ORACLE®

Java™ EE Connector Architecture Specification

Version 1.7

March 2013

ORACLE IS WILLING TO LICENSE THIS SPECIFICATION TO YOU ONLY UPON THE CONDITION THAT YOU ACCEPT ALL OF THE TERMS CONTAINED IN THIS AGREEMENT. PLEASE READ THE TERMS AND CONDITIONS OF THIS AGREEMENT CAREFULLY. BY DOWNLOADING THIS SPECIFICATION, YOU ACCEPT THE TERMS AND CONDITIONS OF THE AGREEMENT. IF YOU ARE NOT WILLING TO BE BOUND BY IT, SELECT THE "DECLINE" BUTTON AT THE BOTTOM OF THIS PAGE.

Specification: JSR-000322 Java (TM) EE Connector Architecture ("Specification")

Version: 1.7

Status: Final Release

Specification Lead: Oracle America, Inc. ("Specification Lead")

Release: On or before, May 31 2013

Copyright 2013 Oracle America, Inc.

All rights reserved.

LIMITED LICENSE GRANTS

1. License for Evaluation Purposes. Specification Lead hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense), under Specification Lead'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) discussing the Specification with any third party; and (iii) 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 Specification.

2. License for the Distribution of Compliant Implementations. Specification Lead also grants you a perpetual, non-exclusive, non-transferable, worldwide, fully paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights or, subject to the provisions of subsection 4 below, patent rights it may have covering the Specification to create and/or distribute an Independent Implementation of the Specification that: (a) fully implements the Specification including all its required interfaces and functionality; (b) does not modify, subset, superset or otherwise extend the Licensor Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the Licensor Name Space other than those required/authorized by the Specification or Specifications being implemented; and (c) passes the Technology Compatibility Kit (including satisfying the requirements of the applicable TCK Users Guide) for such Specification ("Compliant Implementation"). In addition, the foregoing license is expressly conditioned on your not acting outside its scope. No license is granted hereunder for any other purpose (including, for example, 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 Specification Lead or Specification Lead's licensors is granted hereunder. Java, and Java-related logos, marks and names are trademarks or registered trademarks of Oracle America, Inc. in the U.S. and other countries.

3. Pass-through Conditions. You need not include limitations (a)-(c) from the previous paragraph or any other particular "pass through" requirements in any license You grant concerning the use of your Independent Implementation or products derived from it. However, except with respect to Independent Implementations (and products derived from them) that satisfy limitations (a)-(c) from the previous paragraph, You may neither: (a) grant or otherwise pass through to your licensees any licenses under Specification Lead's applicable intellectual property rights; nor (b) authorize your licensees to make any claims concerning their implementation's compliance with the Specification in question.

4. Reciprocity Concerning Patent Licenses.

a. With respect to any patent claims covered by the license granted under subparagraph 2 above that would be infringed by all technically feasible implementations of the Specification, such license is conditioned upon your offering on fair, reasonable and non-discriminatory terms, to any party seeking it from You, a perpetual, non-exclusive, non-transferable, worldwide license under Your patent rights which are or would be infringed by all technically feasible implementations of the Specification to develop, distribute and use a Compliant Implementation.

Please
Recycle

Adobe PostScript™

b With respect to any patent claims owned by Specification Lead and covered by the license granted under subparagraph 2, whether or not their infringement can be avoided in a technically feasible manner when implementing the Specification, such license shall terminate with respect to such claims if You initiate a claim against Specification Lead that it has, in the course of performing its responsibilities as the Specification Lead, induced any other entity to infringe Your patent rights.

c Also with respect to any patent claims owned by Specification Lead and covered by the license granted under subparagraph 2 above, where the infringement of such claims can be avoided in a technically feasible manner when implementing the Specification such license, with respect to such claims, shall terminate if You initiate a claim against Specification Lead that its making, having made, using, offering to sell, selling or importing a Compliant Implementation infringes Your patent rights.

5. Definitions. For the purposes of this Agreement: "Independent Implementation" shall mean an implementation of the Specification that neither derives from any of Specification Lead's source code or binary code materials nor, except with an appropriate and separate license from Specification Lead, includes any of Specification Lead's source code or binary code materials; "Licensor Name Space" shall mean the public class or interface declarations whose names begin with "java", "javax", "com.<Specification Lead>" or their equivalents in any subsequent naming convention adopted by Oracle through the Java Community Process, or any recognized successors or replacements thereof; and "Technology Compatibility Kit" or "TCK" shall mean the test suite and accompanying TCK User's Guide provided by Specification Lead which corresponds to the Specification and that was available either (i) from Specification Lead's 120 days before the first release of Your Independent Implementation that allows its use for commercial purposes, or (ii) more recently than 120 days from such release but against which You elect to test Your implementation of the Specification.

This Agreement will terminate immediately without notice from Specification Lead if you breach the Agreement or act outside the scope of the licenses granted above.

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED "AS IS". SPECIFICATION LEAD 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 SPECIFICATION LEAD 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, ARISING OUT OF OR RELATED IN ANY WAY TO YOUR HAVING, IMPELEMENTING OR OTHERWISE USING USING THE SPECIFICATION, EVEN IF SPECIFICATION LEAD AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will indemnify, hold harmless, and defend Specification Lead and its licensors from any claims arising or resulting from: (i) your use of the Specification; (ii) the use or distribution of your Java application, applet and/or implementation; and/or (iii) any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

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 Specification Lead 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 Specification Lead 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.

GENERAL TERMS

Any action related to 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.

The Specification is subject to U.S. export control laws and may be subject to export or import regulations in other countries. Licensee agrees to comply strictly with all such laws and regulations and acknowledges that it has the responsibility to obtain such licenses to export, re-export or import as may be required after delivery to Licensee.

This Agreement is the parties' entire agreement relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, conditions, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification to this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

Rev. April, 2006

Contents

Tables

Figures

Code Samples

CHAPTER **1**

Introduction

The Java™ Platform, Enterprise Edition (Java™ EE platform) provides containers for client applications, web components based on Java™ Servlets and JavaServer Pages™ (JSP™) and Enterprise JavaBeans™ (EJB™) components. These containers provide deployment and runtime support for application components. They provide a federated view of the services provided by the underlying application server for the application components.

Containers can run on existing systems; for example, web servers for the web containers; application servers, TP monitors, and database systems for EJB containers. This enables enterprises to leverage both the advantages of their existing systems and those of Java EE. Enterprises can write, or rewrite, new applications using Java EE capabilities and can also encapsulate parts of existing applications in Enterprise JavaBeans, JavaServer Pages or servlets.

Enterprise applications access functions and data associated with applications running on Enterprise Information Systems (EIS). Application servers extend their containers and support connectivity to heterogeneous EISs. Enterprise tools and Enterprise Application Integration (EAI) vendors add value by providing tools and frameworks to simplify the EIS integration task.

For enterprise application integration, bi-directional connectivity between enterprise applications and EIS is essential. The Java EE Connector architecture defines standard contracts that allow bi-directional connectivity between enterprise applications and EISs. It also formalizes the relationships, interactions, and the packaging of the integration layer, thus enabling enterprise application integration.

1.1 Overview

The Java EE Connector architecture defines a standard architecture for connecting the Java EE platform to heterogeneous EISs. Examples of EISs include Enterprise Resource Planning (ERP), mainframe transaction processing (TP), and database systems.

The connector architecture defines a set of scalable, secure, and transactional mechanisms that enable the integration of EISs with application servers[1] and enterprise applications.

The connector architecture also defines a Common Client Interface (CCI) for EIS access. The CCI defines a client API for interacting with heterogeneous EISs.

The connector architecture enables an EIS vendor to provide a standard resource adapter for its EIS. A resource adapter is a system-level software driver that is used by a Java application to connect to an EIS. The resource adapter plugs into an application server and provides connectivity between the EIS, the application server, and the enterprise application. The resource adapter serves as a protocol adapter that allows any arbitrary EIS communication protocol to be used for connectivity.

An application server vendor extends its system once to support the connector architecture and is then assured of seamless connectivity to multiple EISs. Likewise, an EIS vendor provides one standard resource adapter which has the capability to plug in to any application server that supports the connector architecture.

1.2 Scope

Version 1.0 of the connector architecture defines:

- A standard set of system-level contracts between an application server and EIS. These contracts focus on the important system-level aspects of integration: connection management, transaction management, and security.
- A Common Client Interface (CCI) that defines a client API for interacting with multiple EISs.
- A standard deployment and packaging protocol for resource adapters.

Refer to section 2.2.2 for the rationale behind the Common Client Interface.

Version 1.5 of the connector architecture defines:

---

1. Application server is a generic term used in this document to refer to a middle-tier component server that is compliant with the Java 2 Platform, Enterprise Edition.

- **Lifecycle management contract.** A contract between an application server and a resource adapter that allows an application server to manage the lifecycle of a resource adapter. This contract provides a mechanism for the application server to bootstrap a resource adapter instance during its deployment or application server startup, and to notify the resource adapter instance during its undeployment or during an orderly shutdown of the application server.
- **Work management contract.** A contract between an application server and a resource adapter that allows a resource adapter to do work (monitor network endpoints, call application components, etc.) by submitting `Work` instances to an application server for execution. The application server dispatches threads to execute submitted `Work` instances. This allows a resource adapter to avoid creating or managing threads directly, and allows an application server to efficiently pool threads and have more control over its runtime environment. The resource adapter can control the security context and transaction context with which `Work` instances are executed.
- **Transaction inflow contract.** A contract between an application server and a resource adapter that allows a resource adapter to propagate an imported transaction to an application server. This contract also allows a resource adapter to transmit transaction completion and crash recovery calls initiated by an EIS, and ensures that the ACID (Atomicity, Consistency, Isolation and Durability) properties of the imported transaction are preserved.
- **Message inflow contract.** A standard, generic contract between an application server and a resource adapter that allows a resource adapter to asynchronously deliver messages to message endpoints residing in the application server 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 wide range of message providers (Java Message Service (JMS), Java API for XML Messaging (JAXM), etc.) to be plugged into any Java EE compatible application server by way of a resource adapter.
- **Packaging Model.** Describes the packaging model for different types of resource adapters (outbound only, inbound only, or both).

Version 1.6 of the Connector Architecture defines:
- **Generic work context contract.** A generic contract that enables a resource adapter to control the execution context of a `Work` instance that it has submitted to the application server for execution. The Generic work contract provides the mechanism for a resource adapter to augment the runtime context of a `Work` instance with additional contextual information flown-in from the EIS. This contract enables a resource adapter to control, in a more flexible manner, the contexts in which the `Work` instances submitted by it are executed by the application server's `WorkManager`.
- **Security work context.** A standard contract that enables a resource adapter to establish security information while submiting a `Work` instance for execution to a `WorkManager` and while delivering messages to message endpoints residing in the application server. This contract provides a mechanism to support the

execution of a `Work` instance in the context of an established identity. It also supports the propagation of user information/Principal information from an EIS to a MessageEndpoint during Message Inflow.

1.3 Target Audience

The target audience for this specification includes:

- EIS vendors and resource adapter providers
- Messaging system vendors
- Application server vendors and container providers
- Enterprise application developers and system integrators
- Enterprise tool and EAI vendors

The system-level contracts between an application server and an EIS are targeted towards EIS vendors (or resource adapter providers, if the two roles are different) and application server vendors. The CCI is targeted primarily towards enterprise tools and EAI vendors.

1.4 JDBC and Connector Architecture

The Java^{TM} DataBase Connectivity ("JDBC^{TM}") API defines a standard Java API for accessing relational databases. The JDBC technology provides an API for sending SQL statements to a database and processing the tabular data returned by the database.

The connector architecture is a standard architecture for integrating Java EE applications with EISs that are not relational databases. Each of these EISs currently provides a native function call API for identifying a function to call, specifying its input data, and processing its output data. The goal of the Common Client Interface (CCI) is to provide an EIS independent API for coding these EIS function calls.

The CCI is targeted at EIS development tools and other sophisticated users of EISs. The CCI provides a way to minimize the EIS specific code required by such tools. Most Java EE developers will access EISs using these tools rather than using CCI directly.

It is expected that many Java EE applications will combine relational database access using JDBC with EIS access using EIS access tools based on CCI.

The connector architecture defines a standard SPI (Service Provider Interface) for integrating the transaction, security, and connection management facilities of an application server with those of a transactional resource manager. The JDBC 3.0 specification (see "JDBC API Specification, version 4.1" on page E-1) specifies the relationship of JDBC to the SPI specified in the connector architecture.

1.5 Relationship With Other Integration Technologies (JBI and SCA)

The Enterprise Application Integration (EAI) and Business to Business integration (B2B) functional space may be considered, in an abstract sense, as forms of network service composition. That is, in a typical EAI/B2B scenario, an enterprise application may make use of network resources to realize some of its functionality. In this context, the network resource may be a REST service, a SOAP service, a database server, a JMS topic/queue, some legacy application, etc.

The Java Business Integration (JBI) and Service Component Architecture (SCA) are integration technologies that come to mind in the EAI/B2B space. They allow the creation and consumption of such network services. They enable the building of applications through composition of services in an enterprise by adopting a Service Oriented Architecture (SOA). These technologies can be used to implement integration with various forms of network resources that are not tied to a specific external architectural style.

The Connector architecture covers the category of network resources that expose some form of connection oriented protocol. Database servers, JMS systems, legacy apps, etc. typically fall into this category of network resource. The Connector architecture is the mechanism that the Java EE platform provides to simplify use of such network resources.

1.6 Organization

This document begins by describing the rationale and goals for creating a standard architecture to integrate an application server with multiple heterogeneous EISs. It then describes the key concepts relevant to the connector architecture. These sections provide an overview of the architecture.

This document then describes typical scenarios for using the connector architecture. This chapter introduces the various roles and responsibilities involved in the development and deployment of enterprise applications that integrate with multiple EISs.

After these descriptive sections, this document focuses on the prescriptive aspects of the connector architecture.

1.7 Document Conventions

A regular Palatino font is used for describing the connector architecture.

An *italic* font is used for paragraphs that contain descriptive notes providing clarifications.

A regular `Courier` font is used for Java source code, class, interface and method names.

The requirements section occurring in various chapters of this document highlight only the salient requirements, but do not contain all the requirements. So, this entire document must be used as a requirements specification.

Note that the scenarios described in this document are illustrative in scope. The intent of the scenarios is not to specify a prescriptive way of implementing a particular contract.

This document uses the EJB component model to describe some scenarios. The EJB specification (see "Enterprise JavaBeans (EJB) Specification, version 3.2" on page E-1) provides the latest details of the EJB component model.

1.8 Connector Architecture Version 1.7 Contributor Details

1.8.1 Expert Group Details (JSR-322)

Refer to the URL `http://www.jcp.org/en/jsr/detail?id=322` for details on the first maintenance release of JSR-322. The following table lists the members of the Connector Expert Group.

TABLE 1-1   Connector 1.7 Expert Group Members

Company	Expert Member
IBM	Fred Rowe
Oracle	Weijun Tian, Sivakumar Thygaarajan (**Specification Lead**)
Red Hat Middleware LLC	Jesper Pedersen

1.8.2 Acknowledgements

This maintenance release of the specification is the result of the collaborative efforts of the Java EE Connector Architecture 1.7 Expert Group (JSR-322) (individual members listed above) through discussions in the java.net project at `http://connector-spec.java.net`.

The specification leads of the Java EE 7 Platform, Bill Shannon and Linda DeMichiel, provided overall guidance and helped review the specification. Nigel Deakin and Marina Vatkina provided valuable inputs for this specification related to enhancements in the Message Inflow chapter. Byung-Woo Jun from Ericsson Infotech AB participated as an expert group member earlier in this maintenance release and gave valuable inputs for this specification.

Binod PG reviewed drafts of the specification and gave valuable feedback. Jagadish Ramu and Dapeng Hu from the Java EE Reference Implementation team (Project GlassFish) and Paul Hendley from the Java EE Compatibility Test Suite team provided a lot of useful comments and inputs to the content of this specification.

Murthy Narayanan and Prasad Subramanian provided management support.

1.9 Connector Architecture Version 1.6 Contributor Details

1.9.1 Expert Group Details (JSR-322)

Refer to the URL http://www.jcp.org/en/jsr/detail?id=322 for details on JSR-322. The following table lists the members of the Connector Expert Group.

TABLE 1-2 Connector 1.6 Expert Group Members

Company	Expert Member
Ericsson Infotech AB	Eltjo Boersma
IBM	Piotr Przybylski
Oracle	Jim Gish, Michael Gilbode, Weijun Tian
Red Hat Middleware LLC	Adrian Brock, Jesper Pedersen
SAP AG Walldorf	Nikolai Tankov
Sun Microsystems, Inc.	Binod PG and Sivakumar Thyagarajan (**Specification Leads**)
Tmax Soft, Inc	Bongjae Chang, Eugene Chung

1.9.2 Acknowledgements

Many people contributed to the preparation and review of this specification and we would like to thank them.

This specification is the result of the collaborative efforts of the Java EE Connector Architecture 1.6 Expert Group whose individual members, listed above, contributed countless hours to ensure the success of this specification.

The specification leads of the Java EE 6 Platform, Bill Shannon and Roberto Chinnici, provided overall guidance and helped review the specification. Ron Monzillo provided technical and design assistance for the Security Inflow Contract of the specification. Kenneth Saks, Lance Andersen, Rajiv Mordani, George Tharakan, Frank Kieviet and Jan Luehe provided guidance and reviewed drafts of the specification.

Jagadish Ramu and Shalini Muthukrishnan from the Java EE Reference Implementation team (Project GlassFish) and Paul Hendley from the Java EE Compatibility Test Suite team provided a lot of useful comments and feedback on the various drafts of the specification.

Prabha MS, Paromita Dutta and Chinmayee Srivathsa helped with the editorial tools and Janice Helb provided excellent editorial assistance. Clemens Tator, Stephen Connolly and M. Ganesh Sudhakar provided invaluable comments through the JSR 322 comments alias.

Tom Kincaid, Murthy Narayanan and Srikanth Anandal provided management support.

1.10 Connector Architecture Version 1.5 Contributor Details

1.10.1 Expert Group Details (JSR-112)

Refer to the URL http://www.jcp.org/en/jsr/detail?id=112 for details on JSR-112. The following table lists the members of the Connector Expert Group.

TABLE 1-3   Connector 1.5 Expert Group Members

Company	Expert Member
BEA	Jim Gish, Mitch Upton, Tom Mitchell
Bull S.A.	Michael Giroux
Ericsson Infotech AB	Peter Kristiansson
Fujitsu Limited	Ivar Alexander
Hewlett-Packard	John Speidel
IBM	Michael Beisiegel, Piotr Przybylski
Bahwan Cybertek Technologies Inc.	Vijay Sundhar
Individual Expert	Charlton Barreto
IONA Technologies PLC	Gary Tully
MicroFocus	Stephen Gennard
NEON Systems, Inc.	Yongtao You

TABLE 1-3   Connector 1.5 Expert Group Members

Company	Expert Member
Oracle	Anthony Lai
SAP AG Walldorf	Nikolai Tankov
Siemens AG	Ulf Lange
Silverstream Software	Roman Kishchenko
Sonic Software	Jaime Meritt
Sun Microsystems, Inc.	Ram Jeyaraman (**Specification Lead**)
Sybase	Mark DeLaFranier
TIBCO Software Inc.	Jon Dart, Anuradha M. Sastri
Unisys	Albert DeNigris
WebMethods Corp	Bruce Tran, Bruno Kurtic

1.10.2   Acknowledgements

1.11 Connector Architecture Version 1.0 Contributor Details

1.11.1 Expert Group Details (JSR-16)

Refer to the URL http://www.jcp.org/en/jsr/detail?id=16 for details on JSR-16. The following are part of the expert group and have made invaluable contributions to the Connector architecture specification:

TABLE 1-4 Connector 1.0 Expert Group Members

Company	Expert Member
BEA	Pete Homan
Fujitsu	Yoshi Otagiri, Ivar Alexander
IBM	Tom Freund, Michael Beisiegel
Inline	Jack Greenfield
Inprise	Charlton Barreto
IPlanet	Tony Pan, Pavan Bhatnagar
Motorola	Guy Bieber
Oracle	Dan Coyle
SAP	Marek Barwicki
Sun	Rahul Sharma (**Specification Lead**) Fred H. Carter
Sybase	Rajini Balay, K. Swaminathan
Tibco	Jon Dart
Unisys	Lester Lee

1.11.2 Acknowledgements

Shel Finkelstein, Mark Hapner, Vlada Matena, Tony Ng, Bill Shannon, and Sekhar Vajjhala (all from Sun Microsystems) have provided invaluable technical input and guidance to the Connector architecture specification. Jean Zeng and Pong Ching also provided useful input to the specification.

Rick Cattell, Shel Finkelstein, Bonnie Kellett, and Jeff Jackson have provided huge support to the specification lead in the management of the Connectors expert group.

Tony Ng is leading the effort of providing a reference implementation for the Connector architecture as part of J2EE 1.3 platform. Liz Blair has worked on providing the Compatibility Test Suite (CTS) plan for the Connector architecture.

Beth Stearns was a great help in doing an editorial review of this document.

Overview

This chapter introduces key concepts that are required to understand the connector architecture. It lays down a reference framework to facilitate a formal specification of the connector architecture in the subsequent chapters of this document.

2.1 Definitions

2.1.1 Enterprise Information System (EIS)

An EIS provides the information infrastructure for an enterprise. An EIS offers a set of services to its clients. These services are exposed to clients as local and/or remote interfaces. Examples of an EIS include:

- Enterprise Resource Planning (ERP) system
- Mainframe transaction processing (TP) system
- Legacy database system

There are two aspects of an EIS:

- System level services - for example, SAP RFC, CICS ECI
- An application specific interface - for example, the table schema and specific stored procedures of a database, the specific CICS TP program

2.1.2   Connector Architecture

An architecture for integrating Java EE servers with EISs. There are two parts to this architecture: an EIS vendor-provided resource adapter and an application server that allows this resource adapter to be plugged in. This architecture defines a set of contracts (such as transactions, security, connection management) that a resource adapter has to support to plug in to an application server.

These contracts support bi-directional communication (outbound and inbound) between an application server and an EIS by way of a resource adapter. That is, the application server may use the resource adapter for outbound communication to the EIS, and it may also use the resource adapter for inbound communication from the EIS.

2.1.3   EIS Resource

An EIS resource provides EIS-specific functionality to its clients. Examples are:

- A record or set of records in a database system
- A business object in an ERP system
- A transaction program in a transaction processing system

2.1.4   Resource Manager (RM)

A resource manager manages a set of shared EIS resources. A client requests access to a resource manager to use its managed resources. A transactional resource manager can participate in transactions that are externally controlled and coordinated by a transaction manager.

In the context of the connector architecture, a client of a resource manager can either be a middle-tier application server or a client-tier application. A resource manager is typically in a different address space or on a different machine from the client that accesses it.

This document refers to an EIS as a resource manager when it is mentioned in the context of transaction management. Examples of resource managers are a database system, a mainframe TP system, and an ERP system.

2.1.5　Managed Environment

A managed environment defines an operational environment for a Java EE-based, multi-tier, web-enabled application that accesses EISs. The application consists of one or more application components—EJBs, JSPs, servlets—which are deployed on containers. These containers can be one of the following:

- Web containers that host JSPs, servlets, and static HTML pages
- EJB containers that host EJB components
- Application client containers that host standalone application clients

2.1.6　Non-Managed Environment

A non-managed environment defines an operational environment for a two-tier application. An application client directly uses a resource adapter to access the EIS, which defines the second tier of a two-tier application.

2.1.7　Connection

A connection provides connectivity to a resource manager. It enables an application client to connect to a resource manager, perform transactions, and access services provided by that resource manager. A connection can be either transactional or non-transactional. Examples include a database connection and an SAP R/3 connection. A connection to a resource manager may be used by a client for bi-directional communication, depending on the capabilities of the resource manager.

2.1.8　Application Component

An application component can be a server-side component, such as an EJB, JSP, or servlet, that is deployed, managed, and executed on an application server. It can also be a component executed on the web-client tier but made available to the web-client by an application server. Examples of the latter type of application component include a Java applet, and a DHTML page.

2.1.9　Container

A container is a part of an application server that provides deployment and runtime support for application components. It provides a federated view of the services provided by the underlying application server for the application components. For

more details on different types of standard containers, refer to the EJB (see "Enterprise JavaBeans (EJB) Specification, version 3.2" on page E-1), JSP, and servlet specifications.

2.2 Rationale

This section describes the rationale behind the connector architecture.

2.2.1 System Contracts

A standard architecture is needed to integrate various EISs with an application server. Without a standard, EIS vendors and application server vendors may have to use vendor-specific architectures to provide EIS integration.

The connector architecture provides a Java solution to the problem of bi-directional connectivity between the multitude of application servers and EISs. By using the connector architecture, it is no longer necessary for EIS vendors to customize their product for each application server. An application server vendor who conforms to the connector architecture also does not need to add custom code whenever it wants to extend its application server to support connectivity to a new EIS.

The connector architecture enables an EIS vendor to provide a standard resource adapter for its EIS. The resource adapter plugs into an application server and provides the underlying infrastructure for the integration between an EIS and the application server.

An application server vendor extends its system only once to support the connector architecture and is then assured of connectivity to multiple EISs. Likewise, an EIS vendor provides one standard resource adapter and it has the capability to plug in to any application server that supports the connector architecture.

The following figure shows that a standard EIS resource adapter can plug into multiple application servers. Similarly, multiple resource adapters for different EISs can plug into an application server. This system-level pluggability is made possible through the connector architecture.

If there are *m* application servers and *n* EISs, the connector architecture reduces the scope of the integration problem from an *m* x *n* problem to an *m* + *n* problem.

FIGURE 2-1   System Level Pluggability Between Application Servers and EISs

Application Server   Resource Adapters
                                     Enterprise Information
                                     Systems

Application Servers   Resource Adapter

Application Servers   Enterprise Information
                                     System

| | Application server extension for resource adapter pluggability |
| | Standard resource adapter |
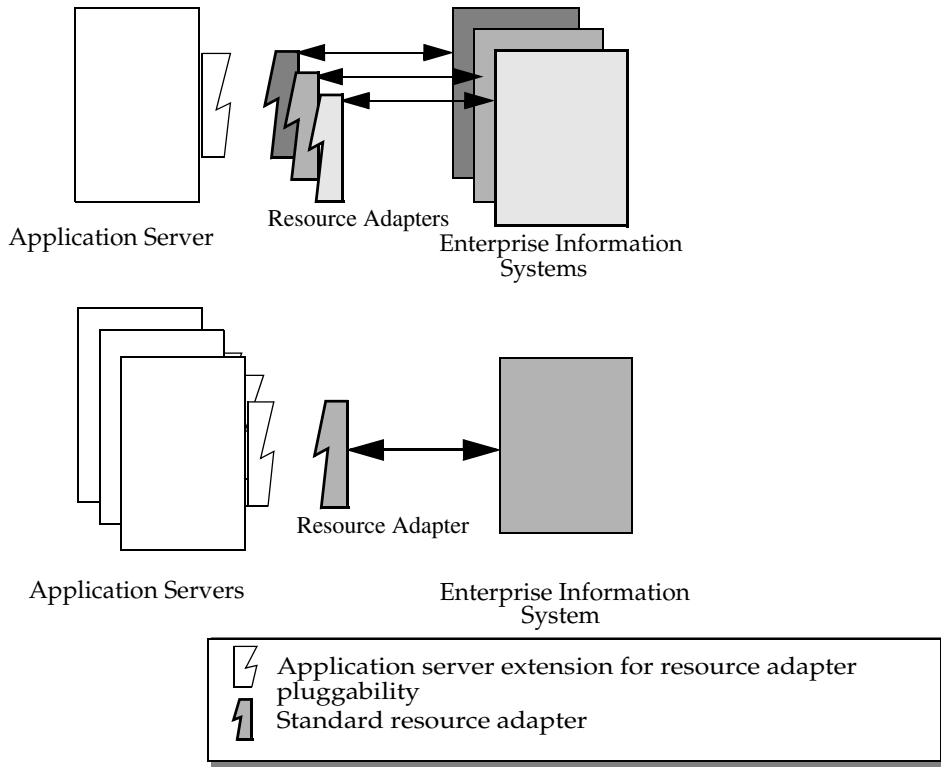
2.2.2   Common Client Interface

An enterprise tools vendor provides tools that lead to a simple application programming model for EIS access, thereby reducing the effort required in EIS integration. An EAI vendor provides a framework that supports integration across multiple EISs. Both types of vendors need to integrate across heterogeneous EISs.

Each EIS typically has a client API that is specific to the EIS. Examples of EIS client APIs are RFC for SAP R/3 and ECI for CICS.

An enterprise tools vendor adapts different client APIs for target EISs to a common client API. The adapted API is typically specific to a tools vendor and supports an application programming model common across all EISs. Adapting the API requires significant effort on the part of a tools vendor. In this case, the *m* x *n* integration problem applies to tools vendors.

The connector architecture provides a solution for the m x n integration problem for tools and EAI vendors. The architecture specifies a standard Common Client Interface (CCI) that supports a common client API across heterogeneous EISs.

All EIS resource adapters that support CCI are capable of being plugged into enterprise tools and EAI frameworks in a standard way. A tools vendor need not do any API adoption; the vendor can focus on providing its added value of simplifying EIS integration.

The CCI drastically reduces the effort and learning requirements for tools vendor by narrowing the scope of an m x n problem to an $m + n$ problem if there are m tools and n EISs.

2.3 Goals

The connector architecture has been designed with the following goals:

- Simplify the development of scalable, secure, and transactional resource adapters for a wide range of EISs—ERP systems, database systems, mainframe-based transaction processing systems.
- Be sufficiently general to cover a wide range of heterogeneous EISs. The sufficient generality of the architecture ensures that there are various implementation choices for different resource adapters; each choice is based on the characteristics and mechanisms of an underlying EIS.
- Be not tied to a specific application server implementation, but applicable to all Java EE platform compliant application servers from multiple vendors.
- Provide a standard client API for enterprise tools and EAI vendors. The standard API will be common across heterogeneous EISs.
- Express itself in a manner that allows an organization to unambiguously determine whether or not an implementation is compatible.
- Be simple to understand and easy to follow, regardless of whether one is designing a resource adapter for a particular EIS or developing/deploying application components that need to access multiple EISs. This simplicity means the architecture introduces only a few new concepts, and places minimal implementation requirements so that it can be leveraged across different integration scenarios and environments.
- Define contracts and responsibilities for various roles that provide pieces for standard bi-directional connectivity to an EIS. This enables a standard resource adapter from a EIS vendor to be pluggable across multiple application servers.

- Enable an enterprise application programmer in a non-managed application environment to directly use the resource adapter to access the underlying EIS. This is in addition to managed access to an EIS, with the resource adapter deployed in the middle-tier application server.

The Connector Architecture

This chapter gives an overview of the connector architecture.

Multiple resource adapters—that is, one resource adapter per type of EIS—are pluggable into an application server. This capability enables application components deployed on the application server to access the underlying EISs.

An application server and an EIS collaborate to keep all system-level mechanisms—transactions, security, and connection management—transparent from the application components. As a result, an application component provider focuses on the development of business and presentation logic for its application components and need not get involved in the system-level issues related to EIS integration. This leads to an easier and faster cycle for the development of scalable, secure, and transactional enterprise applications that require connectivity with multiple EISs.

FIGURE 3-1   Overview of the Connector Architecture
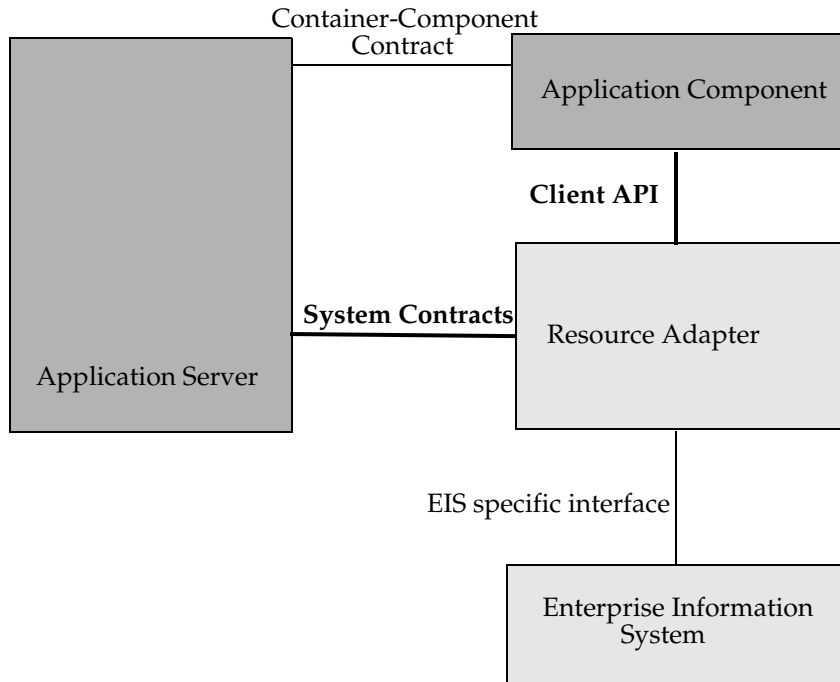
3.1   System Contracts

To achieve a standard system-level pluggability between application servers and EISs, the connector architecture defines a standard set of system-level contracts between an application server and an EIS. The EIS side of these system-level contracts are implemented in a resource adapter.

A resource adapter is specific to an underlying EIS. It is a system-level software driver that is used by an application server or an application component to connect to an EIS.

A resource adapter plugs into an application server. The resource adapter and application server collaborate to provide the underlying mechanisms—transactions, security, connection pooling, and dispatch to application components.

A resource adapter is used within the address space of the application server. Examples of resource adapters are:

- A JDBC driver to connect to a relational database, as specified in the JDBC specification. For more information on JDBC, see "JDBC API Specification, version 4.1" on page E-1

- A resource adapter to connect to an ERP system

- A resource adapter to connect to a TP system

- A resource adapter to plug-in a messaging system

A resource adapter may provide different types of connectivity between an application and an EIS.

- **Outbound communication.** The resource adapter allows an application to connect to an EIS system and perform work. All communication is initiated by the application. In this case, the resource adapter serves as a passive library for connecting to an EIS, and executes in the context of the application threads.

- **Inbound communication.** The resource adapter allows an EIS to call application components and perform work. All communication is initiated by the EIS. The resource adapter may request threads from the application server or create its own threads.

- **Bi-directional communication.** The resource adapter supports both outbound and inbound communication.

The connector architecture defines the following set of standard contracts between an application server and EIS:

- A connection management contract that enables an application server to pool connections to an underlying EIS, and enables application components to connect to an EIS. This leads to a scalable application environment that can support a large number of clients requiring access to EISs.

- A transaction management contract between the transaction manager and an EIS that supports transactional access to EIS resource managers. This contract enables an application server to 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 that reduces security threats to the EIS and protects valuable information resources managed by the EIS.

- A lifecycle management contract that allows an application server to manage the lifecycle of a resource adapter. This contract provides a mechanism for the application server to bootstrap a resource adapter instance during its deployment or application server startup, and to notify the resource adapter instance during its undeployment or during an orderly shutdown of the application server.

- A work management contract that allows a resource adapter to do work (monitor network endpoints, call application components, etc.) by submitting `Work` instances to an application server for execution. The application server dispatches

threads to execute submitted `Work` instances. This allows a resource adapter to avoid creating or managing threads directly, and allows an application server to efficiently pool threads and have more control over its runtime environment. The resource adapter can control the security context and transaction context with which `Work` instances are executed.

- A generic work context contract that enables a resource adapter to control the execution context of a `Work` instance that it has submitted to the application server for execution. The Generic Work Context Contract provides the mechanism for a resource adapter to augment the runtime context of a `Work` instance with additional contextual information flown-in from the EIS. This contract enables a resource adapter to control, in a more flexible manner, the contexts in which the `Work` instances submitted by it are executed by the application server's `WorkManager`.

- A transaction inflow contract that allows a resource adapter to propagate an imported transaction to an application server. This contract also allows a resource adapter to transmit transaction completion and crash recovery calls initiated by an EIS, and ensures that the ACID properties of the imported transaction are preserved.

- A security work context that enables a resource adapter to establish security information while submiting a `Work` instance for execution to a `WorkManager` and while delivering messages to message endpoints residing in the application server. This contract provides a mechanism to support the execution of a `Work` instance in the context of an established identity. It also supports the propagation of user information/Principal information from an EIS to a MessageEndpoint during Message Inflow.

- A message inflow contract that allows a resource adapter to asynchronously deliver messages to message endpoints residing in the application server 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 wide range of message providers (Java Message Service (JMS), Java API for XML Messaging (JAXM), etc.) to be plugged into any Java EE compatible application server by way of a resource adapter.

FIGURE 3-1 does not illustrate any contracts that are internal to an application server implementation. The specific mechanisms and contracts within an application server are outside the scope of the connector architecture specification. This specification focuses on the system-level contracts between the application server and the EIS.

In FIGURE 3-1, the application server, application component and resource adapter are shown as separate entities. This is done to illustrate that there is a logical separation of the respective roles and responsibilities defined for the support of the system level contracts. However, this separation does not imply a physical separation, as in an application server, application component and a resource adapter running in separate processes.

3.2   Client API

The client API used by application components for EIS access may be defined as:

- The standard Common Client Interface (CCI) as specified in Chapter 17, "Common Client Interface".
- A client API specific to the type of a resource adapter and its underlying EIS. An example of such an EIS specific client API is JDBC for relational databases.

The Common Client Interface (CCI) defines a common client API for accessing EISs. The CCI is targeted towards Enterprise Application Integration (EAI) and enterprise tools vendors.

3.3   Requirements

The connector architecture requires that the connector architecture-compliant resource adapter and the application server support the system contracts. Detailed requirements for each system contract are specified in later chapters.

The connector architecture recommends, though it does not mandate, that a resource adapter support CCI as the client API. The recommendation enables the connector architecture to provide a solution for the m x n integration problem for application development tools and EAI vendors.

The connector architecture allows a resource adapter with an EIS-specific client API to support system contracts and to be capable of standard connector architecture-based pluggability into an application server.

3.4   Non-Managed Environment

The connector architecture supports access to EISs from non-managed application clients; for example, Java applications and applets.

In a non-managed two-tier application environment, an application client directly uses a resource adapter library. A resource adapter, in this case, exposes its low-level transactions and security APIs to its clients. An application client has to take responsibility for managing security and transactions (and rely on connection

pooling if done by the resource adapter internally) by using the low-level APIs exposed by the resource adapter. This model is similar to the way a two-tier JDBC application client accesses a database system in a non-managed environment.

3.5 Standalone Container Environment

Server Providers can provide a Connector container within a product that implements the Java EE Full Profile or within a subset profile such as the Java EE Web Profile. The complete set of application server requirements in this specification is required for a compliant Java EE Connector Architecture 1.6 container within an implementation of the Java EE Full Profile. The minimum set, listed below, must be supported for a compliant Java EE Connector Architecture 1.6 container within an implementation of any subset of the Java EE Full Profile. Overall profile requirements are described within the "Java Platform, Enterprise Edition (Java EE) Specification, version 7" on page E-1.

Non-"Full Profile" implementations may only support a subset of the component specifications that were mandated to be present in a full Java EE platform product implementation. An implementation of the Connector specification bundled in such a managed environment is described as standalone connector container below.

Based on the availability of other dependent component specification implementations, the following requirements must be satisfied by a standalone connector container.

- If a `MessageEndpointFactory` implementation (such as support for message-driven beans) is available, the Message Inflow requirements specified in Chapter 13, "Message Inflow" must be satisfied by it.
- If an implementation of the Bean Validation specification is provided, the requirements in Section 5.3.7.5, "JavaBean Validation" on page 5-15 must be supported.

An existing resource adapter archive RAR may not be fully functional in a standalone implementation, though. For example a bi-directional resource adapter archive deployed on a standalone implementation that does not support Message Inflow would not have the corresponding Message Inflow support (`endpointActivation`) provided to the resource adapter.

A standalone connector container implementation that does not support one of the dependent component specification implementations listed above must not fail the deployment of a resource adapter that uses the capabilities in the unsupported specifications. For instance, if a bi-directional resource adapter is deployed to a standalone connector container that does not support Message Inflow, the container will not be able to make calls to the `endpointActivation` method in the `ResourceAdapter` JavaBean because the implementation does not support Message

Inflow (and therefore `MessageEndpoint` deployment). However, the container must support the deployment of a bi-directional resource adapter and support other capabilities of the resource adapter that do not rely on support for Message Inflow (outbound communication, use of the WorkManager etc.).

The standalone connector container must support the baseline compatibility requirements as defined by the Java Authentication Service Provider Interface for Containers (JASPIC) specification and support the Security Inflow requirements specified in Chapter 16, "Security Inflow". See "Java Authentication Service Provider Interface for Containers Specification, version 1.4" on page E-2 for more information on the JASPIC specification.

This specification does not define new application components or require any particular existing application component to be supported in the standalone connector container environment.

Roles and Scenarios

This chapter describes a set of roles specific to the connector architecture. The goal of this chapter is to specify contracts that ensure that the output of each role is compatible with the input of the other role. Later chapters specify a detailed set of responsibilities for each role, relative to the system-level contracts.

4.1 Roles

This section describes the roles and responsibilities specific to the connector architecture.

4.1.1 Resource Adapter Provider

The resource adapter provider is an expert in the technology related to an EIS and is responsible for providing a resource adapter for an EIS. Since this role is highly EIS specific, an EIS vendor typically provides the resource adapter for its system.

A third-party vendor (who is not an EIS vendor) may also provide an EIS resource adapter and its associated set of application development tools. Such a provider typically specializes in writing resource adapters and related tools for a large number of EISs.

4.1.2   Application Server Vendor

The application server vendor provides an implementation of a Java EE-compliant application server that provides support for component based enterprise applications. A typical application server vendor is an OS vendor, middleware vendor, or database vendor. The role of an application server vendor is typically the same as that of a container provider.

The Java EE platform specification (see "Java Platform, Enterprise Edition (Java EE) Specification, version 7" on page E-1) specifies requirements for a Java EE platform provider.

4.1.3   Container Provider

The container provider is responsible for providing a container implementation for a specific type of application component. For example, the container provider may provide a container for EJB components. Each type of application component—EJB, servlet, JSP, applet—has its own set of responsibilities for its container provider. The respective specifications outline these responsibilities.

A container implementation typically provides the following functionality:

- It provides deployed application components with transaction and security management, distribution of clients, scalable management of resources, and other services that are generally required as part of a managed server platform.
- It provides application components with connectivity to an EIS by transparently managing security, resources, and transactions using the system-level contracts with the EIS-specific resource adapter.
- It insulates application components from the specifics of the underlying system-level mechanisms by supporting a simple, standard contract with the application component. Refer to the Enterprise JavaBeans specification (see "Enterprise JavaBeans (EJB) Specification, version 3.2" on page E-1) for more details on the EJB component contract.

The expertise of the container provider is system-level programming, with its focus on the development of a scalable, secure, and transaction-enabled container.

The container provider is also responsible for providing deployment tools necessary for the deployment of application components and resource adapters. It is also required to provide runtime support for the deployed application components.

The container provider typically provides tools that allow the system administrator to monitor and manage a container and application components during runtime.

4.1.4   Application Component Provider

In the context of the connector architecture, the application component provider produces an application component that accesses one or more EISs to provide its application functionality.

The application component provider is an application domain expert. In the case of application components targeted towards integration with multiple EISs, various business tasks and entities are implemented based on access to EIS data and functions.

The application component provider typically programs against easy-to-use Java abstractions produced by application development tools. These Java abstractions are based on the Common Client interface (CCI).

The application component provider is not required to be an expert at system level programming. The application component provider does not program transactions, security, concurrency, or distribution, but relies on a container to provide these services transparently.

The application component provider is responsible for specifying structural information for an application component and its external dependencies. This information includes, for example, the name and type of the connection factories, and security information.

The output of an application component provider is a Java^{TM} Archive (JAR) file that contains the application components and any additional Java classes required to connect to EISs.

4.1.5   Enterprise Tools Vendors

The application component provider relies on tools to simplify application development and EIS integration. Since programming client access to EIS data and functions is a complex application development task, an application development tool reduces the effort and complexity involved in this task.

Enterprise tools serve different roles in the application development process, as follows:

- **Data and function mining tool** - enables application component providers to look at the scope and structure of data and functions existing in an EIS
- **Analysis and design tool** - enables application component providers to design an application in terms of EIS data and functions
- **Code generation tool**- generates Java classes for accessing EIS data and functions. A mapping tool that bridges across two different programming models (object to relational or vice-versa) falls into this category of tools.

