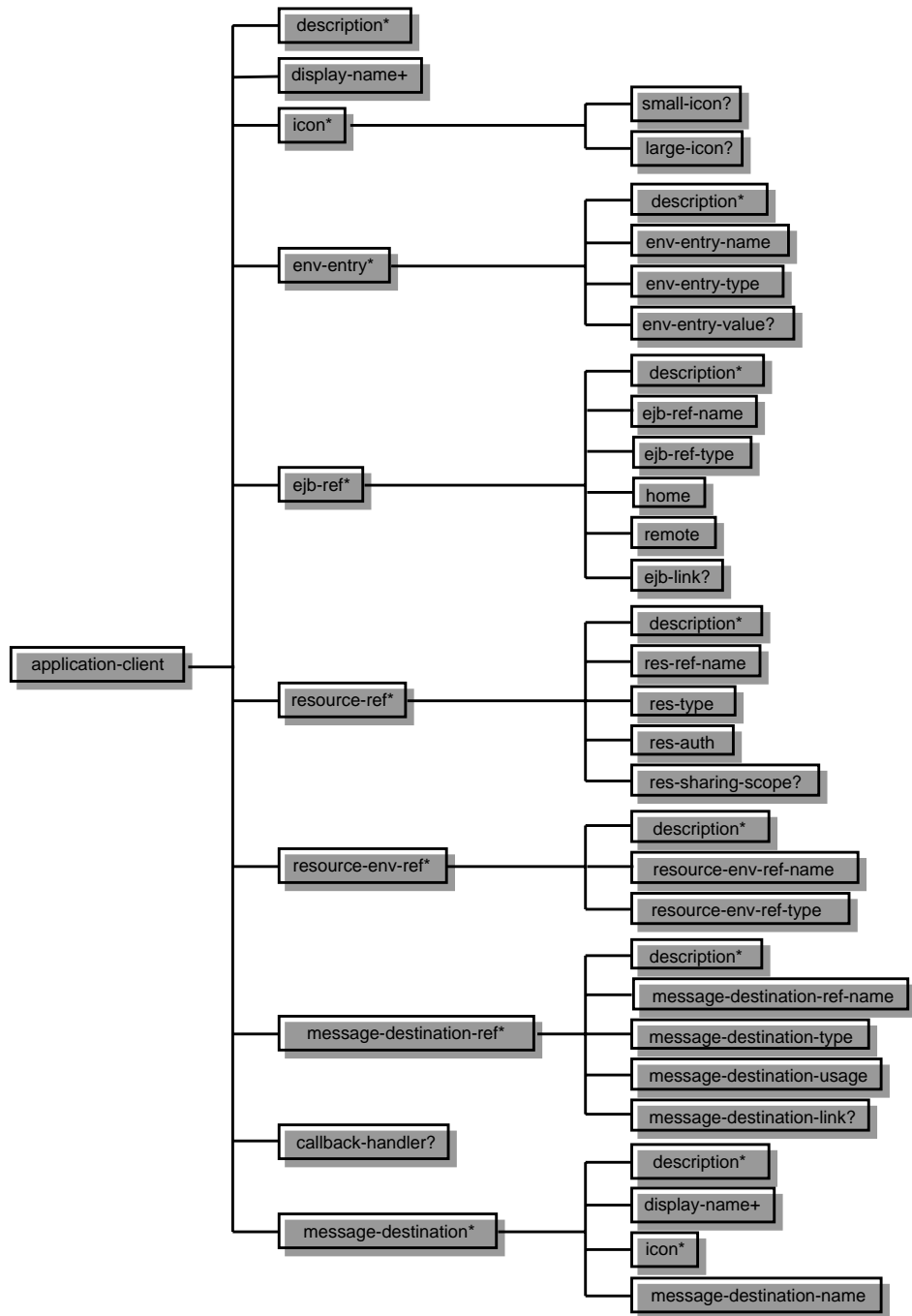
## **J2EE.A.5     J2EE 1.4 Application Client XML Schema**

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.

**Figure J2EE.0-2** shows the structure of the J2EE application-client XML

Schema.



**Figure J2EE.0-2** J2EE Application Client XML Schema Structure

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://java.sun.com/xml/ns/j2ee"
    xmlns:j2ee="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified"
    version="1.4">

    <xsd:annotation>
        <xsd:documentation>

            @(#)application-client_1_4.xsds    1.17 02/11/03

        </xsd:documentation>
    </xsd:annotation>

    <xsd:annotation>
        <xsd:documentation>

            This is the XML Schema for the application client 1.4
            deployment descriptor. The deployment descriptor must
            be named "META-INF/application-client.xml" in the
            application client's jar file. All application client
            deployment descriptors must indicate the application
            client schema by using the J2EE namespace:

            http://java.sun.com/xml/ns/j2ee

            and indicate the version of the schema by
            using the version element as shown below:

            <application-client xmlns="http://java.sun.com/xml/ns/j2ee"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                    http://java.sun.com/xml/ns/j2ee/application-
client_1_4.xsd"
                version="1.4">
                ...
            </application-client>

```



*The instance documents may indicate the published version of the schema using the `xsi:schemaLocation` attribute for J2EE namespace with the following location:*

*`http://java.sun.com/xml/ns/j2ee/application-client_1_4.xsd`*