- **Application composition tool** - enables application component providers to compose application components from Java classes generated by a code generation tool. This type of tool typically uses the JavaBeans^{TM} component model to enhance the ease of programming and composition.
- **Deployment tool** - used by application component providers and deployers to set transaction, security, and other deployment time requirements.

A number of these tools may be integrated together to form an end-to-end application development environment.

In addition, various tools and middleware vendors offer EAI frameworks that simplify integration across heterogeneous EISs.

4.1.6   Application Assembler

The application assembler combines various application components into a larger set of deployable units. The input of the application assembler is one or more JAR files produced by an application component provider and the output is one or more JAR files with a deployment descriptor. A deployment descriptor may not be provided by the application assembler if metadata annotations (see Chapter 18, "Metadata Annotations") are used to describe deployment information.

The application assembler is typically a domain expert who assembles application components to produce an enterprise application. To achieve this goal, the application assembler takes application components, possibly from multiple application component providers, and assembles these components.

4.1.7   Deployer

The deployer takes one or more deployable units of application components, produced by the application assembler or component provider, and deploys the application components in a target operational environment. An operational environment is comprised of an application server and multiple connected EISs.

The deployer is responsible for resolving all external dependencies declared by the application component provider. For example, the deployer ensures that all connection factories used by the application components are present in an operational environment. To perform its role, the deployer typically uses the application server-provided deployment tools.

The deployer is also responsible for the deployment of resource adapters. Since an operational environment may include multiple EISs, the role of the deployer is more intensive and complex than that in a non-EIS scenario. The deployer has to understand security, transaction, and connection management-related aspects of multiple EISs that are configured in an operational environment.

4.1.8 System Administrator

The system administrator is responsible for the configuration and administration of a complete enterprise infrastructure that includes multiple containers and EISs.

In an operational environment that has multiple EISs, the deployer should manage the operational environment by working closely with the system administrators of respective EISs. This enables the deployer to resolve deployment issues while deploying application components and resource adapters in a target operational environment.

This chapter introduced the roles involved in the connector architecture. The later chapters specify responsibilities for each role in more detail.

4.2 Scenario: Integrated Purchase Order System

This section describes a scenario that illustrates the use of the connector architecture. The following description is kept at a high level. Specific scenarios related to transaction management, security, connection management, and inbound communications are described in subsequent chapters.

The following diagram shows the different pieces that comprise this scenario:

FIGURE 4-1   Illustration of a Scenario Based on the Connector Architecture
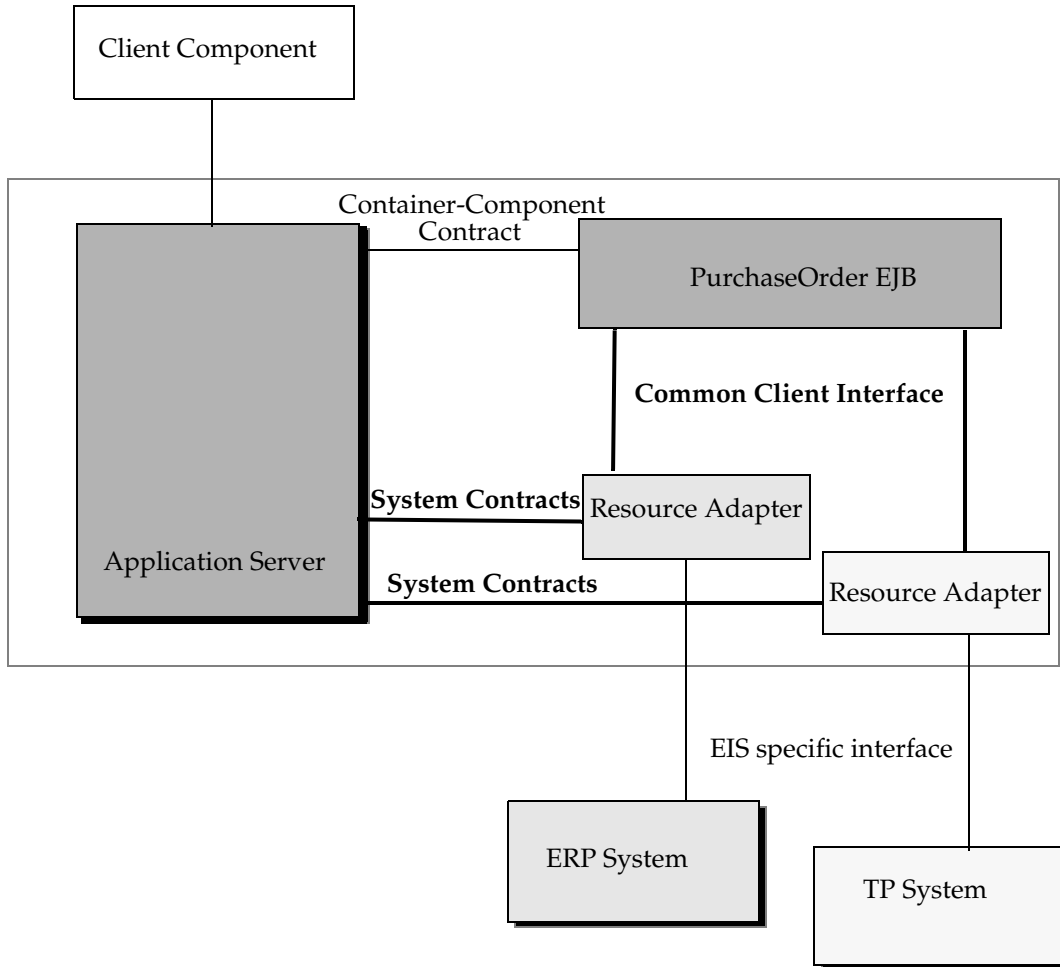
ERP Software Inc. is an enterprise system vendor that provides an enterprise resource planning (ERP) system. ERP Software wants to integrate its ERP system with various application servers. It achieves this goal by providing a standard resource adapter for its ERP system. The resource adapter for ERP systems supports the standard inbound communication, transaction, connection management and security contracts. The resource adapter also supports the Common Client Interface (CCI) as its client API.

TPSoft Inc. is another enterprise system vendor that provides a transaction processing (TP) system. TPSoft has also developed a standard resource adapter for its TP system. The resource adapter library supports CCI as part of its implementation.

AppServer Inc. is a system vendor that has an application server product which supports the development and deployment of component-based enterprise applications. This application server product has an EJB container that provides deployment and runtime support for EJB components. The application server supports the system-level contracts that enable a resource adapter, which also supports these contracts, to plug into the application server and provide bi-directional connectivity to the underlying EIS. The EJB container insulates EJB components from the communication, transaction, security, and connection management mechanisms required for connecting to the EIS.

Manufacturer Corp. is a big manufacturing firm that uses a purchase order processing system based on the ERP system for its business processes. Recently, Manufacturer has acquired a firm that uses TPSoft's TP system for its purchase order processing. Manufacturer aims to integrate these two systems together into a single integrated purchase order system. It requires a scalable, multi-user, secure, transaction-enabled integrated purchase order system that is not tied to a specific computing platform. Manufacturer plans to deploy the middle-tier of this system on the application server from AppServer Inc.

The MIS department of Manufacturer develops a PurchaseOrder EJB that provides an integrated view of the two underlying purchase order systems. While developing PurchaseOrder EJB, the bean provider does not program the transactions, security, connection management or inbound communication mechanisms required for connectivity to the ERP and TP systems; it relies on the EJB container and application server to provide these services.

The bean provider uses an application programming model based on the CCI to access the business objects and function modules for purchase order processing in the ERP system. The bean provider uses a similar application programming model based on the CCI to access the purchase order processing programs in the TP system.

The MIS department of Manufacturer assembles an integrated web-based purchase order application using PurchaseOrder EJB with other types of application components, such as JSPs and servlets.

The MIS department installs and configures the application server, ERP, and TP system as part of its operational environment. It then deploys the integrated purchase order application on this operational environment. As part of the deployment, the MIS department configures the operational environment based on the deployment requirements for the various application components that have been assembled into the integrated enterprise application.

After deploying and successfully testing the integrated purchase order system, the MIS department makes the system available for other departments to use.

4.3 Scenario: Business Integration

This scenario illustrates the use of the connector architecture in a business integration scenario.

Wombat Systems is a manufacturing firm that aims to adopt an e-business strategy. Wombat has huge existing investments in its EIS systems. The EISs include ERP systems, mainframe transaction processing systems, and message providers.

Wombat requires to interact with its various partners. In order to do this, it requires support for different interaction mechanisms. It also requires a mechanism to involve all its EIS systems in the interaction. Further, it requires an application sever to host its business applications which participate in the various interactions.

Wombat buys a Java EE based application server from EComm, Inc. to host its business applications which interact with its EISs and its various partners. The application server supports the connector architecture contracts which make it possible to use appropriate resource adapters to drive interactions with its partners and its EISs.

The connector architecture enables Wombat to integrate its existing infrastructure with the application server. Wombat buys off-the-shelf resource adapters for its existing set of EISs and to support interactions with its partners and uses them to integrate its business applications (deployed on the application server).

FIGURE 4-2 Connector Architecture Usage in Business Integration Scenario
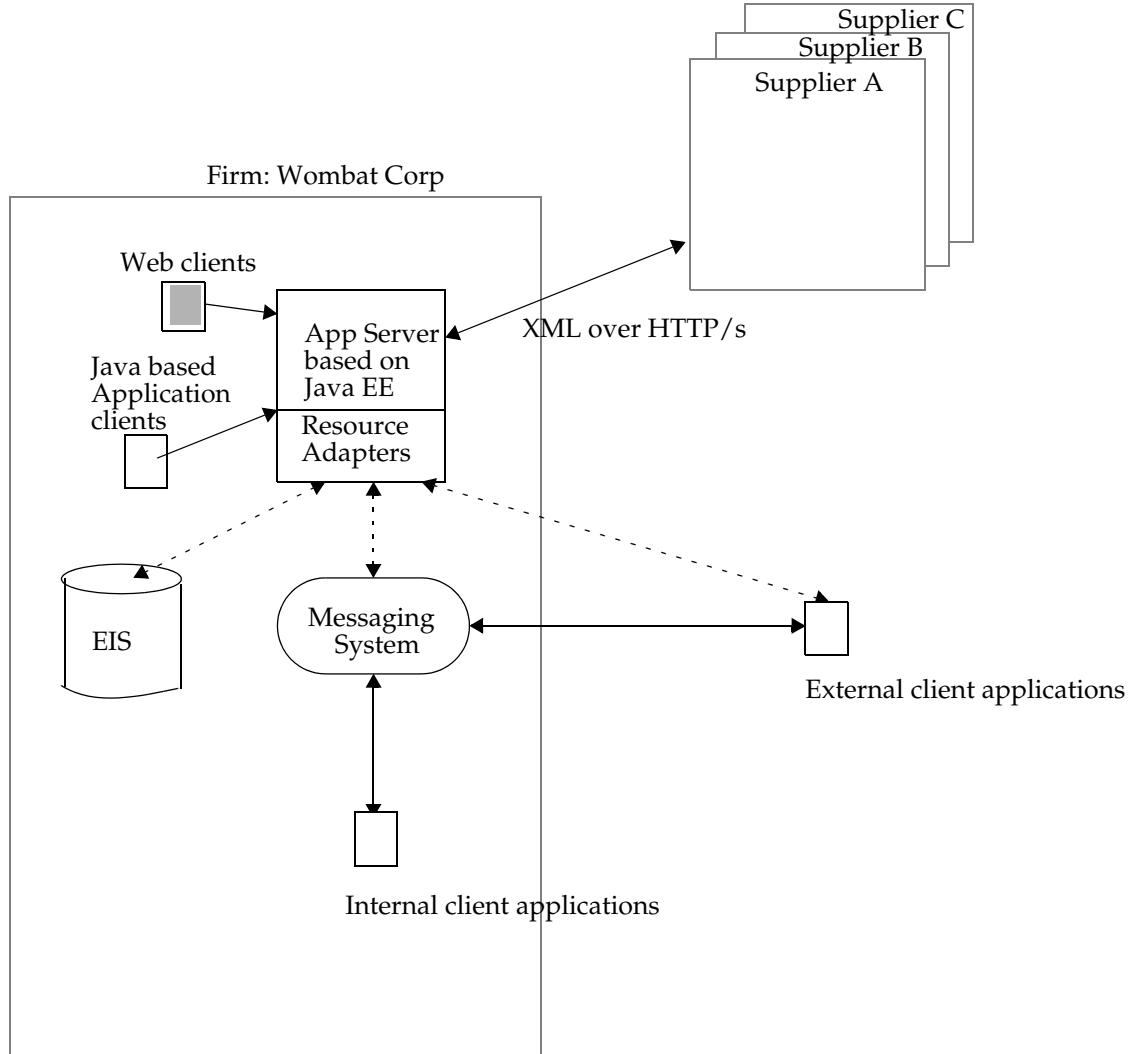
Lifecycle Management

This chapter specifies a contract between an application server and a resource adapter that allows an application server to manage the lifecycle of a resource adapter. This contract provides a mechanism for the application server to bootstrap a resource adapter instance during its deployment or application server startup, and to notify the resource adapter instance during its undeployment or during an orderly shutdown of the application server.

5.1 Overview

A resource adapter is a system component which is deployed in an application server. When a resource adapter is deployed, or during application server startup, an application server requires to bootstrap an instance of the resource adapter in its address space. When a resource adapter is undeployed, or during application server shutdown, the application server requires a mechanism to notify the resource adapter instance to stop functioning so that it can be safely unloaded.

The lifecycle management contract provides such a mechanism for an application server to manage the lifecycle of a resource adapter instance. This allows an application server to bootstrap a resource adapter instance during resource adapter deployment or application server startup and also to expose some of its useful facilities to the resource adapter instance. It also provides a mechanism to notify the resource adapter instance while it is undeployed or during an orderly shutdown of the application server.

5.2 Goals

- Provide a mechanism for an application server to manage the lifecycle of a resource adapter instance.

5.3 Lifecycle Management Model
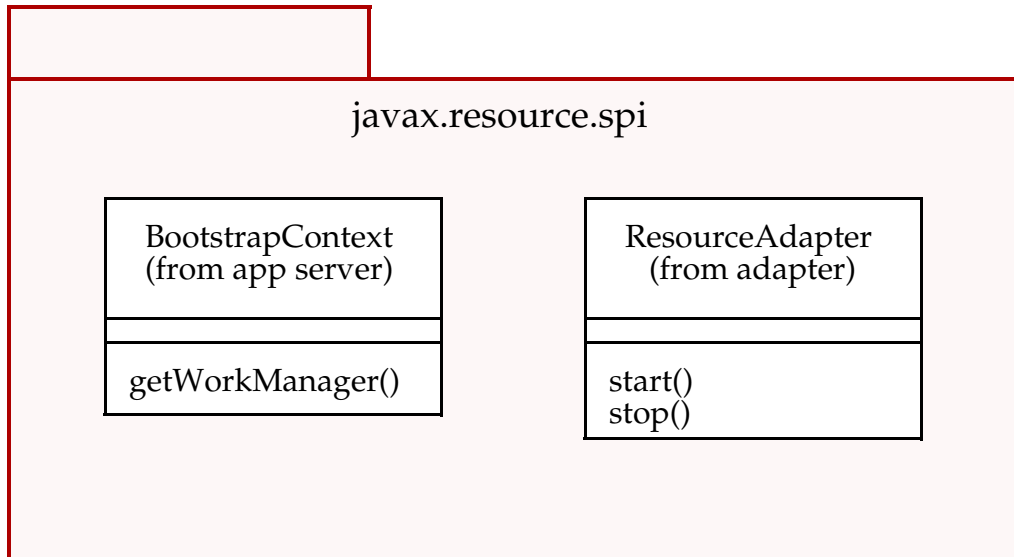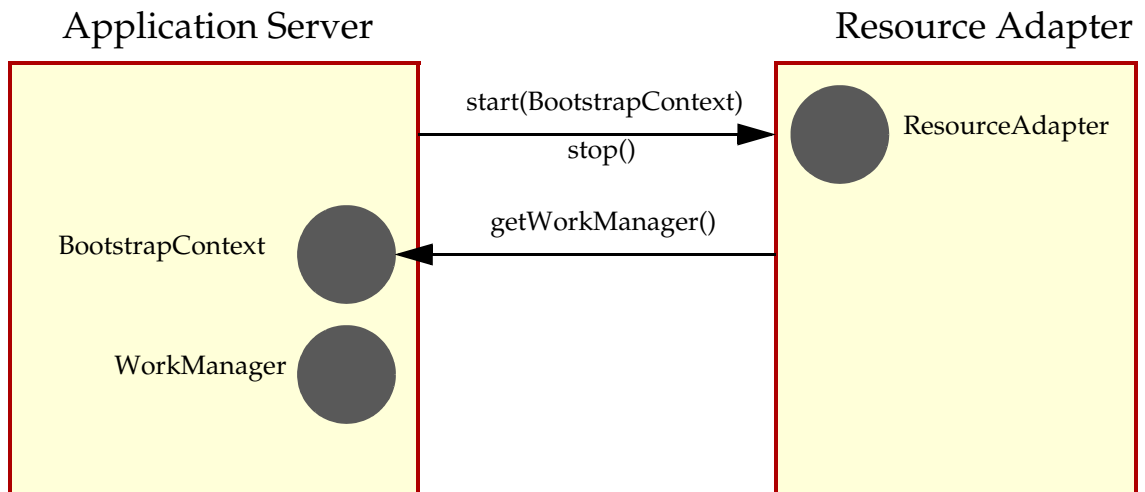
```
package javax.resource.spi;

import javax.resource.spi.work.WorkManager;

public interface ResourceAdapter {
      void start(BootstrapContext) // startup notification
          throws ResourceAdapterInternalException;
      void stop(); // shutdown notification
      ... // other operations
}

public interface BootstrapContext {
      WorkManager getWorkManager();
      ... // other operations
}
```

An application server implements the `BootstrapContext` and `WorkManager` interfaces. A resource adapter implements the `ResourceAdapter` interface.

5.3.1 `ResourceAdapter` JavaBean and Bootstrapping a Resource Adapter Instance

The implementation class name of the `ResourceAdapter` interface is specified in the resource adapter deployment descriptor or through the `Connector` annotation described in Section 18.4, "@Connector" on page 18-6. The `ResourceAdapter` class must be a JavaBean. Refer to Section 19.3, "JavaBean Requirements" on page 19-2. During resource adapter deployment, the resource adapter deployer creates a `ResourceAdapter` JavaBean and configures it with the appropriate properties.

When a resource adapter is deployed, or during application server startup, an application server bootstraps an instance of the resource adapter in its address space. In order to bootstrap a resource adapter instance, the application server must use the configured `ResourceAdapter` JavaBean and call its `start` method. The `start` method call is a startup notification from the application server, and this method is called by an application server thread.

During the `start` method call the `ResourceAdapter` JavaBean is responsible for initializing the resource adapter instance. This may involve creating resource adapter instance specific objects, creating threads (refer to Chapter 10, "Work Management"), and setting up network endpoints. A `ResourceAdapter` JavaBean represents exactly one functional resource adapter unit or instance. The application server must instantiate exactly one `ResourceAdapter` JavaBean per functional resource adapter

instance. The application server must create at least one functional resource adapter instance per resource adapter deployment. An application server may create more than one functional resource adapter instance per resource adapter deployment, in order to create replicas of a single functional resource adapter instance on multiple Java^{TM} Virtual Machines[1]. In general, however, there should be just one functional resource adapter instance per deployment.

The application server is allowed to have multiple instances of a `ResourceAdapter` JavaBean active simultaneously, in the same JVM^{TM} instance, provided the instances are not equal. Their equality is determined using the `equals` method, and therefore, the `ResourceAdapter` JavaBean is required to implement the `equals` method.

During the `start` method call, an application server must provide a `BootstrapContext` instance containing references to some of the application server facilities (for example, `WorkManager`) for use by the resource adapter instance. The application server facilities exposed through the `BootstrapContext` instance may be used by the resource adapter instance during its lifetime.

During the `start` method call, the resource adapter instance initializes itself, and may use the `WorkManager` to submit `Work` instances for execution (see Chapter 10, "Work Management"). The `start` method call should return in a timely manner, and should avoid blocking calls, such as use of `doWork` method call on the `WorkManager` instance. The application server may throw a `WorkRejectedException` in response to any or all `doWork` method calls on the `WorkManager` instance, in order to enforce that a `start` method call does not block. Resource adapter implementations are strongly recommended to use `startWork` and `scheduleWork` methods on the `WorkManager`, instead of the `doWork` method.

Any exception thrown during the `start` method call indicates an error condition, and the attempt by the application server to create a resource adapter instance fails. A future version of the specification may add a two-phase startup procedure.

A resource adapter instance at runtime may contain several objects that may be created and discarded during its lifetime. Such objects include `ManagedConnectionFactory` JavaBean (refer to Chapter 6, "Connection Management"), `ActivationSpec` JavaBean (refer to Chapter 13, "Message Inflow"), various connection objects, resource adapter private objects, and other resource adapter specific objects that are exposed to applications.

The `ResourceAdapter` JavaBean represents a resource adapter instance and contains the configuration information pertaining to that resource adapter instance. This configuration information may also be used as global defaults for `ManagedConnectionFactory` and `ActivationSpec` JavaBeans. That is, when `ManagedConnectionFactory` or `ActivationSpec` JavaBeans are created they may inherit the global defaults (`ResourceAdapter` JavaBean configuration information), which make it easier to configure them.

---

1. The terms "Java^{TM} Virtual Machine" and "JVM" mean a Virtual Machine for the Java^{TM} platform.

A resource adapter instance may provide bi-directional connectivity to multiple EIS instances. A `ManagedConnectionFactory` JavaBean can be used to provide outbound connectivity to a single EIS instance. An `ActivationSpec` JavaBean can be used to provide inbound connectivity from an EIS instance. A resource adapter instance may contain several such `ManagedConnectionFactory` and `ActivationSpec` JavaBeans. The following figure describes the association between a resource adapter instance and its various `ManagedConnectionFactory` and `ActivationSpec` JavaBeans.

FIGURE 5-3   Resource Adapter Instance (Composition)

Resource adapter instance
(within an application server)

ResourceAdapter
JavaBean (exactly one)

Outbound communication

Application

ManagedConnectionFactory
JavaBean

EIS instance

Application

ManagedConnectionFactory
JavaBean

EIS instance

Inbound communication

Application

ActivationSpec
JavaBean

EIS instance

Application

ActivationSpec
JavaBean
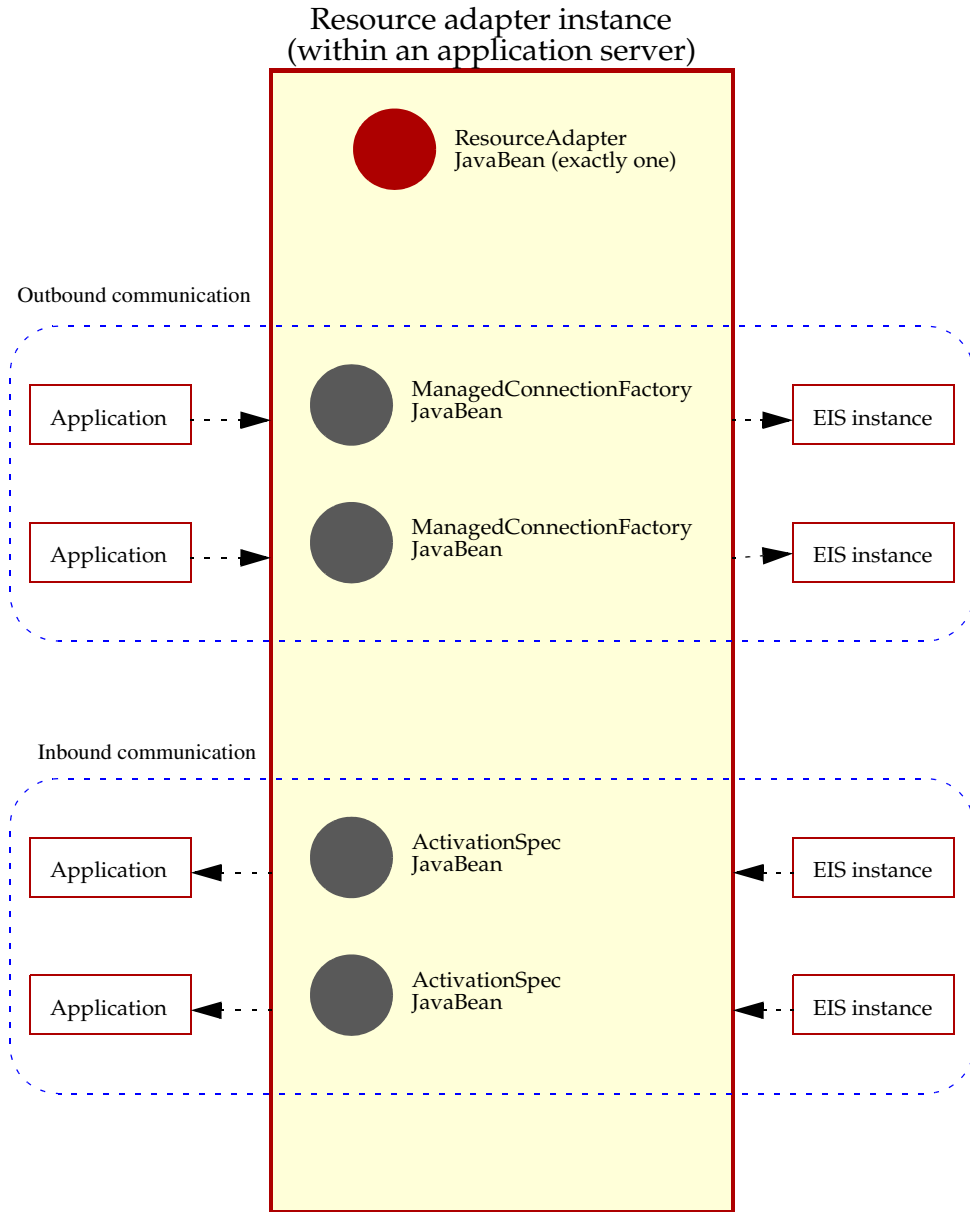
EIS instance

5.3.2 `ManagedConnectionFactory` JavaBean and Outbound Communication

A `ManagedConnectionFactory` JavaBean represents outbound connectivity information to an EIS instance from an application by way of a specific resource adapter instance. This contains the configuration information pertaining to outbound connectivity to an EIS instance. Refer to Chapter 6, "Connection Management" for more details on the `ManagedConnectionFactory` JavaBean.

When a `ManagedConnectionFactory` JavaBean is created, it may inherit the `ResourceAdapter` JavaBean (which represents the resource adapter instance) configuration information, and overrides specific global defaults, if any, and may add other configuration information specific to outbound connectivity.

That is, in the case of outbound communication, the outbound connectivity configuration is a union of `ResourceAdapter` JavaBean and `ManagedConnectionFactory` JavaBean configuration, with the intersecting configuration properties based on the `ManagedConnectionFactory` JavaBean settings.

Outbound communication is initiated by an application and the communication occurs in the context of an application thread, even though resource adapter threads may be involved in the interaction. Note, a resource adapter may use the work management contract (refer to Chapter 10, "Work Management") to request threads to do work.

```
import javax.resource.spi.ResourceAdapterAssociation;
import javax.resource.spi.ManagedConnectionFactory;

public class ManagedConnectionFactoryImpl
        implements ManagedConnectionFactory,
ResourceAdapterAssociation {

    ResourceAdapter getResourceAdapter();
    void setResourceAdapter(ResourceAdapter) throws
ResourceException;
    ... // other methods
}
```

The `ResourceAdapterAssociation` interface specifies the methods to associate a `ManagedConnectionFactory` JavaBean with a `ResourceAdapter` JavaBean.

Prior to using a `ManagedConnectionFactory` JavaBean, the application server must create an association between the `ManagedConnectionFactory` JavaBean and a `ResourceAdapter` JavaBean, by calling the `setResourceAdapter` method on

the `ManagedConnectionFactory` JavaBean. A successful association is established only when the `setResourceAdapter` method on the `ManagedConnectionFactory` JavaBean returns without throwing an exception.

The `setResourceAdapter` method on the `ManagedConnectionFactory` JavaBean must be called exactly once; that is, the association must not change during the lifetime of a `ManagedConnectionFactory` JavaBean.

5.3.3   `ActivationSpec` JavaBean and Inbound Communication

An `ActivationSpec` JavaBean represents inbound connectivity information from an EIS instance to an application by way of a specific resource adapter instance. This contains the configuration information pertaining to inbound connectivity from an EIS instance. Refer to Chapter 13, "Message Inflow" for more details on the `ActivationSpec` JavaBean.

When an `ActivationSpec` JavaBean is created, it may inherit the `ResourceAdapter` JavaBean (which represents the resource adapter instance) configuration information, and overrides specific global defaults, if any, and may add other configuration information specific to inbound connectivity.

That is, in the case of inbound communication, the inbound connectivity configuration is a union of `ResourceAdapter` JavaBean and `ActivationSpec` JavaBean configuration, with the intersecting configuration properties based on the `ActivationSpec` JavaBean settings.

Inbound communication is initiated by an EIS instance and the communication occurs in the context of a resource adapter thread. There are no application threads involved. Note, a resource adapter may use the work management contract (refer to Chapter 10, "Work Management") to request threads to do work.

```
import javax.resource.spi.ActivationSpec;

// ActivationSpec interface extends ResourceAdapterAssociation
interface.

public class ActivationSpecImpl implements ActivationSpec {
    ResourceAdapter getResourceAdapter();
    void setResourceAdapter(ResourceAdapter) throws
ResourceException;
    ... // other methods
}
```

The `ResourceAdapterAssociation` interface specifies the methods to associate an `ActivationSpec` JavaBean with a `ResourceAdapter` JavaBean.

Prior to using an `ActivationSpec` JavaBean, the application server must create an association between the `ActivationSpec` JavaBean and a `ResourceAdapter` JavaBean, by calling the `setResourceAdapter` method on the `ActivationSpec` JavaBean. A successful association is established only when the `setResourceAdapter` method on the `ActivationSpec` JavaBean returns without throwing an exception.

The `setResourceAdapter` method on the `ActivationSpec` JavaBean must be called exactly once; that is, the association must not change during the lifetime of an `ActivationSpec` JavaBean.

5.3.4 Resource Adapter Shutdown Procedure

The following are some likely situations during which an application server would shutdown a resource adapter instance:

- The application server is being shutdown.
- The resource adapter is being undeployed.

Irrespective of what causes a resource adapter instance to be shutdown, the application server must use the following two phases to shutdown a resource adapter instance.

5.3.4.1 Phase One

Before calling the `stop` method on the `ResourceAdapter` JavaBean, the application server must ensure that all dependant applications using the specific resource adapter instance are stopped. This includes deactivating all message endpoints receiving messages by way of the specific resource adapter. Note, however, since dependant applications typically cannot be stopped until they are undeployed, the application server may have to delay stopping the resource adapter instance, until all such dependant applications are undeployed.

Completion of phase one guarantees that application threads will not use the resource adapter instance, even though the resource adapter instance specific objects may still be in the memory heap. This ensures that all application activities including transactional activities are completed.

Thus, phase one ensures that even if a resource adapter instance does not properly shutdown during phase two, the resource adapter instance is practically unusable.

5.3.4.2 Phase Two

The application server calls the `stop` method on the `ResourceAdapter` JavaBean to notify the resource adapter instance to stop functioning so that it can be safely unloaded. This is a graceful shutdown notification from the application server, and this method is called by an application server thread.

The `ResourceAdapter` JavaBean is responsible for performing an orderly shutdown of the resource adapter instance during the `stop` method call. This may involve closing network endpoints, relinquishing threads, releasing all active `Work` instances, allowing resource adapter internal in-flight transactions to complete if they are already in the process of doing a commit, and flushing any cached data to the EIS.

The resource adapter instance is considered fully functional until the application server calls the `stop` method on the `ResourceAdapter` JavaBean.

Any unchecked exception thrown by the `stop` method call does not alter the processing of the application server shutdown or resource adapter undeployment that caused the `stop` method call. The application server may log the exception information for error reporting purposes.

Note, it is possible for a resource adapter instance to become non-functional during its lifetime even before the `stop` method is called, due to EIS failure or other reasons. In such cases, the resource adapter instance should throw exceptions to indicate the failure condition, when it is accessed by an application (during outbound communication) or the application server.

A future version of the specification may add a forced shutdown method in addition to the current graceful `stop` method.

5.3.5 Requirements

- The application server must use a new `ResourceAdapter` JavaBean for managing the lifecycle of each resource adapter instance and must discard the `ResourceAdapter` JavaBean after its `stop` method has been called. That is, the application server must not reuse the same `ResourceAdapter` JavaBean object to manage multiple instances of a resource adapter, since the `ResourceAdapter` JavaBean object may contain resource adapter instance specific state information.
- The application server must call the `start` method on the `ResourceAdapter` JavaBean (in order to create a functional resource adapter instance), before accessing other methods on the `ResourceAdapter` JavaBean instance or before using other objects that belong to the same resource adapter instance.
- The application server thread which calls the `start` and the `stop` method on the `ResourceAdapter` JavaBean executes in an unspecified context. However, the application server thread must have at least the same level of security permissions as that of the resource adapter instance.

FIGURE 5-4 Resource Adapter Lifecycle (State Diagram)

Unconfigured resource adapter

The resource adapter deployer configures the various JavaBean classes

Configured resource adapter

resource adapter

deploy

undeploy

Deployment tool

Resource adapter deployer

Application server

Resource adapter is deployed in the application server

Application server calls start method on the ResourceAdapter JavaBean

functional resource adapter instance

Non-functional resource adapter

Application server calls stop method on the ResourceAdapter JavaBean

The start method of the ResourceAdapter JavaBean is called each time a resource adapter instance is created. This may be during resource adapter deployment, application server startup, or other situations.

The stop method of the ResourceAdapter JavaBean is called each time a resource adapter instance is removed. This may be during resource adapter undeployment, application server shutdown, or other situations.

5.3.6   Resource Adapter Implementation Guidelines

The `ResourceAdapter` JavaBean should be treated as a central authority or registry for resource adapter instance specific information, and it should have access to the overall state of the resource adapter instance (network endpoints, etc.). This helps in the manageability of the resource adapter instance, and in performing an orderly shutdown.

Some conventions to follow:

■ Any resource adapter specific object (for example, `ManagedConnectionFactory` JavaBean, `ActivationSpec` JavaBean, or others) which creates network endpoints should register them with the `ResourceAdapter` JavaBean.
■ The resource adapter threads should periodically scan the `ResourceAdapter` JavaBean state and behave accordingly. It is desirable that such threads avoid boundless blocking on I/O calls, and instead use a bounded blocking duration. This helps in resource adapter shutdown, and also potentially avoids deadlock situations during shutdown.

The above conventions enable a `ResourceAdapter` JavaBean to effectively manage the resource adapter instance and to perform an orderly shutdown of the resource adapter instance.

5.3.7   JavaBean Configuration and Deployment

There is at most one `ResourceAdapter` JavaBean instance per resource adapter instance. But there can be many `ManagedConnectionFactory`, `ActivationSpec` or administered object instances (Section 13.4.2.3, "Administered Objects" on page 13-13) per resource adapter instance.

The `ResourceAdapter` JavaBean instance is created and configured during resource adapter deployment. The `ManagedConnectionFactory`, `ActivationSpec` and administered object instances are created and configured during the lifetime of a resource adapter instance.

At runtime, the resource adapter internally uses a union of the configured `ResourceAdapter` and `ManagedConnectionFactory` JavaBean properties, to represent outbound communication configuration.

Similarly, at runtime, the resource adapter internally uses a union of the configured `ResourceAdapter` and `ActivationSpec` JavaBean properties, to represent inbound communication configuration.

5.3.7.1 `ResourceAdapter` JavaBean Instance Configuration

■ Create a `ResourceAdapter` JavaBean instance. This will initialize the instance with the defaults specified by way of the JavaBean mechanism.
■ Apply the `ResourceAdapter` class configuration properties specified in the resource adapter deployment descriptor, on the `ResourceAdapter` instance. This may override some of the default values specified through the JavaBean mechanism. The application server is required to merge values specified by way of annotations and deployment descriptors as specified in Section 18.3, "Deployment Descriptors and Annotations" on page 18-2, before applying the ResourceAdapter class configuration properties.
■ The `ResourceAdapter` deployer may further override the values of the `ResourceAdapter` instance before deployment.

5.3.7.2 Resource Adapter Deployment

The `ResourceAdapter` instance property values may be stored separately and reused later while configuring `ManagedConnectionFactory`, `ActivationSpec`, or administered object instances.

5.3.7.3 `ManagedConnectionFactory` JavaBean Instance Configuration

■ Create a `ManagedConnectionFactory` JavaBean instance. This will initialize the instance with the defaults specified by way of the JavaBean mechanism.
■ Apply the `ResourceAdapter` instance property values, that were stored earlier, on the `ManagedConnectionFactory` instance. Note, that the `ManagedConnectionFactory` JavaBean may have none, some or all of the properties of the `ResourceAdapter` JavaBean.
■ Apply the `ManagedConnectionFactory` class configuration properties specified in the resource adapter deployment descriptor, on the `ManagedConnectionFactory` instance.
■ The application server is required to merge values specified by way of annotations and deployment descriptors as specified in Section 18.3, "Deployment Descriptors and Annotations" on page 18-2, before applying the ManagedConnectionFactory class configuration properties.
■ The `ManagedConnectionFactory` deployer may further override the values of the `ManagedConnectionFactory` instance before deployment.

At runtime, the resource adapter internally uses a union of the configured `ResourceAdapter` and `ManagedConnectionFactory` JavaBean properties, to represent outbound communication configuration. Note, the `ManagedConnectionFactory` instance and the `ResourceAdapter` instance may have intersecting property names. In such a situation, the values specified in the `ManagedConnectionFactory` instance takes precedence.

5.3.7.4   `ActivationSpec` JavaBean Instance Configuration

- Create an `ActivationSpec` JavaBean instance. This will initialize the instance with the defaults specified by way of the JavaBean mechanism.
- Apply the `ResourceAdapter` instance property values, that were stored earlier, on the `ActivationSpec` instance. Note, that the `ActivationSpec` JavaBean may have none, some, or all of the properties of the `ResourceAdapter` JavaBean.
- Apply the `ActivationSpec` class configuration properties specified in the application deployment descriptor, on the `ActivationSpec` instance.
- The application server is required to merge values specified by way of annotations and deployment descriptors as specified in Section 18.3, "Deployment Descriptors and Annotations" on page 18-2, before applying the ActivationSpec class configuration properties.
- The `ActivationSpec` deployer may further override the values of the `ActivationSpec` instance before deployment.

At runtime, the resource adapter internally uses a union of the configured `ResourceAdapter` and `ActivationSpec` JavaBean properties, to represent inbound communication configuration. Note, the `ActivationSpec` instance and the `ResourceAdapter` instance may have intersecting property names. In such a situation, the values specified in the `ActivationSpec` instance takes precedence.

5.3.7.5   JavaBean Validation

The Bean Validation specification (see "Bean Validation Specification, version 1.1" on page E-2) defines "a metadata model and API for JavaBean validation. The default metadata source is annotations, with the ability to override and extend the meta-data through the use of XML validation descriptors."

The JavaBeans provided by the resource adapter implementation, like `ResourceAdapter`, `ManagedConnectionFactory` etc, may use the annotations or the XML validation descriptor facilities defined by the Bean Validation specification to express their validation requirements of its configuration properties to the application server. A constraint annotation, can be applied to a JavaBean type, on any of the type's fields or on any of the JavaBeans-compliant properties.The use of Bean Validation constraint annotations by the resource adapter implementation as a self-validation check behavior is optional.

The Bean Validation specification defines a set of standard built-in constraints. The resource adapter implementation is encouraged to use them instead of redefining custom annotations for the same use cases. The resource adapter implementation may (but is not limited to) use the Bean Validation facilities for the following use cases:

- **Range or limits specification.** To ensure that the value provided by a deployer for a configuration property falls within prescribed limits. The resource adapter implementation may use `@Min`, `@Max`, `@Size` constraints for this purpose.

- **Mandatory attributes.** To require the deployer to provide a value for a configuration property. The resource adapter implementation may use the `@NotNull` constraint for this use case.

In the Java EE 6 environment, as specified in the Java EE platform specification, the Bean Validation facilities are available. The application server must check the validity of the configuration settings provided by the deployer for a JavaBean, using the capabilities provided by the Bean Validation specification. This validation must be performed before using the JavaBean. This helps to catch configuration errors earlier on without having to wait until the JavaBean is put to use. As the application server may check the validation of the configuration settings at deployment time and runtime, the constraint validation implementation must not make any assumptions of the availability of a live resource adapter instance. The application server must support the decoration of the following JavaBeans with constraint annotations:

- `ResourceAdapter`
- `ManagedConnectionFactory`
- `ActivationSpec`
- Administered Objects

The application server must, by default, target the `javax.validation.groups.Default` group for validation. The application server must validate the JavaBean by obtaining a `Validator` instance from its `ValidatorFactory` and invoking the `validate` method with the targeted groups. If the set of `ConstraintViolation` objects returned by the `validate` method is not empty, the application server must fail validation by throwing the `javax.validation.ConstraintValidationException` containing a reference to the returned set of `ConstraintViolation` objects, and must not put the JavaBean in use. The application server must treat all JavaBean properties as "reachable" and "cascadable" as defined by the BeanValidation Specification. For more details on reachability and cascaded validation, see Section 3.5 of the "Bean Validation Specification, version 1.1" on page E-2.

Application server configuration tools and third-party tools are recommended to leverage the constraint metadata request API defined in the Bean Validation specification to provide a richer interaction model during configuration of the JavaBeans.

5.3.7.6   Configuration Property Attributes

Dynamic Reconfigurable Configuration Properties

Configuration properties whose values could be configured dynamically during the lifetime of the JavaBean are referred to as dynamically reconfigurable configuration properties. A resource adapter may indicate that a configuration property is

dynamically reconfigurable through the `config-property-supports-dynamic-updates` attribute in the deployment descriptor (see Section 20.7, "Resource Adapter XML Schema Definition" on page 20-24) or the `supportsDynamicUpdates` annotation element in the `ConfigProperty` annotation (see Section 18.5, "@ConfigProperty" on page 18-10).

Neither the application server nor the resource adapter must support the dynamic reconfiguration of configuration properties. If an application server supports this feature and the resource adapter employs JavaBean Validation (see Section 5.3.7.5, "JavaBean Validation" on page 5-15), the application server must performJavaBean Validation after reconfiguring all the modified values of the JavaBean. When the JavaBean is validated, the resource adapter can deduce that the reconfiguration has been completed by the deployer or administrator.

Invalid reconfiguration of the state of a JavaBean by an application server may be indicated by the resource adapter through the following means:

■ Throwing an exception when the field is updated
■ For configuration properties that can only be validated based on the state of other configuration properties, throwing an exception during the validation phase.

Confidential Properties

Certain configuration properties of a JavaBean, such as `Password` (see Section 20.5.4, "Standard Properties" on page 20-14 for more information on `Password`), may be confidential and must not be presented as clear text in configuration tools. The resource adapter may indicate such properties as "Confidential Properties" through the `config-property-confidential` attribute in the deployment descriptor (see Section 20.7, "Resource Adapter XML Schema Definition" on page 20-24) or the `confidentialProperty` annotation element in the `ConfigProperty` annotation (see Section 18.5, "@ConfigProperty" on page 18-10). The application server's configuration tool may use this attribute to use special visual aids denoting confidentiality.

5.3.7.7   Resource Adapter Implementation Guidelines

A resource adapter implementation may choose to use common properties, that is, a `ManagedConnectionFactory` or an `ActivationSpec` JavaBean, may contain some or all of the properties of the `ResourceAdapter` JavaBean. The choice is up to the resource adapter implementation.

In general, there is no need for common properties, since these various objects are associated at runtime with the `ResourceAdapter` JavaBean. However, there may be situations, for example, a `ManagedConnectionFactory` JavaBean may need to override the `ResourceAdapter` JavaBean values in order to successfully connect to a different EIS. In such a scenario, providing common properties between the

`ResourceAdapter` and `ManagedConnectionFactory` JavaBeans, allows the `ManagedConnectionFactory` deployer to override the `ResourceAdapter` property values and configure the `ManagedConnectionFactory` appropriately.

5.3.8 Lifecycle Management in a Non-Managed Environment

Although the lifecycle management contract is primarily intended for a managed environment, it may still be used in a non-managed environment provided that the application that bootstraps a resource adapter instance is capable of managing its lifecycle.

5.3.9   A Sample Resource Adapter Implementation

CODE EXAMPLE 5-1   Sample Resource Adapter

```
package com.xyz.adapter;

import javax.resource.spi.ResourceAdapter;
import javax.resource.spi.BootstrapContext;
import javax.resource.spi.work.*;

public class MyResourceAdapterImpl implements ResourceAdapter {

    void start(BootstrapContext serverCtx) {
        // 1. setup network endpoints
      ...

        // 2. get WorkManager reference
        WorkManager wm = serverCtx.getWorkManager();

         // 3. provide Work objects to WorkManager
        for (i = 0; i < 10; i++) {
           Work work = new MyWork(...);
           try {
               wm.startWork(work);
           } catch (WorkException we) { // handle the exception }
        }
    }

   void stop() { // release Work instances, do cleanup and return.}
}

public class MyWork implements Work {

    void release() {
      // set a flag to hint the Work instance to complete.
      // Note, the calling thread is different from
      // the active thread in which this instance is executing.
    }

    void run() {
      // do work (call application components, monitor
     // network ports, etc.).
    }
}
```

FIGURE 5-5 Lifecycle Management Model (Sequence Diagram)
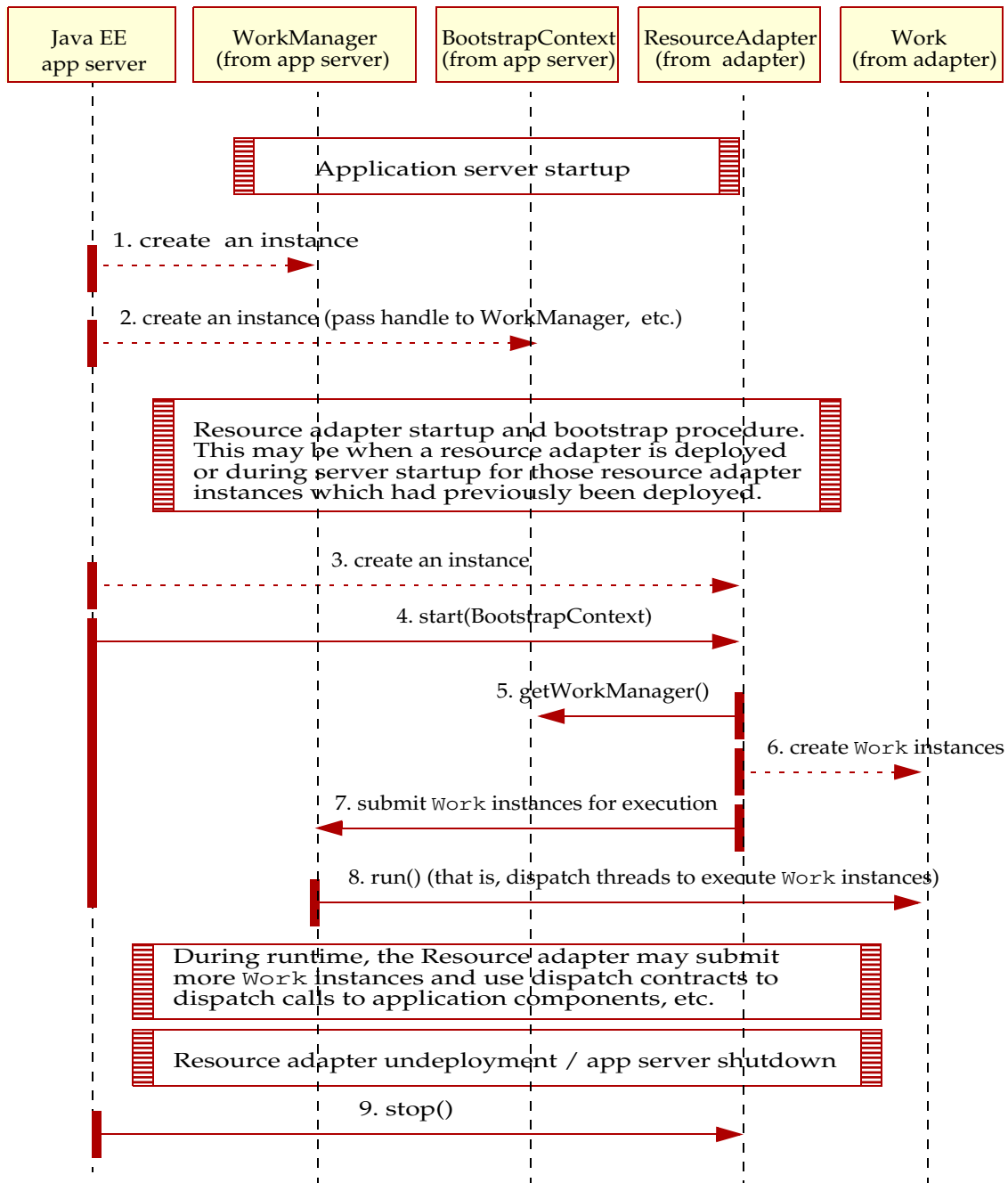
Connection Management

This chapter specifies the connection management contract between an application server and a resource adapter. It introduces the concepts and mechanisms relevant to this contract, and delineates the responsibilities of the roles of the resource adapter provider and application server vendor in terms of their system-level support for the connection management contract. To complete the description of the connection management contract, this chapter also refers to the responsibilities of the application component provider and deployer. The chapter includes scenarios to illustrate the connection management contract.

6.1 Overview

An application component uses a connection factory to access a connection instance, which the component then uses to connect to the underlying EIS. A resource adapter acts as a factory of connections. Examples of connections include database connections, JMS (Java Message Service) connections, and SAP R/3 connections.

Connection pooling manages connections that are expensive to create and destroy. Connection pooling of expensive connections leads to better scalability and performance in an operational environment. The connection management contract provides support for connection pooling.

6.2 Goals

The connection management contract has been designed with the following goals:

- To provide a consistent application programming model for connection acquisition for both managed and non-managed (two-tier) applications.

- To enable a resource adapter to provide a connection factory and connection interfaces based on the CCI specific to the type of resource adapter and EIS. This enables JDBC drivers to be aligned with the connector architecture with minimum impact on the existing JDBC APIs.
- To provide a generic mechanism by which an application server can provide different services—transactions, security, advanced pooling, error tracing/logging—for its configured set of resource adapters.
- To provide support for connection pooling.

The goal of the connector architecture is to enable efficient, scalable, and extensible connection pooling mechanisms, not to specify a mechanism or implementation for connection pooling. The goal is accomplished by defining a standard contract for connection management with the providers of connections—that is, resource adapters. An application server should use the connection management contract to implement a connection pooling mechanism in its own implementation-specific way.

6.3   Architecture: Connection Management

The connection management contract specifies an architected contract between an application server and a resource adapter. This connection management contract is shown with bold flow lines in FIGURE 6-1. It includes the set of interfaces shown in the architecture diagram.

6.3.1   Overview: Managed Application Scenario

The application server uses the deployment information specified by way of the deployment descriptor mechanism (specified in section Section 20.6.4, "Requirements" on page 20-24) and metadata annotations (specified in Section 18.3, "Deployment Descriptors and Annotations" on page 18-2) to configure the resource adapter in the operational environment.

The resource adapter provides connection and connection factory interfaces. A connection factory acts as a factory for EIS connections. For example, `javax.sql.DataSource` and `java.sql.Connection` interfaces are JDBC-based interfaces for connecting to a relational database.

The CCI (specified in Chapter 17, "Common Client Interface") defines `javax.resource.cci.ConnectionFactory` and `javax.resource.cci.Connection` as interfaces for a connection factory and a connection, respectively.

The application component does a lookup of a connection factory in the Java Naming and Directory Interface^{TM} (JNDI) name space. It uses the connection factory to get a connection to the underlying EIS. The connection factory instance delegates the connection creation request to the `ConnectionManager` instance.

The `ConnectionManager` enables the application server to provide different quality-of-services in the managed application scenario. These quality-of-services include transaction management, security, error logging and tracing, and connection pool management. The application server provides these services in its own implementation-specific way. The connector architecture does not specify how the application server implements these services.

The `ConnectionManager` instance, on receiving a connection creation request from the connection factory, does a lookup in the connection pool provided by the application server. If there is no connection in the pool that can satisfy the connection request, the application server uses the `ManagedConnectionFactory` interface (implemented by the resource adapter) to create a new physical connection to the underlying EIS. If the application server finds a matching connection in the pool, it uses the matching `ManagedConnection` instance to satisfy the connection request.

If a new `ManagedConnection` instance is created, the application server adds the new `ManagedConnection` instance to the connection pool.

The application server registers a `ConnectionEventListener` with the `ManagedConnection` instance. This listener enables the application server to get event notifications related to the state of the `ManagedConnection` instance. The application server uses these notifications to manage connection pooling, manage transactions, cleanup connections, and handle any error conditions.

The application server uses the `ManagedConnection` instance to get a connection instance that acts as an application-level handle to the underlying physical connection. An instance of type `javax.resource.cci.Connection` is an example of such a connection handle. An application component uses the connection handle to access EIS resources.

The resource adapter implements the `XAResource` interface to provide support for transaction management. The resource adapter also implements the `LocalTransaction` interface so that the application server can manage transactions internal to a resource manager. The chapter on transaction management describes this transaction management contract between the application server (and its transaction manager) and the resource adapter (and its underlying resource manager).

FIGURE 6-1   Architecture Diagram: Managed Application scenario
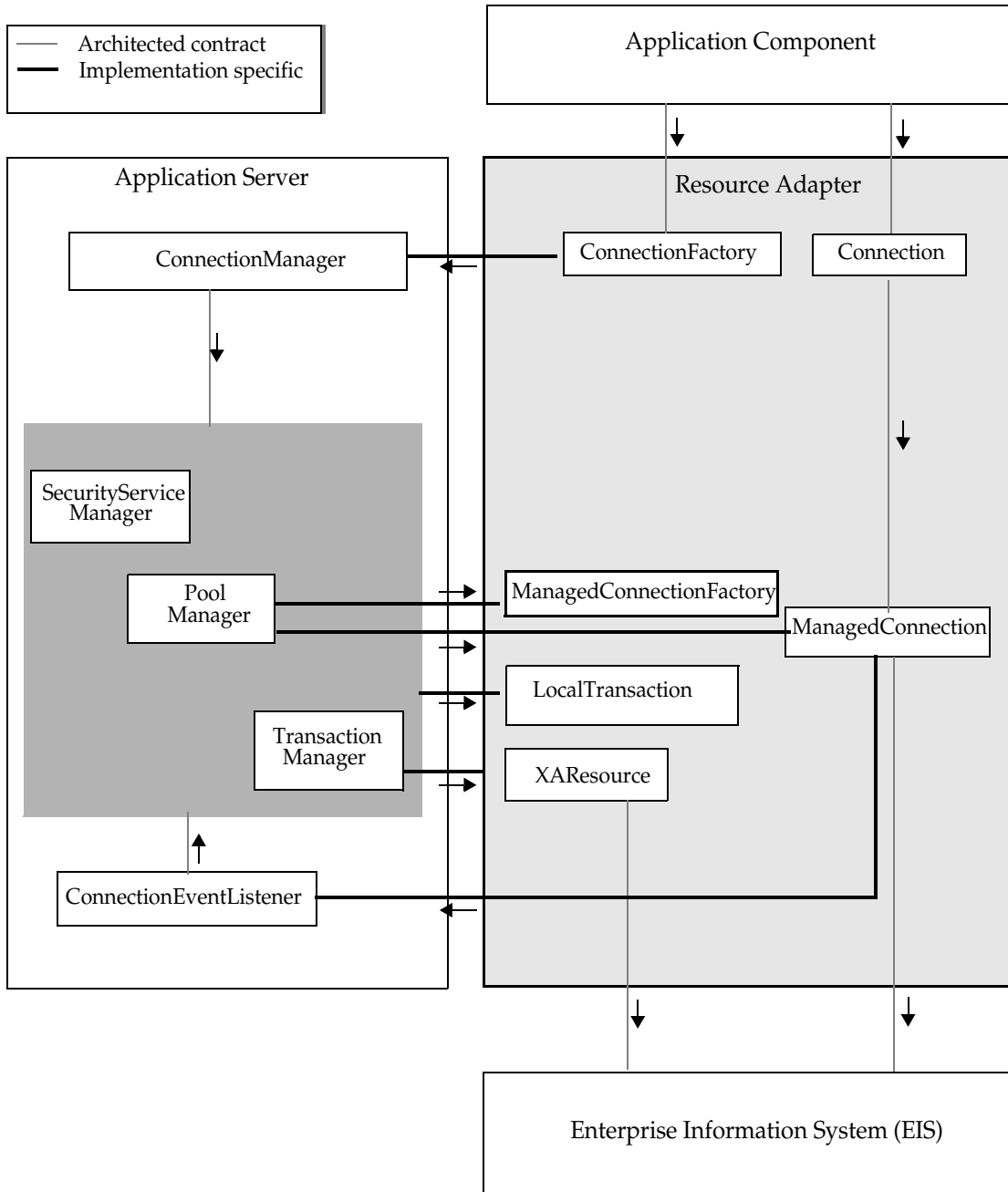
6.4 Application Programming Model

The application programming model for getting an EIS connection is similar across both managed (application server based) and non-managed scenarios. The following sections explain a typical application programming model scenario.

6.4.1 Managed Application Scenario

The following steps are involved in a managed scenario:

1. The application assembler or component provider specifies connection factory requirements for an application component using a deployment descriptor mechanism. For example, a bean provider specifies the following elements in the deployment descriptor for a connection factory reference. Note that the connection factory reference is part of the deployment descriptor for EJB components and not the resource adapter. Refer EJB specification (see "Enterprise JavaBeans (EJB) Specification, version 3.2" on page E-1) for details on the deployment mechanism for EJB components:

   - `res-ref-name: eis/MyEIS`
   - `res-type: javax.resource.cci.ConnectionFactory`
   - `res-auth: Application` or `Container`

2. During resource adapter deployment, the deployer sets the configuration information (example: server name, port number) for the resource adapter. The application server uses a configured resource adapter to create physical connections to the underlying EIS. Refer to Chapter 19, "API Requirements" for details on packaging and deployment of a resource adapter.

3. The application component looks up a connection factory instance in the component's environment using the JNDI interface.

```
// obtain the initial JNDI Naming context
Context initctx = new InitialContext();

// perform JNDI lookup to obtain the connection factory
javax.resource.cci.ConnectionFactory cxf =
          (javax.resource.cci.ConnectionFactory)
              initctx.lookup("java:comp/env/eis/MyEIS");
```

The JNDI name passed in the method `NamingContext.lookup` is the same as that specified in the `res-ref-name` element of the deployment descriptor. The JNDI lookup results in a connection factory instance of type `java.resource.cci.ConnectionFactory` as specified in the `res-type` element.

4. The application component invokes the `getConnection` method on the connection factory to get an EIS connection. The returned connection instance represents an application-level handle to an underlying physical connection.

   An application component obtains multiple connections by calling the method `getConnection` on the connection factory multiple times.

```
javax.resource.cci.Connection cx = cxf.getConnection();
```

5. The application component uses the returned connection to access the underlying EIS by way of the resource adapter. Chapter 17, "Common Client Interface" specifies in detail the application programming model for EIS access.

Note – The JNDI context of an accessing application is available to a resource adapter through the application thread that uses its connection object. The resource adapter may use the JNDI context to access other resources.

---

6. After the component finishes with the connection, it closes the connection using the `close` method on the `Connection` interface.

```
cx.close();
```

7. If an application component fails to close an allocated connection after its use, that connection is considered an unused connection. The application server manages the cleanup of unused connections. When a container terminates a component instance, the container cleans up all connections used by that component instance. Refer section Section 6.5.4 "ManagedConnection" and Section 6.8.3 "Scenario: Connection Event Notifications and Connection Close" for details on the cleanup of connections.

6.4.2   Non-Managed Application Scenario

In a non-managed application scenario, the application developer follows a similar programming model to the managed application scenario. The non-managed case involves looking up of a connection factory instance, getting an EIS connection, using the connection for EIS access, and finally closing the connection.

6.4.3   Guidelines

Connection handles are application level handles to underlying physical connections and are light-weight objects, especially when dissociated from the `ManagedConnection`. Creation of a connection handle does not necessarily result in the creation of a new physical connection to the EIS. The `ManagedConnection`, which represents the actual underlying physical connection, should maintain any session or transaction state data associated with that connection to the EIS. An application component may not derive much benefit from caching these handles, although this is allowed in this specification. Application components are recommended to obtain and cache the Connection Factory objects instead. For more information, see Section 6.5.1 "ConnectionFactory and Connection".

An application component is recommended to obtain a connection handle from the connection factory, use the connection handle to interact with the EIS by way of the resource adapter, and close the connection handle after finishing with it.

```
//recommended: connection handle creation, use and close
Connection con = null;
try {
    con = cf.getConnection();
    //use the con handle to interact with the EIS
} finally {
    if (con != null){
        con.close();
    }
}
```

The application component is recommended to explicitly close the connection handle as soon as the handle has been used and is not required later. This reduces the possibility of connection leaks and enhances the application server's ability to pool physical connections to the EIS (see Section 6.5.3.3 "Connection Pool Implementation").

6.5   Interface/Class Specification

This section specifies the Java classes and interfaces defined as part of the connection management contract. For a complete specification of these classes and interfaces, refer to the API documentation distributed with this document.

The following figure shows the class hierarchy for the connection management contract. The diagram also illustrates the responsibilities for the definition of an interface and its implementation:

FIGURE 6-2 Class Diagram: Connection Management Architecture

package: javax.resource.spi

package: javax.resource.cci

<interface>
ConnectionFactory

<interface>
Connection

<interface>
ConnectionManager

<interface>
ManagedConnectionFactory

<interface>
ManagedConnection

<interface>
ManagedConnectionMetaData

<interface>
ConnectionEventListener

<interface>
LocalTransaction

package: Resource Adapter Specific

DefaultConnectionManager

ManagedConnectionFactoryImpl

ConnectionImpl

0-1

ManagedConnectionImpl

0-1

0-1

ManagedConnection-
MetaDataImpl

ConnectionFactoryImpl

package: (Application Server specific)

ConnectionManagerImpl

ConnectionEventListenerImpl

0-1

LocalTransactionImpl

package: javax.transaction.xa

<interface>
XAResource

0-1

XAResourceImpl

- implements
association or use
relationship
inherits
contains

6.5.1   `ConnectionFactory` and `Connection`[1]

A connection factory provides an interface to get a connection to an EIS instance. A connection provides connectivity to an underlying EIS.

One goal of the connector architecture is to support a consistent application programming model across both CCI and EIS specific client APIs. To achieve this goal, the connector architecture recommends a design pattern (specified as an interface template) for both the connection factory and connection interfaces.

The CCI connection factory and connection interfaces (defined in the package `javax.resource.cci`) are based on the above design pattern. Refer to Section 17.5, "Connection Interfaces" on page 17-9 for details on the CCI connection factory and connection interfaces. The following code sample shows the CCI interfaces:

```
public interface javax.resource.cci.ConnectionFactory
    extends java.io.Serializable, javax.resource.Referenceable {

    public javax.resource.cci.Connection getConnection()
        throws javax.resource.ResourceException;
    ...
}

public interface javax.resource.cci.Connection {
    public void close() throws javax.resource.ResourceException;
    ...
}
```

---

1. In this document, the term 'physical connection' refers to a `ManagedConnection` instance, while the term 'connection handle' refers to an application-level connection handle. When the distinction between 'physical connection' and 'connection handle' is not important, the term 'connection' is used to refer to an EIS connection.

An example of a non-CCI interface is a resource adapter that uses the package `com.myeis` for its EIS specific interfaces, as follows:

```
public interface com.myeis.ConnectionFactory
    extends java.io.Serializable, javax.resource.Referenceable {

    public com.myeis.Connection getConnection()
        throws com.myeis.ResourceException;
    ...
}

public interface com.myeis.Connection {
    public void close() throws com.myeis.ResourceException;
    ...
}
```

The JDBC interfaces—`javax.sql.DataSource`, `java.sql.Connection`—are examples of non-CCI connection factory and connection interfaces.

Note that the methods defined on a non-CCI interface are not required to throw a `ResourceException`. The exception can be specific to a resource adapter, for example: `java.sql.SQLException` for JDBC (see "JDBC API Specification, version 4.1" on page E-1) interfaces.

The following are additional guidelines for the recommended interface template:

- A resource adapter is allowed to add additional `getConnection` methods to its definition of a connection factory interface. These additional methods are specific to a resource adapter and its EIS. For example, CCI defines a variant of the `getConnection` method that takes `java.resource.cci.ConnectionSpec` as a parameter.
- A resource adapter should only introduce additional `getConnection` methods if it requires additional flexibility (beyond that offered by the default `getConnection` method) in the connection request invocations.
- A connection interface must provide a `close` method to close the connection. The behavior of such an application-level connection closure is described in the OID FIGURE 6-7.

The above design pattern leads to a consistent application programming model for connection creation and connection closing.

6.5.1.1   Requirements

A resource adapter must provide implementations for both the connection factory and connection interfaces.

In the connector architecture, a resource adapter provides an implementation of the connection factory interface in both managed and non-managed scenarios. This differs from the JDBC (see "JDBC API Specification, version 4.1" on page E-1) architecture.

In the JDBC architecture, an application server provides the implementation of `javax.sql.DataSource` interface. Using a similar design approach for the connector architecture would have required an application server to provide implementations of various connection factory interfaces defined by different resource adapters. Since the connection factory interface may be defined as specific to an EIS, the application server may find it difficult to provide implementations of connection factory interfaces without any code generation.

The connection factory implementation class delegates the `getConnection` method invocation from an application component to the associated `ConnectionManager` instance. The `ConnectionManager` instance is associated with a connection factory instance at its instantiation [refer to the OID shown in FIGURE 20-3 on page 20-23].

Note that the connection factory implementation class must call the `ConnectionManager.allocateConnection` method in the same thread context in which the application component had called the `getConnection` method.

The connection factory implementation class is responsible for taking connection request information and passing it in a form required by the `ConnectionManager.allocateConnection` method.

```
public interface javax.resource.spi.ConnectionManager
    extends java.io.Serializable {

    public Object allocateConnection(
    ManagedConnectionFactory mcf,
    ConnectionRequestInfo cxRequestInfo)
        throws ResourceException;
}

public interface javax.resource.spi.ConnectionRequestInfo {
    public boolean equals(Object other);
    public int hashCode();
}
```

6.5.1.2　ConnectionRequestInfo

The `ConnectionRequestInfo` parameter to the `ConnectionManager.allocateConnection` method enables a resource adapter to pass its own request-specific data structure across the connection request flow.

A resource adapter extends the `ConnectionRequestInfo` interface to support its own data structure for the connection request.

This is typically used to allow a resource adapter to handle application component-specified per-connection request properties (for example, `clientID` and `language`). The application server passes these properties to the `createManagedConnection` and `matchManagedConnections` method calls on the `ManagedConnectionFactory`. These properties remain opaque to the application server during the connection request flow.

It is important to note that the properties passed through the `ConnectionRequestInfo` instance should be client-specific (for example, user name, password, language) and not related to the configuration of a target EIS instance (for example, port number, server name).

The `ManagedConnectionFactory` instance is configured with properties required for the creation of a connection to a specific EIS instance. Note that a configured `ManagedConnectionFactory` instance must have the complete set of properties that are needed for the creation of the physical connections. This enables the container to manage connection request without requiring an application component to pass any explicit connection parameters. Configured properties on a `ManagedConnectionFactory` can be overridden through `ConnectionRequestInfo` in cases when a component provides client-specific properties in the `getConnection` method invocation. Refer to Section 20.5.1, "ResourceAdapter" on page 20-12 for details on the configuration of a `ManagedConnectionFactory`.

When the `ConnectionRequestInfo` reaches the `createManagedConnection` or `matchManagedConnections` methods on the `ManagedConnectionFactory` instance, the resource adapter uses this additional per-request information to create and match connections.

A resource adapter must implement the `equals` and `hashCode` methods defined in the `ConnectionRequestInfo` interface. The equality must be defined in the complete set of properties for the `ConnectionRequestInfo` instance. An application server can use these methods to structure its connection pool in an implementation-specific way. Since `ConnectionRequestInfo` represents a resource adapter specific data structure, the conditions for equality are defined and implemented by a resource adapter.

6.5.1.3   Additional Requirements

A resource adapter implementation is not required to support the mechanism for passing resource adapter-specific connection request information. It can choose to pass `null` for `ConnectionRequestInfo` in the `allocateConnection` invocation.

An implementation class for a connection factory interface must implement `java.io.Serializable`. This enables a connection factory instance to be stored in the JNDI naming environment. A connection factory implementation class must implement the interface `javax.resource.Referenceable`. Note that the `javax.resource.Referenceable` interface extends the `javax.naming.Referenceable` interface. Refer to section Section 20.6.3, "Scenario: Referenceable" on page 20-19 for details on the JNDI reference mechanism.

A connection implementation class implements its methods in a resource adapter implementation-specific way. It must use a `javax.resource.spi.ManagedConnection` instance as its underlying physical connection.

6.5.2   ConnectionManager

The `javax.resource.spi.ConnectionManager` interface provides a hook for a resource adapter to pass a connection request to an application server. An application server provides different quality-of-service as part of its handling of the connection request.

6.5.2.1   Interface

The connection management contract defines a standard interface for the `ConnectionManager` as follows:

```
public interface javax.resource.spi.ConnectionManager
    extends java.io.Serializable {

    public Object allocateConnection(
    ManagedConnectionFactory mcf,
    ConnectionRequestInfo cxRequestInfo)
        throws ResourceException;
}
```

The method `allocateConnection` is called by a resource adapter's connection factory instance so that the instance can delegate a connection request to the `ConnectionManager` instance.

The `ConnectionRequestInfo` parameter represents information specific to a resource adapter to handle the connection request.

6.5.2.2   Requirements

An application server must provide an implementation of the `ConnectionManager` interface. This implementation is not specific to any particular resource adapter or connection factory interface.

The `ConnectionManager` implementation delegates to the internal mechanisms of an application server to provide various services: security, connection pool management, transaction management, and error logging and tracing.

An application server should implement these services in a generic manner, independent of any resource adapter and EIS-specific mechanisms. The connector architecture does not specify how an application server implements these services; the implementation is specific to each application server.

After an application server hooks-in its services, the connection request is delegated to a `ManagedConnectionFactory` instance either for the creation of a new physical connection or for the matching of an already existing physical connection.

An implementation class for the `ConnectionManager` interface must implement the `java.io.Serializable` interface.

A resource adapter must provide a default implementation of the `javax.resource.spi.ConnectionManager` interface. The implementation class comes into play when a resource adapter is used in a non-managed two-tier application scenario. In an application server-managed environment, the resource adapter must not use the default `ConnectionManager` implementation class. A default implementation of `ConnectionManager` enables the resource adapter to provide services specific to itself. These services can include connection pooling, error logging and tracing, and security management. The default `ConnectionManager` delegates to the `ManagedConnectionFactory` the creation of physical connections to the underlying EIS.

An implementation of the ConnectionManager interface may only be provided by a resource adapter, for the purpose described in this section, or by an application server that fully meets the requirements of this specification.

FIGURE 6-3 ConnectionManager and Application Server Specific Services
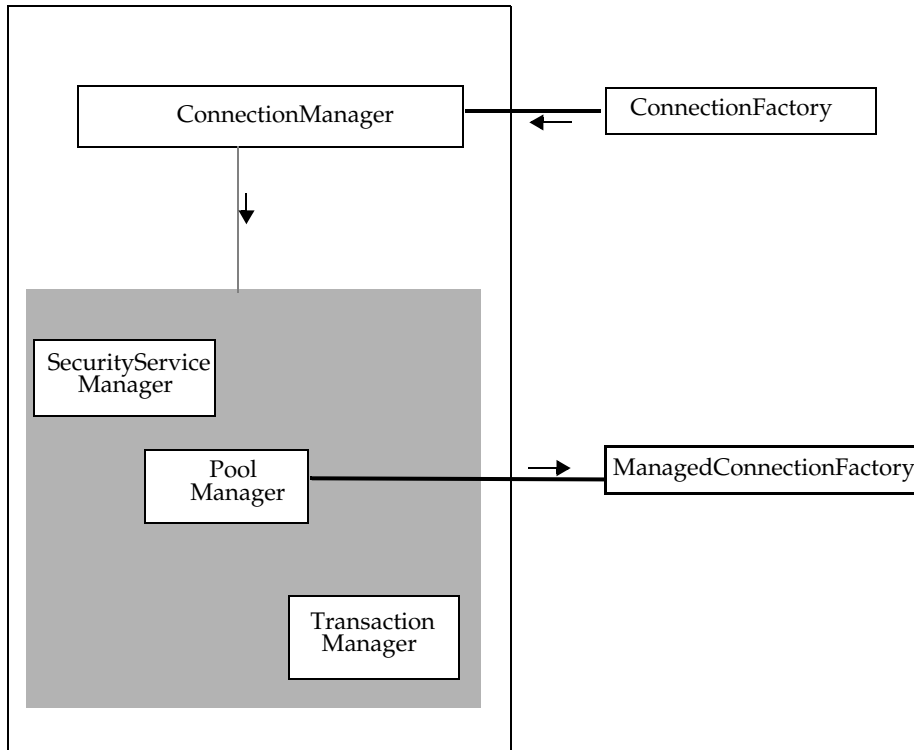
6.5.3 ManagedConnectionFactory

A javax.resource.spi.ManagedConnectionFactory instance is a factory of both ManagedConnection and connection factory instances. This interface supports connection pooling by defining methods for matching and creating connections.

6.5.3.1   Interface

The following code extract shows the interface specification for the
`ManagedConnectionFactory`.

```
public interface javax.resource.spi.ManagedConnectionFactory
    extends java.io.Serializable {

    public Object createConnectionFactory(
    ConnectionManager connectionManager)
          throws ResourceException;

    public Object createConnectionFactory()
          throws ResourceException;

    public ManagedConnection createManagedConnection(
    javax.security.auth.Subject subject,
    ConnectionRequestInfo cxRequestInfo)
          throws ResourceException;

    public ManagedConnection matchManagedConnections(
    java.util.Set connectionSet,
    javax.security.auth.Subject subject,
    ConnectionRequestInfo cxRequestInfo)
        throws ResourceException;

    public boolean equals(Object other);
    public int hashCode();

}
```

The method `createConnectionFactory` creates a connection factory instance.
For CCI, the connection factory instance is of the type
`javax.resource.cci.ConnectionFactory`. The connection factory instance is
initialized with the `ConnectionManager` instance provided by the application
server.

When the `createConnectionFactory` method takes no arguments,
`ManagedConnectionFactory` provides a default `ConnectionManager` instance.
This occurs in a non-managed application scenario.

The method `createManagedConnection` creates a new physical connection to the
underlying EIS instance. The `ManagedConnectionFactory` instance uses the
security information (passed as a `Subject` instance) and an optional
`ConnectionRequestInfo` instance to create this new physical connection (refer to
Chapter 9, "Security Contract" for more details).

A created `ManagedConnection` instance typically maintains internal information about the security context (under which the connection has been created) and any connection-specific parameters (for example, the socket connection).

The `matchManagedConnections` method enables the application server to use resource adapter-specific criteria for matching a `ManagedConnection` instance to service a connection request. The application server finds a candidate set of `ManagedConnection` instances from its connection pool based on application server-specific criteria, and passes this candidate set to the `matchManagedConnections` method. If the application server implements connection pooling, it must use the `matchManagedConnections` method to choose a suitable connection.

The `matchManagedConnections` method matches a candidate set of connections using criteria known internally to the resource adapter. The criteria used for matching connections is specific to a resource adapter and is not specified by the connector architecture.

A `ManagedConnection` instance has specific internal state information based on its security context and physical connection. The `ManagedConnectionFactory` implementation compares this information for each `ManagedConnection` instance in the candidate set against the information passed in through the `matchManagedConnections` method and the configuration of this `ManagedConnectionFactory` instance. The `ManagedConnectionFactory` uses the results of this comparison to choose the `ManagedConnection` instance that can best satisfy the current connection request.

If the resource adapter cannot find an acceptable `ManagedConnection` instance, it returns a `null` value. In this case, the application server requests the resource adapter to create a new connection instance.

If the resource adapter does not support connection matching, it must throw a `NotSupportedException` when `matchManagedConnections` method is invoked. This allows an application server to avoid pooling connections obtained from that resource adapter.

6.5.3.2   Requirements

A resource adapter must provide an implementation of the `ManagedConnectionFactory` interface.

It is required that the `ManagedConnectionFactory` implementation class extend the implementation of the `hashCode` and `equals` methods defined in `java.lang.Object`. These two methods are used by an application server to structure its connection pool in an implementation-specific way. The `equals` and

`hashCode` method implementation should be based on a complete set of configuration properties that make a `ManagedConnectionFactory` instance unique and specific to an EIS instance.

An implementation class for `ManagedConnectionFactory` interface must be a JavaBean. Refer to Section 19.3, "JavaBean Requirements" on page 19-2.

6.5.3.3   Connection Pool Implementation

The connector architecture does not specify how an application server implements connection pooling. However, it recommends that an application server should structure its connection pool such that it uses the connection creation and matching facility in an efficient manner and does not cause resource starvation.

The following paragraphs provide non-prescriptive guidelines for the connection pool implementation by an application server.

An application server may partition its pool on a per `ManagedConnectionFactory` instance (and thereby on a per EIS instance) basis. An application server may choose to guarantee, in an implementation specific way, that it will always partition connection pools with at least per `ManagedConnectionFactory` instance granularity.

The per-`ManagedConnectionFactory` instance pool may be further partitioned based on the transaction or security context or any client-specific parameters (as associated with the `ConnectionRequestInfo`). When an application server calls the matching facility, it is recommended that the application server narrow down the candidate set of `ManagedConnection` instances to a reasonable limit, and achieves matching efficiently. For example, an application server may pass only those `ManagedConnection` instances to the `matchManagedConnections` method that are associated with the target `ManagedConnectionFactory` instance (and thereby a specific target EIS instance).

An application server may use additional parameters for its search and matching criteria used in its connection pool management. These parameters may be EIS- or application server- specific. The `equals` and `hashCode` methods defined in both `ManagedConnectionFactory` and `ConnectionRequestInfo` facilitate connection pool management and structuring by an application server.

6.5.3.4 Detecting Invalid Connections

```
import java.util.Set;

interface ValidatingManagedConnectionFactory {
    Set getInvalidConnections(Set connectionSet) throws
ResourceException;
}
```

This interface may be implemented by a `ManagedConnectionFactory` instance that supports the ability to validate `ManagedConnection` objects. The `getInvalidConnections` method returns a set of invalid `ManagedConnection` objects chosen from a specified set of `ManagedConnection` objects.

This optional functionality may be used by the application server to prune invalid `ManagedConnection` objects from its connection pool periodically. The application server may use this functionality to test for the validity of a `ManagedConnection` by passing in a `Set` of size one (with the `ManagedConnection` that has to be tested for validity as the only member of the `Set`).

6.5.3.5 Requirement for XA Recovery

The `ManagedConnectionFactory` implementation for a transaction authority (XA) protocol capable resource adapter (refer to Chapter 7, "Transaction Management" for more details on transactions) must support the `createManagedConnection` method that takes a `Subject` and a `null` for the parameter `ConnectionRequestInfo`. This enables the application server to get an `XAResource` instance using `ManagedConnection.getXAResource` and then call the `XAResource.recover` method. Note that the application server uses this special case only to get to the `XAResource` instance for the underlying resource manager.

The reason for this requirement is that the application server may not have a valid `ConnectionRequestInfo` instance when it is required to get the `ManagedConnection` instance to initiate recovery. Refer to Section 9.1.8, "ManagedConnectionFactory" on page 9-9 for additional details on the `ManagedConnectionFactory.createManagedConnection` method.

6.5.4 ManagedConnection

A `javax.resource.spi.ManagedConnection` instance represents a physical connection to an underlying EIS.

Note – The connector architecture allows one or more `ManagedConnection` instances to be multiplexed over a single physical pipe to an EIS. However, for simplicity, this specification describes a `ManagedConnection` instance as being mapped 1-1 to a physical connection.

The creation of a `ManagedConnection` instance typically results in the allocation of EIS and resource adapter resources (for example, memory and network sockets) for each physical connection. Since these resources can be costly and scarce, an application server pools `ManagedConnection` instances in a managed environment.

Connection pooling improves the scalability of an application environment. An application server uses the `ManagedConnectionFactory` and `ManagedConnection` interfaces to implement connection pool management.

An application server also uses the transaction management-related methods (`getXAResource` and `getLocalTransaction`) on the `ManagedConnection` interface to manage transactions. These methods are discussed in more detail in Chapter 7, "Transaction Management".

The `ManagedConnection` interface also provides methods to support error logging and tracing in a managed environment.

6.5.4.1 Interface

The connection management contract defines the following interface for a
`ManagedConnection`. The following code extract shows only the methods that are
used for connection pool management. The remaining methods are introduced in
other parts of the specification.

```
public interface javax.resource.spi.ManagedConnection {
    public Object getConnection(
        javax.security.auth.Subject subject,
        ConnectionRequestInfo cxRequestInfo)
                throws ResourceException;
    public void destroy() throws ResourceException;
    public void cleanup() throws ResourceException;

    // Methods for Connection and transaction event notifications
    public void addConnectionEventListener(
        ConnectionEventListener listener);
    public void removeConnectionEventListener(
        ConnectionEventListener listener);

    public ManagedConnectionMetaData getMetaData()
        throws ResourceException;

    // Additional methods - specified in the other sections
    ...
}
```

The `getConnection` method creates a new application-level connection handle. A
connection handle is tied to an underlying physical connection represented by a
`ManagedConnection` instance. For CCI, the connection handle created by a
`ManagedConnection` instance is of the type `javax.resource.cci.Connection`.
A connection handle is tied to its `ManagedConnection` instance in a resource
adapter implementation-specific way.

A `ManagedConnection` instance may use the `getConnection` method to change
the state of the physical connection based on the `Subject` and
`ConnectionRequestInfo` arguments. For example, a resource adapter can re-
authenticate a physical connection to the underlying EIS when the application server
calls the `getConnection` method. Section 9.1.9, "ManagedConnection" on page 9-14
specifies re-authentication requirements in more detail.

The method `addConnectionEventListener` allows a connection event listener to
register with a `ManagedConnection` instance. The `ManagedConnection` instance
notifies connection close/error and local transaction-related events to its registered
set of listeners.

The `removeConnectionEventListener` method removes a registered `ConnectionEventListener` instance from a `ManagedConnection` instance. Since an application server may modify the list of event listeners at a time when the `ManagedConnection` may be iterating through its list of event listeners, the resource adapter is recommended to handle this scenario by synchronizing access to its list of event listeners.

The method `getMetaData` returns the metadata information (represented by the `ManagedConnectionMetaData` interface) for a `ManagedConnection` and the connected EIS instance.

6.5.4.2   Connection Sharing and Multiple Connection Handles

To support connection sharing, the application server can call `getConnection` multiple times on a `ManagedConnection` instance. In this case, a call to the method `ManagedConnection.getConnection` does not invalidate any previously created connection handles. Multiple connection handles can exist concurrently for a single `ManagedConnection` instance. This design supports the connection sharing mechanism. Refer to Section 7.9, "Connection Sharing" on page 7-32 for more details.

Because multiple connection handles to a single `ManagedConnection` can exist concurrently, a resource adapter implementation may:

- Provide thread-safe semantics for a `ManagedConnection` implementation to support concurrent access to a `ManagedConnection` instance from multiple connection handles. It is strongly recommended that resource adapters provide support for concurrent access to a `ManagedConnection` instance from multiple connection handles. This may be required in a future release of the specification.
- Ensure that there is at most one connection handle associated actively with a `ManagedConnection` instance. The active connection handle is the only connection using the `ManagedConnection` instance until an application-level `close` is called on this connection handle. The active connection handle may also be modified by the container as a result of Connection Association (see Section 7.11.2, "Connection Association" on page 7-43) or the dissociation of a lazily associatable `ManagedConnection` (see Section 7.16.1, "Lazy Connection Association Optimization" on page 7-49). For example, a `ManagedConnection.getConnection` method implementation associates a newly created connection handle as the active connection handle. Any operations on the `ManagedConnection` from any previously created connection handles should result in an application level exception. An example application level exception extends the `javax.resource.ResourceException` interface and is specific to a resource adapter. A scenario illustrating this implementation is shown in the Section 7.10.3, "Scenario: Local Transaction" on page 7-38.

6.5.4.3   Connection Matching Contract

The application server invokes the
`ManagedConnectionFactory.matchManagedConnections` method (implemented
by a resource adapter) to find a matching `ManagedConnection` for servicing a
connection request. The application server passes a candidate set of
`ManagedConnection` instances to the `matchManagedConnections` method.

The application server should use the connection matching contract for
`ManagedConnection` instances that have no existing connection handles. A
candidate set passed to the `matchManagedConnections` method should not have
any `ManagedConnection` instances with existing connection handles.

There is no requirement that the `matchManagedConnections` implementation be
capable of performing a match across a candidate set that includes
`ManagedConnection` instances with existing connection handles. Note that a
resource adapter can return a successful match with the requirement that the
`ManagedConnection.getConnection` method will later change the state of the
matched `ManagedConnection`. To avoid any unexpected matching behavior, the
application server should not pass a `ManagedConnection` instance with existing
connection handles to the `matchManagedConnections` method as part of a
candidate set.

A connection request can lead to the creation of additional connection handles for a
`ManagedConnection` instance that already has one or more existing connection
handles. In this case, the application server should take the responsibility of
checking whether or not the chosen `ManagedConnection` instance can service such
a request. Refer to Section 7.9, "Connection Sharing" on page 7-32 for details.

6.5.4.4   Cleanup of `ManagedConnection`

A resource adapter typically allocates system resources (outside a JVM instance) for
a `ManagedConnection` instance. Additionally, a `ManagedConnection` instance can
have state specific to a client, such as security context, data/function access
structures, and result set from a query.

The method `ManagedConnection.cleanup` initiates a cleanup of any client-specific
state maintained by a `ManagedConnection` instance. The `cleanup` must invalidate
all connection handles created using the `ManagedConnection` instance. Any
attempt by an application component to use the associated connection handle after
cleanup of the underlying `ManagedConnection` should result in an exception.

The container always drives the cleanup of a `ManagedConnection` instance. The
container keeps track of created connection handles in an implementation specific
mechanism. It invokes `ManagedConnection.cleanup` when it has to invalidate all
connection handles associated with this `ManagedConnection` instance and put the

`ManagedConnection` instance back in to the pool. This may be called after the end of a connection sharing scope or when the last associated connection handle is closed for a `ManagedConnection` instance.

The invocation of the `ManagedConnection.cleanup` method on an already cleaned-up connection should not throw an exception.

The cleanup of a `ManagedConnection` instance resets its client-specific state and prepares the connection to be put back into a connection pool. The `cleanup` method should not cause the resource adapter to close the physical pipe and reclaim system resources associated with the physical connection.

An application server should explicitly call `ManagedConnection.destroy` to destroy a physical connection. An application server should destroy a physical connection to manage the size of its connection pool and to reclaim system resources.

A resource adapter should destroy all allocated system resources for this `ManagedConnection` instance when the method `destroy` is called.

6.5.4.5   Requirements

A resource adapter must provide an implementation of the `ManagedConnection` interface.

6.5.5   `ManagedConnectionMetaData`

The method `ManagedConnection.getMetaData` returns a `javax.resource.spi.ManagedConnectionMetaData` instance. The `ManagedConnectionMetaData` provides information about a `ManagedConnection` and the connected EIS instance. This information is only available to the caller of this method if a valid physical connection exists for an EIS instance.

6.5.5.1   Interface

The `ManagedConnectionMetaData` interface provides the following information about an EIS instance:

- Product name of the EIS instance
- Product version of the EIS instance
- Maximum number of concurrent connections from different processes that an EIS instance can support
- User name for this connection, as known to the EIS instance

The method `getUserName` returns the user name known to the underlying EIS instance for an active connection. The name corresponds to the resource principal under whose security context the connection to the EIS instance has been established.

6.5.5.2 Requirements

A resource adapter must provide an implementation of the `ManagedConnectionMetaData` interface. An instance of this implementation class should be returned from the `ManagedConnection.getMetaData` method.

6.5.6 ConnectionEventListener

The connector architecture provides an event callback mechanism that enables an application server to receive notifications from a `ManagedConnection` instance. An application server uses these event notifications to manage its connection pool, to clean up invalid or terminated connections, and to manage local transactions. Chapter 7, "Transaction Management" discusses local transaction-related event notifications in more detail.

An application server implements the `javax.resource.spi.ConnectionEventListener` interface. It uses the `ManagedConnection.addConnectionEventListener` method to register a connection listener with a `ManagedConnection` instance.