```
</xsd:documentation>
</xsd:annotation>
```

```
<xsd:annotation>
  <xsd:documentation>
```

*The following conventions apply to all J2EE deployment descriptor elements unless indicated otherwise.*

- *In elements that specify a pathname to a file within the same JAR file, relative filenames (i.e., those not starting with "/" ) are considered relative to the root of the JAR file's namespace. Absolute filenames (i.e., those starting with "/" ) also specify names in the root of the JAR file's namespace. In general, relative names are preferred. The exception is .war files where absolute names are preferred for consistency with the Servlet API.*

```
</xsd:documentation>
</xsd:annotation>
```

```
<xsd:include schemaLocation="j2ee_1_4.xsd"/>
```

```
<!-- ***** -->
```

```
<xsd:element name="application-client" type="j2ee:application-
clientType">
```

```
<xsd:annotation>
  <xsd:documentation>
```

*The `application-client` element is the root element of an application client deployment descriptor. The application client deployment descriptor describes the EJB components and external resources referenced by the application client.*

```

    </xsd:documentation>
  </xsd:annotation>

  <xsd:unique name="env-entry-name-uniqueness">

    <xsd:annotation>
      <xsd:documentation>

        The env-entry-name element contains the name of an
        application client's environment entry. The name is a JNDI
        name relative to the java:comp/env context. The name must
        be unique within an application client.

      </xsd:documentation>
    </xsd:annotation>

    <xsd:selector xpath="j2ee:env-entry"/>
    <xsd:field    xpath="j2ee:env-entry-name"/>
  </xsd:unique>
  <xsd:unique name="ejb-ref-name-uniqueness">

    <xsd:annotation>
      <xsd:documentation>

        The ejb-ref-name element contains the name of an EJB
        reference. The EJB reference is an entry in the application
        client's environment and is relative to the
        java:comp/env context. The name must be unique within the
        application client.

        It is recommended that name is prefixed with "ejb/".

      </xsd:documentation>
    </xsd:annotation>

    <xsd:selector xpath="j2ee:ejb-ref"/>
    <xsd:field    xpath="j2ee:ejb-ref-name"/>
  </xsd:unique>
  <xsd:unique name="res-ref-name-uniqueness">

    <xsd:annotation>
      <xsd:documentation>

```

*The res-ref-name element specifies the name of a resource manager connection factory reference. The name is a JNDI name relative to the java:comp/env context. The name must be unique within an application client.*

```

    </xsd:documentation>
  </xsd:annotation>

  <xsd:selector xpath="j2ee:resource-ref"/>
  <xsd:field    xpath="j2ee:res-ref-name"/>
</xsd:unique>
<xsd:unique name="resource-env-ref-uniqueness">

```

```

<xsd:annotation>
  <xsd:documentation>

```

*The resource-env-ref-name element specifies the name of a resource environment reference; its value is the environment entry name used in the application client code. The name is a JNDI name relative to the java:comp/env context and must be unique within an application client.*

```

    </xsd:documentation>
  </xsd:annotation>

  <xsd:selector xpath="j2ee:resource-env-ref"/>
  <xsd:field    xpath="j2ee:resource-env-ref-name"/>
</xsd:unique>
<xsd:unique name="message-destination-ref-uniqueness">

```

```

<xsd:annotation>
  <xsd:documentation>

```

*The message-destination-ref-name element specifies the name of a message destination reference; its value is the message destination reference name used in the application client code. The name is a JNDI name relative to the java:comp/env context and must be unique within an application client.*

```

    </xsd:documentation>
  </xsd:annotation>

```

```

        <xsd:selector xpath="j2ee:message-destination-ref"/>
        <xsd:field xpath="j2ee:message-destination-ref-name"/>
    </xsd:unique>
</xsd:element>

<!-- ***** -->
<xsd:complexType name="application-clientType">
    <xsd:sequence>
        <xsd:group ref="j2ee:descriptionGroup"/>
        <xsd:element name="env-entry"
            type="j2ee:env-entryType"
            minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="ejb-ref"
            type="j2ee:ejb-refType"
            minOccurs="0" maxOccurs="unbounded"/>
        <xsd:group ref="j2ee:service-refGroup"/>
        <xsd:element name="resource-ref"
            type="j2ee:resource-refType"
            minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="resource-env-ref"
            type="j2ee:resource-env-refType"
            minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="message-destination-ref"
            type="j2ee:message-destination-refType"
            minOccurs="0"
            maxOccurs="unbounded"/>
        <xsd:element name="callback-handler"
            type="j2ee:fully-qualified-classType"
            minOccurs="0">

        <xsd:annotation>
            <xsd:documentation>

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

            </xsd:documentation>
        </xsd:annotation>
    
```

```

</xsd:element>
<xsd:element name="message-destination"
              type="j2ee:message-destinationType"
              minOccurs="0"
              maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="version"
               type="j2ee:dewey-versionType"
               fixed="1.4"
               use="required">

<xsd:annotation>
  <xsd:documentation>

    The required value for the version is 1.4.