6.5.6.1 Interface

The following code extract specifies the `ConnectionEventListener` interface:

```
public interface javax.resource.spi.ConnectionEventListener {
    public void connectionClosed(ConnectionEvent event);
    public void connectionErrorOccurred(ConnectionEvent event);

    // Local Transaction Management related events
    public void localTransactionStarted(ConnectionEvent event);
    public void localTransactionCommitted(ConnectionEvent event);
    public void localTransactionRolledback(ConnectionEvent
event);
}
```

A `ManagedConnection` instance calls the
`ConnectionEventListener.connectionClosed` method to notify its
registered set of listeners when an application component closes a connection
handle. The application server uses this connection close event to make a decision on
whether or not to put the `ManagedConnection` instance back into the connection
pool.

The `ManagedConnection` instance calls the
`ConnectionEventListener.connectionErrorOccurred` method to notify its
registered listeners of the occurrence of a physical connection-related error. The
event notification happens just before a resource adapter throws an exception to the
application component using the connection handle.

The `connectionErrorOccurred` method indicates that the associated
`ManagedConnection` instance is now invalid and unusable. The application server
handles the connection error event notification by initiating application server-
specific cleanup (for example, removing `ManagedConnection` instance from the
connection pool) and then calling `ManagedConnection.destroy` method to
destroy the physical connection.

A `ManagedConnection` instance also notifies its registered listeners for transaction-
related events by calling the following methods—`localTransactionStarted`,
`localTransactionCommitted`, and `localTransactionRolledback`. An
application server uses these notifications to manage local transactions. See
Section 7.7, "Local Transaction Management Contract" on page 7-28 for details on
the local transaction management.

The processing of event notifications by the registered event listeners may be
synchronous or asynchronous. That is, a listener may process an event notification
immediately (as part of the notification method call) or it may defer event processing
to a later in time. The resource adapter must not assume the processing of event
notifications by its listeners to be synchronous or asynchronous.

6.5.7   `ConnectionEvent`

A `javax.resource.spi.ConnectionEvent` class provides information about the
source of a connection-related event. A `ConnectionEvent` instance contains the
following information:

- Type of the connection event
- `ManagedConnection` instance that has generated the connection event. A
  `ManagedConnection` instance is returned from the
  `ConnectionEvent.getSource` method.
- Connection handle associated with the `ManagedConnection` instance; required
  for the `CONNECTION_CLOSED` event and optional for the other event types.

- Optionally, an exception indicating a connection related error. Refer to Section 22.2, "System Exceptions" on page 22-2 for details on the system exception. Note that the exception is used for the CONNECTION_ERROR_OCCURRED notification.

This class defines the following types of event notifications:

- CONNECTION_CLOSED
- LOCAL_TRANSACTION_STARTED
- LOCAL_TRANSACTION_COMMITTED
- LOCAL_TRANSACTION_ROLLEDBACK
- CONNECTION_ERROR_OCCURRED

6.6 Error Logging and Tracing

The connector architecture provides basic support for error logging and tracing in both managed and non-managed environments. This support enables an application server to detect errors related to a resource adapter and its EIS, and to use error information for debugging.

6.6.1 ManagedConnectionFactory

The javax.resource.spi.ManagedConnectionFactory interface defines the following methods for error logging and tracing:

```
public interface javax.resource.spi.ManagedConnectionFactory
    extends java.io.Serializable {

    public void setLogWriter(java.io.PrintWriter out)
            throws ResourceException;
    public java.io.PrintWriter getLogWriter()
            throws ResourceException;
    ...
}
```

The log writer is a character output stream to which all logging and tracing messages for a ManagedConnectionFactory instance are printed.

A character output stream can be registered with a ManagedConnectionFactory instance using the setLogWriter method. A ManagedConnectionFactory implementation uses this character output stream to output error log and trace information.

An application server manages the association of a log writer with a `ManagedConnectionFactory`. When a `ManagedConnectionFactory` instance is created, the log writer is initially `null` and logging is disabled. Associating a log writer with a `ManagedConnectionFactory` instance enables logging and tracing for the `ManagedConnectionFactory` instance.

An application server administrator primarily uses the error and trace information printed on a log writer by a `ManagedConnectionFactory` instance. This information is typically system-level in nature (for example, information related to connection pooling and transactions) rather than of direct interest to application developers.

6.6.2   ManagedConnection

The `javax.resource.spi.ManagedConnection` interface defines the following methods to support error logging and tracing specific to a physical connection.

```
public interface javax.resource.spi.ManagedConnection {
    public void setLogWriter(java.io.PrintWriter out)
            throws ResourceException;
    public java.io.PrintWriter getLogWriter()
            throws ResourceException;
    ...
}
```

A newly created `ManagedConnection` instance gets the default log writer from the `ManagedConnectionFactory` instance that creates the `ManagedConnection` instance. The default log writer can be overridden by an application server using the `ManagedConnection.setLogWriter` method. The setting of the log writer on a `ManagedConnection` enables an application server to manage error logging and tracing specific to the physical connection represented by a `ManagedConnection` instance.

An application server can optionally disassociate the log writer from a `ManagedConnection` instance when this connection instance is put back into the connection pool by using `setLogWriter` and passing `null`.

6.7 Object Diagram

The following shows the object diagram for the connection management architecture. It shows invocations across the various object instances that correspond to the architected interfaces in the connection management contract, as opposed to those instances specific to implementations of the application server and the resource adapter.

To keep the diagram simple, it does not show the transaction management contract-related interfaces (`XAResource` and `LocalTransaction`) and invocations.

FIGURE 6-4 Object Diagram: Connection Management Architecture

Application
Component

Application Server

Resource Adapter

ConnectionManager

ConnectionFactory

Connection

allocateConnection

application server
specific

SecurityService
Manager

create new instance

ResourceAdapter
specific

ManagedConnectionFactory

create new instance

createManagedConnection-
matchManagedConnections
createConnectionFactory

create
new instance

create new instance

ManagedConnection

Pool
Manager

add/removeConnectionEventListener
getConnection

Transaction
Manager

application server
specific

EIS specific

ConnectionEventListener

Connection Event notifications

Architected interface
Instantiation
Implementation specific

Enterprise Information System (EIS)

6.8 Illustrative Scenarios

This section uses sequence diagrams to illustrate various interactions between the object instances involved in the connection management contract.

Some sequence diagrams include a box labeled "Application Server". This box refers to various modules and classes internal to an application server. These modules and classes communicate through contracts that are application server implementation specific.

In this section, the CCI interfaces—`javax.resource.cci.ConnectionFactory` and `javax.resource.cci.Connection`—represent connection factory and connection interfaces respectively.

The description of these sequence diagrams does not include transaction-related details. These are covered in Chapter 7, "Transaction Management".

6.8.1 Scenario: Connection Pool Management

The following object interactions are involved in the scenario shown in FIGURE 6-7:

- The application component calls the `getConnection` method on the `javax.resource.cci.ConnectionFactory` instance (returned from the JNDI lookup) to get a connection to the underlying EIS instance. Refer to Section 20.6, "JNDI Configuration and Lookup" on page 20-15 for details on the JNDI configuration and lookup.
- The `ConnectionFactory` instance initially handles the connection request from the application component in a resource adapter specific way. It then delegates the connection request to the associated `ConnectionManager` instance. The `ConnectionManager` instance has been associated with the `ConnectionFactory` instance when the `ConnectionFactory` was instantiated.

  The `ConnectionFactory` instance receives all connection request information passed through the `getConnection` method and, in turn, passes it in a form required by the method `ConnectionManager.allocateConnection`. The `ConnectionRequestInfo` parameter to the `allocateConnection` method enables a `ConnectionFactory` implementation class to pass on client-specific connection request information. This information is opaque to an application server and is used subsequently by a resource adapter to do connection matching and creation.

- The `ConnectionManager` instance (provided by the application server) handles the `allocateConnection` request by interacting with the application server specific connection pool manager. The interaction between a `ConnectionManager` instance and pool manager is internal and specific to an application server.
- The application server finds a candidate set of `ManagedConnection` instances from its connection pool. The candidate set includes all `ManagedConnection` instances that the application server considers suitable for handling the current connection allocation request. The application server finds the candidate set using its own implementation-specific structuring and lookup criteria for the connection pool. Refer to Section 6.5.3 "ManagedConnectionFactory" for guidelines of connection pool implementation by an application.
- If the application server finds no matching `ManagedConnection` instance that can best handle this connection allocation request, or if the candidate set is empty, the application server calls the `ManagedConnectionFactory.createManagedConnection` method to create a new physical connection to the underlying EIS instance. The application server passes necessary security information (as JAAS `Subject`) as part of this method invocation. For details on the security contract, refer to the `Security Management` chapter. It can also pass the `ConnectionRequestInfo` information to the resource adapter. The connection request information has been associated with the connection allocation request by the resource adapter and is used during connection creation.
- The `ManagedConnectionFactory` instance creates a new physical connection to the underlying EIS to handle the `createManagedConnection` method. This new physical connection is represented by a `ManagedConnection` instance. The `ManagedConnectionFactory` uses the security information (passed as a `Subject` instance), `ConnectionRequestInfo`, and its default set of configured properties (port number, server name) to create a new `ManagedConnection` instance. Refer to Chapter 9, "Security Contract" for more details on the `createManagedConnection` method.
- The `ManagedConnectionFactory` instance initializes the created `ManagedConnection` instance and returns it to the application server.
- The application server registers a `ConnectionEventListener` instance with the `ManagedConnection` instance, enabling it to receive notifications for events on this connection. The application server uses these event notifications to manage connection pooling and transactions.
- The `ManagedConnection` instance obtains its log writer (for error logging and tracing support) from the `ManagedConnectionFactory` instance that created this connection. However, an application server can set a new log writer with a `ManagedConnection` instance to do additional error logging and tracing at the level of a `ManagedConnection`.
- The application server does the necessary transactional setup for the `ManagedConnection` instance. Chapter 7, "Transaction Management" explains this step in more detail.

- Next, the application server calls `ManagedConnection.getConnection` method to get an application level connection handle of type `javax.resource.cci.Connection`. A `ManagedConnection` instance uses the `Subject` and `ConnectionRequestInfo` parameters to the `getConnection` method to change the state of the `ManagedConnection`.

  Calling the `getConnection` method does not necessarily create a new physical connection to the EIS instance. Calling `getConnection` produces a temporary connection handle that is used by an application component to access the underlying physical connection. The actual underlying physical connection is represented by a `ManagedConnection` instance.

- The application server returns the connection handle to the resource adapter. The resource adapter then passes the connection handle to the application component that initiated the connection request.

FIGURE 6-5 OID: Connection Pool Management with New Connection Creation

Application Component

Resource Adapter
javax.resource.cci.
ConnectionFactory

Application Server

Transaction Manager

ManagedConnection Factory

Resource Adapter
Managed Connection

XAResource

getConnection

ConnectionManager.allocateConnection

Application server looks up a candidate connection set from the connection pool

Note: Following steps happen if no matching connection is found or if candidate set is empty

createManagedConnection

create a new instance

addConnectionEventListener(ConnectionEventListener)

Optional: setLogWriter(PrintWriter)

Application server performs transactional setup for the ManagedConnection instance. For example, application server performs following setup for JTA transactions.

getXAResource

Transaction.enlistResource(XAResource)

XAResource.start(XID)

getConnection(Subject, ConnectionRequestInfo)

return javax.resource.cci.Connection

return javax.resource.cci.Connection

6.8.2   Scenario: Connection Matching

FIGURE 6-6 shows the object interactions for a connection matching scenario—that is, a scenario in which the application server finds a non-empty candidate connection set and calls the resource adapter to do matching on the candidate set. The following steps are involved in this scenario:

1. The application server handles the connection allocation request by creating a candidate set of `ManagedConnection` instances from the connection pool. The candidate set includes the `ManagedConnection` instances that the application server considers suitable for handling the current connection allocation request. The application server finds this candidate set using its own implementation-specific structuring and lookup criteria for the connection pool. Refer to Section 6.5.3 "ManagedConnectionFactory" for guidelines on connection pool implementation by an application.

2. The application server calls the `ManagedConnectionFactory.matchManaged-Connections` method to enable the resource adapter to do the connection matching. It passes the candidate connection set, security information (as a `Subject` instance associated with the current connection request), and any `ConnectionRequestInfo`.

3. The `ManagedConnectionFactory` instance matches the candidate set of connections using the criteria known internally to the resource adapter. The `matchManagedConnections` method returns a `ManagedConnection` instance that the resource adapter considers to be an acceptable match for the current connection allocation request.

4. The application server can set a new log writer with the `ManagedConnection` instance to do error logging and tracing at the level of the `ManagedConnection`.

5. The application server does the necessary transactional setup for the `ManagedConnection` instance. Chapter 7, "Transaction Management" explains this step in more detail.

6. The application server calls the `ManagedConnection.getConnection` method to get a new application level connection handle.

7. The `ManagedConnection.getConnection` method implementation uses the `Subject` parameter and any `ConnectionRequestInfo` to set the state of the `ManagedConnection` instance based on the current connection allocation request. Refer to Section 9.1.9, "ManagedConnection" on page 9-14 for details if a resource adapter implements support for re-authentication of a `ManagedConnection` instance.

8. The application server returns the connection handle to the resource adapter. The resource adapter then passes the connection handle to the application component that initiated the connection request.

FIGURE 6-6 OID: Connection Pool Management with Connection Matching
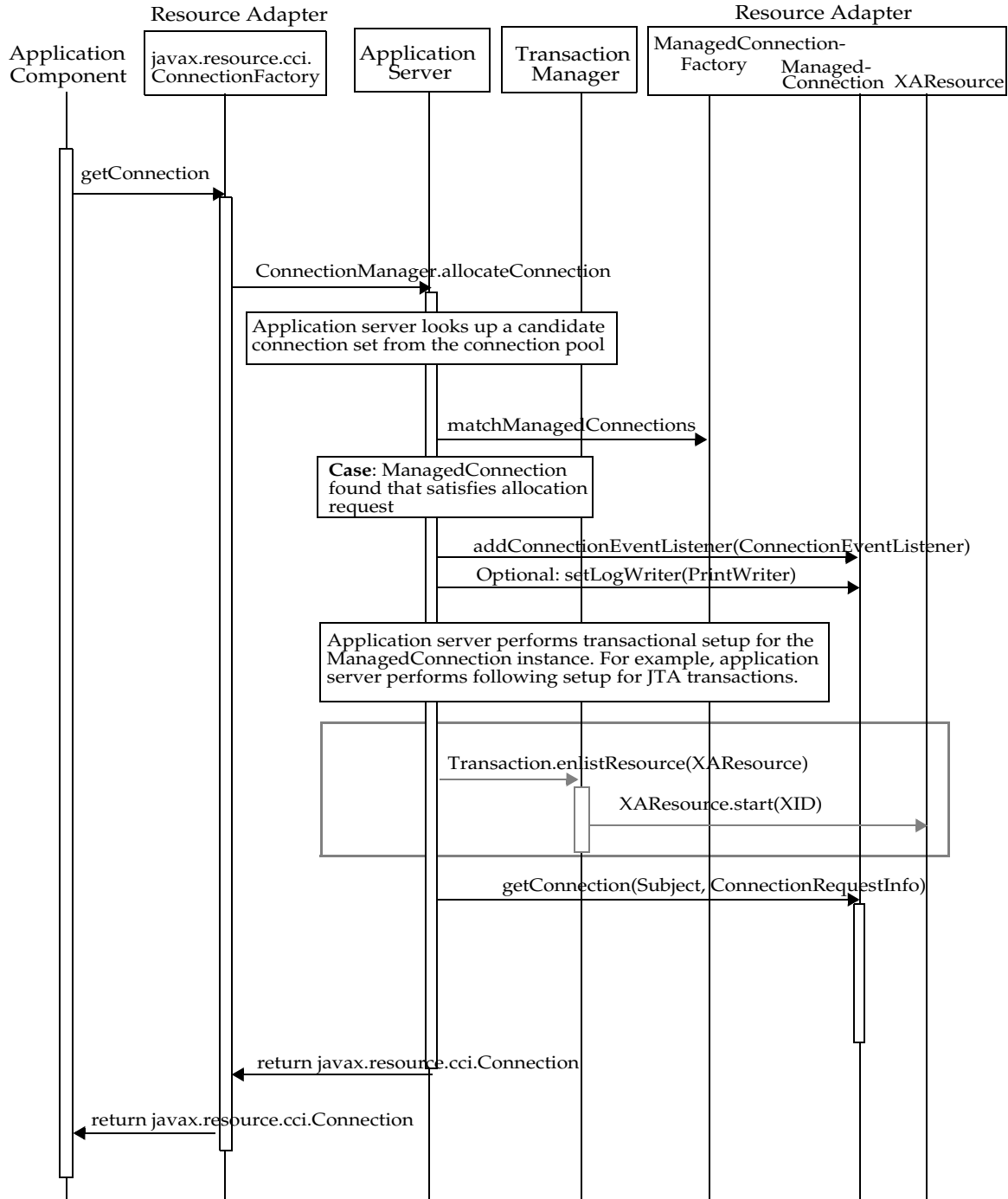
6.8.3 Scenario: Connection Event Notifications and Connection Close

For each `ManagedConnection` instance in the pool, the application server registers a `ConnectionEventListener` instance to receive close and error events on the connection. This scenario explains how the connection event callback mechanism enables an application server to manage connection pooling.

The scenario involves the following steps (see FIGURE 6-7) when an application component initiates a connection close:

1. The application component releases an allocated connection handle using the `close` method on the `javax.resource.cci.Connection` instance. The `Connection` instance delegates the close method to the associated `ManagedConnection` instance. The delegation happens through an association between `ManagedConnection` instance and the corresponding connection handle `Connection` instance. The mechanism by which this association is achieved is specific to the implementation of a resource adapter.

2. The connection management contract places a requirement that a `ManagedConnection` instance must not alter the state of a physical connection while handling the connection close.

3. The `ManagedConnection` instance notifies all its registered listeners of the application's connection close request using the `ConnectionEventListener.connectionClosed` method. It passes a `ConnectionEvent` instance with the event type set to `CONNECTION_CLOSED`.

4. On receiving the connection close event notification, the application server performs the transaction management-related cleanup of the `ManagedConnection` instance. Refer to FIGURE 6-7 for details on the cleanup of a `ManagedConnection` instance participating in a JTA transaction.

5. The application server also uses the connection close event notification to manage its connection pool. On receiving the connection close notification, the application server calls the `ManagedConnection.cleanup` method (depending on whether the `ManagedConnection` is shared and the presence of other active connection handles) to perform cleanup on the `ManagedConnection` instance that raised the connection close event. The application server-initiated cleanup of a `ManagedConnection` instance prepares this `ManagedConnection` instance to be reused for subsequent connection requests. See Section 7.9, "Connection Sharing" on page 7-32 for a discussion of connection sharing and its implications on `ManagedConnection` cleanup.

6. After initiating the necessary cleanup for the `ManagedConnection` instance, the application server puts the `ManagedConnection` instance back into the connection pool. The application server should be able to use this available `ManagedConnection` instance to handle future connection allocation requests from application components.

6.8.3.1   Connection Cleanup

The application server can also initiate cleanup of a `ManagedConnection` instance when the container terminates the application component instance that has the corresponding connection handle. The application server should call `ManagedConnection.cleanup` to initiate the connection cleanup. After the cleanup, the application server puts the `ManagedConnection` instance into the pool to serve future allocation requests.

6.8.3.2   Connection Destroy

To manage the size of the connection pool, the application server can call `ManagedConnection.destroy` method to destroy a `ManagedConnection`. A `ManagedConnection` instance handles this method call by closing the physical connection to the EIS instance and releasing all system resources held by this instance.

The application server also calls `ManagedConnection.destroy` when it receives a connection error event notification that signals a fatal error on the physical connection.

FIGURE 6-7 OID: Connection Event Notification

Application Component

Resource Adapter
javax.resource.cci. Connection

Application Server

Transaction Manager

Resource Adapter
ManagedConnection Factory

Managed Connection

XAResource

close()

Internal: Resource Adapter implementation specific

ManagedConnection notifies all registered ConnectionEventListeners

connectionClosed(ConnectionEvent: CONNECTION_CLOSED)

Case: JTA Transaction

Transaction.delistResource(XAResource)

XAResource.end(XID)

ManagedConnection.cleanup

Application Server returns ManagedConnection instance to the connection pool

6.9 Architecture: Non-Managed Environment

The connection management contract enables a resource adapter to be used in a two-tier application directly from an application client.

In a non-managed application scenario, the `ConnectionManager` implementation class may be provided either by a resource adapter (as a default `ConnectionManager` implementation) or by application developers. Note that a default implementation of the `ConnectionManager` should be defined for a resource adapter (in terms of the functionality provided and third-party components added) only at development time.

The default `ConnectionManager` instance interposes on the connection request and delegates the request to the `ManagedConnectionFactory` instance. The `ManagedConnectionFactory` creates a physical connection (represented by a `ManagedConnection` instance) to the underlying EIS. The `ConnectionManager` gets a connection handle (of type `javax.resource.cci.Connection` for CCI) from the `ManagedConnection` and returns it to the connection factory. The connection factory returns the connection handle to the application.

A resource adapter supports interactions (shown as light shaded lines in the following figure) between its internal objects in an implementation-specific way. For example, a resource adapter can use the connection event listening mechanism as

part of its `ManagedConnection` implementation for connection management. However, the resource adapter is not required to use the connection event mechanism to drive its internal interactions.

FIGURE 6-8   Architecture Diagram: Non-Managed Application Scenario
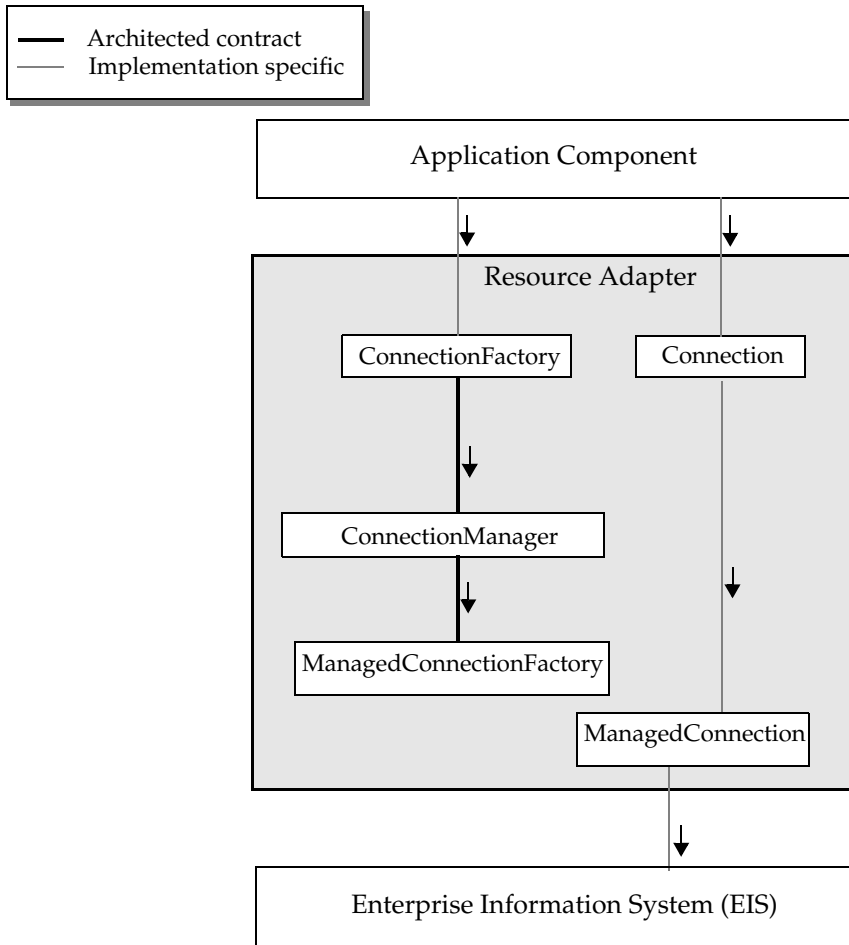
6.9.1   Scenario: Programmatic Access to `ConnectionFactory`

To maintain the consistency of the application programming model across both managed and non-managed environments, application code should use the JNDI namespace to look-up a connection factory instance.

The following code extract shows how an application client accesses a connection factory instance in a non-managed environment. The code extract does not show the use of JNDI. It is used as an example to illustrate the use of ManagedConnectionFactory and ConnectionFactory interfaces in the application code. Refer to section Section 20.6, "JNDI Configuration and Lookup" on page 20-15 for details on JNDI configuration and lookup.

```
// Application Client Code
// Create an instance of the ManagedConnectionFactory
// implementation class passing in initialization parameters
// (if any) for this instance
com.myeis.ManagedConnectionFactoryImpl mcf =
    new com.myeis.ManagedConnectionFactoryImpl(...);

// Set properties on the ManagedConnectionFactory instance
// Note: Properties are defined on the implementation class
// and not on the javax.resource.spi.ManagedConnectionFactory
// interface
mcf.setServerName(...);
mcf.setPortNumber(...);

// set remaining properties
...

// Get access to connection factory. The ConnectionFactory
instance
// gets initialized with the default ConnectionManager provided
// by the resource adapter
javax.resource.cci.ConnectionFactory cxf =
    (javax.resource.cci.ConnectionFactory)
    mcf.createConnectionFactory();

// Get a connection using the ConnectionFactory instance
javax.resource.cci.Connection cx = cxf.getConnection(...);

// use connection to access the underlying EIS instance
...

// Close the connection
cx.close();
```

6.9.2   Scenario: Connection Creation in Non-Managed Application Scenario

The following object interactions are involved in the scenario shown in FIGURE 6-9:

- The application client calls a method on the
  `javax.resource.cci.ConnectionFactory` instance, returned from the JNDI
  lookup, to get a connection to the underlying EIS instance.
- The `ConnectionFactory` instance delegates the connection request from the
  application to the default `ConnectionManager` instance. The resource adapter
  provides the default `ConnectionManager` implementation.
- The `ConnectionManager` instance creates a new physical connection to the
  underlying EIS instance by calling the
  `ManagedConnectionFactory.createManagedConnection` method.
- The `ManagedConnectionFactory` instance handles the
  `createManagedConnection` method by creating a new physical connection to
  the underlying EIS, represented by a `ManagedConnection` instance. The
  `ManagedConnectionFactory` uses the security information, passed as a
  `Subject` instance, any `ConnectionRequestInfo` instance, and its configured
  set of properties, such as port number, server name, to create a new
  `ManagedConnection` instance.
- The `ManagedConnectionFactory` initializes the state of the created `Managed–
  Connection` instance and returns it to the default `ConnectionManager` instance.
- The `ConnectionManager` instance calls the
  `ManagedConnection.getConnection` method to get an application-level
  connection handle. Calling the `getConnection` method does not necessarily
  create a new physical connection to the EIS instance. Calling `getConnection`
  produces a temporary handle that is used by an application to access the
  underlying physical connection. The actual underlying physical connection is
  represented by a `ManagedConnection` instance.

- The `ConnectionManager` instance returns the connection handle to the `ConnectionFactory` instance, which then returns the connection to the application that initiated the connection request.

FIGURE 6-9   OID: Connection Creation in a Non-Managed Application Scenario
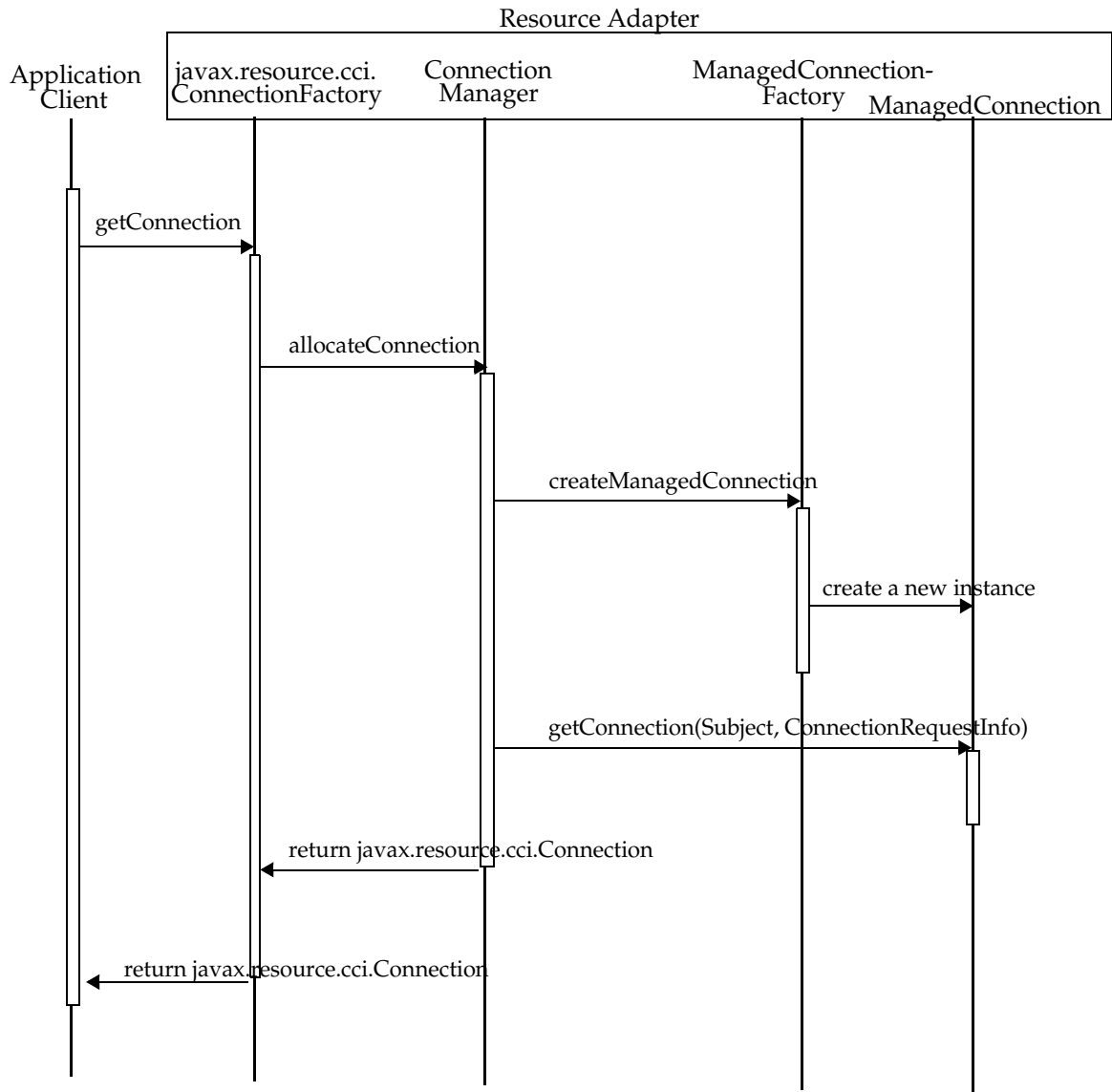
6.10   Requirements

This section outlines requirements for the connection management contract.

6.10.1   Resource Adapter

The requirements for a resource adapter are as follows:

- A resource adapter must provide implementations of the following interfaces:

  - `javax.resource.spi.ManagedConnectionFactory`
  - `javax.resource.spi.ManagedConnection`
  - `javax.resource.spi.ManagedConnectionMetaData`

- The `ManagedConnection` implementation provided by a resource adapter must use the following interface and classes to provide support to an application server for connection management and transaction management, as explained later:

  - `javax.resource.spi.ConnectionEvent`
  - `javax.resource.spi.ConnectionEventListener`

  To support non-managed environments, a resource adapter is not required to use the above two interfaces to drive its internal object interactions.

- A resource adapter must provide support for basic error logging and tracing by implementing the following methods:

  - `ManagedConnectionFactory.set/getLogWriter`
  - `ManagedConnection.set/getLogWriter`

- A resource adapter must provide a default implementation of the `javax.resource.spi.ConnectionManager` interface. The implementation class comes into play when a resource adapter is used in a non-managed two-tier application scenario. In an application server-managed environment, the resource adapter must not use the default `ConnectionManager` implementation class.

  A default implementation of `ConnectionManager` enables the resource adapter to provide services specific to itself. These services can include connection pooling, error logging and tracing, and security management. The default `ConnectionManager` delegates to the `ManagedConnectionFactory` the creation of physical connections to the underlying EIS.

- In a managed environment, with the exception of application client containers, a resource adapter must not asynchronously (that is, using a separate thread other than the application thread) call application objects other than message-driven beans. However, this restriction does not apply to a non-managed scenario, as well as application client containers. A resource adapter deployer may use the

`ResourceAdapter` JavaBean to configure the resource adapter during its deployment to set the desired behavior, based on the requirements of the deployment environment.

- A resource adapter is not allowed to support its own internal connection pooling in a managed environment. In this case, the application server is responsible for connection pooling. However, a resource adapter may multiplex connections (one or more `ManagedConnection` instances per physical connection) over a single physical pipe transparent to the application server and components.

  In a non-managed two tier application scenario, a resource adapter is allowed to support connection pooling internal to the resource adapter.

6.10.2   Application Server

The requirements for an application server are as follows:

- An application server must use the interfaces defined in the connection management contract to use services provided by a resource adapter. These interfaces are as follows:

  - `javax.resource.spi.ManagedConnectionFactory`
  - `javax.resource.spi.ManagedConnection`
  - `javax.resource.spi.ManagedConnectionMetaData`

- An application server must provide an implementation of the `javax.resource.spi.ConnectionManager` interface. This implementation should not be specific to any particular type of resource adapter, EIS, or connection factory interface.

- An application server must implement the `javax.resource.spi.-ConnectionEventListener` interface and to register `ConnectionEventListener` with a resource adapter to get connection-related event notifications. An application server uses these event notifications to do its pool management, transaction management, and connection cleanup.

- An application server must use the following interfaces (supported by the resource adapter) to provide basic error logging and tracing for its configured set of resource adapters:

  - `ManagedConnectionFactory.set/getLogWriter`
  - `ManagedConnection.set/getLogWriter`

- An application server must use the `javax.resource.spi.ConnectionManager` hook-in mechanism to provide its specific quality-of-services. The connector architecture does not specify the set of services the application server provides, nor does it specify how the application server implements these services.

CHAPTER **7**

Transaction Management

This chapter specifies the transaction management contract between an application server (and supported transaction manager) and an EIS resource manager.

This chapter focuses only on the system-level aspects of transaction management. The Java EE component model specifications describe the application level transaction model. For example, the EJB specification (see "Enterprise JavaBeans (EJB) Specification, version 3.2" on page E-1) specifies the transaction model for EJB components.

7.1 Overview

The following figure shows an application component deployed in a container provided by an application server. The application component performs transactional access to multiple resource managers. The application server uses a transaction manager that takes the responsibility of managing transactions across multiple resource managers.

FIGURE 7-1   Transaction Management Contract
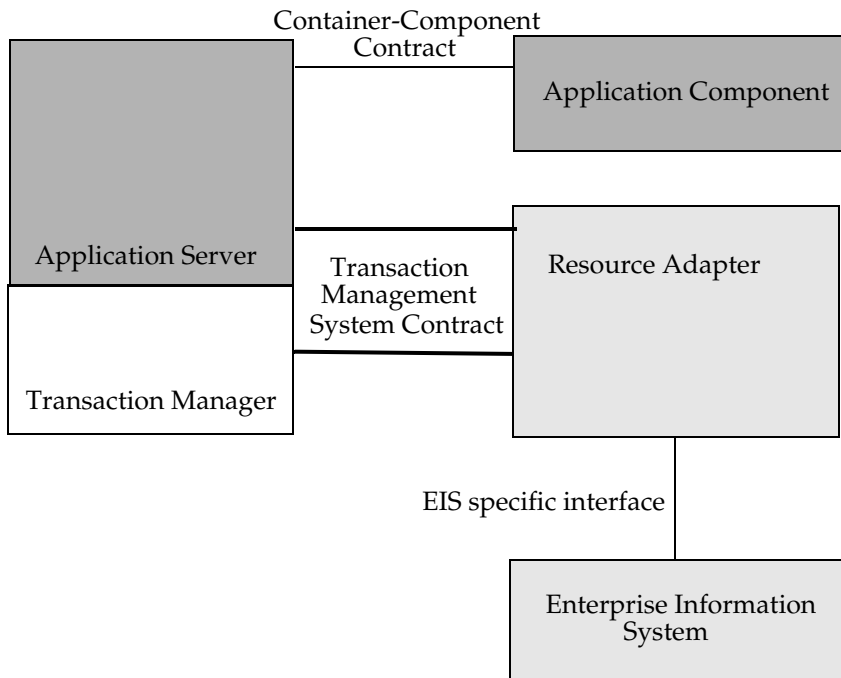
A resource manager can support two types of transactions:

■   A transaction that is controlled and coordinated by a transaction manager external to the resource manager. This document refers to such a transaction as JTA or XA transaction.

■   A transaction that is managed internal to a resource manager. The coordination of such transactions involves no external transaction managers. This document refers to such transactions as RM local transactions (or local transactions).

A transaction manager coordinates transactions across multiple resource managers. It also provides additional low-level services that enable transactional context to be propagated across systems. The services provided by a transaction manager are not visible directly to the application components.

The connector architecture defines a transaction management contract between an application server and a resource adapter and its underlying resource manager. The transaction management contract has two parts, depending on the type of transaction:

- a JTA `javax.transaction.xa.XAResource` based contract between a transaction manager and a resource manager
- a local transaction management contract

These contracts enable an application server to provide the infrastructure and runtime environment for transaction management. Application components rely on this transaction infrastructure to support their component-level transaction model. Connection Handles obtained in the context of an application component should not be passed between application component boundaries, especially if the connection handles are involved in a transaction, and an application server is not required to support this usage.

7.2 Transaction Management Scenarios

This section uses a set of scenarios to present an overview of the transaction management architecture.

7.2.1 Transactions Across Multiple Resource Managers

In the following figure, an application client invokes EJB component X. EJB X accesses transaction programs managed by a TP system and calls EJB Y to access an ERP system.

FIGURE 7-2   Scenario: Transactions Across Multiple Resource Managers
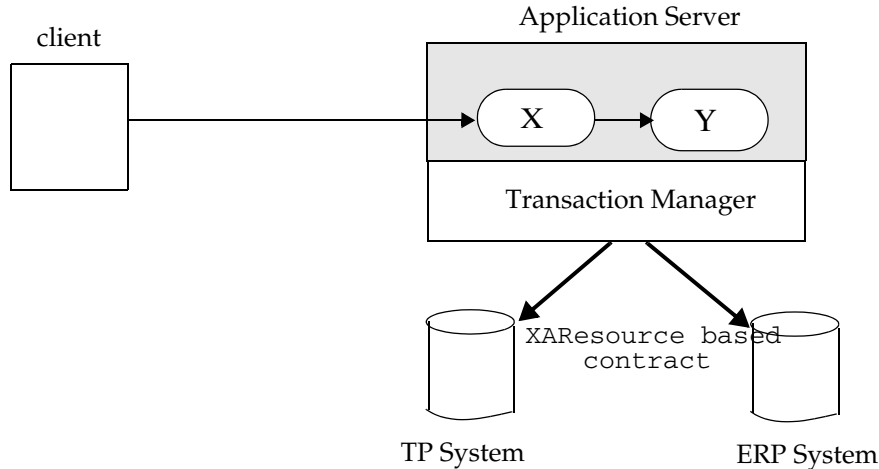
The application server uses a transaction manager to support a transaction management infrastructure that enables an application component to perform transactional access across multiple EIS resource managers. The transaction manager manages transactions across multiple resource managers and supports propagation of the transaction context across distributed systems.

The transaction manager supports a JTA XAResource-based transaction management contract with a resource adapter and its underlying resource manager. The ERP system supports JTA transactions by implementing an XAResource interface through its resource adapter. The TP system also implements an XAResource interface. This interface enables the two resource managers to participate in transactions that are coordinated by an external transaction manager. The transaction manager uses the XAResource interface to manage transactions across the two underlying resource managers.

The EJBs X and Y access the ERP and TP system using the respective client access API for the two systems. Behind the scenes, the application server enlists the connections to both systems, obtained from their respective resource adapters, as part of the transaction. When the transaction commits, the transaction manager performs a two-phase commit protocol across the two resource managers, ensuring that all read/write access to resources managed by both the TP system and ERP system is either entirely committed or entirely rolled back.

7.2.2   Local Transaction Management

The transactions are demarcated either by the container (called container-managed demarcation) or by a component (called component-managed demarcation). In component-managed demarcation, an application component can use the JTA `UserTransaction` interface or a transaction demarcation API specific to an EIS (for example, JDBC transaction demarcation using `java.sql.Connection`).

The EJB specification requires an EJB container to support both container-managed and component-managed transaction demarcation models. The JSP and servlet specifications require a web container to support component-managed transaction demarcation.

If multiple resource managers participate in a transaction, the EJB container uses a transaction manager to coordinate the transaction. The contract between the transaction manager and resource manager is defined using the `XAResource` interface.

If a single resource manager instance participates in a transaction (either component-managed or container-managed), the container has two choices:

- Using the transaction manager to manage this transaction. The transaction manager uses one-phase commit-optimization, described in Section 7.6.1 "Scenarios Supported", to coordinate the transaction for this single resource manager instance.
- Letting the resource manager coordinate this transaction internally without involving an external transaction manager.

If an application accesses a single resource manager using an XA transaction, it has more performance overhead compared to using a local transaction. The overhead is due to the involvement of an external transaction manager in the coordination of the XA transaction.

To avoid the overhead of using an XA transaction in a single resource manager scenario, the application server may optimize this scenario by using a local transaction instead of an XA transaction. This scenario is shown in the following figure.

FIGURE 7-3   Scenario: Local Transaction on a Single Resource Manager
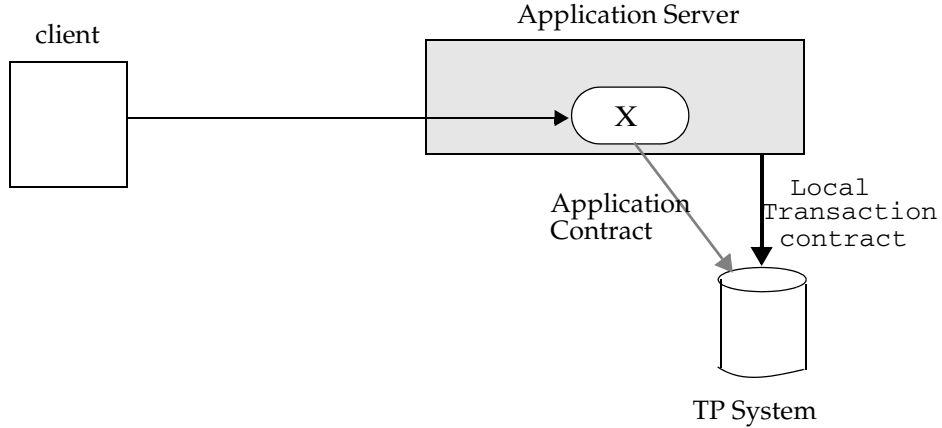
7.3   Transaction Management Contract

This section specifies the transaction management contract. The transaction management contract builds on the connection management contract specified in Chapter 6, "Connection Management".

The following figure shows the interfaces and flows in the transaction management contract. It does not show the interfaces, classes, and flows that are the same in the connection management contract.