  </xsd:documentation>
</xsd:annotation>

</xsd:attribute>
<xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

</xsd:schema>

```

## J2EE.A.6 J2EE:application-client 1.3 XML DTD

This section contains 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:

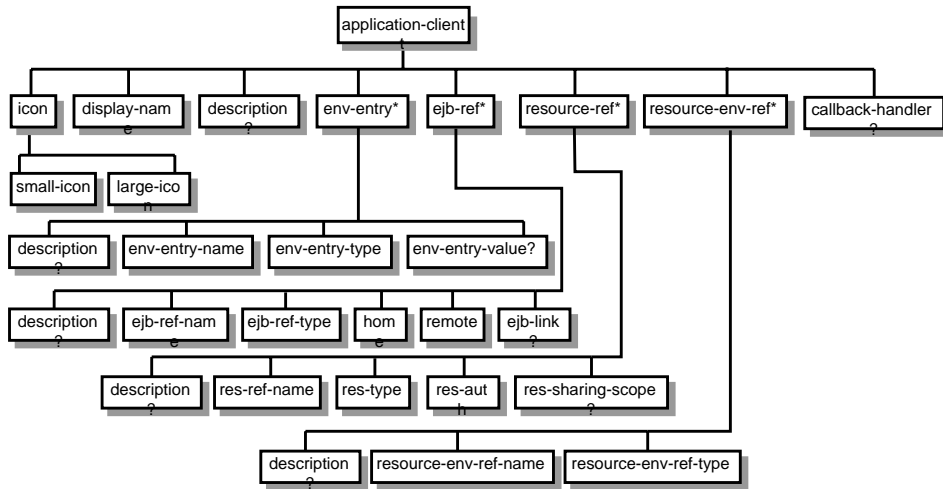
```

<!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.

**Figure J2EE.A-3** shows the structure of the J2EE:application-client XML DTD.



**Figure J2EE.A-3** J2EE:application-client XML DTD Structure

```
<!--
```

This is the XML DTD for the J2EE 1.3 application client deployment descriptor. All J2EE 1.3 application client deployment descriptors must include a DOCTYPE of the following form:

```
<!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 following conventions apply to all J2EE deployment descriptor elements unless indicated otherwise.

- In elements that contain PCDATA, leading and trailing whitespace in the data may be ignored.
- In elements whose value is an "enumerated type", the value is case sensitive.
- In elements that specify a pathname to a file within the same JAR file, relative filenames (i.e., those not starting with "/") are considered relative to the root of the JAR file's namespace. Absolute filenames (i.e., those starting with "/") also specify names in the root of the JAR file's namespace. In general, relative

names are preferred. The exception is .war files where absolute names are preferred for consistency with the servlet API.

-->

<!--

The application-client element is the root element of an application client deployment descriptor. The application client deployment descriptor describes the EJB components and external resources referenced by the application client.

-->

**<!ELEMENT application-client (icon?, display-name, description?, env-entry\*, ejb-ref\*, resource-ref\*, resource-env-ref\*, callback-handler?)>**

<!--

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

Used in: application-client

-->

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

<!--

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

Used in: application-client, ejb-ref, env-entry, resource-env-ref, resource-ref

-->

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

```
<!--
The display-name element contains a short name that is intended to
be displayed by tools. The display name need not be unique.
Used in: application-client
Example:
<display-name>Employee Self Service</display-name>
-->
```

#### **<!ELEMENT display-name (#PCDATA)>**

```
<!--
The ejb-link element is used in the ejb-ref or ejb-local-ref elements
to specify that an EJB reference is linked to another enterprise
bean.
The name in the ejb-link element is composed of a
path name specifying the ejb-jar containing the referenced
enterprise bean with the ejb-name of the target bean appended and
separated from the path name by "#". The path name is relative to
the jar file containing the application client that is referencing
the enterprise bean. This allows multiple enterprise beans with the
same ejb-name to be uniquely identified.
Used in: ejb-ref
Examples:
    <ejb-link>EmployeeRecord</ejb-link>
    <ejb-link>../products/product.jar#ProductEJB</ejb-link>
-->
```

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

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



**<!ELEMENT ejb-ref (description?, ejb-ref-name, ejb-ref-type, home, remote, ejb-link?)>**

<!--

The ejb-ref-name element contains the name of an EJB reference. The EJB reference is an entry in the application client's environment and is relative to the java:comp/env context. The name must be unique within the application client.

It is recommended that name is prefixed with "ejb/".

Used in: ejb-ref

Example:

```
<ejb-ref-name>ejb/Payroll</ejb-ref-name>
```

-->

**<!ELEMENT ejb-ref-name (#PCDATA)>**

<!--

The ejb-ref-type element contains the expected type of the referenced enterprise bean.

The ejb-ref-type element must be one of the following:

```
<ejb-ref-type>Entity</ejb-ref-type>
```

```
<ejb-ref-type>Session</ejb-ref-type>
```

Used in: ejb-ref

-->

**<!ELEMENT ejb-ref-type (#PCDATA)>**

<!--

The env-entry element contains the declaration of an application client's environment entry. The declaration consists of an optional description, the name of the environment entry, and an optional value. If a value is not specified, one must be supplied during deployment.

Used in: application-client

-->

**<!ELEMENT env-entry (description?, env-entry-name, env-entry-type, env-entry-value?)>**

```
<!--
```

The `env-entry-name` element contains the name of an application client's environment entry. The name is a JNDI name relative to the `java:comp/env` context. The name must be unique within an application client.

Used in: `env-entry`

Example:

```
<env-entry-name>minAmount</env-entry-name>
-->
```

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

```
<!--
```

The `env-entry-type` element contains the fully-qualified Java type of the environment entry value that is expected by the application client's code.

The following are the legal values of `env-entry-type`:

```
java.lang.Boolean
java.lang.Byte
java.lang.Character
java.lang.String
java.lang.Short
java.lang.Integer
java.lang.Long
java.lang.Float
java.lang.Double
```

Used in: `env-entry`

Example:

```
<env-entry-type>java.lang.Boolean</env-entry-type>
-->
```

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

```
<!--
```

The `env-entry-value` element contains the value of an application client's environment entry. The value must be a `String` that is valid for the constructor of the specified type that takes a single `String` parameter, or for `java.lang.Character`, a single character.

Used in: `env-entry`

Example:

```
<env-entry-value>100.00</env-entry-value>
-->
```

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

<!--

The home element contains the fully-qualified name of the enterprise bean's home interface.

Used in: ejb-ref

Example:

<home>com.aardvark.payroll.PayrollHome</home>

-->

**<!ELEMENT home (#PCDATA)>**

<!--

The icon element contains small-icon and large-icon elements that specify the file names for small and a large GIF or JPEG icon images used to represent the parent element in a GUI tool.

Used in: application-client

-->

**<!ELEMENT icon (small-icon?, large-icon?)>**

<!--

The large-icon element contains the name of a file containing a large (32 x 32) icon image. The file name is a relative path within the application client's jar file.

The image may be either in the JPEG or GIF format. The icon can be used by tools.

Used in: icon

Example:

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

-->

**<!ELEMENT large-icon (#PCDATA)>**

<!--

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

Used in: ejb-ref

Example:

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

-->

**<!ELEMENT remote (#PCDATA)>**

<!--

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

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

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

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

Used in: `resource-ref`

-->

**<!ELEMENT res-auth (#PCDATA)>**

<!--

The `res-ref-name` element specifies the name of a resource manager connection factory reference. The name is a JNDI name relative to the `java:comp/env` context. The name must be unique within an application client.

Used in: `resource-ref`

-->

**<!ELEMENT res-ref-name (#PCDATA)>**

<!--

The `res-sharing-scope` element specifies whether connections obtained through the given resource manager connection factory reference can be shared. The value of this element, if specified, must be one of the two following:

`<res-sharing-scope>Shareable</res-sharing-scope>`

`<res-sharing-scope>Unshareable</res-sharing-scope>`

The default value is `Shareable`.

Used in: `resource-ref`

-->

**<!ELEMENT res-sharing-scope (#PCDATA)>**

<!--

The `res-type` element specifies the type of the data source. The type is specified by the fully qualified Java language class or interface expected to be implemented by the data source.

Used in: `resource-ref`

-->

**<!ELEMENT res-type (#PCDATA)>**

<!--

The resource-env-ref element contains a declaration of an application client's reference to an administered object associated with a resource in the application client's environment. It consists of an optional description, the resource environment reference name, and an indication of the resource environment reference type expected by the application client code.

Used in: application-client

Example:

```
<resource-env-ref>
  <resource-env-ref-name>jms/StockQueue</resource-env-ref-name>
  <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
</resource-env-ref>
-->
```

**<!ELEMENT resource-env-ref (description?, resource-env-ref-name, resource-env-ref-type)>**

<!--

The resource-env-ref-name element specifies the name of a resource environment reference; its value is the environment entry name used in the application client code. The name is a JNDI name relative to the java:comp/env context and must be unique within an application client.

Used in: resource-env-ref

-->

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

<!--

The resource-env-ref-type element specifies the type of a resource environment reference. It is the fully qualified name of a Java language class or interface.

Used in: resource-env-ref

-->

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

<!--

The resource-ref element contains a declaration of an application client's reference to an external resource. It consists of an optional description, the resource manager connection factory reference name, the indication of the resource manager connection factory type expected by the application client code, the type of authentication (Application or Container), and an optional specification of the shareability of connections obtained from the resource (Shareable or Unshareable).

Used in: application-client

Example:

```
<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> </resource-ref>
-->
```

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

<!--

The small-icon element contains the name of a file containing a small (16 x 16) icon image. The file name is a relative path within the application client's jar file.

The image may be either in the JPEG or GIF format. The icon can be used by tools.

Used in: icon

Example:

```
<small-icon>employee-service-icon16x16.jpg</small-icon>
-->
```

**<!ELEMENT small-icon (#PCDATA)>**

<!--

The ID mechanism is to allow tools that produce additional deployment information (i.e., information beyond the standard deployment descriptor information) to store the non-standard information in a separate file, and easily refer from these tool-specific files to the information in the standard deployment descriptor.

Tools are not allowed to add the non-standard information into the standard deployment descriptor.

-->

```

<!ATTLIST application-client id ID #IMPLIED>
<!ATTLIST callback-handler id ID #IMPLIED>
<!ATTLIST description id ID #IMPLIED>
<!ATTLIST display-name id ID #IMPLIED>
<!ATTLIST ejb-link id ID #IMPLIED>
<!ATTLIST ejb-ref id ID #IMPLIED>
<!ATTLIST ejb-ref-name id ID #IMPLIED>
<!ATTLIST ejb-ref-type id ID #IMPLIED>
<!ATTLIST env-entry id ID #IMPLIED>
<!ATTLIST env-entry-name id ID #IMPLIED>
<!ATTLIST env-entry-type id ID #IMPLIED>
<!ATTLIST env-entry-value id ID #IMPLIED>
<!ATTLIST home id ID #IMPLIED>
<!ATTLIST icon id ID #IMPLIED>
<!ATTLIST large-icon id ID #IMPLIED>
<!ATTLIST remote id ID #IMPLIED>
<!ATTLIST res-auth id ID #IMPLIED>
<!ATTLIST res-ref-name id ID #IMPLIED>
<!ATTLIST res-sharing-scope id ID #IMPLIED>
<!ATTLIST res-type id ID #IMPLIED>
<!ATTLIST resource-env-ref id ID #IMPLIED>
<!ATTLIST resource-env-ref-name id ID #IMPLIED>
<!ATTLIST resource-env-ref-type id ID #IMPLIED>
<!ATTLIST resource-ref id ID #IMPLIED>
<!ATTLIST small-icon id ID #IMPLIED>

```

## J2EE.A.7 J2EE:application-client 1.2 XML DTD

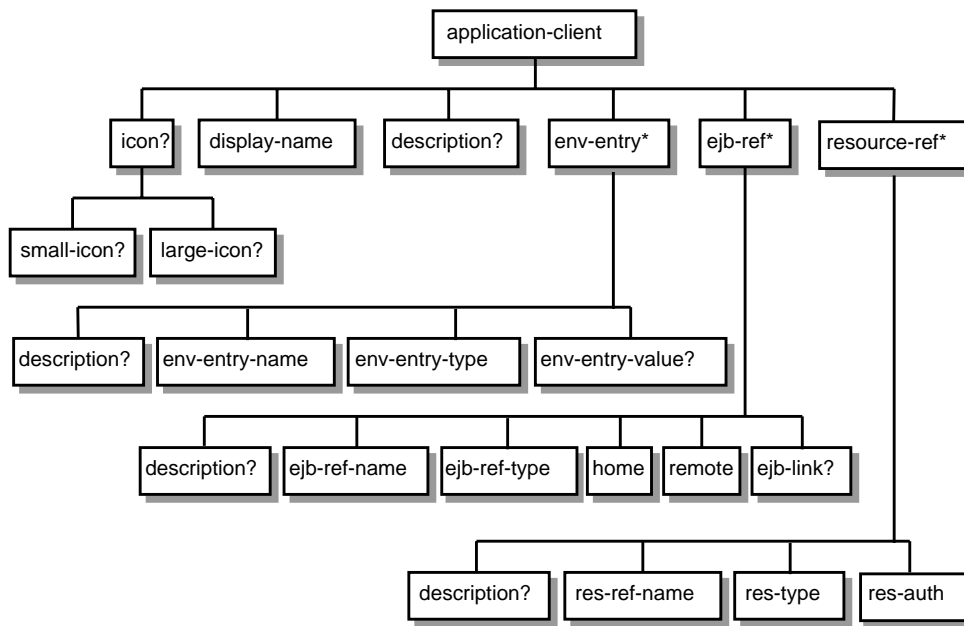
This section contains the XML DTD for the J2EE 1.2 version of the application client deployment descriptor. A valid application client deployment descriptor may contain the following DOCTYPE declaration:

```

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

```

**Figure J2EE.A-4** shows the structure of the J2EE:application-client XML DTD.



**Figure J2EE.A-4** J2EE:application-client XML DTD Structure

```

<!--
The application-client element is the root element of an application
client deployment descriptor.
The application client deployment descriptor describes the EJB
components and external resources referenced by the application
client.
-->

```

```

<!ELEMENT application-client (icon?, display-name, description?,
    env-entry*, ejb-ref*, resource-ref*)>

```