FIGURE 7-4   Architecture Diagram: Transaction Management
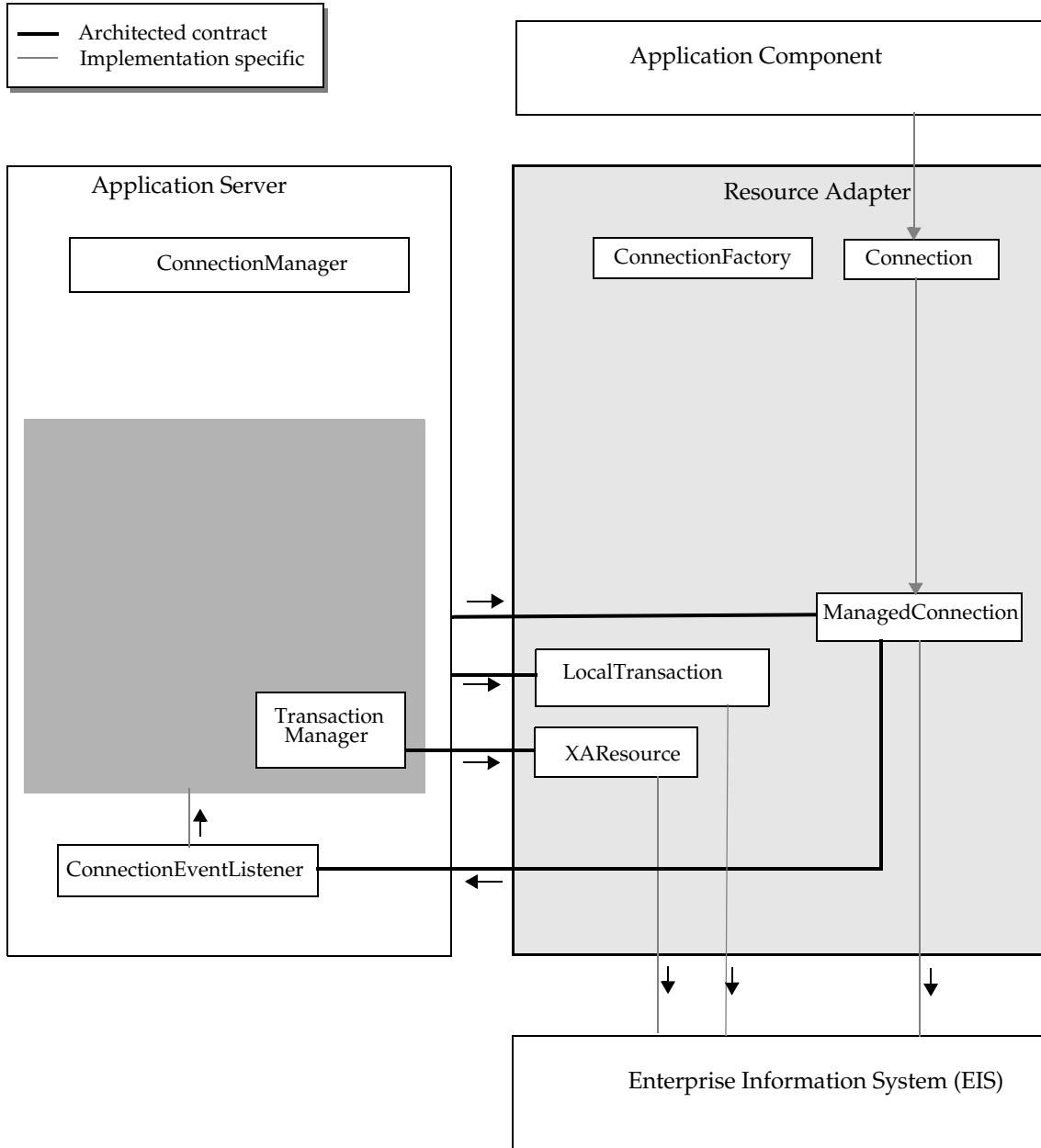
7.3.1 Interface: `ManagedConnection`

The `javax.resource.spi.ManagedConnection` instance represents a physical connection to an EIS and acts as a factory for connection handles.

The following code extract shows the methods on the `ManagedConnection` interface that are defined specifically for the transaction management contract:

```
public interface javax.resource.spi.ManagedConnection {
    public XAResource getXAResource() throws ResourceException;
    public LocalTransaction getLocalTransaction()
            throws ResourceException;
    ...
}
```

A `ManagedConnection` instance provides access to a pair of interfaces: `javax.transaction.xa.XAResource` and `javax.resource.spi.LocalTransaction`.

Depending on the transaction support level of a resource adapter, these methods should raise appropriate exceptions. For example, if the transaction support level for a resource adapter is `NoTransaction`, an invocation of `getXAResource` method should throw a `ResourceException`. Refer to Chapter 22, "Exceptions" for details on the exception hierarchy.

The following figure illustrates this concept:

FIGURE 7-5 ManagedConnection Interface for Transaction Management
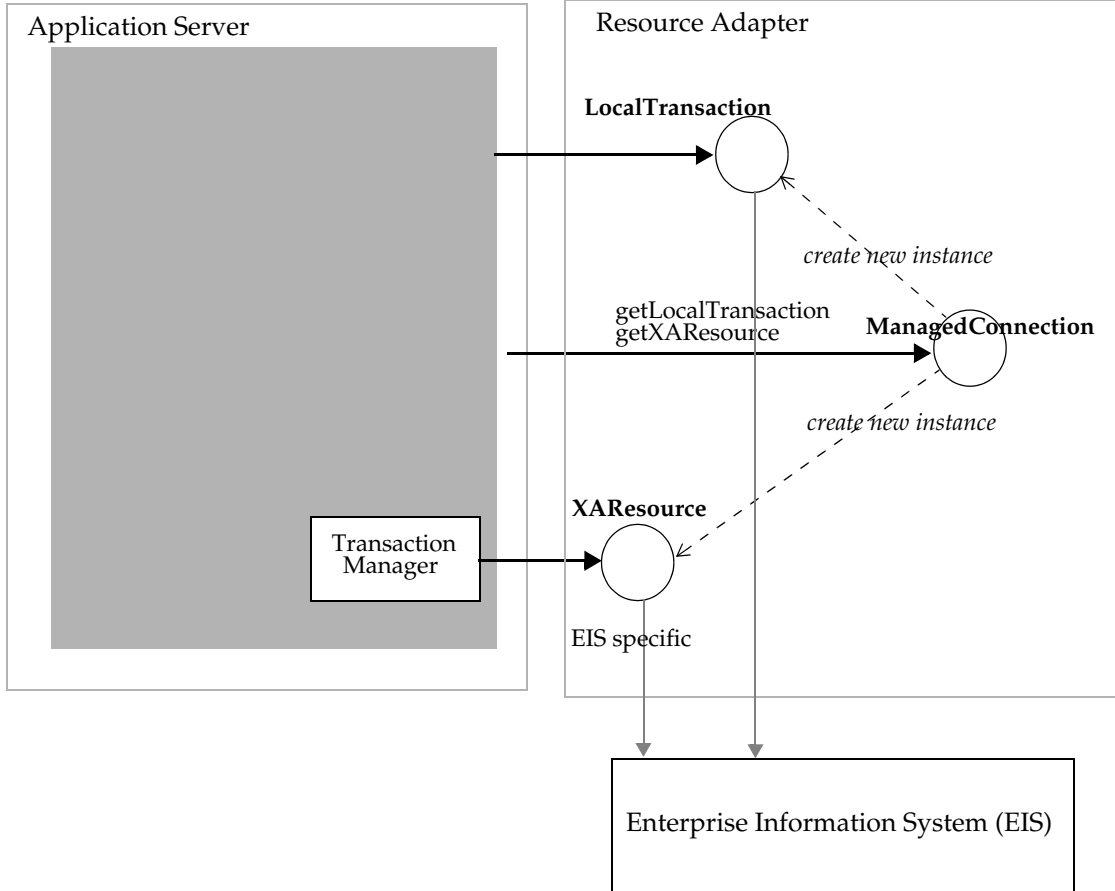
The transaction manager uses the XAResource interface to associate and dissociate a transaction with the underlying EIS resource manager instance and to perform a two-phase commit protocol. The transaction manager does not directly use the ManagedConnection interface. The next section describes the XAResource interface in more detail.

The application server uses the LocalTransaction interface to manage local transactions.

7.3.2 Interface: `XAResource`

The `javax.transaction.xa.XAResource` interface is a Java mapping of the industry standard `XA` interface based on `X/Open CAE specification` (see "X/Open CAE Specification -- Distributed Transaction Processing: the XA Specification, X/Open document" on page E-1).

The following code extract shows the interface specification for the `XAResource` interface. For more details and API documentation, refer to the JTA (see "Java Transaction API (JTA) Specification, version 1.2" on page E-1) and XA (see "X/Open CAE Specification -- Distributed Transaction Processing: the XA Specification, X/Open document" on page E-1) specifications:

```
public interface javax.transaction.xa.XAResource {
    public void commit(Xid xid, boolean onePhase) throws
XAException;
    public void end(Xid xid, int flags) throws XAException;
    public void forget(Xid xid) throws XAException;
    public int prepare(Xid xid) throws XAException;
    public Xid[] recover(int flag) throws XAException;
    public void rollback(Xid xid) throws XAException;
    public void start(Xid xid, int flags) throws XAException;
}
```

7.3.2.1 Implementation

A resource adapter for an EIS resource manager implements the `XAResource` interface. This interface enables the resource manager to participate in transactions that are controlled and coordinated by an external transaction manager. The transaction manager uses the `XAResource` interface to communicate transaction association, completion, and recovery to the resource manager.

A resource adapter typically implements the `XAResource` interface using a low-level library available for the underlying EIS resource manager. This low-level library either supports a native implementation of the `XA` interface or provides a proprietary vendor-specific interface for transaction management.

A resource adapter is responsible for maintaining a 1-1 relationship between the `ManagedConnection` and `XAResource` instances. Each time a `ManagedConnection.getXAResource` method is called, the same `XAResource` instance has to be returned.

A transaction manager can use any `XAResource` instance (if it refers to the proper resource manager instance) to initiate transaction completion. The `XAResource` instance used during the transaction completion process need not be the one initially enlisted with the transaction manager for this transaction.

7.3.3 Interface: `LocalTransaction`

The following code extract shows the `javax.resource.spi.LocalTransaction` interface:

```
public interface javax.resource.spi.LocalTransaction {
    public void begin() throws ResourceException;
    public void commit() throws ResourceException;
    public void rollback() throws ResourceException;
}
```

A resource adapter implements the `LocalTransaction` interface to provide support for local transactions that are performed on the underlying resource manager. An application server uses the `LocalTransaction` interface to manage local transactions for a resource manager.

Section 7.7.1 "Interface: LocalTransaction" has more details on the local transaction management contract.

7.4 Relationship to JTA and JTS

The Java^{TM} Transaction API (JTA) (see "Java Transaction API (JTA) Specification, version 1.2" on page E-1) is a specification of interfaces between a transaction manager and the other parties involved in a distributed transaction processing system: application programs, resource managers, and an application server.

The Java^{TM} Transaction Service (JTS) API is a Java binding of the Common Object Request Broker Architecture (CORBA) Object Transaction Service (OTS) 1.1 specification. JTS provides transaction interoperability using the standard Internet Inter-ORB Protocol (IIOP) for transaction propagation between servers. The JTS API is intended for vendors who implement transaction processing infrastructure for enterprise middleware. For example, an application server vendor can use a JTS implementation as the underlying transaction manager.

7.4.1 JTA Interfaces

The application server uses the `javax.transaction.TransactionManager` and `javax.transaction.Transaction` interfaces, specified in the JTA specification, for its contract with the transaction manager.

The application server uses the `javax.transaction.TransactionManager` interface to control the transaction boundaries on behalf of the application components that are being managed by the application server. For example, an EJB container manages the transaction states for transactional EJB components. The EJB container uses the `TransactionManager` interface to demarcate transaction boundaries based on the calling thread's transaction context.

The application server also uses the `javax.transaction.Transaction` interface to enlist and delist transactional connections with the transaction manager. This enables the transaction manager to coordinate transactional work performed by all enlisted resource managers within a transaction.

7.5 Object Diagram

The following figure shows the object instances and their interactions related to transaction management. Since the transaction management contract builds upon the connection management contract, the following diagram does not show object interactions that have already been discussed in Chapter 6, "Connection Management".

FIGURE 7-6   Object Diagram: Transaction Management

Application
Component

Application Server

ConnectionManager

Resource Adapter

Connection

Resource
Adapter specific

create
new instance

LocalTransaction

create new instance

ManagedConnection

Pool
Manager

getLocalTransaction
getXAResource

create new instance

XAResource

Transaction
Manager

EIS specific

EIS specific

ConnectionEventListener

Connection Event notifications

Architected interface
Instantiation
Implementation specific

Enterprise Information System (EIS)

7.6   XAResource-based Transaction Contract

This section specifies detailed requirements for a resource manager and a transaction manager for the `XAResource`-based transaction management contract. In this section, the following abbreviations are used: RM (Resource Manager), TM (Transaction Manager), 1PC (one-phase commit protocol), and 2PC (two-phase commit protocol).

7.6.1 Scenarios Supported

The following table specifies various transaction management scenarios and mentions whether these scenarios are within the scope of the connector architecture.

Description	Supported / NotSupported
TM does two-phase commit (2PC) on RMs that support two-phase commit (as defined in RM's requirements for `XAResource` implementation in the subsection below) Examples of RM: Oracle and DB2 installations that support 2PC in their `XAResource` implementations.	Supported based on TM's requirement to be JTA/JTS and X/Open compliant, and RM's support for 2PC in the `XAResource` interface.
TM does one-phase commit (1PC) optimization on the only RM involved in a transaction. RM supports 2PC in its `XAResource` implementation (as defined in RM's requirements for the `XAResource` implementation in the subsection below). Example of RM: DB2 installation that supports 2PC in its `XAResource` implementation.	Supported based on TM's requirement to be JTA/JTS and X/Open compliant, and RM's support for the `XAResource` interface. **Note**: This scenario will also work if TM does 2PC on RM.
TM does one-phase commit optimization on the only RM involved in a transaction. RM does not support 2PC but supports 1PC in its `XAResource` implementation. Example of RM: ERP system or mainframe TP system that does not support 2PC, but implements 1PC in its XAResource implementation as defined in the RM's requirements for 1PC.	Supported by requiring that TM must support 1PC optimization. A successful transaction coordination of 1PC only RM comes as a result of required 1PC optimization for a TM. The rationale behind this requirement is that this scenario will be an important scenario to support for the connector architecture.
TM does last-resource commit optimization across multiple RMs involved in a transaction—RMs that support 2PC (for example: Oracle and DB2) and a single RM that supports only 1PC (for example: an ERP system).	Out of the scope of the connector architecture specification
More than one RM that support only 1PC involved in a transaction with none or multiple 2PC enabled RMs	Out of the scope of the connector architecture specification

7.6.2 Resource Adapter Requirements

The connector architecture does not require that all resource adapters must support JTA `XAResource` based transaction contract.

If a resource adapter decides to support an `XAResource` based contract, then the connector architecture places certain requirements on a resource adapter and its underlying resource manager (RM).

The following requirements refer to a resource adapter and its resource manager together as a resource manager (RM). The division of responsibility between a resource adapter and its underlying resource manager for supporting the transaction contract is implementation-specific and is out of the scope of the connector architecture.

These requirements assume that a transaction manager (TM) supports JTA/XA and JTS requirements.

The following set of requirements are based on the JTA and XA specifications and should be read in conjunction with these specifications. These detailed requirements are included in this document to clearly specify the requirements from the connector architecture perspective.

7.6.2.1 General

- If an RM supports an `XAResource` contract, then it must support the one-phase commit protocol by implementing `XAResource.commit` when the boolean flag `onePhase` is set to `True`. The RM is not required to implement the two-phase commit protocol support in its `XAResource` implementation.
- However, if an RM supports the two-phase commit protocol, then the RM must use the `XAResource` interface for supporting the two-phase commit protocol.
- An RM is allowed to combine the implementation of 2PC protocol with 1PC optimization by implementing `XAResource.commit(onePhase=True)` in addition to the implementation requirements for 2PC.

7.6.2.2 One-phase Commit

- An RM should allow `XAResource.commit (onePhase=True)` even if it has not received `XAResource.prepare` for the transaction branch.
- If the RM fails to commit a transaction during a 1PC commit, then the RM should throw one of the `XA_RB*` exceptions. In the exception case, an RM should roll back the transaction branch's work and release all held RM resources.
- The RM is responsible for deciding the outcome of a transaction branch on an `XAResource.commit` method. The RM can discard knowledge of the transaction branch once it returns from the `commit` call.

- The RM is not required to maintain knowledge of transaction branches to support failure recovery for the TM.
- If an `XAResource.prepare` method is called on an RM that supports only one-phase commit, then the RM should throw an `XAException` with `XAER_PROTO or XA_RB*` flag.
- The RM should return an empty list of XIDs for `XAResource.recover`, because the RM is not required to maintain stable knowledge about transaction branches.

7.6.2.3 Two-phase Commit

- If the RM supports 2PC, then its implementation of 2PC must be compliant with the 2PC protocol definition with presumed rollback as specified in the OSI TP (Transaction Protocol defined by ISO (ISO92)) specification.
- The RM must implement the `XAResource.prepare` method and must be able to report whether it can guarantee its ability to commit the transaction branch. If the RM reports that it can, the RM must hold and record in a stable way all the resources necessary to commit the branch. It must hold all these resources until the TM directs it to commit or rollback the branch.
- An RM that reports a heuristic completion to the TM must not discard its knowledge of the transaction branch. The RM should discard its knowledge of the branch only when the TM calls `XAResource.forget`. The RM must notify the TM of all heuristic decisions.
- On the TM's `XAResource.commit` and `XAResource.rollback` calls, the RM is allowed to report through an `XAException` that it has heuristically completed the transaction branch. This feature is optional.

A TM supporting the OSI TP specification uses the one-phase commit optimization by default to manage an RM that is the only resource involved in the transaction. The mechanism to identify to the TM a particular RM that only supports 1PC is beyond the scope of this specification.

7.6.2.4 Transaction Association and Calling Protocol

- The RM `XAResource` implementation must support `XAResource.start` and `XAResource.end` for association and disassociation of a transaction, as represented by, unique XID, with recoverable units of work being done on the RM.
- The RM must ensure that the TM invokes `XAResource` calls in the legal sequence, and must return `XAER_PROTO` or another suitable error if the caller TM violates the state tables, as defined in Chapter 6 of the XA specification (see "Java Transaction API (JTA) Specification, version 1.2" on page E-1).

7.6.2.5　Unilateral Roll-back

- The RM need not wait for global transaction completion to report an error. The RM can return a rollback-only flag as a result of any `XAResource.start` or `XAResource.end` call. This can happen anytime except after a successful `prepare`.
- The RM is allowed to unilaterally rollback and forget a transaction branch any time before it prepares it.

7.6.2.6　Read-Only Optimization

Support for read-only optimization is optional for RM implementation. An RM can respond to the TM's request to prepare a transaction by asserting that the RM was not asked to update shared resources in this transaction branch. This response concludes the RM's involvement in the transaction, and the RM can release all resources and discard its knowledge of the transaction.

7.6.2.7　`XID` Support

- The RM must accept XIDs from TMs. The RM is responsible for using the XID to maintain an association between a transaction branch and recoverable units of work done by the application programs.
- The RM must not alter in any way the bits associated in the data portion of an XID. For example, if an RM remotely communicates an XID, it must ensure that the data bits of the XID are not altered by the communication process.

7.6.2.8　Support for Failure Recovery

- A full JTA compliant `XAResource` implementation that supports 2PC must maintain the status of all transaction branches in which it is involved. After responding affirmatively to the TM `prepare` call, an RM should not erase its knowledge of the branch or of the work done in support of the branch until it successfully receives a TM's invocation to commit or rollback the branch.
- If an RM that supports 2PC heuristically completes a branch, it should not forget a branch until the TM explicitly tells it to by calling `XAResource.forget`.
- On the TM's `XAResource.recover` call, an RM that supports 2PC must return a list of all transaction branches that it has prepared or has heuristically completed.
- When an RM recovers from its own failure, it must recover prepared and heuristically completed branches. It should discard its knowledge of all other branches.

7.6.3　Transaction Manager Requirements

The following section specifies requirements of a TM. This section assumes that the TM is compliant with JTA/JTS and X/Open (see "X/Open CAE Specification -- Distributed Transaction Processing: the XA Specification, X/Open document" on page E-1) specifications.

7.6.3.1　Interfaces

The TM must use the `XAResource` interface supported by an RM for transaction coordination and recovery. The TM must be written to handle consistently any information or status that an RM can legally return. The TM must assume that it can support RMs that have different capabilities as allowed by the RM requirements specification section, for instance RMs that make heuristic decisions and RMs that use the read-only optimization. [Requirement derived from Section 7.3, XA specification]

7.6.3.2　XID Requirements

The TM must generate XIDs conforming to the structure defined in section 4.2 on page 19 of the XA specification (see "Java Transaction API (JTA) Specification, version 1.2" on page E-1). The generated XIDs must be globally unique and must adequately describe a transaction branch.

7.6.3.3　One-phase Commit Optimization

- The TM must support one-phase commit protocol optimization. The TM uses the 1PC optimization when the TM knows there is only one RM registered in a transaction that is making changes to shared resources. In this optimization, the TM makes its phase 2 commit request to that RM without having made a phase 1 prepare request.
- The TM is not required to record in a stable manner such transactions, and in some failure cases, the TM may not record the outcome of the transaction completion.

7.6.3.4　Implementation Options

The support of *last-resource optimization* is an implementation-specific option for a TM. A detailed specification of TM and RM requirements for this optimization is outside the scope of the connector architecture.

7.6.4   Scenario: Transactional Setup for a ManagedConnection

The following object interactions are involved in the scenario shown in FIGURE 7-7.

1. The runtime scenario begins with a client method invocation on an EJB instance. This invocation has a transaction context, represented by a unique transaction `Xid`, associated with it if the invocation came from a client that was already participating in the transaction. Alternatively, the EJB container starts a transaction before dispatching the client request to the EJB method.

2. The EJB instance calls the `getConnection` method on the `ConnectionFactory` instance. The resource adapter delegates the connection request to the application server using the connection management contract. FIGURE 6-6 on page 6-38 explains this step.

3. The application server gains control and handles the connection allocation request.

4. To handle the connection allocation request, the application server gets a `Managed-Connection` instance either from the connection pool or creates a new `Managed-Connection` instance. FIGURE 6-6 on page 6-38 describes this step.

5. The application server registers itself as a `ConnectionEventListener` with the `ManagedConnection` instance. This enables the application server to receive notifications for various events on this connection instance. The application server uses these event notifications to manage connection pooling and transactions.

6. Based on the current transaction context associated with the connection-requesting thread and the EJB instance, the application server decides whether or not the transaction manager will participate in the coordination of the currently active transaction.

7. If the application server decides that the transaction manager will manage the current transaction, it conducts the following transactional setup on the `ManagedConnection` instance:

8. The application server invokes the `ManagedConnection.getXAResource` method to get the `XAResource` instance associated with the `ManagedConnection` instance.

9. The application server enlists the `XAResource` instance with the transaction manager for the current transaction context. The application server uses the `Transaction.enlistResource` method (specified in the JTA specification) to enlist the `XAResource` instance with the transaction manager. This enlistment informs the transaction manager about the resource manager instance participating in the transaction.

10. The transaction manager invokes `XAResource.start` to associate the current transaction with the underlying resource manager instance. This enables the transaction manager to inform the participating resource manager that all units of work performed by the application on the underlying `ManagedConnection` instance should now be associated with this transaction.

11. The application server calls the `ManagedConnection.getConnection` method to get a new application-level connection handle. The underlying physical connection is represented by a `ManagedConnection` instance.

12. The application server returns the connection handle to the resource adapter. The resource adapter then passes the connection handle to the application component that had initiated the connection request.

FIGURE 7-7 OID: Transactional Setup For Newly Created ManagedConnection Instances
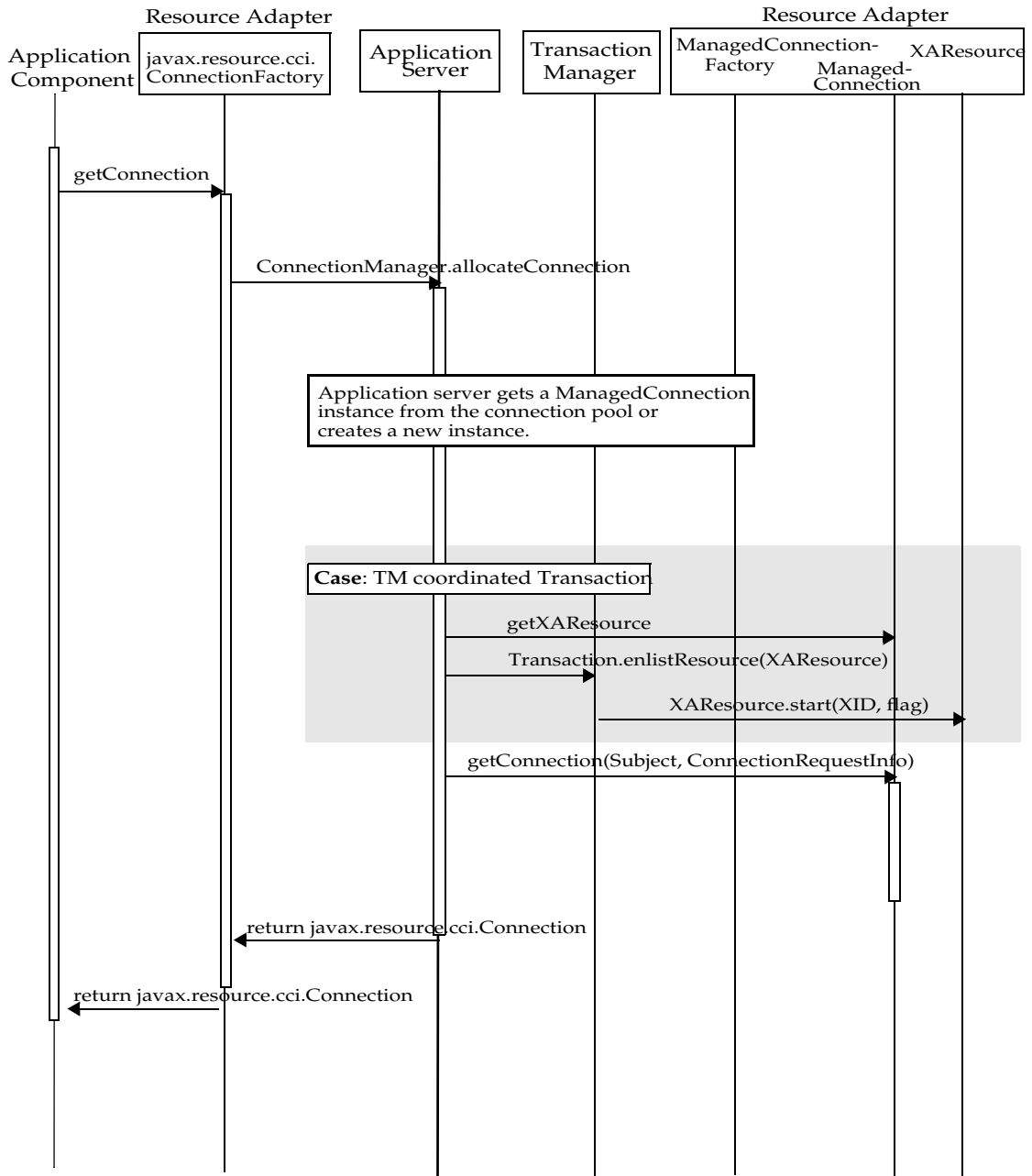
7.6.5 Scenario: Connection Close and JTA Transactional Cleanup

For each `ManagedConnection` instance in the pool, the application server registers a `ConnectionEventListener` instance to receive specific events on the connection. The connection event callback mechanism enables the application server to manage connection pooling and transactions.

FIGURE 6-4 on page 6-31 describes the following steps when an application component closes a connection:

1. The application component releases a `Connection` instance by calling the `close` method. The `Connection` instance delegates the connection close request to its associated `ManagedConnection` instance. A `ManagedConnection` must not alter any state on the physical connection while handling a delegated connection close request.

2. The `ManagedConnection` instance notifies all its registered listeners of the application's connection close request using the `ConnectionEventListener.connectionClosed` method. It passes a `ConnectionEvent` instance with the event type set to `CONNECTION_CLOSED`.

3. On receiving the connection close notification, the application server performs transactional cleanup for the `ManagedConnection` instance. If the `ManagedConnection` instance was participating in a transaction manager-enlisted JTA transaction, the application server takes the following steps:

4. The application server dissociates the `XAResource` instance, corresponding to the `ManagedConnection` instance, from the transaction manager using the method `Transaction.delistResource`.

5. The transaction manager calls `XAResource.end(Xid,flag)` to inform the resource manager that any further operations on the `ManagedConnection` instance are no longer associated with the transaction, represented by the `Xid` passed in `XAResource.end` call. This method invocation dissociates the transaction from the resource manager instance.

6. After the JTA transaction completes, the application server initiates a cleanup of the physical connection instance by calling `ManagedConnection.cleanup` method. After calling the method `cleanup` on the `ManagedConnection` instance, the application server returns the `ManagedConnection` instance to the connection pool.

7. The application server can now use the `ManagedConnection` instance to handle future connection allocation requests from either the same or another component instance.

FIGURE 7-8   OID: Connection Close and Transactional Cleanup
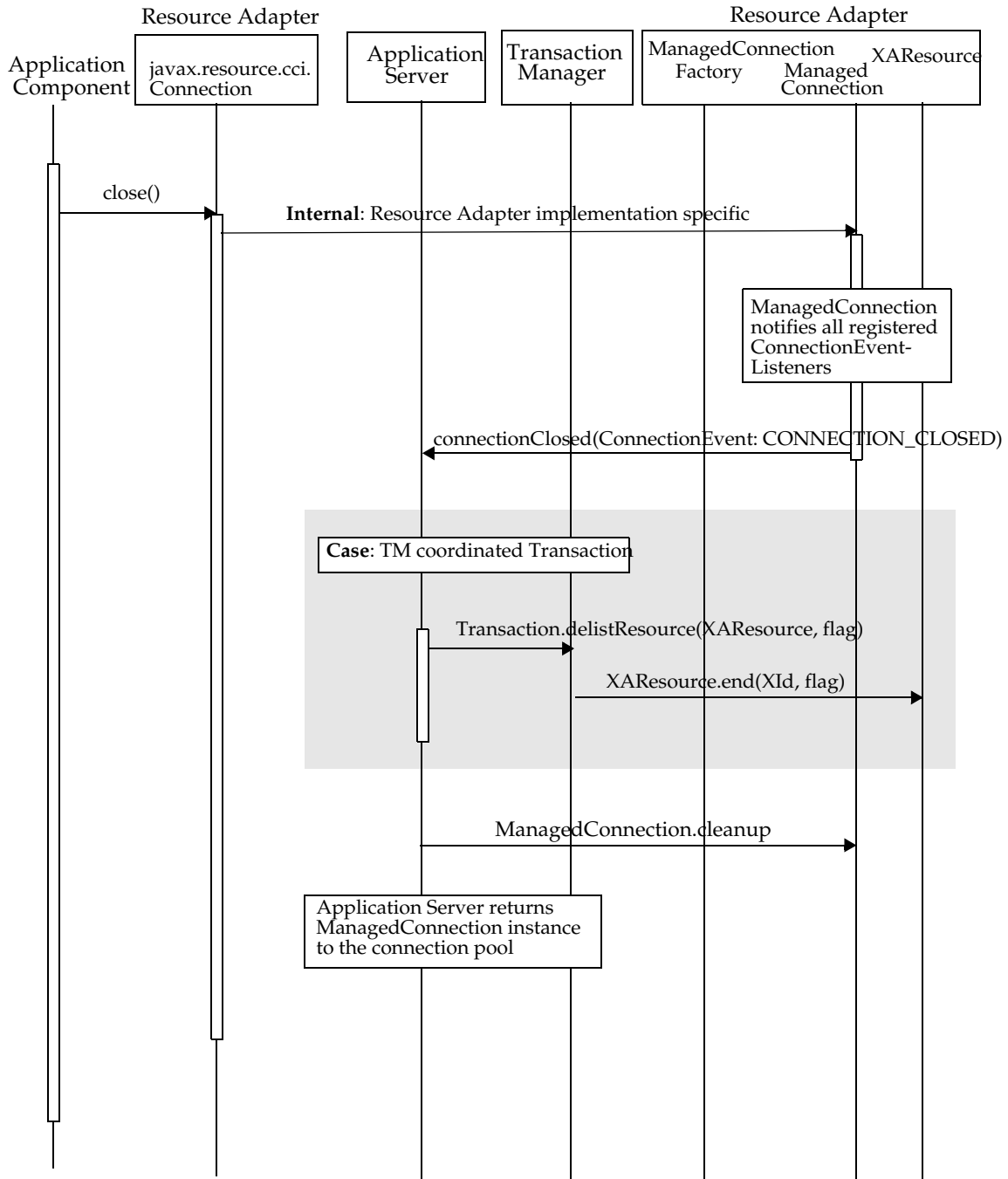
7.6.6 OID: Transaction Completion

The scenario in the following figure illustrates the steps taken by the transaction manager to commit a transaction across multiple resource manager instances. These steps are executed after the transaction manager calls the `XAResource.end` method for each enlisted resource manager instance.

The following steps happen in this scenario:

1. The transaction manager calls `XAResource.prepare` to begin the first phase of the transaction completion protocol. The transaction manager can call any `XAResource` instance is associated with the proper underlying resource manager instance, and is not restricted to the `XAResource` instance initially involved with the transaction. The application server can assume that all `XAResource` instances produced by a `ManagedConnectionFactory` instance refer to the same underlying resource manager instance.

2. Assuming that all resource manager instances involved in the transaction agree to commit, the transaction manager calls `XAResource.commit` to commit the transaction. Otherwise, the transaction manager calls `XAResource.rollback`.

FIGURE 7-9   OID: Transaction Completion

FIGURE 7-9   OID: Transaction Completion
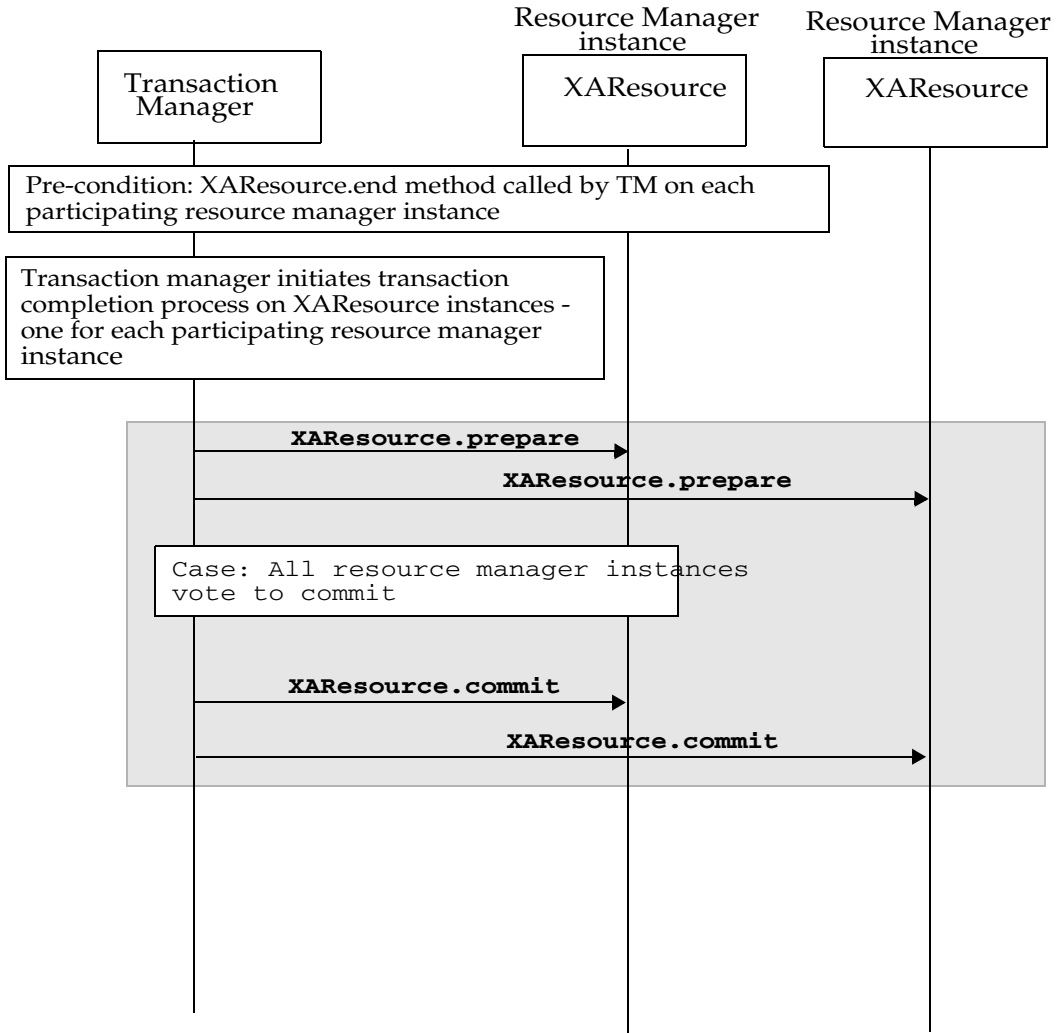
7.7   Local Transaction Management Contract

The main motivation for defining a local transaction contract between an application server and a resource manager is to enable an application server to manage resource manager local transactions, hereafter called local transactions.

The local transaction management contract has two parts:

■ The application server uses the `javax.resource.spi.LocalTransaction` interface to manage local transactions transparently to an application component. The scenarios in Section 7.10 "Transaction Scenarios" illustrate this part of the local transaction management contract.

■ The other part of the contract relates to notifications for local transaction-related events. If the resource adapter supports a local transaction demarcation API, for example, `javax.resource.cci.LocalTransaction` for the Common Client Interface, the resource adapter is required to notify the application server of the events (transaction begin, commit, and rollback) related to the local transaction. An application server uses this part of the contract, as explained in Section 7.8 "Scenarios: Local Transaction Management".

7.7.1   Interface: `LocalTransaction`

The `javax.resource.spi.LocalTransaction` interface defines the contract between an application server and resource adapter for local transaction management. This interface is defined in Section 7.3.3 "Interface: LocalTransaction".

7.7.2   Interface: `ConnectionEventListener`

An application server implements the `javax.resource.spi.ConnectionEventListener` interface. It registers this listener instance with the `ManagedConnection` instance by using `ManagedConnection.addConnectionEventListener` method.

The following code extract specifies the `ConnectionEventListener` interface related to the local transaction management contract:

```
public interface javax.resource.spi.ConnectionEventListener {
    // Local Transaction Management related events
    public void localTransactionStarted(ConnectionEvent event);
    public void localTransactionCommitted(ConnectionEvent event);
    public void localTransactionRolledback(ConnectionEvent
event);

    ...
}
```

The `ManagedConnection` instance notifies its registered listeners for transaction related events by calling the methods `localTransactionStarted`, `localTransactionCommitted`, and `localTransactionRolledback`.

The `ConnectionEvent` class defines the following types of event notifications related to the local transaction management contract:

- **`LOCAL_TRANSACTION_STARTED`** - a local transaction was started using the `ManagedConnection` instance
- **`LOCAL_TRANSACTION_COMMITTED`** - a local transaction was committed using the `ManagedConnection` instance
- **`LOCAL_TRANSACTION_ROLLEDBACK`** - a local transaction was rolled back using the `ManagedConnection` instance

7.7.2.1   Requirements

The connector specification requires an application server to implement the `ConnectionEventListener` interface and handle local transaction related events. This enables the application server to achieve local transaction cleanup and transaction serial interleaving, as illustrated in Section 7.8 "Scenarios: Local Transaction Management". The connector specification provides the necessary mechanisms for transaction management. Whether these mechanisms are used in an application server depends on the application server's implementation of the transaction requirements of the Java EE component specifications.

The resource adapter must send local transaction events through the `ConnectionEventListener` interface when an application component starts a local transaction using the application level transaction demarcation interface. An exception to this requirement is when the transaction demarcation API supports the concept of an implicit begin of a local transaction. The JDBC API is an example where there is no explicit local transaction begin method.

However, resource adapters that allow implicit begin of a local transaction, for instance, JDBC drivers, are strongly encouraged to provide support for local transaction events. This may be required in a future release of the specification.

The resource adapter must not send local transaction events for local transactions managed by the container.

7.8   Scenarios: Local Transaction Management

This section illustrates how an application server uses the event notifications from the resource adapter to manage local transactions and to restrict illegal transaction demarcations by an application component.

In these scenarios, an application component starts a local transaction using an application-level transaction demarcation interface, for example, `javax.resource.cci.LocalTransaction` as defined in the CCI, supported by the resource adapter. The resource adapter, in its implementation of the transaction demarcation interface, sends event notifications related to the local transaction, namely, local transaction begin, commit, and rollback. The application server is notified of these local transaction-related events through the `ConnectionEventListener` mechanism.

7.8.1   Local Transaction Cleanup

A stateless session bean with bean-managed transaction demarcation starts a local transaction in a method invocation. It returns from the business method without completing the local transaction.

The application server implements the `ConnectionEventListener` interface. The resource adapter notifies the application server with a `LOCAL_TRANSACTION_STARTED` event when the local transaction is started by the session bean instance.

When the session bean instance returns from the method invocation without completing the local transaction, the application server detects this as an incomplete local transaction because it has not received any matching `LOCAL_TRANSACTION_COMMITTED` or `LOCAL_TRANSACTION_ROLLEDBACK` events from the resource adapter.

On detecting an incomplete local transaction, the application server aborts the transaction, terminates the stateless session bean instance, and throws an exception to the client.

7.8.2   Component Termination

The application server terminates a component instance, for example, because of some system exception in a method invocation.

On termination of a component instance, the application server cleans up all `ManagedConnection` instances being used by this component instance. The cleanup of a connection involves resetting all local transaction and client-specific state. This state is maintained internal to the `ManagedConnection` instance.

The application server initiates a cleanup of a `ManagedConnection` instance by calling `ManagedConnection.cleanup`. After cleanup, the application server returns this connection to the pool to serve future allocation requests.

7.8.3 Transaction Interleaving

The application server uses the connection event listener mechanism, specified through the interfaces `ConnectionEventListener` and `ConnectionEvent`, to flag illegal cases of transaction demarcation. The application server implements the `ConnectionEventListener` interface to support this scenario.

The following subsection illustrates a scenario for component-managed transaction demarcation.

7.8.3.1 Scenario

An EJB component with bean managed transaction demarcation starts a local transaction using the application-level transaction demarcation interface, for example, `javax.resource.cci.LocalTransaction` as defined in the CCI, supported by the resource adapter. It then calls the `UserTransaction.begin` method to start a JTA transaction before it has completed the local transaction.

In this scenario, the EJB component has started but not completed the local transaction. When the application component attempts to start a JTA transaction by invoking the `UserTransaction.begin` method, the application server detects it as a transaction demarcation error and throws an exception from the `UserTransaction.begin` method.

When the application component starts the local transaction, the resource adapter notifies the application server of the `LOCAL_TRANSACTION_STARTED` connection event. When the component invokes the `UserTransaction.begin` method, the application server detects an error condition, because it has not received the matching `LOCAL_TRANSACTION_COMMITTED` or `LOCAL_TRANSACTION_ROLLEDBACK` event from the resource adapter for the currently active local transaction.

7.9 Connection Sharing

Sharing connections typically results in efficient use of resources and better performance. An application can indicate the ability to share its various resource references, or connections, in its deployment descriptor. A connection can be marked either as `shareable` or `unshareable`. The default is `shareable`.

When multiple shareable connections x and y acquired by an application are used within a global transaction scope (for instance, container-managed or bean-managed), the application server must provide a single shared connection behavior under the following conditions:

- x and y are collocated in a single Java Virtual Machine process address space
- x and y are using a single transactional resource manager
- x and y have identical properties
- x and y are marked as `shareable`
- x and y are used within a container-managed or bean-managed transaction scope

The ability to share is unspecified for connections marked `shareable` that are used outside a global transaction scope. Sharing is not supported for connections obtained from a non-transactional resource adapter, that is, transaction support level is `NoTransaction`.

The intent of the connection sharing requirement is to avoid resource manager lock contentions and read isolation problems, and thus ensure portable behavior for transactional applications. The application server may implement the connection sharing semantics either using a single shared connection or through other mechanisms[1].

If a connection is marked as `shareable`, it must be transparent to the application whether a single shared connection is used or not. The application must not make assumptions about a single shared connection being used, and hence must use the connection in a shareable manner.

However, a Java EE application component that intends to use a connection in an unshareable way must leave a deployment hint to that effect, which will prevent the connection from being shared by the container. Examples of unshareable usage of a connection include changing the security attributes, isolation levels, character settings, and localization configuration.

Containers must not attempt to share connections that are marked `unshareable`.

Java EE application components may use the optional deployment descriptor element `res-sharing-scope` or the `shareable` annotation element of `Resource` annotation defined in the Common Annotations specification (see "Common annotations for the Java Platform Specification, version 1.2" on page E-2), 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. Refer to EJB specification (see "Enterprise JavaBeans (EJB) Specification, version 3.2" on page E-1) and the servlet specification (see "Java Servlet Specification, version 3.1" on page E-1) for a description of the deployment descriptor element.

---

1. An application server may not share connections that are marked `shareable`, but still ensure the portability of the transaction application across application servers. For instance, the application server may choose to end XA association of connections when control flows from one application component to another, in order to avoid concurrent access problems on XA Resource Managers, and thus provide single shared connection behavior.