```

<!--
The description element is used to provide text describing the parent
element.
The description element should include any information that the
application-client file producer wants to provide to the consumer of
the application-client file (i.e., to the Deployer).

```



Typically, the tools used by the application-client file consumer will display the description when processing the parent element that contains the description.

-->

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

<!--

The display-name element contains a short name that is intended to be displayed by tools.

-->

**<!ELEMENT display-name (#PCDATA)>**

<!--

The ejb-link element is used in the ejb-ref element to specify that an EJB reference is linked to an enterprise bean in the encompassing J2EE Application package.

The value of the ejb-link element must be the ejb-name of an enterprise bean in the same J2EE Application package.

Used in: ejb-ref

Example: <ejb-link>EmployeeRecord</ejb-link>

-->

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

<!--

The ejb-ref element is used for the declaration of a reference to an enterprise bean's home.

The declaration consists of an optional description; the EJB reference name used in the code of the referencing application client; the expected type of the referenced enterprise bean; the expected home and remote interfaces of the referenced enterprise bean; and an optional ejb-link information.

The optional ejb-link element is used to specify the referenced enterprise bean.

-->

**<!ELEMENT ejb-ref (description?, ejb-ref-name, ejb-ref-type, home, remote, ejb-link?)>**

<!--

The ejb-ref-name element contains the name of an EJB reference.

The EJB reference is an entry in the application client's environment.

It is recommended that name is prefixed with "ejb/".

Used in: ejb-ref

Example: <ejb-ref-name>ejb/Payroll</ejb-ref-name>

-->

**<!ELEMENT ejb-ref-name (#PCDATA)>**

<!--

The ejb-ref-type element contains the expected type of the referenced enterprise bean.

The ejb-ref-type element must be one of the following:

<ejb-ref-type>Entity</ejb-ref-type>

<ejb-ref-type>Session</ejb-ref-type>

Used in: ejb-ref

-->

**<!ELEMENT ejb-ref-type (#PCDATA)>**

<!--

The env-entry element contains the declaration of an application client's environment entries.

The declaration consists of an optional description, the name of the environment entry, and an optional value.

-->

**<!ELEMENT env-entry (description?, env-entry-name, env-entry-type, env-entry-value?)>**

<!--

The env-entry-name element contains the name of an application client's environment entry.

Used in: env-entry

Example: <env-entry-name>EmployeeAppDB</env-entry-name>

-->

**<!ELEMENT env-entry-name (#PCDATA)>**

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

#### **<!ELEMENT env-entry-type (#PCDATA)>**

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

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

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

#### **<!ELEMENT home (#PCDATA)>**

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

#### **<!ELEMENT icon (small-icon?, large-icon?)>**

<!--

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

Example:

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

**<!ELEMENT large-icon (#PCDATA)>**

<!--

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

Used in: `ejb-ref`

Example:

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

**<!ELEMENT remote (#PCDATA)>**

<!--

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

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

```
<res-auth>Application</res-auth>
<res-auth>Container</res-auth>
-->
```

**<!ELEMENT res-auth (#PCDATA)>**

<!--

The `res-ref-name` element specifies the name of the resource factory reference name. The resource factory reference name is the name of the application client's environment entry whose value contains the JNDI name of the data source.

Used in: `resource-ref`

-->

**<!ELEMENT res-ref-name (#PCDATA)>**

**<!--**

The res-type element specifies the type of the data source. The type is specified by the Java interface (or class) expected to be implemented by the data source.

Used in: resource-ref

**-->**

**<!ELEMENT res-type (#PCDATA)>**

**<!--**

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

Example:

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

**<!ELEMENT resource-ref (description?, res-ref-name, res-type, res-auth)>**

**<!--**

The small-icon element contains the name of a file containing a small (16 x 16) icon image.

The file name is a relative path within the application-client jar file.

The image must be either in the JPEG or GIF format, and the file name must end with the suffix ".jpg" or ".gif" respectively.

The icon can be used by tools.

Example:

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

**<!ELEMENT small-icon (#PCDATA)>**

```
<!--
```

```
The ID mechanism is to allow tools to easily make tool-specific
references to the elements of the deployment descriptor.
```