Java EE application components may cache connection objects and reuse them across multiple transactions. Containers that provide connection sharing should transparently switch such cached connection objects, at dispatch time, to point to an appropriate shared connection with the correct transaction scope. Refer to Section 7.11 "Connection Association" for more details on connection association.

Refer to Section 7.10 "Transaction Scenarios" for a special case of connection sharing as applied to resource adapters that support local transactions.

7.9.1 Sharing Violation Detection

A resource adapter may detect sharing violations. Any operation on a shareable connection which violates shareability is a sharing violation, for example, mutable operations like changing connection attributes, security settings, isolation levels, etc.

When such a mutable operation is performed on a `ManagedConnection`, it may throw a `SharingViolationException` when both the following conditions are true:

- The number of connection handle objects associated with the `ManagedConnection` is more than one.
- The `ManagedConnection` is associated with a transaction, either local or XA.

Further, a resource adapter may reject creation of a connection handle, by throwing a `SharingViolationException`, if the connection is already in a unshareable condition. Any mutable operation performed on a connection makes it unshareable.

7.9.1.1 Scenario 1

Application component A gets a shareable connection to a resource and invokes component B which also gets a shareable connection to the same resource. Both A and B are involved in a common transaction scope, either local or XA. The application server shares the connections acquired by both A and B. From this point onwards, any attempt to change a mutable property, such as isolation level, by either component, results in a `SharingViolationException` being thrown by the resource adapter to the offending component.

7.9.1.2 Scenario 2

Application component A gets a shareable connection to a resource. A is involved in a transaction, either local or XA. A then modifies one of the mutable properties of the resource, such as isolation level. This makes the connection unshareable. The resource adapter does not throw an exception since only one connection handle is present.

Later, A invokes B under the same transaction scope. B also attempts to acquire a shareable connection to the same resource. The application server chooses to share the connection that is already in use by A. At this point, the resource adapter throws a `SharingViolationException` to B since sharing had been attempted on an unshareable connection. The resource adapter does this by saving that the connection had been made unshareable earlier.

The resource adapter might throw a `SharingViolationException` to B, even if A had closed its connection handle before it invoked B, since the connection acquired by A had become unshareable.

7.10 Transaction Scenarios

This section specifies requirements for various transaction scenarios.

7.10.1 Requirements

The Java EE platform specification (see "Java Platform, Enterprise Edition (Java EE) Specification, version 7" on page E-1) identifies the following as transactional resources:

- JDBC connections
- JMS sessions
- Resource adapter connections at the `XATransaction` level

The Java EE platform specification requires that Java EE product providers must transparently support transactions that span multiple components and transactional resources. These requirements must be met regardless of whether a Java EE product is implemented as a single process, multiple processes on the same node, or multiple processes on multiple nodes.

In addition, Java EE product providers must support transactional applications that are comprised of servlets or JSP pages accessing multiple enterprise beans within a single transaction. Each component may also acquire one or more connections to access transactional resources. Java EE product providers must support scenarios where multiple components in an application access transactional resources as part of a single transaction.

The Java EE platform specification requires Java EE platform products to support resource adapters at the `XATransaction` level as a transactional resource. It must be possible to access such resource adapters from multiple application components within a single transaction.

The connector architecture has an additional requirement that is applicable to resource adapters that support local transactions. Note that both LocalTransaction and XATransaction resource adapters support local transactions. and they are both referred to as "local transaction capable" resource adapters in the section below.

Application server must use a single local transaction in a scenario where the following conditions hold:

■ Multiple components are involved in a global transaction scope.
■ All components use a single resource adapter that is local transaction capable. There is no other XAResource or local transaction capable resource adapter involved in the global transaction scope.
■ All components get connections to the same EIS instance.
■ Components have not specified the res-sharing-scope flag as unshareable. This condition accounts for potential sharing of connections in terms of security context, client-specific connection parameters, and EIS specific configuration.

Note that this requirement does not apply to a local transaction that is started by a component using an application level transaction demarcation API that is specific to a resource adapter.

Application server determines this scenario in an implementation-specific manner.

Application server may use connection sharing mechanisms to implement this local transaction requirement. Please refer to Section 7.10.3 "Scenario: Local Transaction" for an illustration.

Application servers must support transaction scenarios where access to a non-transactional resource is combined with access to one or more transactional resources within a single transaction. For example, in a container-managed transaction, an EJB accesses JDBC and JMS resources and also accesses a non-transactional EIS using its resource adapter. If there is a failure during the above scenario, transactional resource managers operating under the transaction should rollback, but the recovery of the non-transactional resource is unspecified in this specification.

The application server is not required to support any additional transaction scenarios beyond the above set of scenarios. A Java EE application should not depend on an application server's support for any optional transaction scenarios. The application should also not depend on whether or not the container detects that a specific optional transaction scenario is illegal. Any errors in optional transaction scenarios are considered application programming errors.

7.10.2 Illustrative Scenarios

The following are examples of optional transaction scenarios. The following section also describes, in a non-prescriptive manner, issues in support for these scenarios by an application server:

- Within a transaction, an EJB component acquires connections to two different resource managers X and Y using their respective non-XA local transaction capable resource adapters.

  The container cannot manage a local transaction across two different resource managers. Since resource adapters and underlying resource managers are not XA capable, the container cannot use XA in this case. However, a Java EE application should not depend on the container to detect this illegal scenario.

- Within a transaction, EJB component A acquires a connection to a resource manager X using a non-XA local transaction capable resource adapter. Next, EJB component B under the same transaction context acquires a connection to a different resource manager Y using a non-XA local transaction capable resource adapter

  The container cannot manage a local transaction across two different resource managers. Since resource adapters are not XA capable, the container cannot use XA in this case. However, a Java EE application should not depend on the container to detect this illegal scenario.

- Within a transaction, EJB component A acquires a connection to a resource manager X using a non-XA local transaction capable resource adapter. Next, the same EJB (or EJB B) under the same transaction context acquires a connection to a different resource manager Y using an XA capable resource adapter

  This scenario may be supported if the transaction manager supports last resource commit optimization. Since this optimization feature is optional and not specified in the connector architecture, a Java EE application should not depend on support for this scenario.

- Within a transaction, EJB A acquires a connection to a resource manager X using an XA capable resource adapter. Next, the same EJB component (or another EJB component B) under the same transaction context acquires a connection to a different resource manager Y using a non-XA local transaction capable resource adapter

  This scenario may be supported if the transaction manager supports last resource commit optimization. Since this optimization feature is optional and not specified in the connector architecture, a Java EE application should not depend on support for this scenario.

7.10.3   Scenario: Local Transaction

This scenario illustrates the use of the connection sharing mechanism to implement requirement for a local transaction to span components.

In this scenario, two EJB components get connections to the same EIS resource manager within a single transaction. Both EJB components use the same local transaction capable resource adapter.

A local transaction is associated with a single physical connection. Both EJB components in this scenario share the same physical connection under the local transaction scope. The container has the responsibility of managing connection sharing as illustrated in this scenario.

To share a physical connection in the local transaction scope, the container assumes the connection to be shareable unless it has been marked `unshareable` in the `res-sharing-scope`. The container uses connection sharing in a manner that is transparent to application components.

In the following figure, the stateful session beans A and B have container-managed transaction demarcation with the transaction attribute set to `Required`. Both A and B access a single EIS resource manager as part of their business logic.

FIGURE 7-10 Scenario to illustrate Local Transaction Management
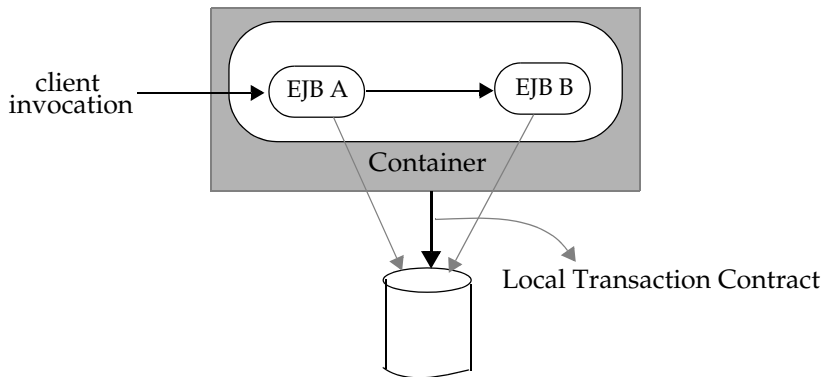
The following steps happen in this scenario:

1. The client invokes a method on session bean A with no transaction context. In its method implementation, the EJB A acquires a connection to the EIS instance.

2. When acquiring the connection, the container starts a local transaction by invoking the `begin` method of the `javax.resource.spi.LocalTransaction` instance. The local transaction is tied to the `ManagedConnection` instance that is associated with the connection handle acquired by the component in the previous step.

3. After the local transaction starts, any recoverable unit of work performed by A on the EIS resource manager using the acquired connection is automatically included under the local transaction context.

4. Session bean A now invokes a method on the session bean B instance. In this scenario, A does not close the connection handle before invoking the method on B.

Note – A container should ensure that the connection sharing mechanism is equally applicable if A were to close the connection handle before calling the B instance.

---

5. In the invoked method, B makes a request to acquire a connection to the same EIS resource manager.

6. The container returns a connection handle using the same `ManagedConnection` instance that was used for handling the connection request from A.

7. The container retains the association of the `ManagedConnection` instance with the local transaction context across the method invocation from A to B. This means that any unit of work that B will perform on the EIS resource manager using its acquired connection handle will be automatically included as part of the current local transaction. The connection state, for example, any open cursors, can also be retained across method invocations when the physical connection is shared.

8. Before the method invocation on B completes, B calls the `close` method on the connection handle. The container should not initiate any cleanup of the physical connection at this time since there is still an uncompleted local transaction associated with the shared physical connection. In this scenario, the cleanup of a physical connection refers to the dissociation of the local transaction context from the `ManagedConnection` instance. In the absence of support for Lazy Connection Association (see Section 7.16.1 "Lazy Connection Association Optimization") from the resource adapter and the application server, the component B should not cache the connection handle. See Section 6.4.3, "Guidelines" on page 6-7 for a suggested scheme of obtaining and closing connection handles. A component caching a connection handle in this scenario is not portably supported.

9. When A regains control, A can use the same connection handle, provided A had not called the `close` method on the connection handle, to access EIS resources. All recoverable units of work on the EIS resource manager will be included in the existing local transaction context.

Note – If A closes the connection handle before calling B, and then reacquires the connection handle when regaining control, the container should ensure that the local transaction context stays associated with the shared connection.

---

10. A eventually calls the `close` method on its connection handle. The container gets a connection close event notification based on the scenario described in Section 6.8.3, "Scenario: Connection Event Notifications and Connection Close" on page 6-39.

11. Since there is an incomplete local transaction associated with the underlying physical connection, the container does not initiate a cleanup of the `ManagedConnection` on receiving the connection close event notification. The container must still go through the completion process for the local transaction.

12. When the business method invocation on A completes successfully without any application error, the container starts the completion protocol for the local transaction. The container calls the `LocalTransaction.commit` method to commit the transaction.

13. After the local transaction completes, the container initiates a cleanup of the physical connection instance by calling the `ManagedConnection.cleanup` method.

Note – The container should initiate cleanup of the `ManagedConnection` instance in the case where A does not call the `close` method on the connection handle before returning. The container identifies the need for cleaning up the `ManagedConnection` instance based on the scope of connection sharing.

---

14. On the `cleanup` method invocation, the `ManagedConnection` instance does a cleanup of its local transaction related state and resets itself to a default state.

15. The container returns the physical connection to the pool for handling subsequent connection requests.

FIGURE 7-11 Connection Sharing Across Component Instances
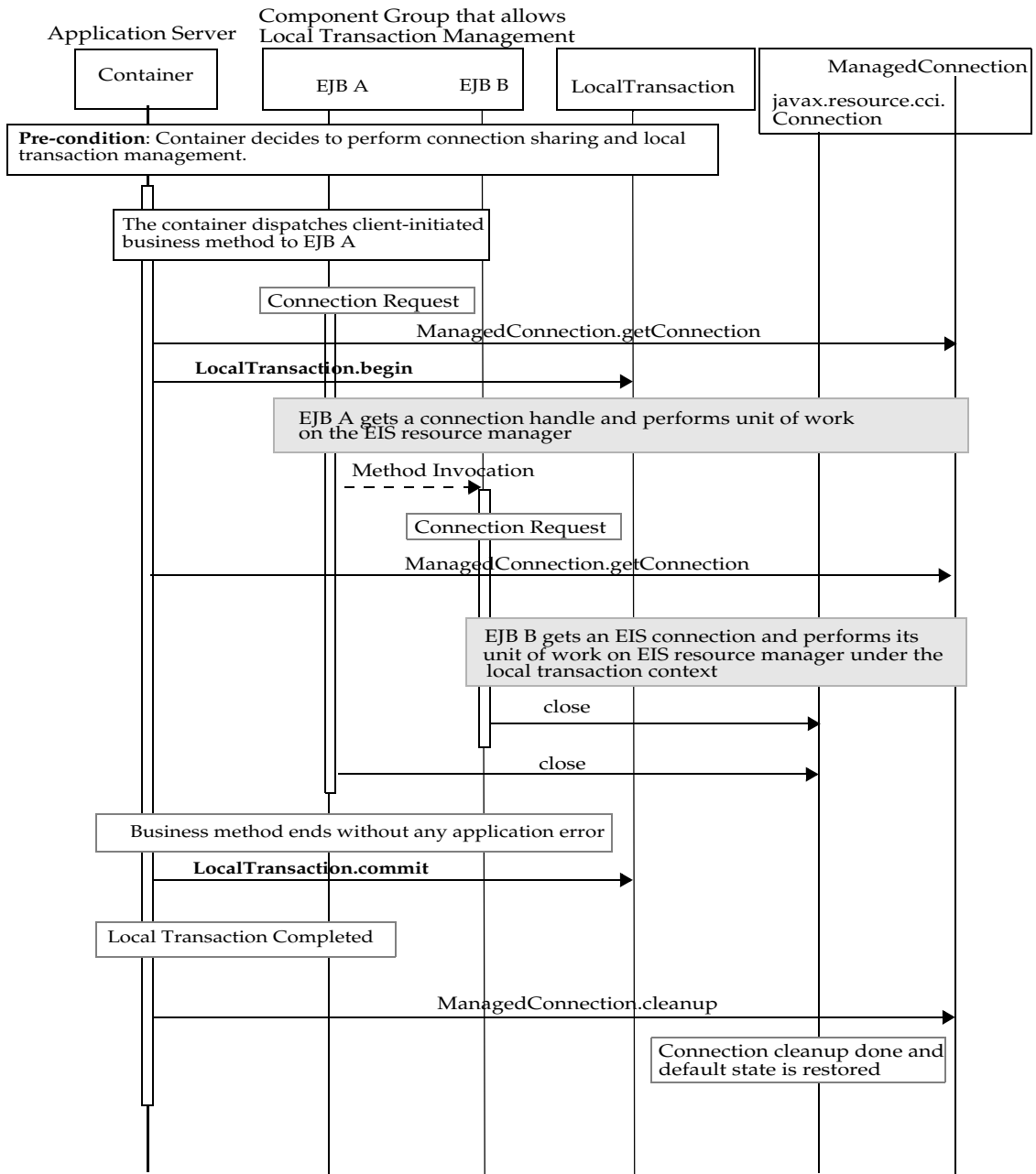
7.11 Connection Association

According to the connection management contract, a connection handle is created from a `ManagedConnection` instance using the `ManagedConnection.getConnection` method. A connection handle maintains an association with the underlying `ManagedConnection` instance.

7.11.1 Scenario

In the scenario shown in the following figure, session bean A acts as a client of entity bean C and makes invocations on methods of entity bean C. Another session bean B also acts as a client of entity bean C. The C is an entity bean that may be shared across multiple clients.

A, B and C get connections to the same EIS. These EJB components have marked `res-sharing-scope` for these connections to be `shareable`.

A and C define a connection sharing scope. Both A and C share the same physical connection across a transaction that spans methods on A and C. Similarly, B and C define another connection sharing scope. B and C also share the same physical connection across a transaction that spans two components.

FIGURE 7-12 Connection Sharing Scenario
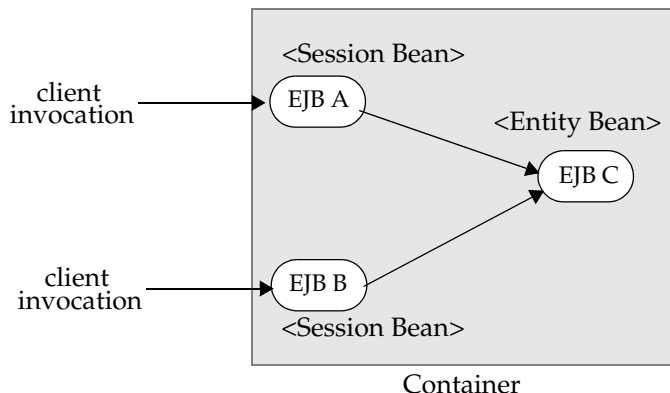
In this scenario, entity bean C obtains an application-level connection handle using the method `getConnection` on the `ConnectionFactory` during its creation. Entity bean C holds the connection handle during its lifetime.

A gets a connection handle and invokes a method on C. At a different time, B gets a connection handle and invokes a method on C.

In both cases, depending on the connection sharing scope, defined in terms of the shared physical ManagedConnection instance, in which C is called, the container supports a mechanism to associate the connection handle held by C as part of its state with the current ManagedConnection instance.

FIGURE 7-13 State Diagram of Application-Level Connection Handle
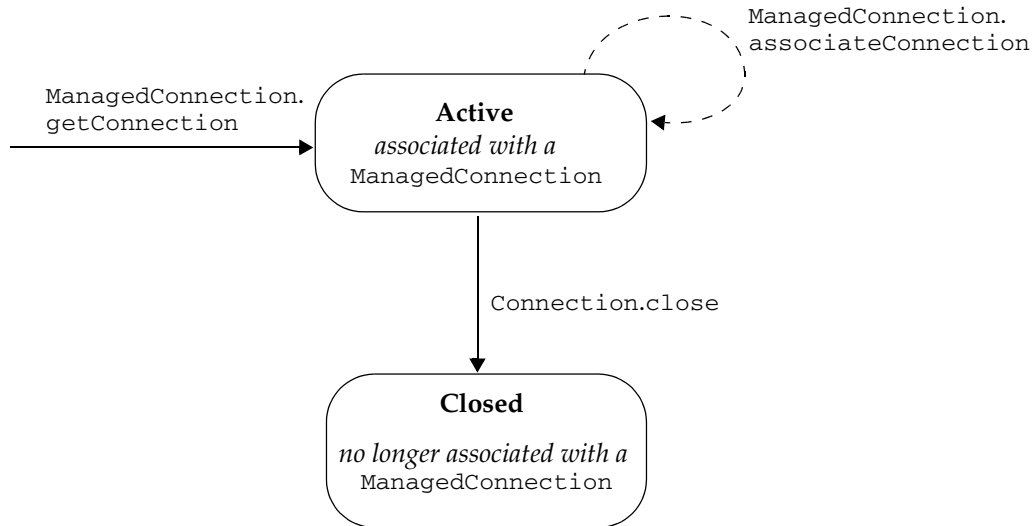
7.11.2   Connection Association

The interface ManagedConnection defines method associateConnection as follows:

```
public interface javax.resource.spi.ManagedConnection {
    public void associateConnection(Object connection)
    throws ResourceException;
    ...
}
```

The container typically uses the associateConnection method to change the association of an application-level connection handle with a ManagedConnection instance. The container finds the right ManagedConnection instance, depending on the connection sharing scope, and calls the associateConnection method. To achieve this, the container is required to keep track of connection handles acquired

by component instances and `ManagedConnection` instances using an implementation-specific mechanism. In order to set a Connection Handle as the active connection handle (see Section 6.5.4.2, "Connection Sharing and Multiple Connection Handles" on page 6-23), the container may also use the `associateConnection` method to set the same `ManagedConnection` associated with the Connection handle.

The `associateConnection` method implementation for a `ManagedConnection` should dissociate the connection handle passed as a parameter from its currently associated `ManagedConnection` and associate the new connection handle with itself.

Note that the switching of connection associations must happen only for connection handles and `ManagedConnection` instances that correspond to the same `ManagedConnectionFactory` instance. The container should enforce this restriction in an implementation-specific manner. If a container cannot enforce the restriction, the container should not use the connection association mechanism.

7.11.3   Requirements

The container must provide a mechanism to change the association of a connection handle to different `ManagedConnection` instances depending on the connection sharing and transaction scope. This mechanism is used in scenarios where components hold on to connection handles across different local transaction and connection sharing scopes.

The container may use the connection association mechanism in the `XAResource`-based transaction management contract.

The resource adapter must implement the `associateConnection` method to support connection sharing. The container makes a decision on whether or not to use the `associateConnection` method implemented by a resource adapter. The support for this method is required independent of the transaction support level of the resource adapter. Note that the container makes the decision to invoke the `associateConnection` method.

7.12 Local Transaction Optimization

If all the work done as a part of a transaction uses a single resource manager, the application server can use a local transaction in place of an externally coordinated JTA transaction. The use of a local transaction avoids the overhead of initiating a global transaction, and involving the TM for transaction coordination, and leads to more optimized performance.

Since a typical application accesses a single resource manager, the local transaction optimization is a useful performance enhancement for transaction management.

The application server manages local transaction optimization transparent to the Java EE application. Whenever a container-managed or bean-managed transaction is started, the container may attempt local transaction optimization.

When the transaction begins, a container cannot determine beforehand whether or not the unit of work done as part of this transaction will use a single resource manager. The container uses an implementation-specific mechanism to achieve local transaction optimization. For example, the container can choose to start a local transaction when the first resource manager is accessed and lazily start a JTA transaction only when more than one resource managers are accessed in an existing transaction. The mechanism through which the application server and its transaction manager coordinates the initial local transaction and lazily started JTA transactions is outside the scope of the connector specification. Refer to the Java EE platform specification (see "Java Platform, Enterprise Edition (Java EE) Specification, version 7" on page E-1) for more details on the local transaction optimization.

7.12.1 Requirements

The container is not required to support the local transaction optimization.

7.13 Runtime Transaction Support Level Specification

A resource adapter may determine and classify the level of transaction support it can provide at runtime. The resource adapter can use the configuration details, provided by a deployer, to determine the transactional capabilities and the requirements of the underlying EIS and then specify the level of transaction support at runtime.

```
package javax.resource.spi;

public interface TransactionSupport extends java.io.Serializable {
    public enum TransactionSupportLevel
        {NoTransaction, LocalTransaction, XATransaction}

    public TransactionSupportLevel getTransactionSupport();
}
```

To specify the level of transaction support at runtime, a
`ManagedConnectionFactory` must implement the `TransactionSupport`
interface. It is optional for the `ManagedConnectionFactory` to implement this
interface.

When a `ManagedConnectionFactory` does not implement this interface, the
container must use the resource adapter's level of transaction support classification.
The container must use the transaction support specified in the merged result of the
resource adapter's deployment descriptor and Connector annotations. Refer to
Section 20.4.1, "Resource Adapter Provider" on page 20-7 for more information on
the resource adapter deployment descriptor and Section 18.4, "@Connector" on
page 18-6 for more information on the Connector annotation. If the resource adapter
deployer has overridden the transaction support value, the overriden value must be
used. Refer to Section 5.3.7.1, "ResourceAdapter JavaBean Instance Configuration"
on page 5-14 for details on resource adapter configuration.

For `ManagedConnectionFactory` JavaBeans that implement the
`TransactionSupport` interface, the application server must perform the following
prior to using the JavaBean. The application server must call the
`getTransactionSupport` method to determine its level of transaction support.
The application server must complete the configuration of the
`ManagedConnectionFactory` instance (see Section 5.3.2,
"ManagedConnectionFactory JavaBean and Outbound Communication" on
page 5-8) before invoking the `getTransactionSupport` method. The application
server must use the value returned by the `getTransactionSupport` method and
ignore the value specified by the resource adapter deployment descriptor/Connector
annotation or the deployer configuration. The application server must provide the
transaction levels listed in `TransactionSupport.TransactionSupportLevel`
enum, the same semantics as the levels detailed in Section 7.15.1 "Resource
Adapter".

A resource adapter must always return a level of transaction support whose ordinal
value in the TransactionSupport.TransactionSupportLevel enum is equal to or lesser
than the resource adapter's transaction support classification.

7.14 Interface: TransactionSynchronizationRegistry

The `TransactionSynchronizationRegistry` interface is defined in the JTA specification (see "Java Transaction API (JTA) Specification, version 1.2" on page E-1) and could be used by system level components to interact with the transaction manager. This interface provides the ability to register synchronization objects, associate resource objects with the current transaction, get the transaction context of the current transaction, get current transaction status, and mark the current transaction for rollback.

This interface is implemented by the application server by a stateless service object. A resource adapter may obtain the `TransactionSynchronizationRegistry` through the `getTransactionSynchronizationRegistry` method (shown below) of `BootstrapContext` (see Section 5.3.1, "ResourceAdapter JavaBean and Bootstrapping a Resource Adapter Instance" on page 5-4). The application server is required to make a `TransactionSynchronizationRegistry` object available through its `BootstrapContext` implementation. The same `TransactionSynchronizationRegistry` object can be used by any number of artifacts in the resource adapter module with thread safety.

```
public interface javax.resource.spi.BootstrapContext {
    TransactionSynchronizationRegistry
getTransactionSynchronizationRegistry();
    ...
}
```

7.15 Requirements

This section outlines the requirements for the transaction management contract.

7.15.1 Resource Adapter

A resource adapter can be classified based on the level of transaction support, as follows:

- **NoTransaction.** The resource adapter supports neither resource manager local nor JTA transactions. It implements neither the `XAResource` nor `LocalTransaction` interfaces.
- **LocalTransaction** - The resource adapter supports resource manager local transactions by implementing the `LocalTransaction` interface. The local transaction management contract is specified in Section 7.7 "Local Transaction Management Contract".
- **XATransaction** - The resource adapter supports both resource manager local and JTA transactions by implementing the `LocalTransaction` and `XAResource` interfaces. The requirements for supporting the `XAResource`-based contract are specified in Section 7.6 "XAResource-based Transaction Contract".

Note – Other levels of support (includes any transaction optimizations supported by an underlying resource manager) are outside the scope of the connector architecture.

The above levels reflect the major steps of transaction support that a resource adapter is required to make to allow external transaction coordination. Depending on its transactional capabilities and the requirements of its underlying EIS, a resource adapter can choose to support any one of the above transaction support levels.

7.15.1.1   Auto Commit

When a connection is in an auto-commit mode, an operation on the connection automatically commits after it has been executed. The auto-commit mode must be off if multiple interactions have to be grouped in a single transaction, either local or XA, and committed or rolled back as a unit.

A resource adapter must manage the auto-commit mode as follows:

- A transactional resource adapter, either at `XATransaction` or `LocalTransaction` level, must set the auto-commit mode to false within a transaction, either local or XA, on a connection participating in the transaction. This requirement holds for both container-managed and bean-managed transaction demarcation.
- A transactional resource adapter must set the auto-commit mode to true, on connections that are used outside a transaction.

7.15.2   Application Server

An application server must support resource adapters with all three levels of transaction support—`NoTransaction`, `LocalTransaction`, and `XATransaction`.

The following are the requirements for an application server for the transaction management contract:

- The application server must support a transaction manager that manages transactions using the JTA `XAResource`-based contract. The requirements for a transaction manager to support an `XAResource`-based contract are specified in Section 7.6.3 "Transaction Manager Requirements".
- The application server must use the `LocalTransaction` interface-based contract to manage local transactions for a resource manager.
- The application server must use the deployment descriptor mechanism and the values in the Connector metadata annotation to ascertain the transactional capabilities of a resource adapter. Refer to Section 20.4, "Deployment" on page 20-7 for details on the deployment descriptor specification and Section 18.4, "@Connector" on page 18-6 for details on the Connector annotation.
- If a `ManagedConnectionFactory` chooses to specify its transactional capability in a dynamic fashion at runtime (see Section 7.13 "Runtime Transaction Support Level Specification"), the application server must ascertain the transactional capability provided by the `ManagedConnectionFactory` instance.
- The application server must implement the `ConnectionEventListener` interface to get transaction-related event notifications.

7.16 Connection Optimizations

This section describes two optional connection optimizations:

- Lazy connection association optimization
- Lazy transaction enlistment optimization

7.16.1 Lazy Connection Association Optimization

Application components may acquire connections through a `ConnectionFactory` object (resource-ref) obtained from the JNDI namespace. The connection(s) thus obtained may be closed by the application before method completion, or may be cached by the application for later use.

When a connection is cached by the application component, the cached connection handle is considered active and remains associated with a `ManagedConnection` instance from the application server's connection pool. If the cached connection handle is used infrequently, then the associated `ManagedConnection` instance remains in hibernation during periods of non-use. This is because the application server cannot detect when the hibernating `ManagedConnection` instance will be used again by the application.

Such hibernating ManagedConnection instances result in suboptimal usage of system resources. Avoiding hibernation of ManagedConnection instances leads to more optimal resource utilization and better performance.

The following describes a mechanism that allows an application server to avoid hibernating ManagedConnection instances (by dissociating the ManagedConnection from its connection handles and using the freed ManagedConnection instance for other applications). This mechanism also provides a way to notify the application server when a dissociated connection handle is used by the application, so that it can be associated with an appropriate ManagedConnection instance.

FIGURE 7-14 describes the processing of a getConnection method call initiated by an application component (that is, when the application component first acquires a connection). At a later point in time, the connection may be dissociated by the application server by calling the dissociateConnections method on the appropriate ManagedConnection instance. This dissociates the ManagedConnection instance from all its connection handle objects.

When such a dissociated connection is used by the application (upon method re-entry), it is required to be re-associated with an appropriate ManagedConnection instance. FIGURE 7-15 describes connection re-association processing. The connection re-association processing depends on the connection notifying the application server upon re-use (lazy re-association trigger). The connection object invokes the associateConnection method on the ConnectionManager instance in order to lazily re-associate itself with an appropriate ManagedConnection instance.

Thus, a connection handle that can be dissociated can exist in one of three states: Active, Inactive or Closed. FIGURE 7-16 describes the state transitions of a dissociatable connection handle. Note that the state Inactive applies only to dissociatable connection handles.

The application server may dissociate connections that are shareable. It must not dissociate connections that are marked unshareable, however, since application-specific state may be retained by a ManagedConnection instance. The application server may also call the dissociateConnections method even when an active transaction is in progress in the ManagedConnection.

When a disassociated connection handle is closed, the resource adapter must notify the application server by calling the inactiveConnectionClosed method on the LazyAssociatableConnectionManager interface. The application server can then perform any cleanup operations related to the disassociated connection handle in its connection pool.

FIGURE 7-14 Connection Acquisition Processing

ConnectionManager

4. getConnection(
Subject, ConnectionRequestInfo)

ManagedConnection

3. createManagedConnection(Subject, ConnectionRequestInfo)
OR matchManagedConnections(ConnectionSet, Subject,
ConnectionRequestInfo)

2. allocateConnection(
ManagedConnectionFactory,
ConnectionRequestInfo)

ConnectionFactory

1. getConnection(config)

App Component

ManagedConnectionFactory

FIGURE 7-15 Connection Re-association Processing
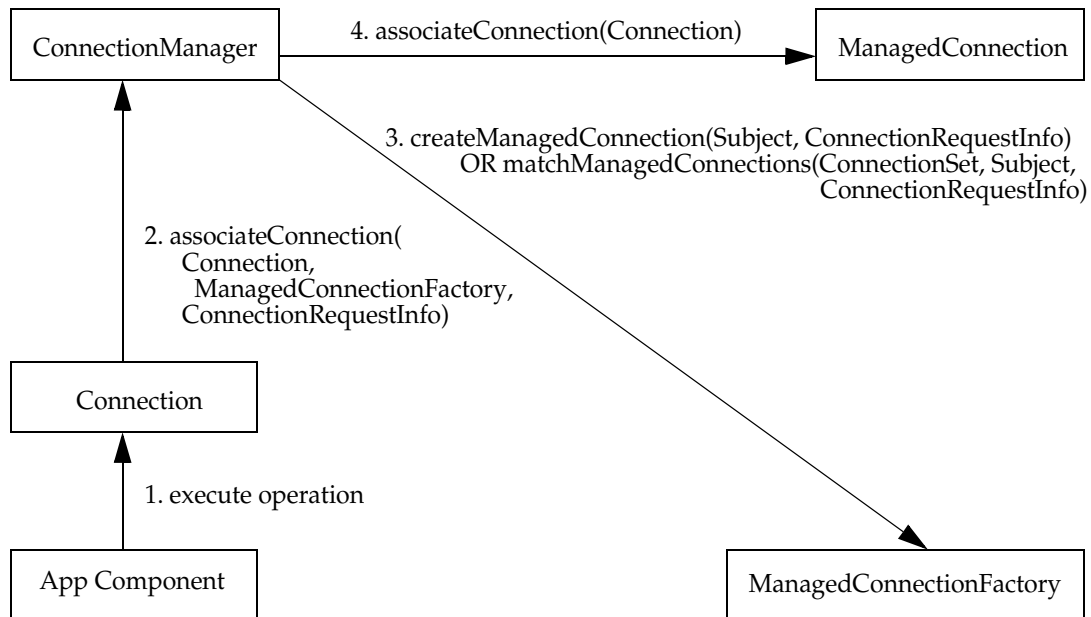
FIGURE 7-16 State Diagram of a Dissociatable Application-level Connection Handle
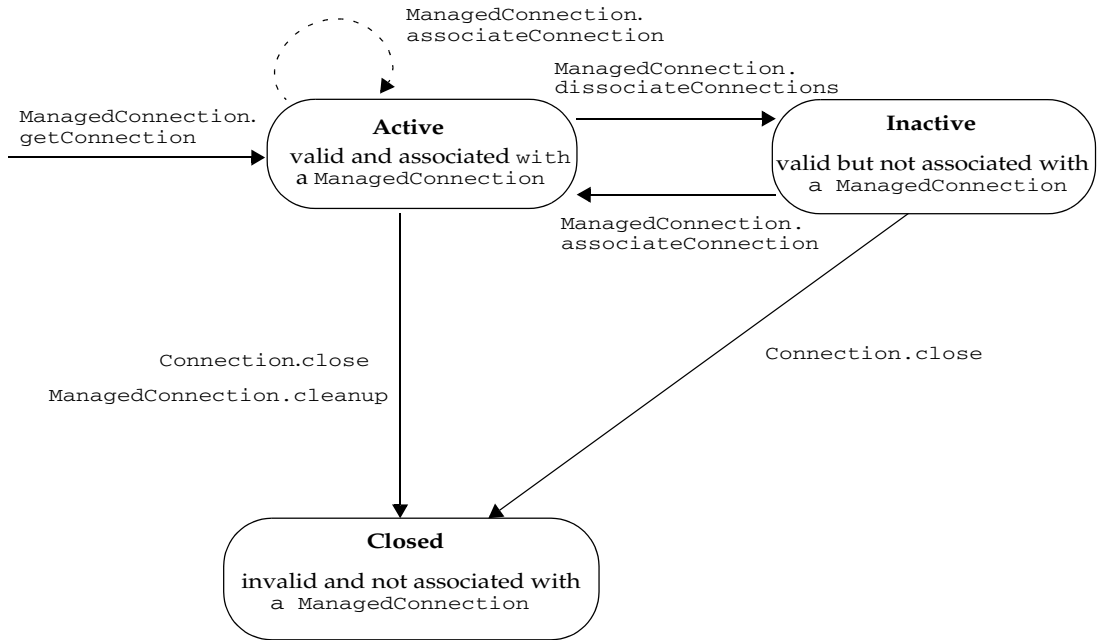
7.16.1.1   API Additions

```
package javax.resource.spi;

import javax.resource.ResourceException;

interface LazyAssociatableConnectionManager { // application
server
    void associateConnection(
        Object connection, ManagedConnectionFactory mcf,
            ConnectionRequestInfo info)
    throws ResourceException;

    void inactiveConnectionClosed(Object connection,
            ManagedConnectionFactory mcf);
}

interface DissociatableManagedConnection { // resource adapter
    void dissociateConnections() throws ResourceException;
}
```

Neither the application server nor the resource adapter must support this optimization.

A resource adapter that does not support this optimization must provide a `ManagedConnection` implementation that does not implement the `DissociatableManagedConnection` interface. This allows an application server to detect that the resource adapter does not support this optimization.

An application server that does not support this optimization must provide a `ConnectionManager` implementation that does not implement the `LazyAssociatableConnectionManager` interface. This allows a resource adapter to detect that the application server does not support this optimization. In reality, a resource adapter will not call this method (in order to re-associate a connection) since an application server that does not support this optimization would never dissociate a connection.

There are no changes to the resource adapter deployment descriptor since the application server can programmatically detect whether a resource adapter supports this optimization or not.

7.16.2   Lazy Transaction Enlistment Optimization

Transactions may be started by an application server before a method call on an application component or it may be started by an application component during a method call. It is also possible that an application server may use a transaction imported from a different server during a method call.

Irrespective of how a transaction is started, an application server enlists all connections (cached or newly acquired by an application component) with the transaction, so that the work done using those connections will be part of the transaction. This enlistment happens before the method call in the case of cached connections and during the method call when connections are newly acquired within the transaction.

But not all the connections that are cached or newly acquired by an application component may be used within a transaction. Since the application server cannot detect whether these connections would be used within the transaction, it statically (eagerly) enlists all such connections with the transaction. Thus, connections that are not used in a transaction are unnecessarily enlisted, which leads to sub-optimal performance.

The following describes a dynamic mechanism that allows the application server to enlist only those connections that are used within a transaction. A `ManagedConnection` that supports this optimization must invoke the `lazyEnlist` method on the `ConnectionManager` every time it is used outside of a local or XA transaction. The application server uses this method call to lazily enlist the connection in the transaction (if there is one). The application server may delist the `ManagedConnection` instances from the transaction at a later point.

This optimization can be used only on connections that are lazily enlist-able.

7.16.2.1 API Additions

```
package javax.resource.spi;

import javax.resource.ResourceException;
import javax.transaction.xa.Xid;

interface LazyEnlistableConnectionManager { // application server
    void lazyEnlist(ManagedConnection) throws ResourceException;
}

interface LazyEnlistableManagedConnection { // resource adapter
}
```

Neither the application server nor the resource adapter must support this optimization.

A resource adapter that does not support this optimization must provide a ManagedConnection implementation which does not implement the LazyEnlistableManagedConnection interface. This allows an application server to detect that the resource adapter does not support this optimization.

An application server that does not support this optimization must provide a ConnectionManager implementation that does not implement the LazyEnlistableConnectionManager interface. This allows a resource adapter to detect that the application server does not support this optimization.

There are no changes to the resource adapter deployment descriptor since the application server can programmatically detect whether a resource adapter supports this optimization or not.

Security Architecture

This chapter specifies the security architecture for the integration of EISs with the Java EE platform. It adds EIS integration-specific security details to the security requirements specified in other Java EE specifications.

8.1 Overview

It is critical that an enterprise be able to depend on the information in its EIS for its business activities. Any loss or inaccuracy of information or any unauthorized access to the EIS can be extremely costly to an enterprise. There are several mechanisms that can be used to protect an EIS against such security threats, including:

- Identification and authentication of principals, human users to verify they are who they claim to be.
- Authorization and access control to determine whether a principal is allowed to access an application server and/or an EIS.
- Secure communication between an application server and an EIS. Communication over insecure links can be protected using a protocol, for example, Kerberos, that provides authentication, integrity, and confidentiality services. Communication can also be protected by using a secure link protocol, for example, SSL.

8.2 Goals

The security architecture is designed to meet the following goals:

- Extend the end-to-end security model for Java EE applications to include integration with EISs based on the connector architecture.
- Support authentication and authorization of users who are accessing EISs.

- Keep the security architecture technology neutral and enable the specified security contract to be supported by various security technologies.
- Enable the security architecture to support a range of EISs with different levels of security support and existing security environments.
- Support security configuration of a resource adapter in an operational environment.
- Keep the security model for connector architecture-based EIS integration transparent to an application component provider. This includes providing support for single sign-on across multiple EISs.

The security model for EIS integration is not designed to do the following:

- Mandate a specific technology and describe how it can be used to implement the security architecture for connector architecture-based EIS integration.
- Specify and mandate a specific security policy. The security architecture enables an application server and EIS to support the implementation and administration of security policies based on their respective requirements.

8.3 Terminology

The following terms are used in this chapter:

- **Principal.** A principal is an entity that can be authenticated by an authentication mechanism deployed in an enterprise. A principal is identified using a *principal name* and authenticated using *authentication data.* The content and format of the principal name and the authentication data depend upon the authentication mechanism.
- **Security Attributes.** A principal has a set of security attributes associated with it. These security attributes are related to the authentication and authorization mechanisms. Some examples are security permissions, and credentials for a principal.
- **Credential.** A credential contains or references security information that can authenticate a principal to additional services. A principal acquires a credential upon authentication or from another principal that allows its credential to be used. The latter is termed *principal delegation*.
- **End user.** An end user is an entity, human or service, that acts as a source of a request to an application. An end user is represented as a security principal within a `Subject` as specified in the JAAS framework (see "Java Authentication and Authorization Service Specification, version 1.0" on page E-1).
- **Initiating Principal.** The security principal representing the end-user that interacts directly with the application. An end-user can authenticate using either a web client or an application client.

- **Caller Principal.** A principal that is associated with an application component instance during a method invocation. For example, an EJB instance can call the `getCallerPrincipal` method to get the principal associated with the current security context.
- **Resource Principal.** A security principal under whose security context a connection to an EIS instance is established.
- **Security domain.** A scope within which certain common security mechanisms and policies are established. This specification does not specify the scope of a security domain. An enterprise can contain more than one security domain. Thus an application server and an EIS may either be in the same or different security domains. Appendix C, "Security Scenarios" provides illustrative examples of how security domains can be setup and managed.

In a managed environment, application components are deployed in web or EJB containers. When a method gets invoked on a component, the principal associated with the component instance is termed a caller principal.

The relationship between an initiating principal and a caller principal depends on the principal delegation option for inter-container and inter-component calls. This form of principal delegation is out of the scope of the connector architecture.

The relationship of a resource principal and its security attributes, for example, credentials and access privileges, to an initiating or caller principal depends on how the resource principal has been setup by the system administrator or deployer.

Refer to Section 9.1.1, "Interfaces and Classes" on page 9-1 for details on interfaces and classes that are used to represent a resource principal and its credentials.

8.4   Application Security Model

This section is a brief summary of the security model from the perspective of an application component provider. Refer to the relevant specifications for more detail.

The application component requests a connection to be established under the security context of a resource principal. The security context includes security attributes—access privileges, authorization level—for a resource principal. Once a connection is successfully established, all application-level invocations to the EIS instance using the connection happen under the security context of the resource principal.

The application component provider has the following two choices related to EIS sign-on:

- Allow the deployer to set up the resource principal and EIS sign-on information. For example, the deployer sets the user name and password for establishing a connection to an EIS instance.

- Perform sign-on to an EIS from the component code by providing explicit security information for a resource principal.

The application component provider uses a deployment descriptor element or metadata annotations defined in the corresponding application component specifications, for example, `res-auth` for EJB components, to indicate the requirements for one of the above two approaches. If the `res-auth` element is set to `Application`, the component code performs a programmatic sign-on to the EIS. If the `res-auth` element is `Container`, the application server takes on the responsibility of setting up and managing EIS sign-on.

8.4.1   Scenario: Container-Managed Sign-on

The application component provider sets the `res-auth` deployment descriptor element, or the equivalent metadata annotation defined in the relevant application component specification, to be `Container` letting the application server take the responsibility of managing EIS sign-on.

The Deployer sets up the principal mapping such that the user account for connecting to the EIS instance is always `eStoreUser`. The Deployer also configures the authentication data, for example, the password, needed to authenticate the `eStoreUser` to the EIS.

The component code invokes the `getConnection` method on the `ConnectionFactory` instance with no security-related parameters. The component relies on the application server to manage sign-on to the EIS instance based on the security information configured by the Deployer.

```
// Method in an application component
Context initctx = new InitialContext();

// perform JNDI lookup to obtain connection factory
javax.resource.cci.ConnectionFactory cxf =
    (javax.resource.cci.ConnectionFactory)initctx.lookup(
        "java:comp/env/eis/MyEIS");

// Invoke factory to obtain a connection. The security
// information is not passed in the getConnection method
javax.resource.cci.Connection cx = cxf.getConnection();
...
```

8.4.2 Scenario: Component-Managed Sign-on

The application component provider sets the `res-auth` element to be `Application`.