```
-->
```

```
<!--ATTLIST application-client id ID #IMPLIED>
<!--ATTLIST description id ID #IMPLIED>
<!--ATTLIST display-name id ID #IMPLIED>
<!--ATTLIST ejb-link id ID #IMPLIED>
<!--ATTLIST ejb-ref id ID #IMPLIED>
<!--ATTLIST ejb-ref-name id ID #IMPLIED>
<!--ATTLIST ejb-ref-type id ID #IMPLIED>
<!--ATTLIST env-entry id ID #IMPLIED>
<!--ATTLIST env-entry-name id ID #IMPLIED>
<!--ATTLIST env-entry-type id ID #IMPLIED>
<!--ATTLIST env-entry-value id ID #IMPLIED>
<!--ATTLIST home id ID #IMPLIED>
<!--ATTLIST icon id ID #IMPLIED>
<!--ATTLIST large-icon id ID #IMPLIED>
<!--ATTLIST remote id ID #IMPLIED>
<!--ATTLIST res-auth id ID #IMPLIED>
<!--ATTLIST res-ref-name id ID #IMPLIED>
<!--ATTLIST res-type id ID #IMPLIED>
<!--ATTLIST resource-ref id ID #IMPLIED>
<!--ATTLIST small-icon id ID #IMPLIED>
```

# A P P E N D I X J2EE.B

---

## Revision History

### **J2EE.B.1 Changes in Expert Draft 1**

#### **J2EE.B.1.1 Additional Requirements**

- Updated entire specification to require J2SE 5.0, and to reflect that several optional packages are now part of J2SE
- Added requirements for many new APIs, see Chapter J2EE.6, “Application Programming Interface” for details.

#### **J2EE.B.1.2 Removed Requirements**

- None.

#### **J2EE.B.1.3 Editorial Changes**

- Incorporated J2EE 1.4 maintenance review change to make it clear that a security manager is not required.
- Updated Section J2EE.8.2, “Library Support” to make it clear that Class-Path entries may also refer to directories.
- Removed AWT requirements, which are now fully specified in the J2SE specification.
- Made explicit the requirement that J2EE products must be able to deploy JDBC drivers that have been packaged as resource adapters. This has always been true; there’s nothing special about JDBC drivers when packaged as resource adapters. See Section J2EE.6.2.4.2, “JDBC™ API.”

## **J2EE.B.2     Changes in Expert Draft 2**

### **J2EE.B.2.1     Additional Requirements**

- Updated WS-I requirement to match JAX-RPC 2.0. See Section J2EE.7.2.1, “Internet and Web Protocols.”
- EJB containers must now be capable of supporting the same security permissions as the web container, including access to files. See Table J2EE.6-2.
- Significant updates to Chapter J2EE.5, “Resources, Naming, and Injection” and Chapter J2EE.8, “Application Assembly and Deployment” to describe the use of annotations and deployment descriptors to specify resource injection.
- Update version numbers of referenced specifications in Appendix J2EE.C, “Related Documents.”

### **J2EE.B.2.2     Removed Requirements**

- None.

### **J2EE.B.2.3     Editorial Changes**

- Clarified that support for a CORBA Portable Object Adapter is required. See Section J2EE.6.2.4.5, “RMI-IIOP.”
- Moved J2EE 1.4 deployment descriptor schemas to Appendix J2EE.A, “Previous Version Deployment Descriptors.”
- Updated application deployment descriptor to version 5.0 in Section J2EE.8.5, “J2EE Application XML Schema.”
- Update application client deployment descriptor to version 5.0 in Section J2EE.9.7, “J2EE Application Client XML Schema.”
- Fixed many typos, wording problems, etc.



## **J2EE.B.3     Changes in Early Draft Review 1**

### **J2EE.B.3.1     Additional Requirements**

- Added EJB 3.0 Persistence as a separate entry in Chapter J2EE.6, “Application Programming Interface.”
- Application client fields or methods that are injection targets must be static. See Section J2EE.5.2.3, “Annotations and Resource Injection.”

### **J2EE.B.3.2     Removed Requirements**

- None.

### **J2EE.B.3.3     Editorial Changes**

- Updated figures.
- Clearly marked some of the incomplete sections.
- Added references to included specifications in Appendix J2EE.C, “Related Documents.”
- Added note about EJB 3.0 Persistence in section Section J2EE.6.26, “Java Persistence API 1.0.”

## **J2EE.B.4     Changes in Early Draft Review 2**

### **J2EE.B.4.1     Additional Requirements**

- Updated SAAJ to version 1.3 in Section J2EE.6.1.2, “Java Optional Packages.”
- Added requirements for deploying application packages with no deployment descriptor. See Section J2EE.8.4.2, “Deploying a J2EE Application.”
- Added requirement for support of image/png data in Chapter J2EE.6, “Application Programming Interface.”
- Reverted JAX-RPC to version 1.1 and added JAX-WS 2.0 as an additional requirement in Chapter J2EE.6, “Application Programming Interface.”
- Updated JAF to version 1.1 in Section J2EE.6.9, “JavaBeans™ Activation Framework 1.1 Requirements.”

- Expanded and clarified the requirements around resource injection in Section J2EE.5.2.3, “Annotations and Resource Injection.”
- Filled in requirements for JSR-250 in Section J2EE.6.24, “Common Annotations for the Java™ Platform 1.0 Requirements.”

#### **J2EE.B.4.2     Removed Requirements**

- None.

#### **J2EE.B.4.3     Editorial Changes**

- Clarified responsibilities of libraries that dynamically load classes, in Section J2EE.8.2.5, “Dynamic Class Loading.”

### **J2EE.B.5     Changes in Public Review Draft**

#### **J2EE.B.5.1     Additional Requirements**

- It must be possible to store references to EJB 3 business interfaces in an HttpSession object. See Section J2EE.6.4, “Servlet 2.4 Requirements.”

#### **J2EE.B.5.2     Removed Requirements**

- None.

#### **J2EE.B.5.3     Editorial Changes**

- EDR2 was never published to the public because Public Review is coming only a week later.
- Fixed some typos.





# APPENDIX J2EE.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.

*Java™ 2 Platform, Enterprise Edition Specification Version 5.0* (this specification). Available at <http://java.sun.com/j2ee/docs.html>.

*Java™ 2 Platform, Enterprise Edition Technical Overview* (J2EE Overview). Available at <http://java.sun.com/j2ee/white.html>.

*Java™ 2 Platform, Standard Edition, v5.0 API Specification* (J2SE specification). Available at <http://java.sun.com/j2se/5.0/docs/api/index.html>.

*Enterprise JavaBeans™ Specification, Version 3.0* (EJB specification). Available at <http://java.sun.com/products/ejb>.

*JavaServer Pages™ Specification, Version 2.1* (JSP specification). Available at <http://java.sun.com/products/jsp>.

*Java™ Servlet Specification, Version 2.4* (servlet specification). Available at <http://java.sun.com/products/servlet>.

*JDBC™ 3.0 API* (JDBC specification). Available at <http://java.sun.com/products/jdbc>.

*Java™ Naming and Directory Interface 1.2 Specification* (JNDI specification). Available at <http://java.sun.com/products/jndi>.

*Java™ Message Service, Version 1.1* (JMS specification). Available at <http://java.sun.com/products/jms>.

- Java™ Transaction API, Version 1.0.1B* (JTA specification). Available at <http://java.sun.com/products/jta>.
- Java™ Transaction Service, Version 1.0* (JTS specification). Available at <http://java.sun.com/products/jts>.
- JavaMail™ API Specification Version 1.3* (JavaMail specification). Available at <http://java.sun.com/products/javamail>.
- JavaBeans™ Activation Framework Specification Version 1.0* (JAF specification). Available at <http://java.sun.com/beans/glasgow/jaf.html>.
- J2EE™ Connector Architecture 1.5* (Connector specification). Available at <http://java.sun.com/j2ee/connector>.
- Java™ API for XML Processing, Version 1.3* (JAXP specification). Available at <http://java.sun.com/xml>.
- Web Services for J2EE 1.1* (Web Services specification). Available at <http://jcp.org/en/jsr/detail?id=921>.
- Java™ API for XML-based RPC 2.0* (JAX-RPC specification). Available at <http://java.sun.com/xml/jaxrpc>.
- SOAP with Attachments API for Java™ 1.3* (SAAJ specification). Available at <http://java.sun.com/xml/saaj>.
- Java™ API for XML Registries 1.0* (JAXR specification). Available at <http://java.sun.com/xml/jaxr>.
- Java™ 2 Platform, Enterprise Edition Management Specification 1.0* (J2EE Management specification). Available at <http://jcp.org/jsr/detail/77.jsp>.
- Java™ 2 Platform, Enterprise Edition Deployment Specification 1.1* (J2EE Deployment specification). Available at <http://jcp.org/jsr/detail/88.jsp>.
- Java™ Management Extensions 1.2* (JMX specification). Available at <http://java.sun.com/products/JavaManagement/>.
- Java™ Authorization Service Provider Contract for Containers 1.0* (JACC specification). Available at <http://jcp.org/jsr/detail/115.jsp>.

*Java™ Authentication and Authorization Service (JAAS) 1.0* (JAAS specification). Available at <http://java.sun.com/products/jaas>.

*Debugging Support for Other Languages 1.0*. Available at <http://jcp.org/en/jsr/detail?id=45>.

*Standard Tag Library for JavaServer Pages 1.1* (JSTL specification). Available at <http://jcp.org/en/jsr/detail?id=52>.

*Web Services Metadata for the Java Platform 1.0*. Available at <http://jcp.org/en/jsr/detail?id=181>.

*JavaServer Faces 1.2* (JSF specification). Available at <http://jcp.org/en/jsr/detail?id=252>.

*Streaming API for XML 1.0* (StAX specification). Available at <http://jcp.org/en/jsr/detail?id=173>.

*Enterprise JavaBeans 3.0 Persistence API*. Available at <http://java.sun.com/products/ejb>.

*Extension Mechanism Architecture*, Available at <http://java.sun.com/j2se/5.0/docs/guide/extensions>.

*Optional Package Versioning*, Available at <http://java.sun.com/j2se/5.0/docs/guide/extensions>.

*JAR File Specification*, Available at <http://java.sun.com/j2se/5.0/docs/guide/jar/jar.html>.

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

*CORBA 2.6 - Chapter 26 - Secure Interoperability*, Available at <http://www.omg.org/cgi-bin/doc?formal/01-12-30>.

*IDL To Java™ Language Mapping Specification*, Available at <http://www.omg.org/cgi-bin/doc?ptc/2000-01-08>.

*Java™ Language To IDL Mapping Specification*, Available at <http://www.omg.org/cgi-bin/doc?ptc/2000-01-06>.

*Interoperable Naming Service*, Available at <http://www.omg.org/cgi-bin/doc?ptc/00-08-07>.

*Transaction Service Specification* (OTS specification), Available at <http://www.omg.org/cgi-bin/doc?formal/2001-11-03>.

*Designing Enterprise Applications with the Java™ 2 Platform, Enterprise Edition*, Available at <http://java.sun.com/j2ee/blueprints>.

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







**We make the net work.**

Sun Microsystems, Inc.  
4150 Network Circle  
Santa Clara, California 95054, U.S.A.  
650 960-1300

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

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

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