The component code performs a programmatic sign-on to the EIS. The application component passes explicit security information, for example, the username and password, to the `getConnection` method of the `ConnectionFactory` instance.

```
// Method in an application component
Context initctx = new InitialContext();

// perform JNDI lookup to obtain connection factory
javax.resource.cci.ConnectionFactory cxf =
    (javax.resource.cci.ConnectionFactory)initctx.lookup(
        "java:comp/env/eis/MyEIS");

// Invoke factory to obtain a connection
com.myeis.ConnectionSpecImpl properties = .. // get a new
ConnectionSpec
properties.setUserName("...");
properties.setPassword("...");
javax.resource.cci.Connection cx = cxf.getConnection(properties);
...
```

8.5 EIS Sign-on

Creating a new physical connection requires a sign-on to an EIS instance. Changing the security context on an existing physical connection can also require EIS sign-on. The latter is termed re-authentication.

An EIS sign-on typically involves one or more of the following steps:

- Determine a resource principal under whose security context a physical connection to an EIS will be established.
- Authenticate a resource principal if it is not already authenticated.
- Establish a secure association between the application server and the EIS. This enables additional security mechanisms, for example, data confidentiality and integrity, to be applied to communication between the two entities.
- Set the access control to EIS resources.

8.5.1　Authentication Mechanism

An application server and an EIS collaborate to ensure resource principals are properly authenticated when the principal connects to the underlying EIS. The connector architecture identifies the following as the commonly-supported authentication mechanisms:

- **`BasicPassword`** –Basic password based authentication mechanism specific to an EIS
- **`Kerbv5`** – Kerberos version 5-based authentication mechanism

The `authentication-mechanism-type` element is used in the deployment descriptor to specify whether or not a resource adapter supports a specific authentication mechanism. Refer to Section 20.6.4, "Requirements" on page 20-24 for more details on the specification of the deployment descriptor for a resource adapter. The authentication mechanism supported by the resource adapter may also specified through the AuthenticationMechanism annotation (see Section 18.4.3, "@AuthenticationMechanism" on page 18-8) as part of the Connector metadata annotation (see Section 18.4, "@Connector" on page 18-6).

The connector architecture does not require that a specific authentication mechanism be supported by an application server and an EIS. An application server may support any other authentication mechanisms for EIS sign-on. The connector security architecture is independent of security mechanisms.

8.5.2　Resource Principal

When an application component requests a connection from a resource adapter, the connection request is made under the security context of a resource principal. The Deployer can set a resource principal based on the following options:

- **Configured Identity.** In this case, a resource principal has its own configured identity and security attributes independent of the identity of the initiating or caller principal. The identity of the resource principal can be configured either at deployment time or specified dynamically by a component at the connection creation. The scenario described in Section C.1, "eStore Application" on page C-1 illustrates an example where connections to an EIS are always established under the security context of a valid EIS user account. This happens independent of the initiating or caller principal. For example, if a caller principal is A, then the configured resource principals can be B and C on two different EIS instances, where A, B, and C are independent identities.
- **Principal Mapping**. A resource principal is determined by mapping from the identity and/or security attributes of the initiating or caller principal. In this case, a resource principal does not inherit identity or security attributes of a principal that it has been mapped from. The resource principal gets its identity and security

attributes based on the mapping. For example, if the caller principal has identity A, then the mapped resource principal is mapping(A,EIS1) and mapping(A, EIS2) on two different EIS instances.

- **Caller Impersonation**. A resource principal acts on behalf of an initiating or caller principal. Acting on behalf of a caller principal requires that the caller's identity and credentials be delegated to the EIS. The mechanism by which this is accomplished is specific to a security mechanism and an application server implementation. An example of the impersonation is described in Section C.2, "Employee Self-Service Application" on page C-4.

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

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

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

  In the case of credential mapping, the mapped resource principal has the same identity as the initiating or caller principal. For example, a principal with identity A has initial credentials cred(A,mech1) and has credentials cred(A,mech2) after mapping. mech1 and mech2 represents different mechanism types.

8.5.3 Authorization Model

Authorization checking to ensure that a principal has access to an EIS resource can be applied at one or more of the following:

- At the EIS
- At the application server

Authorization checking at the target EIS can be done in an EIS-specific way and is not specified here. For example, an EIS can define its access control policy in terms of its specific security roles and permissions.

Authorization checking can also be done at the application server level. For example, an application server can allow a principal to create a connection to an EIS only if the principal is authorized to do so. Java EE containers such as EJB and servlet containers support both programmatic and declarative security that can be used to

define authorization policies. Programmatic and declarative security are defined in the individual specifications. Refer to the EJB and servlet specifications for more details. An application component developer developing components for EIS access must follow the requirements defined in these specifications.

8.5.4 Secure Association

The communication between an application server and an EIS can be subject to security threats such as data modification and loss of data. Establishing a secure association counters such threats. A secure association is shared security information that allows a component on the application server to communicate securely with an EIS.

Establishing a secure association includes several steps:

- The resource principal is authenticated to the EIS. This may require that the target principal in the EIS domain authenticate itself back to the application server. A target principal can be set up by the system administrator as a security principal associated with a running EIS instance or specific EIS resource.
- Negotiate quality of protection such as confidentiality and integrity.
- A pair of communicating entities—an application server and an EIS instance—establish a shared security context using the credentials of the resource principal. The security context encapsulates shared state information, required so that communication between the application server and the EIS can be protected through integrity and confidentiality mechanisms. Examples of shared state information are cryptographic keys and message sequence numbers.

A secure association between an application server and an EIS is always established by the resource adapter implementation. Note that a resource adapter library runs within the address space of the application server.

A resource adapter can use any security mechanism to establish the secure association. GSS-API (refer to IETF draft on GSS-API v2[5]) is an example of such a mechanism. Note that the connector architecture does not require use of the GSS-API by a resource adapter or application server.

Configuring a mechanism for establishing secure associations is outside the scope of the connector architecture. This includes setting up the desired quality of protection during secure communication.

Once a secure association is successfully established, the connection is associated with the security context of the resource principal. Subsequently, all application-level invocations to the EIS instance using the connection happen under the security context of the resource principal.

8.6 Roles and Responsibilities

This section describes various roles involved in the security architecture. It also describes responsibilities of each role from the security perspective.

The roles and responsibilities of the Application Component Provider and Deployer are specified in detail in the respective Java EE component model specifications.

8.6.1 Application Component Provider

The following features are common across different Java EE component models from the perspective of an Application Component Provider:

- An Application Component Provider invariably avoids the burden of securing its application and focuses on developing the business functionality of its application.
- A security-aware Application Component Provider can use a simple programmatic interface to manage security at an application level. The programmatic interface enables the Application Component Provider to program access control decisions based on the security context—the principal and role—associated with the caller of a method and to manage programmatic sign-on to an EIS.
- An Application Component Provider specifies security requirements for its application declaratively through metadata annotation and deployment descriptor. The security requirements include security roles, method permissions, and an authentication approach for EIS sign-on.
- More qualified roles - Application Server Vendor, Deployer, System Administrator - have the responsibility of satisfying overall security requirements through the deployment mechanism for resource adapters and components, and managing the security environment.

8.6.2 Deployer

The Deployer specifies security policies that ensure secure access to the underlying EISs from application components. The deployer adapts the *intended* security view of an application for EIS access, specified through metadata annotations described in Chapter 18, "Metadata Annotations" or the deployment descriptor, to the *actual* security mechanisms and policies used by the application server and EISs in the target operational environment. The Deployer uses tools to accomplish the above task.

The output of the Deployer's work is a security policy descriptor specific to the operational environment. The format of the security policy descriptor is specific to an application server.

The Deployer performs the following deployment tasks for each connection factory reference declared in the deployment descriptor of an application component:

- Provides a connection factory specific security configuration that is needed for opening and managing connections to an EIS instance.
- Binds the connection factory reference in the deployment descriptor of an application component to the JNDI registered reference for the connection factory. Refer to Section 20.6, "JNDI Configuration and Lookup" on page 20-15 for the JNDI configuration of a connection factory during deployment of a resource adapter. The deployer can use the JNDI `LinkRef` mechanism to create a symbolic link to the actual JNDI name of the connection factory.
- Configures the security information for EIS sign-on, if the value of the `res-auth` deployment descriptor element is `Container`. For example, the Deployer sets up the principal mapping for EIS sign-on.

8.6.3 Application Server

The application server provides a security environment with specific security policies and mechanisms that support the security requirements of the deployed application components and resource adapters, thereby ensuring a secure access to the connected EISs.

The typical responsibilities of an application server are as follows:

- Provide tools to set up security information for a resource principal and EIS sign-on when `res-auth` element is set to `Container`. This includes support for principal delegation and mapping for configuring a resource principal.
- Provide tools to support management and administration of its security domain. For example, security domain administration can include setting up and maintaining both underlying authentication services and trusts between domains, plus managing principals, including identities, keys, and attributes. Such administration is typically security technology specific and is outside the scope of the connector architecture.
- Support a single sign-on mechanism that spans the application server and multiple EISs. The security mechanisms and policies through which single sign-on is achieved are outside the scope of the connector architecture.

Appendix D, "JAAS Based Security Architecture" specifies how JAAS can be used by an application server to support the requirements of the connector security architecture.

8.6.4   EIS Vendor

EIS provides a security infrastructure and environment that supports the security requirements of the client applications. An EIS can have its own security domain with a specific set of security policies and mechanisms, or it can be set up as part of an enterprise-wide security domain.

8.6.5   Resource Adapter Provider

The resource adapter provider provides a resource adapter that supports the security requirements of the underlying EIS.

The resource adapter implements the security contract specified as part of the connector architecture. Chapter 9, "Security Contract" specifies the security contract and related requirements for a resource adapter.

The resource adapter specifies its security capabilities and requirements through metadata annotations or its deployment descriptor. Section 20.6.4, "Requirements" on page 20-24 specifies a standard deployment descriptor for a resource adapter. Chapter 18, "Metadata Annotations" specifies the metadata annotations used to express security requirements of a resource adapter.

8.6.6   System Administrator

The system administrator typically works in close association with administrators of multiple EISs that have been deployed in an operational environment. The system administration tasks can also be performed by the Deployer.

The following tasks are illustrative examples of the responsibilities of the system administrator:

- Set up an operational environment based on the technology and requirements of the authentication service, and if an enterprise directory is supported.
- Configure the user account information for both the application server and the EIS in the enterprise directory. The user account information from the enterprise directory can then be used for authentication of users requesting connectivity to the EIS.
- Establish a password synchronization mechanism between the application server and the EIS. This ensures that the user's security information is identical on both the application server and the EIS. When an EIS requires authentication, the application server passes the user's password to the EIS.

Security Contract

This chapter specifies the security contract between the application server and the EIS. It also specifies the responsibilities of the Resource Adapter Provider and the Application Server Vendor for supporting the security contract.

This chapter references the following chapters and documents:

- The security model specified in the Java EE platform specification (see "Java Platform, Enterprise Edition (Java EE) Specification, version 7" on page E-1).
- Security architecture specified in Chapter 8, "Security Architecture".
- Security scenarios based on the connector architecture (Refer to Appendix C, "Security Scenarios").

9.1 Security Contract

The security contract between the application server and the resource adapter extends the connection management contract (described in Chapter 6, "Connection Management") by adding security-specific details.

This security contract supports EIS sign-on by:

- Passing the connection request from the resource adapter to the application server, enabling the application server to hook-in security services.
- Propagation of the security context, that is, JAAS `Subject` with principal and credentials, from the application server to the resource adapter.

9.1.1 Interfaces and Classes

The security contract includes the following classes and interfaces:

9.1.2 `Subject`

The following text has been taken from the JAAS specification. For detailed information, refer to the JAAS specification (see "Java Authentication and Authorization Service Specification, version 1.0" on page E-1).

A `Subject` represents a grouping of related information for a single entity, such as a person. Such information includes the Subject's identities and its security-related attributes, for example, passwords and cryptographic keys. A `Subject` can have multiple identities. Each identity is represented as a `Principal` within the `Subject`. A `Principal` simply binds a name to a `Subject`.

A `Subject` can also own security-related attributes, which are referred to as `Credentials`. Sensitive credentials that require special protection, such as private cryptographic keys, are stored within a private credential set.

The `Credentials` intended to be shared, such as public key certificates or Kerberos server tickets, are stored within a public credential set. Different permissions are required to access and modify different credential sets.

The `getPrincipals` method retrieves all the principals associated with a `Subject`. The methods `getPublicCredentials` and `getPrivateCredentials` respectively retrieve all the public or private credentials belonging to a `Subject`. The methods defined in the `Set` class modify the returned set of principals and credentials.

9.1.3 Resource Principal

The interface `java.security.Principal` represents a resource principal. The following code extract shows the `Principal` interface:

```
public interface java.security.Principal {
    public boolean equals(Object another);
    public String getName();
    public String toString();
    public int hashCode();
}
```

The method `getName` returns the name of a resource principal.

An application server should use the `Principal` interface, or any derived interface, to pass a resource principal as part of a `Subject` to a resource adapter.

9.1.4   GenericCredential

Note – This interface, introduced in Version 1.0 of this specification, has been deprecated. The preferred way to represent generic credential information is by way of the `org.ietf.jgss.GSSCredential` interface in J2SE Version 1.4, which provides similar functionality.

The interface `javax.resource.spi.security.GenericCredential` defines a security mechanism-independent interface for accessing the security credential of a resource principal.

The `GenericCredential` interface provides a Java wrapper around an underlying mechanism-specific representation of a security credential. For example, the `GenericCredential` interface can be used to wrap Kerberos credentials.

The connector architecture does not define any standard format and requirements for security mechanism specific credentials. For example, a security credential wrapped by a `Generic Credential` interface can have a native representation specific to an operating system.

Note – A contract for the representation of mechanism-specific credentials must be established between an application server and a resource adapter and is outside the scope of the connector architecture. This includes requirements for the exchange of mechanism-specific credentials between a JAAS module and GSS provider. Refer to Appendix D, "JAAS Based Security Architecture" for details on JAAS-based security architecture.

The `GenericCredential` interface enables a resource adapter to extract information about a security credential. The resource adapter can then manage an EIS sign-on for a resource principal by any of the following:

- Using the credentials in an EIS specific manner if the underlying EIS supports the security mechanism type represented by the `GenericCredential` instance
- Using GSS-AP I (see "RFC: Generic Security Service API (GSS-API) Specification, version 2" on page E-1) if the resource adapter and underlying EIS instance support GSS-API.

9.1.4.1 Interface

The following code extract shows the GenericCredential interface:

```
public interface javax.resource.spi.security.GenericCredential {
    public String getName();
    public String getMechType();
    public byte[] getCredentialData()
    throws javax.resource.spi.SecurityException;

    public boolean equals(Object another);
    public int hashCode();

}
```

The GenericCredential interface supports a set of getter methods to obtain information about a security credential.

The method getName returns the name of the resource principal associated with a GenericCredential instance.

The method getMechType returns the mechanism type for the GenericCredential instance. The mechanism type definition for GenericCredential must be consistent with the Object Identifier (OID) based representation specified in the GSS specification (see "RFC: Generic Security Service API (GSS-API) Specification, version 2" on page E-1). In the GenericCredential interface, the mechanism type is returned as a stringified representation of the OID specification.

The GenericCredential interface can be used to get security data for a specific security mechanism. An example is authentication data required for establishing a secure association with an EIS instance on behalf of the associated resource principal. The getCredentialData method returns the credential representation as an array of bytes. Note that the connector architecture does not define a standard format for the returned credential data.

9.1.4.2 Implementation

If an application server supports the deployment of a resource adapter which supports GenericCredential as part of the security contract, the application server must provide an implementation of the GenericCredential interface. Refer to the deployment descriptor specification in Section 20.6.4, "Requirements" on page 20-24 for details on how a resource adapter specifies its support for GenericCredential. Refer to Section 18.4.3, "@AuthenticationMechanism" on page 18-8 for details on how a resource adapter may use the AuthenticationMechanism annotation to specify its support for GenericCredential.

9.1.5 GSSCredential

This interface `org.ietf.jgss.GSSCredential` is in J2SE Version 1.4. This provides a mechanism to represent generic credential information. The functionality provided by this interface is similar to the deprecated `GenericCredential` interface.

9.1.5.1 Implementation

If an application server supports the deployment of a resource adapter which supports `GSSCredential` as part of the security contract, the application server must provide an implementation of the `GSSCredential` interface. Refer to the deployment descriptor specification in Section 20.6.4, "Requirements" on page 20-24 for details on how a resource adapter specifies its support for `GSSCredential`. Refer to Section 18.4.3 "@AuthenticationMechanism"" for details on how a resource adapter may use the AuthenticationMechanism annotation to specify its support for GSSCredential.

9.1.6 PasswordCredential

The class `javax.resource.spi.security.PasswordCredential` acts as a holder of username and password information. This class enables an application server to pass the username and password to the resource adapter through the security contract.

The method `getUserName` gets the name of the resource principal. The interface `java.security.Principal` represents a resource principal.

The PasswordCredential class must implement the equals and hashCode methods.

```
public final class javax.resource.spi.security.PasswordCredential
    implements java.io.Serializable {
    public PasswordCredential(String userName, char[] password) {
... }
    public String getUserName() { ... }
    public char[] getPassword() { ... }

    public ManagedConnectionFactory getManagedConnectionFactory()
        { ... }
    public void setManagedConnectionFactory(
        ManagedConnectionFactory mcf) { ... }

    public boolean equals(Object other) { ... }
    public int hashCode() { ... }
}
```

The getManagedConnectionFactory method returns the ManagedConnectionFactory instance for which the user name and password has been set by the application server. Refer to Section 9.1.8 "ManagedConnectionFactory"to see how a resource adapter uses this method.

9.1.7   ConnectionManager

The method `allocateConnection` is called by the resource adapter's connection factory instance. This method lets the resource adapter pass a connection request to the application server, so the application server can hook-in security and other services.

```
public interface javax.resource.spi.ConnectionManager
    extends java.io.Serializable {

    public Object allocateConnection(
    ManagedConnectionFactory mcf,
    ConnectionRequestInfo cxRequestInfo)
        throws ResourceException;
}
```

FIGURE 9-1   Security Contract
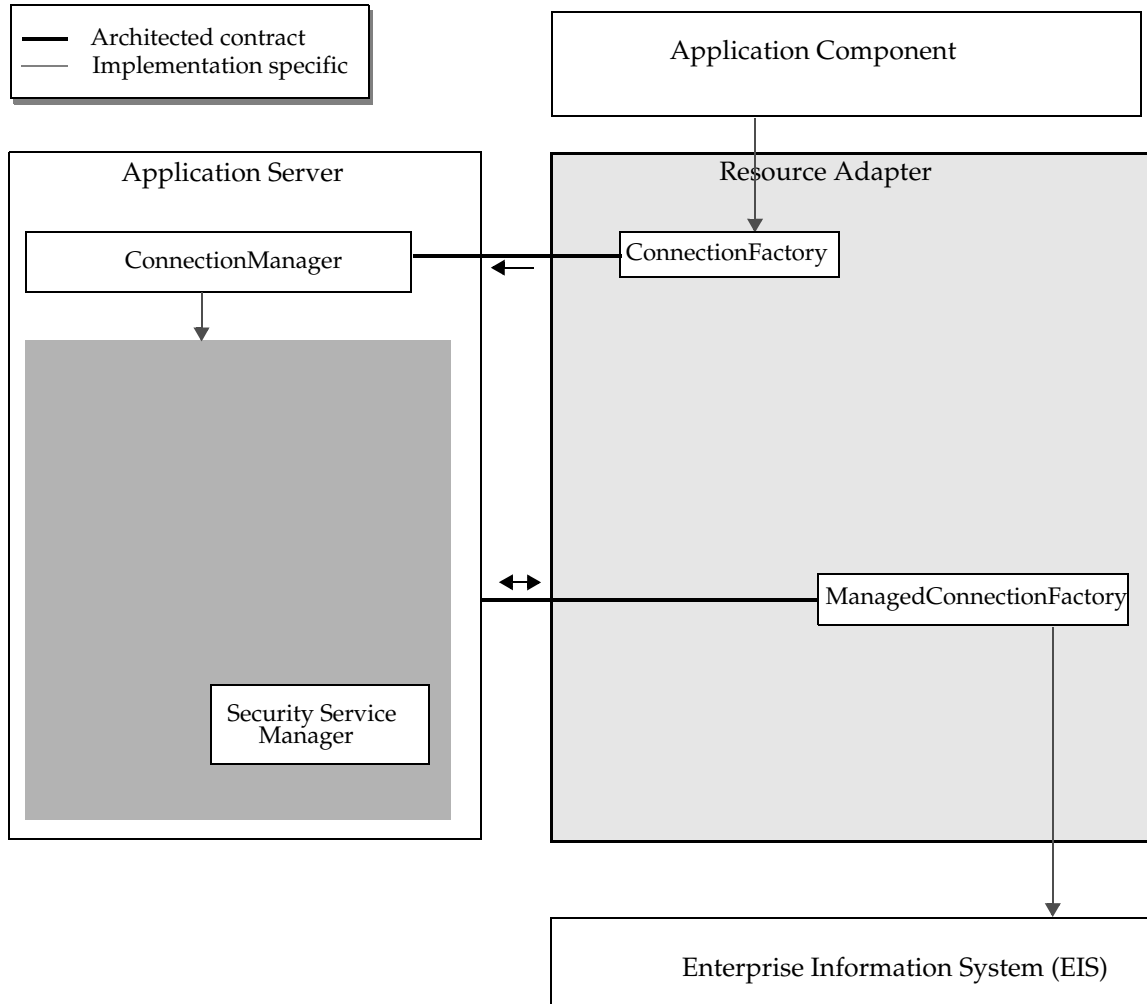
Depending on whether the application server or application component is configured to be responsible for managing EIS sign-on (refer to Section 8.6.1, "Application Component Provider" on page 8-9), the resource adapter calls the ConnectionManager.allocateConnection method in one of the following ways:

■ **Container-managed Sign-on.** The application component passes no security information in the getConnection method and the application server is configured to manage EIS sign-on.

The application server provides the required security information for the resource principal through its configured security policies and mechanisms, for example, principal mapping. The application server requests the authentication of the

resource principal to the EIS either itself or passes authentication responsibility to the resource adapter. This aspect is explained later in the specification of the `ManagedConnectionFactory` interface.

■ **Component-managed Sign-on.** In this case, the application component provides explicit security information in the `getConnection` method. The resource adapter invokes the `allocateConnection` method by passing security information in the `ConnectionRequestInfo` parameter. Since the security information in the `ConnectionRequestInfo` is opaque to the application server, the application server should rely on the resource adapter to manage EIS sign-on, as explained in the `ManagedConnectionFactory` interface specification under option C.

9.1.8 ManagedConnectionFactory

The following code extract shows the methods of the `ManagedConnectionFactory` interface that are relevant to the security contract:

```
public interface javax.resource.spi.ManagedConnectionFactory
    extends java.io.Serializable {

    public ManagedConnection createManagedConnection(
    javax.security.auth.Subject subject,
    ConnectionRequestInfo cxRequestInfo)
        throws ResourceException;
    ...
}
```

During the JNDI lookup, the `ManagedConnectionFactory` instance is configured by the application server with a set of configuration properties. These properties include default security information and EIS instance-specific information, such as hostname and port number, required for initiating a sign-on to the underlying EIS during the creation of a new physical connection.

The default security configuration on a `ManagedConnectionFactory` can be overridden by security information provided either by a component, in component managed sign-on, or by the container, in container-managed sign-on.

The `createManagedConnection` method is used by the application server when it requests the resource adapter to create a new physical connection to the underlying EIS.

9.1.8.1 Contract for the Application Server

The application server may provide specific security services, such as principal mapping and delegation, and single sign-on, before using the security contract with the resource adapter. For example, the application server can map the caller principal to a resource principal before calling the createManagedConnection method to create a new connection under the security context of the resource principal.

In container-managed sign-on, the application server is responsible for creating a Subject instance using its implementation-specific security mechanisms and configuration. This should happen before the application server calls the createManagedConnection method of the ManagedConnectionFactory. The resource adapter is driven by the application server and acts as consumer of security information in the created Subject.

If the application server maintains a cache of the security credentials, such as Kerberos ticket granting ticket (TGT), the application server should reuse the credentials as part of the newly created Subject instance. For example, the application server uses the Subject.getPrivateCredentials().add(credential) method to add a credential to the private credential set.

FIGURE 9-2   Security Contract: Subject Interface and its Containment Hierarchy
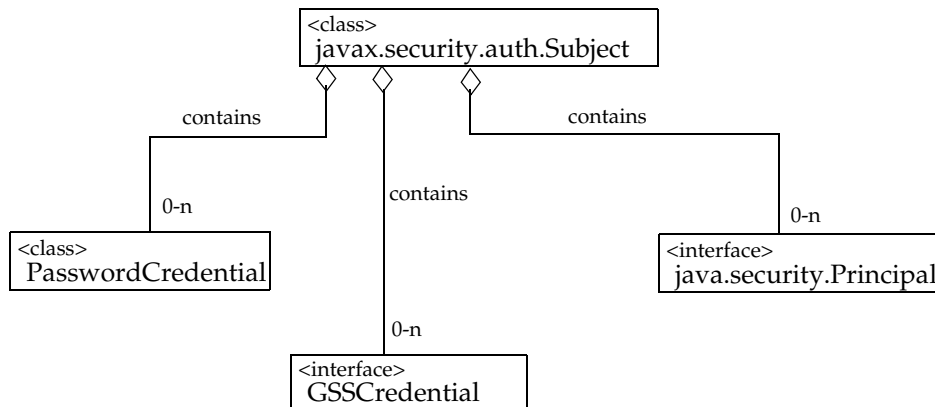
The preceding figure shows the relationship between the Subject, Principal, PasswordCredential and GSSCredential interfaces. Note that in the following options A and B defined for createManagedConnection method invocation, the Subject instance contains a single resource principal, represented as java.security.Principal, and multiple credentials.

The application server has the following options for invoking the `createManagedConnection` method:

- **Option A.** The application server invokes the `createManagedConnection` method by passing in a non-null `Subject` instance that carries a *single* resource principal and its corresponding password-based credentials, represented by the class `PasswordCredential` that provides the user name and password. The `PasswordCredential` should be set in the `Subject` instance as part of the private credential set. Note that the passed `Subject` can contain multiple `PasswordCredential` instances.

  The resource adapter extracts the username and password from this `Subject` instance by looking for the `PasswordCredential` instance in the `Subject`, and uses this security information to sign-on to the EIS instance during connection creation.

- **Option B**. The application server invokes the `createManagedConnection` method by passing in a non-null `Subject` instance that carries a *single* resource principal and its security credentials. In this option, credentials are represented through the `GSSCredential` interface. A typical example is a `Subject` instance with Kerberos credentials.

  For example, an application server may use this option for `createManagedConnection` method invocation when the resource principal is impersonating the caller or initiating principal, and has valid credentials acquired through impersonation. An application server may also use this option for principal mapping scenarios with credentials of a resource principal represented through the `GSSCredential` interface.

  Note that sensitive credentials requiring special protection, such as private cryptographic keys, are stored within a private credential set, while credentials intended to be shared, such as public key certificates or Kerberos server tickets, are stored within a public credential set. The two methods `getPrivateCredentials` and `getPublicCredentials` should be used accordingly.

  In the case of Kerberos mechanism type, the application server must pass the principal's ticket granting ticket (TGT) to a resource adapter in a private credential set.

  The resource adapter uses the resource principal and its credentials from the `Subject` instance to go through the EIS sign-on process before creating a new connection to the EIS.

- **Option C**. The application server invokes the `createManagedConnection` method by passing a `null` `Subject` instance. The application server must use this option for the component-managed sign-on case. In this option, security information is carried in the `ConnectionRequestInfo` instance. The application server does not provide any security information that can be used by the resource adapter for managing EIS sign-on.

During the deployment of a resource adapter, the application server must be configured to use one of the above specified invocation options. Refer to Chapter 20, "Packaging Requirements" for more details.

9.1.8.2 Contract for Resource Adapter

A resource adapter can do EIS sign-on and connection creation in an implementation-specific way, or it can use the GSS-API. The latter option is specified in Appendix D, "JAAS Based Security Architecture". A resource adapter has the following options, corresponding to the options for an application server, for handling the invocation of the `createManagedConnection` method:

- **Option A**. The resource adapter explicitly checks whether the passed `Subject` instance carries a `PasswordCredential` instance using the `Subject.getPrivateCredentials` method.

  Note that the security contract assumes that a resource adapter has the necessary security permissions to extract a private credential set from a `Subject` instance. The specific mechanism through which such permission is set up is outside the scope of the connector architecture.

  If the `Subject` instance contains a `PasswordCredential` instance, the resource adapter extracts the username and password from the `PasswordCredential`. It uses the security information to authenticate the resource principal, corresponding to the username, to the EIS during the creation of a connection. In this case, the resource adapter uses an authentication mechanism that is EIS specific.

  Since a `Subject` instance can carry multiple `PasswordCredential` instances, a `ManagedConnectionFactory` should only use a `PasswordCredential` instance that has been specifically passed to it through the security contract. The `getManagedConnectionFactory` method enables a `ManagedConnectionFactory` instance to determine whether or not a `PasswordCredential` instance is to be used for sign-on to the target EIS instance. The `ManagedConnectionFactory` implementation uses the `equals` method to compare itself with the passed instance.

- **Option B**. The resource adapter explicitly checks whether the passed `Subject` instance carries a `GSSCredential` instance using the `getPrivateCredentials` and `getPublicCredentials` methods defined in the `Subject` interface.

  In the case of Kerberos mechanism type, the resource adapter must extract Kerberos credentials using the `getPrivateCredentials` method in the `Subject` interface.

  The resource adapter uses the resource principal and its credentials, represented by the `GSSCredential` interface, in the `Subject` instance to go through the EIS sign-on process. For example, this option is used for Kerberos-based credentials that have been acquired by the resource principal through impersonation.

A resource adapter uses the getter methods defined in the `GSSCredential` interface to extract information about the credential and its principal. If a resource adapter is using the GSS mechanism, the resource adapter uses a reference to the `GSSCredential` instance in an opaque manner and is not required to handle any mechanism-specific credential representation. However, a resource adapter may need to interpret credential representation if the resource adapter initiates authentication in an implementation-specific manner.

- **Option C**. If the application server invokes the `ManagedConnectionFactory.createManagedConnection` method with a `null Subject` instance, a resource adapter has the following options:

  - The resource adapter should extract security information passed through the `ConnectionRequestInfo` instance. The resource adapter should authenticate the resource principal by combining the configured security information on the `ManagedConnectionFactory` instance with the security information passed through the `ConnectionRequestInfo` instance. The default behavior for the resource adapter is to allow the security information in the `ConnectionRequestInfo` parameter to override the configured security information in the `ManagedConnectionFactory` instance.
  - If the resource adapter does not find any security configuration in the `ConnectionRequestInfo` instance, the resource adapter uses the default security configuration in the `ManagedConnectionFactory` instance.
  - If the EIS does not require authentication, the resource adapter does not need any security information from the `ConnectionRequestInfo` instance, and hence may ignore such security information. This may happen due to a disconnect between the application and the resource adapter.

In the case of option A and option B, a resource adapter should throw a `javax.resource.spi.SecurityException`, if the credential information contained in the `Subject` instance is insufficient to perform authentication. A non-null `Subject` instance with no credentials is not equivalent to a null `Subject` instance, since they indicate different sign-on modes, and hence the resource adapter may handle them differently. A non-null `Subject` instance with no credentials may be interpreted by the resource adapter as follows:

- If the EIS requires authentication, the resource adapter should throw a `javax.resource.spi.SecurityException`. That is, an empty or insufficient credential information is an error.
- If the EIS does not require authentication, the resource adapter does not need any security information from the non-null `Subject` instance, and hence may ignore the `Subject` instance. This may happen due to a disconnect between the application and the resource adapter.

9.1.9 ManagedConnection

A resource adapter can re-authenticate a physical connection (that is, one that already exists in the connection pool under a different security context) to the underlying EIS. A resource adapter performs re-authentication when an application server calls the getConnection method with a security context, passed as a Subject instance, different from the context previously associated with the physical connection.

If a resource adapter supports re-authentication, the matchManagedConnections method in ManagedConnectionFactory may return a matched ManagedConnection instance with the assumption that the ManagedConnection.getConnection method will later switch the security context through re-authentication. Note that the matchManagedConnections method should consider a ManagedConnection instance as immutable. There is no authentication involved in the matchManagedConnections method.

Support for re-authentication depends on whether an underlying EIS supports the re-authentication mechanism for existing physical connections. If a resource adapter does not support re-authentication, the getConnection method should throw a javax.resource.spi.SecurityException if the passed Subject in the getConnection method is different from the security context associated with the ManagedConnection instance.

```
public interface javax.resource.spi.ManagedConnection {
    public Object getConnection(
        javax.security.auth.Subject subject,
        ConnectionRequestInfo cxRequestInfo)
                throws ResourceException;
    ...
}
```

The getConnection method returns a new connection handle. If re-authentication is successful, the resource adapter has changed the security context of the underlying ManagedConnection instance to that associated with the passed Subject instance.

A resource adapter has the following options for handling ManagedConnection.getConnection invocation if it supports re-authentication:

- **Option A**. The resource adapter extracts the PasswordCredential instance from the Subject and performs an EIS-specific authentication. This option is similar to option A defined in the specification of the method createManagedConnection on the interface ManagedConnectionFactory (refer to Section 9.1.8 "ManagedConnectionFactory").

- **Option B**. The resource adapter extracts `GSSCredential` instance from the `Subject` and manages authentication either through the GSS mechanism or an implementation-specific mechanism. This option is similar to option B defined in the specification of the method `createManagedConnection` on the interface `ManagedConnectionFactory` (refer to Section 9.1.8 "ManagedConnectionFactory").
- **Option C**. In this case, the `Subject` parameter is `null`. The resource adapter extracts security information from the `ConnectionRequestInfo` (if there is any) and performs authentication in an implementation-specific manner. This option is similar to option C defined in the specification of the method `createManagedConnection` on the interface `ManagedConnectionFactory` (refer to Section 9.1.8 "ManagedConnectionFactory").

9.2 Requirements

The following are the requirements defined by the security contract:

9.2.1 Resource Adapter

The following are the requirements defined for a resource adapter:

- The resource adapter must support the security contract by implementing the method `ManagedConnectionFactory.createManagedConnection`.
- The resource adapter is not required to support re-authentication as part of its `ManagedConnection.getConnection` method implementation.
- If the security information provided by the component or the container is not adequate to authenticate the caller, or if the security information is erroneous, the resource adapter must throw a `SecurityException` to indicate the error condition.
- The resource adapter must specify its support for the security contract as part of its deployment descriptor or through metadata annotations. The relevant deployment descriptor elements are: `authentication-mechanism`, `authentication-mechanism-type`, `reauthentication-support` and `credential-interface` (refer to Section 20.6.4, "Requirements" on page 20-24 for details). The AuthenticationMechanism annotation described in Section 18.4.3, "@AuthenticationMechanism" on page 18-8 may also be used for this purpose.

9.2.2 Application Server

The following are the requirements defined for an application server:

- The application server must use the method `ManagedConnectionFactory.-createManagedConnection` to pass the security context to the resource adapter during EIS sign-on.
- The application server must be capable of using options A and C as specified in Section 9.1.8 "ManagedConnectionFactory" for the security contract.
- The application server provides an implementation of the `GSSCredential` interface if the following conditions are both true:
  - The application server supports authentication mechanisms, specified as `authentication-mechanism-type` in the deployment descriptor, other than `BasicPassword` mechanism. For example, the application server should implement the `GSSCredential` interface to support the `kerbv5` authentication mechanism type.
  - The application server supports the deployment of resource adapters that are capable of handling `GSSCredential`, and thereby option B as specified in Section 9.1.8 "ManagedConnectionFactory", as part of the security contract.
- The application server must implement the method `allocateConnection` in its `ConnectionManager` implementation.
- The application server must configure its use of the security contract based on the security requirements specified by the resource adapter in its deployment descriptor. For example, if a resource adapter specifies that it supports only `BasicPassword` authentication, the application server should use the security contract to pass a `PasswordCredential` instance to the resource adapter.

Work Management

This chapter specifies a contract between an application server and a resource adapter that allows a resource adapter to do work, such as monitor network endpoints and call application components, by submitting `Work` instances to an application server for execution. The application server dispatches threads to execute submitted `Work` instances. This allows a resource adapter to avoid creating or managing threads directly, provides a mechanism for a resource adapter to perform work, allows an application server to efficiently pool threads, and have more control over its runtime environment. The resource adapter can control the security context and transaction context with which `Work` instances are executed.

10.1   Overview

Some resource adapters merely function as a passive library that executes in the context of an application thread. They do not need to create threads explicitly to do their work. But more sophisticated resource adapters may need threads to function properly. Such resource adapters may use threads to listen to network endpoints, process incoming data, communicate with a network peer, do its internal work, or dispatch calls to application components.

Even though a resource adapter may create Java threads directly and use them to do its work, an application server may prevent it from creating threads for efficiency, security, and manageability reasons. In such situations, a resource adapter requires a mechanism to obtain threads from an application server to do its work.

The work management contract provides such a mechanism which allows a resource adapter to submit `Work` instances to an application server for execution. The application server dispatches threads to execute submitted `Work` instances. This allows a resource adapter to avoid creating or managing threads directly, provides a mechanism for the resource adapter to do its work, and allows an application server more control over its runtime environment.

There are several advantages in allowing an application server to manage threads instead of a resource adapter:

- An application server is optimally designed to manage system resources such as threads. It may pool threads and reuse them efficiently across different resource adapters deployed in its runtime environment.
- A resource adapter may create non-daemon threads that interfere with the orderly shutdown of an application server. It is desirable for an application server to own all the threads to exercise more control over its runtime environment.
- Since an application server knows the overall state of its runtime environment, it may make better decisions on granting threads to a resource adapter, and this leads to better manageability of its runtime environment.
- An application server may need to enforce control over the runtime behavior of its system components, including resource adapters. For example, an application server may choose to intercept operations on a thread object, perform checks, and enforce correct behavior.
- An application server may disallow resource adapters from creating their own threads based on its security policy setting, enforced by a security manager.

10.2   Goals

- Provide a flexible work execution model to handle the thread requirements of a resource adapter.
- Provide a mechanism for an application server to pool and reuse threads.
- Exercise more control over thread behavior in a managed environment.

10.3   Work Management Model

A resource adapter obtains a `WorkManager` instance from the `BootstrapContext` instance provided by the application server during its deployment. The resource adapter may create `Work` instances to do its work and submit them to the `WorkManager` along with an optional execution context for execution.

The application server has a pool of free threads waiting for a `Work` instance to be submitted. When a `Work` instance is submitted, one of the free threads picks up the `Work` instance, sets up an appropriate execution context and calls the `run` method on the `Work` instance. The application server is free to choose an appropriate thread to execute the `Work` instance. There is no restriction on the number of `Work` instances submitted by a resource adapter or when `Work` instances may be submitted. When the `run` method on the `Work` instance completes, the application server reuses the thread.

The application server may decide to reclaim active threads based on load conditions. It calls the `release` method on specific `Work` instances from a separate thread. This serves only as a hint to the resource adapter to release the active thread executing the `Work` instance. The resource adapter should periodically monitor such hints and do the necessary internal cleanup to avoid any inconsistencies. It is expected that a resource adapter uses thread resources carefully and releases them when not in use.

The application server is free to implement its own thread pooling strategy. However, the application server must use threads of the same thread priority level to process `Work` instances submitted by a specific resource adapter. This ensures that multiple threads processing `Work` instances from the same resource adapter have equal claim over CPU resources. This assumption helps the resource adapter build its own internal priority-based task queue without having to worry about thread priority levels.

10.3.1   Requirements

The application server must use threads of the same thread priority level to process `Work` instances submitted by a specific resource adapter.

FIGURE 10-1 Work Management Contract (Object Diagram)
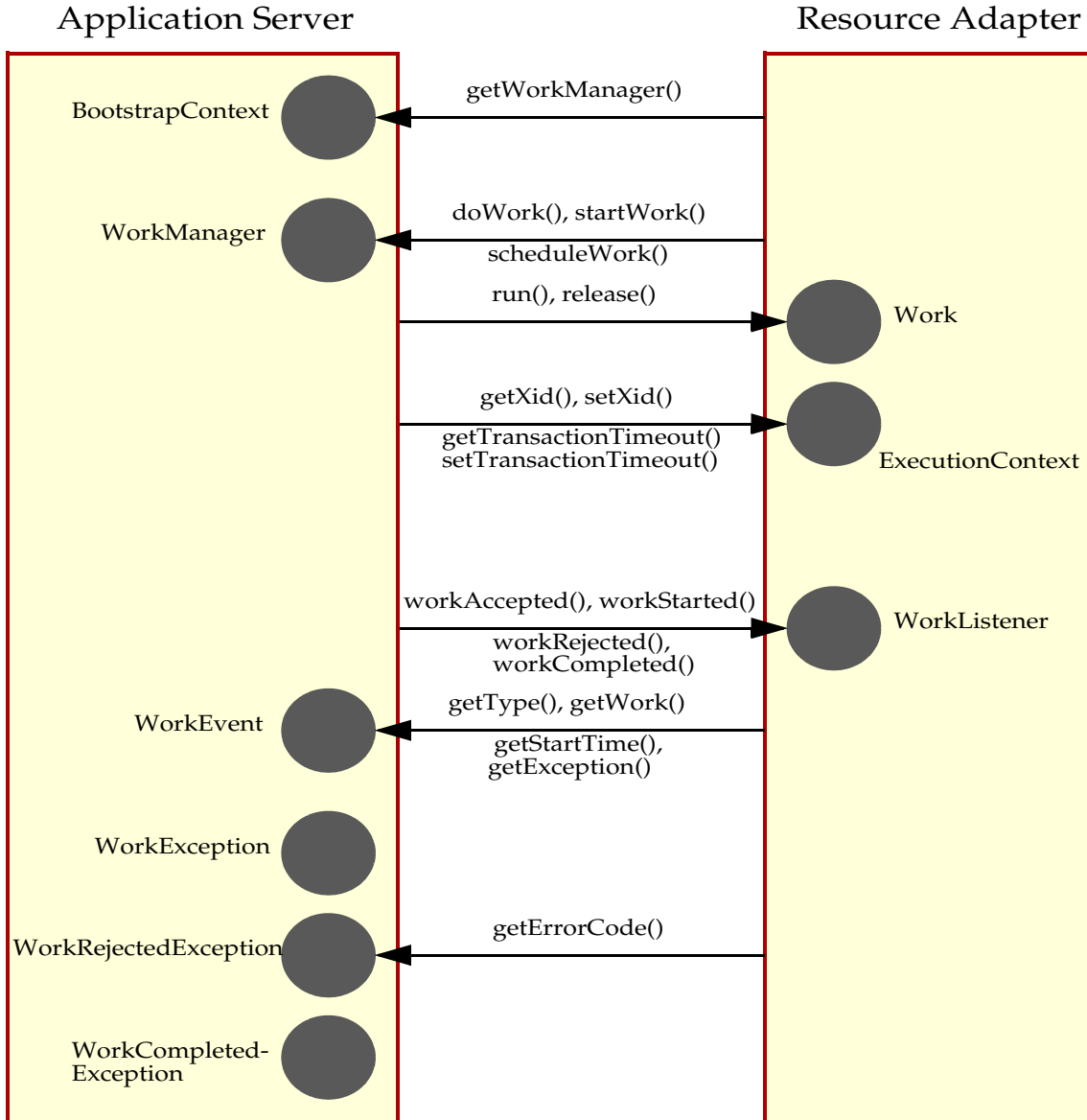
FIGURE 10-2 Work Management Contract (Interfaces)

javax.resource.spi.work

WorkManager (from app server)
doWork() startWork() scheduleWork()

Work extends java.lang.Runnable (from adapter)
release()

ExecutionContext (from adapter)
getXid() getTransactionTimeout()

WorkEvent (from app server)
getType() getWork() getStartTime() getException()

WorkListener (from adapter)
workAccepted() workRejected() workStarted() workCompleted()

WorkException extends java.lang.Exception (from app server)

WorkRejectedException extends WorkException (from app server)
getErrorCode() setErrorCode()

WorkCompletedException extends WorkException (from app server)

CODE EXAMPLE 10-1 `javax.resource.spi.work`

```java
package javax.resource.spi.work;

import java.lang.Object;
import java.lang.Runnable;
import java.lang.Exception;
import java.lang.Throwable;
import java.util.EventObject;
import java.util.EventListener;

import javax.transaction.xa.Xid;
import javax.resource.ResourceException;
import javax.resource.NotSupportedException;

public interface Work extends Runnable {
   void release();
}
public interface WorkManager {

   long IMMEDIATE = 0L; // immediate action
   long INDEFINITE = Long.MAX_VALUE; // no time constraint
   long UNKNOWN = -1; // indicates an unknown value.

   void doWork(Work work) // startTimeout = INDEFINITE
      throws WorkException;
   void doWork(Work work, long startTimeout, ExecutionContext ctx,
      WorkListener lsnr) throws WorkException;
   long startWork(Work work) // startTimeout = INDEFINITE
      throws WorkException;
   long startWork(Work work, long startTimeout,
      ExecutionContext ctx, WorkListener lsnr)
      throws WorkException;
   void scheduleWork(Work work) // startTimeout = INDEFINITE
      throws WorkException;
   void scheduleWork(Work work, long startTimeout,
      ExecutionContext ctx, WorkListener lsnr)
      throws WorkException;
}

public interface WorkListener extends EventListener {
   void workAccepted(WorkEvent e);
   void workRejected(WorkEvent e);
   void workStarted(WorkEvent e);
   void workCompleted(WorkEvent e);
}

public class WorkAdapter implements WorkListener {
   public void workAccepted(WorkEvent e) {}
```

CODE EXAMPLE 10-1 `javax.resource.spi.work`

```
    public void workRejected(WorkEvent e) {}
    public void workStarted(WorkEvent e) {}
    public void workCompleted(WorkEvent e) {}
}

public class WorkEvent extends EventObject {

    public static final int WORK_ACCEPTED = 1;
    public static final int WORK_REJECTED = 2;
    public static final int WORK_STARTED = 3;
    public static final int WORK_COMPLETED = 4;

    public WorkEvent(Object source, int type, Work work,
        WorkException exc) { ... }

    public WorkEvent(Object source, int type, Work work,
        WorkException exc, long startDuration) { ... }

    public int getType() { ... }
    public Work getWork() { ... }
    public long getStartDuration() { ... }
    public WorkException getException() { ... }
}

public class ExecutionContext {

    public void setXid(xid) { ... }
    public Xid getXid() { ... }
    public long getTransactionTimeout() { ... }
    public void setTransactionTimeout(long seconds)
        throws NotSupportedException { ... }
}

public class WorkException extends ResourceException {

    // Indicates an internal error condition.
    public static final String INTERNAL = "-1";

    // Undefined error code.
    public static final String UNDEFINED = "0";

    // Indicates start timeout expiration.
    public static final String START_TIMED_OUT = "1";

    // Indicates that concurrent work within a transaction is
    // disallowed.
    public static final String TX_CONCURRENT_WORK_DISALLOWED = "2";
```

CODE EXAMPLE 10-1   `javax.resource.spi.work`

```
    // Indicates a failure in recreating the specified transaction.
    public static final String TX_RECREATE_FAILED = "3";

    public WorkException() { ... }
    public WorkException(String message) { ... }
    public WorkException(Throwable cause) { ... }
    public WorkException(String message, Throwable cause) { ... }
    public String getMessage() { ... }
}

public class WorkRejectedException extends WorkException {

    public WorkRejectedException() { ... }
    public WorkRejectedException(String message) { ... }
    public WorkRejectedException(Throwable cause) { ... }
    public WorkRejectedException(String message, Throwable cause)
       { ... }
}

public class WorkCompletedException extends WorkException {

    public WorkCompletedException() { ... }
    public WorkCompletedException(String message) { ... }
    public WorkCompletedException(Throwable cause) { ... }
    public WorkCompletedException(String message, Throwable cause)
       { ... }
}

public class RetryableUnavailableException
    extends UnavailableException
    implements javax.resource.spi.RetryableException {
}
```

10.3.2   `Work` Interface

The `Work` interface models a `Work` instance which is executed by a `WorkManager` upon submission. This is implemented by a resource adapter.

```
public interface Work extends Runnable {
    void release();
}
```

- `run` method: The `WorkManager` dispatches a thread that calls the `run` method to begin execution of a `Work` instance. The execution completes when the `run` method returns, with or without an exception. The `Work` instance can treat the calling thread as any Java thread. However, the application server may interpose `java.lang.Thread` methods and perform checks. The `WorkManager` must catch any exception thrown during `Work` processing, which includes execution context setup, and wrap it with a `WorkCompletedException` set to an appropriate error code, which indicates the nature of the error condition.
- `release` method: The `WorkManager` may call the `release` method to request the active `Work` instance to complete execution as soon as possible. This would be called on a separate thread than the one currently executing the `Work` instance. Since this method call causes the `Work` instance to be simultaneously acted upon by multiple threads, the `Work` instance implementation must be thread-safe, and this method must be re-entrant.

The application server thread that calls the `run` method in the `Work` implementation must execute with an unspecified context if no execution context has been specified, or must execute with the specified execution context. It must have at least the same level of security permissions as that of the resource adapter instance. Further, the application server thread that calls the `run` and `release` methods, may or may not have access to a JNDI context.

Note – The JNDI context of an accessing application is available to a resource adapter by way of the thread that uses its connection object. Refer to the note in Section 6.4.1, "Managed Application Scenario" on page 6-5. The thread that accesses the connection object could be an application thread, or, could be a `Work` object accessing an application component. In the latter case, the worker thread gains access to the application's JNDI context during the method call on the component.

Both the `run` and `release` methods in the `Work` implementation may contain synchronization blocks but they must not be declared as synchronized methods.

10.3.3   `WorkManager` Interface

The `WorkManager` interface provides a mechanism to submit `Work` instances for execution. This is implemented by an application server. A `WorkManager` instance can be obtained by calling the `getWorkManager` method of the `BootstrapContext` instance. The `BootstrapContext` instance is provided by the application server when a resource adapter instance is bootstrapped. The `WorkManager` instance is not required to be unique.

This `WorkManager` facility frees the resource adapter from having to create Java threads directly to do its work. Further, this allows efficient pooling of thread resources by the application server and more control over thread usage.

```
public interface WorkManager {

    long IMMEDIATE = 0L; // immediate action (as soon as possible)
    long INDEFINITE = Long.MAX_VALUE; // no time constraint
    long UNKNOWN = -1; // unknown start delay duration

    void doWork(Work work) // startTimeout = INDEFINITE
          throws WorkException;
    void doWork(Work work, long startTimeout, ExecutionContext,
          WorkListener) throws WorkException;
    long startWork(Work work) // startTimeout = INDEFINITE
          throws WorkException;
    long startWork(Work work, long startTimeout, ExecutionContext,
          WorkListener) throws WorkException;
    void scheduleWork(Work work) // startTimeout = INDEFINITE
          throws WorkException;
    void scheduleWork(Work work, long startTimeout,
          ExecutionContext, WorkListener) throws WorkException;
}
```

- `doWork` method: This call blocks until the `Work` instance completes execution. The application server may execute a `Work` instance submitted by way of the `doWork` method using the same calling thread. This method is useful to do work synchronously. For nested `Work` submissions, this provides a first in, first out (FIFO) execution start ordering and last in, first out (LIFO) execution completion ordering guarantee.
- `startWork` method: This call blocks until the `Work` instance starts execution but not until its completion. This returns the time elapsed in milliseconds from `Work` acceptance until the start of execution. Note, this does not offer real-time guarantees. A value of -1 (`WorkManager.UNKNOWN`) must be returned, if the actual start delay duration is unknown. This method is equivalent to the `java.lang.Thread.start` method. For nested `Work` submissions, this provides a FIFO execution start ordering guarantee, but no execution completion ordering guarantee.
- `scheduleWork` method: This call does not block and returns immediately once a `Work` instance has been accepted for processing. This is useful for doing work asynchronously. This does not provide any execution start or execution completion ordering guarantee for nested `Work` submissions.

The optional `startTimeout` parameter specifies a time duration in milliseconds within which the execution of the `Work` instance must start. Otherwise, the `Work` instance is rejected with a `WorkRejectedException` set to an appropriate error code (`WorkException.START_TIMED_OUT`). Note, this does not offer real-time

guarantees. The WorkManager may also indicate that the failure to accept the Work submission is transient and that the resource adapter may retry the Work submission by throwing the RetryableWorkRejectedException.

The optional ExecutionContext parameter provides an execution context with which the Work instance must be executed. The execution context is represented by an ExecutionContext instance containing context information. The resource adapter is responsible for populating the ExecutionContext instance with an appropriate execution context. The default implementation provides a null context, that is, an ExecutionContext instance with null values. A Work instance with null context executes with an unspecified context.

The optional WorkListener parameter provides a callback event listener object which is notified when the various Work processing events (work accepted, work rejected, work started, work completed) occur. Refer to Section 10.3.4 "WorkListener Interface and WorkEvent Class".

The various stages in Work processing are:

10.3.3.1   Work Submit

A Work instance is being submitted for execution. The Work instance may either be accepted or rejected with a WorkRejectedException set to an error code. A submitted Work instance, irrespective of the mode of submission: doWork method, startWork method or scheduleWork method, does not automatically inherit the submitter's execution context. It executes with an unspecified execution context if none is specified, or it executes with the specified context.

FIGURE 10-3 Work Processing Stages and their Outcomes
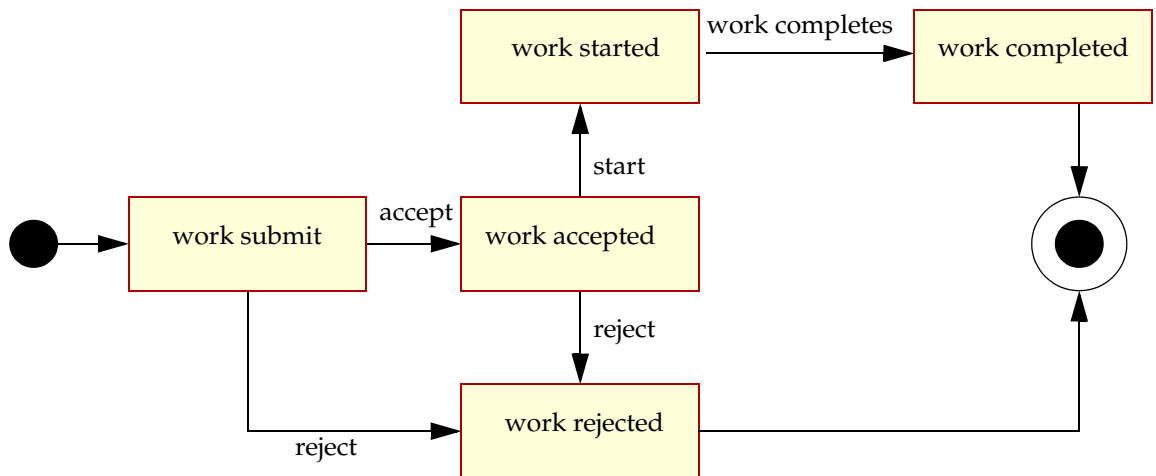
10.3.3.2   `Work` Accepted

The submitted `Work` instance has been accepted for further processing. The accepted `Work` instance may either start execution or may be rejected again with a `WorkRejectedException` set to an appropriate error code.

There is no guarantee on when the execution starts unless a start timeout duration is specified. When a start timeout is specified, the `Work` execution must be started within the specified duration, failing which a `WorkRejectedException` set to an error code `WorkException.TIMED_OUT` is thrown. This is not a real-time guarantee. The start delay duration is measured from the moment a `Work` instance is accepted for processing.

10.3.3.3   `Work` Rejected

The `Work` instance has been rejected. The `Work` instance may be rejected during `Work` submittal or after the `Work` instance has been accepted, but before `Work` instance starts execution. The rejection may be due to internal factors or start timeout expiration. A `WorkRejectedException` with an appropriate error code which indicates the nature of the error condition, is thrown in both cases.

Since the `scheduleWork` method returns after a `Work` instance has been accepted and does not block until a `Work` instance starts, a callback event listener may be used to receive the `WorkRejectedException`. See Section 10.3.4 "WorkListener Interface and WorkEvent Class" for details.

10.3.3.4   `Work` Started

The execution of the `Work` instance has started. This means a thread has been allocated for `Work` execution. But this does not guarantee that the allocated thread has been scheduled to run on a CPU resource. Once execution is started, the allocated thread sets up an appropriate execution context and calls the `run` method on the `Work` instance. Note, any exception thrown during execution context setup or while executing the `run` method on the `Work` instance leads to processing completion.

10.3.3.5   `Work` Completed

The execution of the `Work` instance has been completed. The execution may complete with or without an exception. The `WorkManager` must catch any exception thrown during `Work` processing, which includes execution context setup, and wrap it with a `WorkCompletedException` set to an appropriate error code which indicates the nature of the error condition.

Since the `scheduleWork` method and `startWork` method do not block until execution completion, a callback event listener may be used to receive the `WorkCompletedException.` See Section 10.3.4 "WorkListener Interface and WorkEvent Class" for details).

10.3.3.6   Requirements

- The application server must implement the `WorkManager` interface.
- The application server must allow nested `Work` submissions.
- Both the `run` and `release` methods must be declared as non-synchronized methods.
- When the application server is unable to recreate an execution context if it is specified for the submitted `Work` instance, it must throw a `WorkCompletedException` set to an appropriate error code.
- The `WorkManager` must catch any exception thrown during `Work` processing, which includes execution context setup and wrap it with a `WorkCompletedException` set to an appropriate error code.
- The application server must execute a submitted `Work` instance with an unspecified context if no execution context has been specified, or must execute it with the specified execution context. That is, a submitted `Work` instance must never inherit the submitter's execution context when no execution context is specified.
- If the application server is unable to start `Work` execution when a start timeout is specified for the submitted `Work` instance, it must reject the `Work` instance with a `WorkRejectedException` set to `WorkException.START_TIMED_OUT`.

■ The application server must use a value of -1 (`WorkManager.UNKNOWN`) to indicate an unknown `Work` start delay duration.

FIGURE 10-4 Blocking Durations of Various Work Submissions
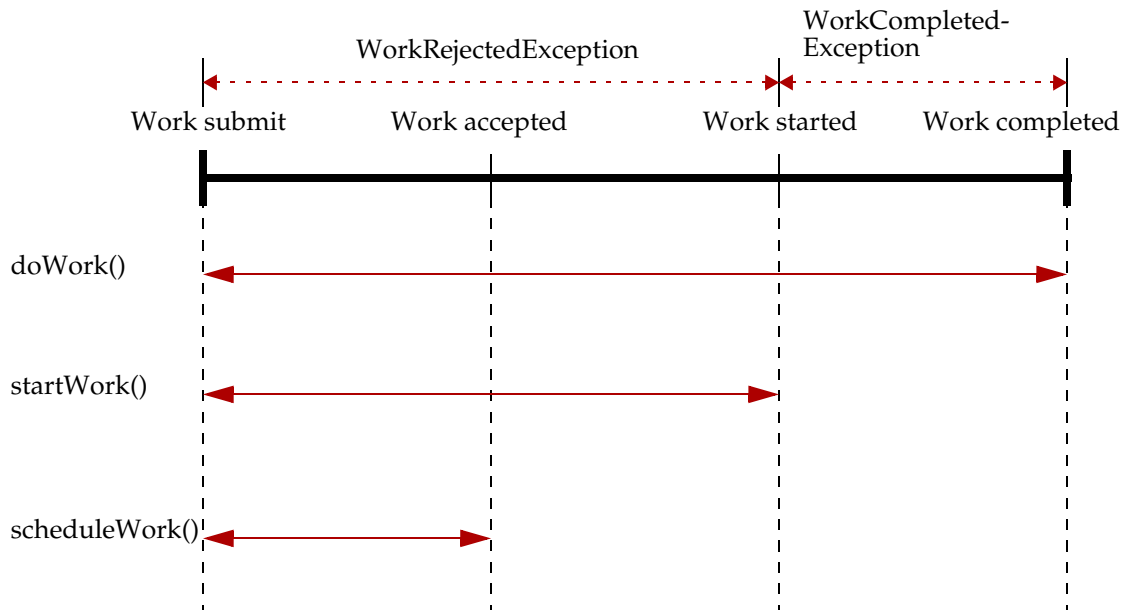
FIGURE 10-5 Work Submission - Blocking Behavior (Sequence Diagram)

| ResourceAdapter (from adapter) | WorkManager (from app server) | Java thread (from app server) | Work (from adapter) |

1. create an instance

2. doWork() [blocks until work completes]

3. dispatches a Java thread

4. set up execution context and call run()

1. create an instance

2. startWork() [blocks until work starts (that is, a thread is allocated)]

3. dispatches a Java thread

4. set up execution context and call run()

1. create an instance

2. scheduleWork() [blocks until work is accepted]

3. dispatches a Java thread

4. set up execution context and call run()

10.3.4   `WorkListener` Interface and `WorkEvent` Class

The `WorkListener` interface is optionally implemented by the resource adapter. The `WorkEvent` and `WorkAdapter` classes are defined by the Connector 1.5 specification. The `WorkListener` instance is supplied to the `WorkManager` during `Work` submittal and provides an event listener callback mechanism in order to be notified when the various `Work` processing events, such as work accepted, work rejected, work started, and work completed, occur. When a `WorkListener` is provided by the resource adapter, the application server must send event notifications to the `WorkListener`. These notifications may occur from any thread with an unspecified context.

```
public interface WorkListener extends EventListener {
    void workAccepted(workEvent);
    void workRejected(WorkEvent);
    void workStarted(WorkEvent);
    void workCompleted(WorkEvent);
}
```

The `WorkEvent` class and `WorkAdapter` abstract class:

```java
public class WorkEvent extends EventObject {

    public static final int WORK_ACCEPTED = 1;
    public static final int WORK_REJECTED = 2;
    public static final int WORK_STARTED = 3;
    public static final int WORK_COMPLETED = 4;

    public WorkEvent(Object source, int type, Work work,
    WorkException exc) { ... }

    public WorkEvent(Object source, int type, Work work,
    WorkException exc, long startDuration) { ... }

    public int getType() { ... }
    public Work getWork() { ... }
    public long getStartDuration() { ... }
    public WorkException getException() { ... }
}

public abstract class WorkAdapter implements WorkListener {
    public void workAccepted(WorkEvent e) {}
    public void workRejected(WorkEvent e) {}
    public void workStarted(WorkEvent e) {}
    public void workCompleted(WorkEvent e) {}
}
```

The `WorkEvent` instance provides the following information:

- The event type.
- The source object, that is, the `Work` instance, on which the event initially occurred.
- A handle to the associated `Work` instance.
- An optional start delay duration in millisecond.
- Any exceptions that were thrown during `Work` processing. Possible exceptions are `WorkRejectedException`, and `WorkCompletedException`.

The type of the event determines the specific contents of a `WorkEvent`.

The `WorkAdapter` class is provided as a convenience for easily creating `WorkListener` instances by extending this class and overriding only those methods of interest. This is a standard event listener pattern used in Java APIs.

10.3.4.1 Requirements

- The `WorkListener` instance must not make any thread assumptions and must be thread-safe. That is, a notification can occur from any arbitrary thread with an unspecified context.
- The application server must send `Work` events to the `WorkListener` instance, if any, provided by the resource adapter.
- The `WorkListener` implementation must not make any assumptions on the ordering of notifications.
- The application server must use a value of -1 (`WorkManager.UNKNOWN`) to indicate an unknown `Work` start delay duration.

10.3.5 ExecutionContext Class

```
public class ExecutionContext {

    public void setXid(xid) { ... }
    public Xid getXid() { ... }
    public long getTransactionTimeout() { ... }
    public void setTransactionTimeout(long seconds)
            throws NotSupportedException { ... }
}
```

The `ExecutionContext` class allows a resource adapter to specify an execution context, such as a transaction context, with which the `Work` instance must be executed. The resource adapter is responsible for populating the `ExecutionContext` instance with an appropriate execution context. The default implementation provides a null context.

It is better for `ExecutionContext` to be a class rather than an interface because:

- There is no need for a resource adapter to implement this class. It is only required to implement the context information, like transaction context.
- The resource adapter code does not have to change when the `ExecutionContext` class evolves. For example, more context types could be added to the `ExecutionContext` class in the future without forcing resource adapter implementations to change.

FIGURE 10-6 Work Submission - Callback Mechanism (Sequence Diagram)

| ResourceAdapter (from adapter) | WorkManager (from app server) | WorkListener (from adapter) | WorkEvent (from app server) | Work (from adapter) |

1. create an instance

2. create an instance

3. scheduleWork() [startWork() or doWork() may be used as well].

4. create an instance

5. workAccepted() / workRejected()

6. when Work is accepted, dispatches a thread which sets up an execution context and calls run()

7. create an instance

8. workStarted()

9. create an instance

10. workCompleted()

10.3.6 Resource Adapter Thread Usage Recommendations

- Resource adapters are strongly recommended to use the work management contract to do work and interact with the application server only from within a `Work` instance, instead of using Java threads directly. This allows the resource adapter to be maximally portable across multiple deployment environments with different security settings.
- Resource adapters are allowed to create Java threads directly as permitted by the server security settings.
- If a resource adapter chooses to use Java threads directly, it is recommended they use the threads as daemon threads, as it does not interfere with an orderly shutdown of the server.

10.3.7 Periodic Execution of `Work` Instances

A resource adapter may need to periodically execute `Work` instances. It may use the `java.util.Timer` facility available in the Java platform or may use the `BootstrapContext` instance provided by the application server to obtain a `Timer` instance.

A resource adapter may not be able to directly create a `Timer` instance, if it does not have adequate runtime permissions to create threads. This is because the `Timer` instance starts a background thread. In such a case, the resource adapter can instead use the `BootstrapContext` instance to obtain a `Timer` instance from the application server.

```
package javax.resource.spi;

import java.util.Timer;
import javax.resource.spi.UnavailableException;

public interface BootstrapContext {
    ... // other methods
    Timer createTimer() // returns a new or an unshared instance
    throws UnavailableException;
}
```

When the `createTimer` method of the `BootstrapContext` instance is invoked, the application server provides a new `Timer` instance or an unshared instance (that is, no one else has a reference) with an empty task queue. The application server must throw an `UnavailableException` if a `Timer` instance is unavailable; the resource

adapter may retry later. The application server must throw an
`java.lang.UnsupportedOperationException`, if it does not support the `Timer`
service.

Sample code to illustrate periodic `Work` executions using a `Timer` instance:

```
package com.xyz.adapter;

import java.util.*;
import javax.resource.spi.*;
import javax.resource.spi.work.WorkManager;

// ResourceAdapter JavaBean
public class MyResourceAdapterImpl implements ResourceAdapter {
    BootstrapContext bootstrapCtx = null;
    public void start(BootstrapContext ctx) {
        bootstrapCtx = ctx;
        ... // other operations
    }
    ... // other methods
}

{ // sample resource adapter code snippet to show Timer usage
    MyResourceAdapterImpl myRA = ... // get ResourceAdapter
JavaBean
    Timer timer = myRA.bootstrapCtx.createTimer(); // get a Timer
instance
    WorkManager workManager = myRA.bootstrapCtx.getWorkManager();

    timer.schedule(
        new TimerTask () {
            public void run() {
                try {
             workManager.scheduleWork(new MyWork());
              } catch (WorkException we) { we.printStackTrace(); }
            }
        }, 0, 1000); // one second interval
}
```

10.3.8 Illustration: Using a `Work` Instance to Listen on Multiple Network Endpoints

J2SE Version 1.4 provides the `java.nio` package that includes a multiplexed, non-blocking I/O facility. Using the `java.nio` package it is possible for a single thread, such as a `Work` instance, to listen on multiple network endpoints or ports. Prior to the `java.nio` facility each network endpoint needed a separate thread to listen to incoming data.

10.3.9 Work Management in a Non-Managed Environment

Although the work management contract is primarily intended for a managed environment, it may still be used in a non-managed environment provided the application that bootstraps a resource adapter instance is capable of functioning as a `WorkManager`.

A resource adapter is free to create Java threads as permitted by the security policy settings of the non-managed environment.

10.3.10 Resource Adapter association

A `Work` or `DistributableWork` instance (see Section 10.3.11 "Distributed Work processing") may implement the `ResourceAdapterAssociation` interface. The `ResourceAdapterAssociation` interface specifies the methods to associate the `Work` instance with a `ResourceAdapter` JavaBean.

The application server must establish an association between the resource adapter instance and the `Work` instance before the exection of the `Work` instance has been started (Refer Section 10.3.3.4 "Work Started").

When a `Work` instance has been distributed to a new `WorkManager` instance (for example, as in Section 10.3.11 "Distributed Work processing"), the resource adapter instance that is associated with the `Work` instance must be available in the `WorkManager` instance that the `Work` has been distributed to. This allows the `Work` instance to use application server facilities like `WorkManager`, `MessageEndpointFactory` etc that are specific to the instance that the `Work` has been distributed to.

10.3.11   Distributed Work processing

An application server instance's `WorkManager` may choose to distribute a `Work` instance submitted by a resource adapter to another `WorkManager` residing in a different application server instance. Distribution of `Work` processing to different instances may be done for achieving optimal utilization of system resources or for providing better response times. These `WorkManager` instances may span across multiple Java virtual machines running on the same host or different hosts.

Neither the application server nor the resource adapter must support distributed Work processing.

10.3.11.1   `DistributableWork` Interface

```
package javax.resource.spi.work;

import java.io.Serializable;

//Marker interface to indicate to the WorkManager that the
//Work may be distributed to a different WorkManager for execution
public interface DistributableWork extends Work, Serializable {

}
```

`Work` instances that may be distributed by a `WorkManager` must implement the `DistributableWork` interface. A `Work` instance that implements the `DistributableWork` interface must not have any reference to local resource-adapter state. This allows the `WorkManager` to delegate processing of the `Work` instance to a different `WorkManager` instance that is running in a different Java virtual machine.

All artifacts that may be coupled to the application server instance where the `Work` is executed in, must be obtained through the `ResourceAdapterAssociation` mechanism discussed in Section 10.3.10 "Resource Adapter association".

10.3.11.2   `DistributableWorkManager` Interface

```
package javax.resource.spi.work;

//Marker interface to indicate that the WorkManager supports the
//distributed processing of Work instances
public interface DistributableWorkManager extends WorkManager {
}
```

A `WorkManager` implementation that supports the submission of
`DistributableWork` instances must implement the `DistributableWorkManager`
marker interface. This allows the resource adapter to programmatically determine
whether the `WorkManager` supports the submission of `DistributableWork`
instances.

When a `DistributableWork` instance is submitted to
`DistributableWorkManager`, the `WorkManager` may finally execute the `Work`
instance in the context of another `WorkManager` instance. This `WorkManager`
instance may reside on a different host, process or JVM instance. This specification
does not define the communication protocol or the mechanics of how a `Work`
instance is transmitted and handled between `DistributableWorkManager`
instances.

The application server that supports `DistributableWorkManager` along with
inputs from the administrator and deployer, must ensure that the environment made
available to the `DistributableWork` instance is consistent irrespective of whether
the `DistributableWork` instance is executed in a local or remote manner.

10.3.11.3   `DistributableWork` Submission and Processing

A resource adapter submits the `DistributableWork` instance to the
`DistributableWorkManager` through the `WorkManager` submission methods
specified in Section 10.3.3 "WorkManager Interface". A
`DistributableWorkManager` may then distribute the submitted
`DistributableWork` instance to another `WorkManager` instance for processing as
shown in the following figure.

When a `DistributableWork` instance is submitted to a `WorkManager` that does not
implement `DistributableWorkManager` interface, the `WorkManager` must execute
the `Work` locally.

Although it is recommended for a `DistributableWorkManager` to process all `Work`
submissions in a distributed fashion, the `DistributableWorkManager` may execute
a `Work` submitted through `doWork()` locally. When a `WorkListener` is provided by

the resource adapter during `Work` submission, the application server must send event notifications to the `WorkListener`. (see Section 10.3.4 "WorkListener Interface and WorkEvent Class").

A `DistributableWork` instance may also use the mechanisms described in Generic Work Context and Security Inflow (see Chapter 11, "Generic Work Context" and Chapter 16, "Security Inflow") chapters to control the execution context of the `Work` instance. A `DistributableWorkManager` must support the requirements in Chapter 11, "Generic Work Context" and Chapter 16, "Security Inflow".

FIGURE 10-7 Distributed Work submission and processing (Sequence Diagram)
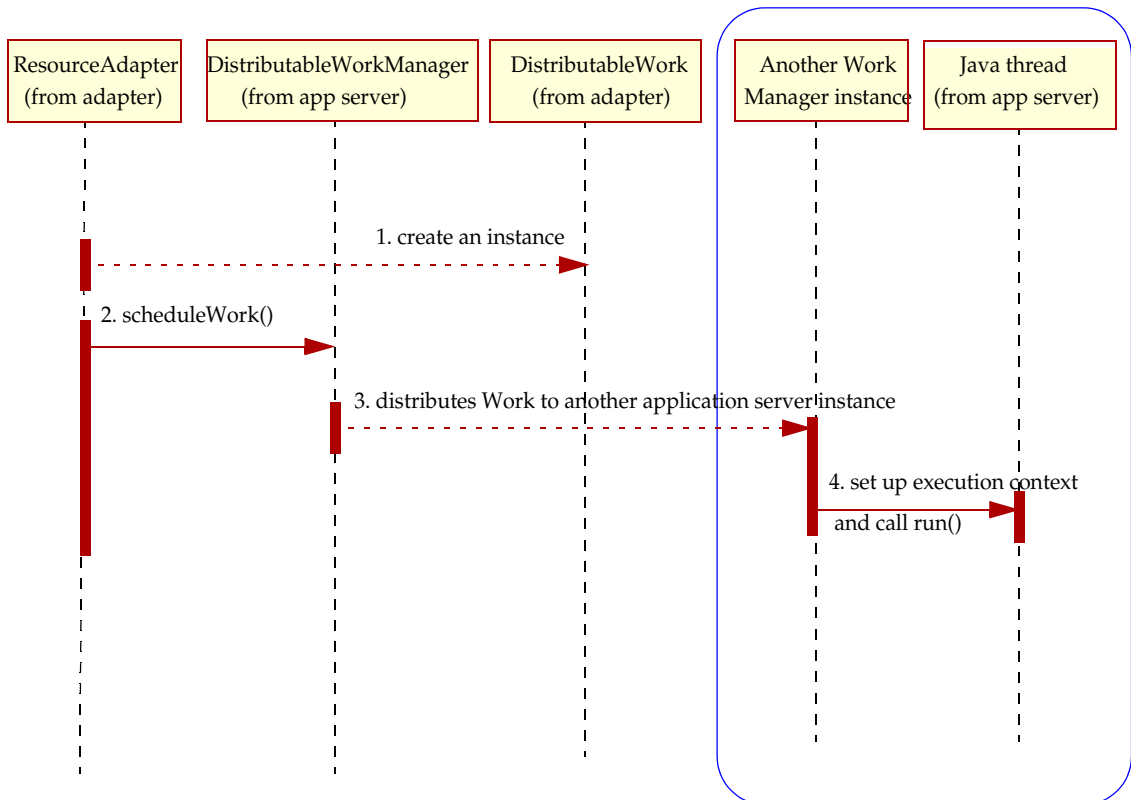
Generic Work Context

This chapter specifies a contract between an application server and a resource adapter that enables a resource adapter to control the execution context of a `Work` instance that it has submitted to the application server for execution. To propagate an imported context to the application server, the resource adapter submits a `Work` instance that implements the `WorkContextProvider` interface. The application server then establishes the provided context as the execution context of the `Work` instance during its execution. The `WorkContext` model is designed to be generic so that a resource adapter can flow in different types of contextual information apart from the standard transaction and security `WorkContexts` defined in this chapter. For more information about `Work` management, see Chapter 10, "Work Management".

11.1   Overview

The Work Management contract between the application server and a resource adapter enables a resource adapter to do a task, such as communicating with the Enterprise Information System (EIS) or delivering messages, by delivering `Work` instances for execution. The Transaction Inflow contract builds upon the interfaces defined in the Work Management contract as described in Chapter 15, "Transaction Inflow". The contract enables the resource adapter to propagate an imported transaction from the EIS to an application server, so that the application server and subsequent participants can do work as part of the imported transaction.

The Generic Work Context Contract provides the mechanism for a resource adapter to augment the runtime context of a `Work` instance with additional contextual information flown-in from the EIS. This contract enables a resource adapter to control, in a more flexible manner, the contexts in which the `Work` instances it submits are executed by the application server's `WorkManager`.

A Generic Work context mechanism also enables an application server to support new message inflow and delivery schemes. It also provides a richer contextual `Work` execution environment to the resource adapter while still maintaining control over concurrent behavior in a managed environment.

Note that the application server is required to support the standard context types listed in Section 11.3.1 "Standard and Custom Work Contexts".

11.2 Goals

The goals of the Generic Work Context Contract are:

- To provide a standard mechanism for a resource adapter to propagate an imported context to an application server.
- To make the existing execution context mechanisms extensible and to provide better metadata to both the application server and the resource adapter of new work context types.
- To design the work context contracts to be independent of the Connectors Work Management Contract so as to enable the resource adapter to use such contexts in other asynchronous task execution approaches. For more information on `Work` Management, see Chapter 10, "Work Management".
- To standardize the most commonly used work contexts, such as Transaction Work Context and Security Work Context. See Chapter 16, "Security Inflow".
- To be backward compatible with the existing `Work` submission and context assignment model described in Chapter 10, "Work Management".
- To enable an application server to support new message inflow and delivery schemes and provide a richer contextual `Work` execution environment to the resource adapter while still maintaining control over concurrent behavior in a managed environment.

11.3 Generic Work Context Model

In this chapter all references to `WorkManager` should be read as references applicable to the Connector `WorkManager`. See Section 10.3.3, "WorkManager Interface" on page 10-9.

When a `Work` is submitted by a resource adapter to a `WorkManager` to be executed asynchronously, one of the free threads picks up the `Work` instance, sets up an appropriate execution context and then calls the run method on the `Work` instance. See Section 10.3, "Work Management Model" on page 10-2 for more information on how a `Work` instance is handled by a `WorkManager`.

A resource adapter submits a `Work` instance that implements `WorkContextProvider`. The `WorkContextProvider` interface indicates to the application server's `WorkManager` that the resource adapter requires additional work contexts to be established in the execution context during `Work` execution.

When one of the free threads from the application server's thread pool picks up the `Work` instance, if the `Work` instance implements `WorkContextProvider` interface, it iterates through the collection of `WorkContexts` provided by the `Work` instance and establishes the contextual information provided by the `WorkContexts` as the execution context of the `Work` instance. It then calls the `run` method to execute the `Work` instance.

The application server is free to use the `WorkContext` during context assignment in any order. The resource adapter must not assume an order in the handling of the `WorkContexts`.

11.3.1 Standard and Custom Work Contexts

Certain EIS integration use cases require the propagation of other contextual information, apart from Transactions, from the EIS to the application server. For example, a resource adapter might require the propagation of security context information from the EIS to the application server during inbound message delivery. The resource adapter might also require the execution of `Work` instances in the context of the "flown-in" Security information. Other use cases that require the flowing in of contextual information are:

- Scenarios where an EIS requires a "conversational" programming model with a `MessageEndpoint` and the resource adapter is required to propagate "correlation" information to the `MessageEndpoint` container to enable the application server to set up or re-create the necessary state in the `MessageEndpoint` to maintain conversational session state.
- Propagating Availability or Quality-of-Service (QoS) related hints or metadata from the EIS so that the application server `WorkManager` can execute the `Work` instance by leveraging those hints.

Transaction and Security work contexts are standardized by means of the `TransactionContext` and `SecurityContext` interfaces. The propagation of Quality-of-Service hints to a `WorkManager` for the execution of a `Work` instance is standardized through the `HintsContext` class. The application server must support these three work contexts. A portable resource adapter can assume an application server's support for these three work contexts defined in the specification. The specification may define additional context types in a future version of the specification.

An application server or a resource adapter may define and use custom `WorkContexts`. However a resource adapter using these custom `WorkContexts` is non-portable and might not function as expected in other application servers that do

not implement the custom `WorkContext`. See Section 11.4.2 "Checking Support for a WorkContext Type" for a discussion about how resource adapters can check with the `WorkContexts` supported by the application server.

11.3.2   Requirements

- The application server must support the establishment of `TransactionContext`, `SecurityContext`, and `HintsContext` contexts.
- The application server must support the `WorkContext` interface. If a resource adapter submits a `Work` instance implementing the `WorkContextProvider` interface, the application server must use the `WorkContexts` provided by the resource adapter to assign the execution context for that `Work` instance.

FIGURE 11-1 Generic Work Context (Object Diagram)
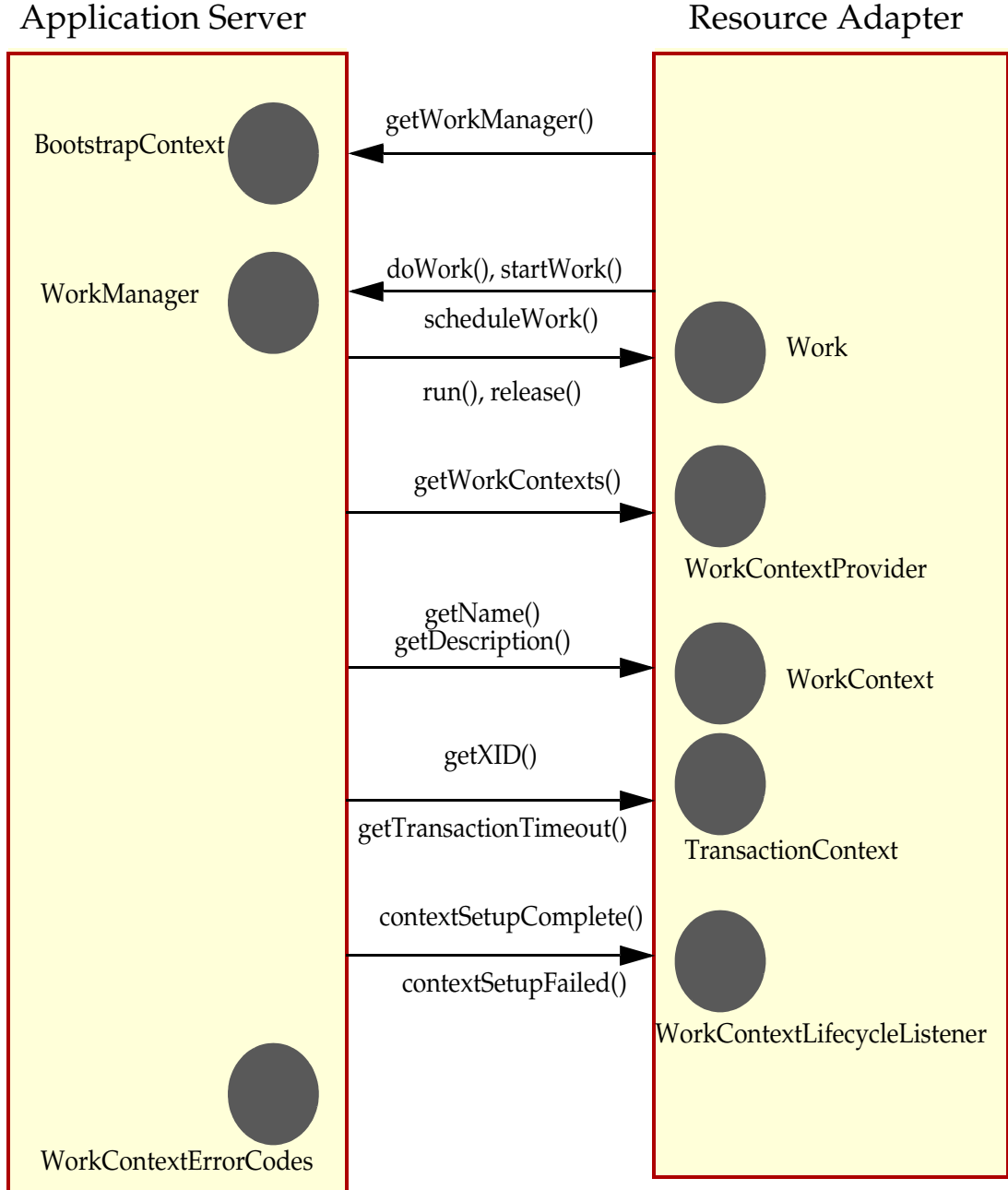
FIGURE 11-2 Generic Work Context (Interfaces)
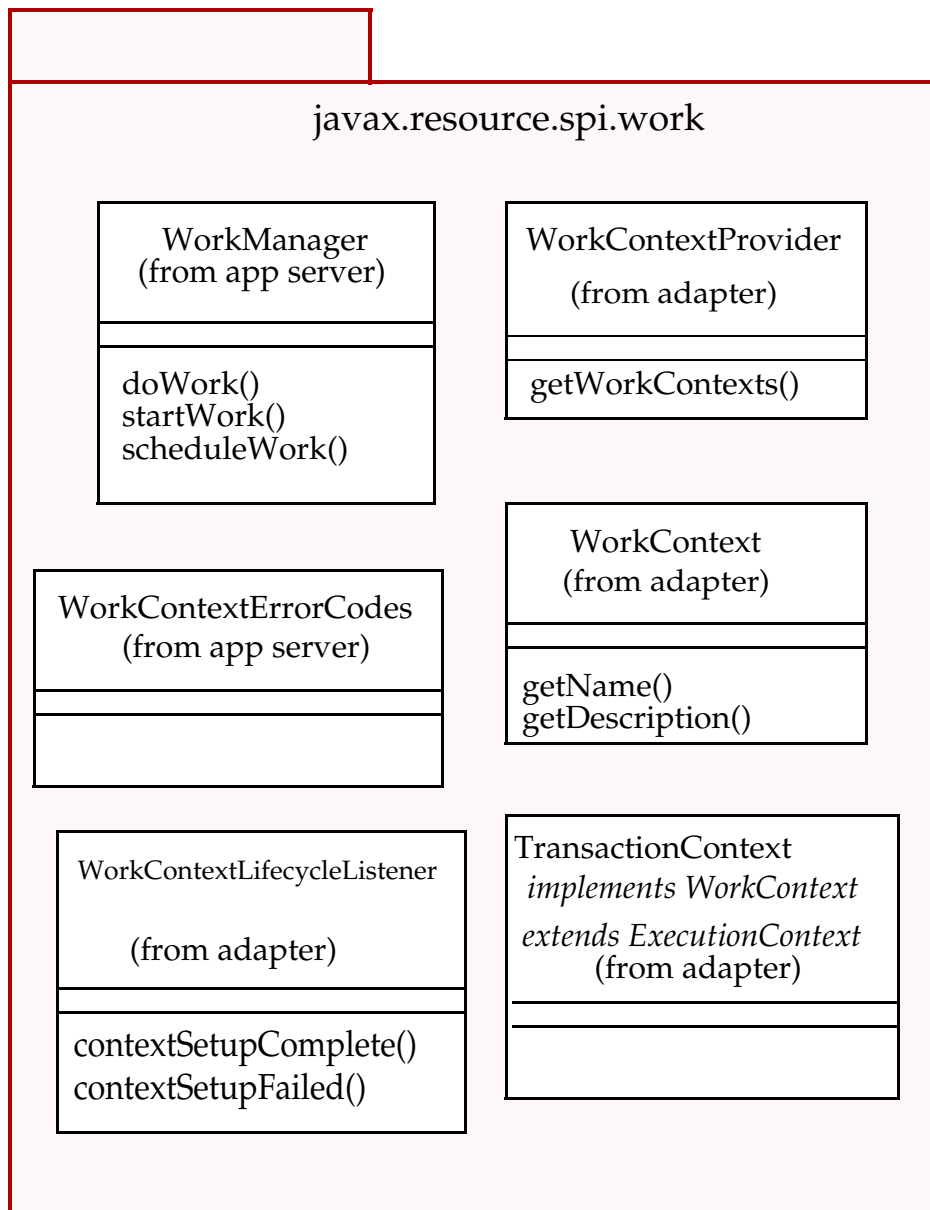
CODE EXAMPLE 11-1 `javax.resource.spi.work`

```
package javax.resource.spi.work;

public interface WorkContextProvider extends Serializable {
        List<WorkContext> getWorkContexts();
}

public interface WorkContext extends Serializable{
      String getName();
      String getDescription();
}

public class TransactionContext extends ExecutionContext
                              implements WorkContext {
      public TransactionContext(Xid xid) { ... }
      public TransactionContext( Xid xid,
                              long timeout){ ... }
      public String getName(){
            return "TransactionContext";
      }
      ... other methods
}

public abstract class SecurityContext implements WorkContext {
      public String getName(){
            return "SecurityContext";
      }
      .... other SecurityContext related methods
}

public class WorkContextErrorCodes {
       // Indicates an unsupported context type
      public static final String
            UNSUPPORTED_CONTEXT_TYPE = "1";

       // Indicates more than one contexts
       // of the same type passed in for Work
      public static final String DUPLICATE_CONTEXTS = "2";

      // Indicates failure in recreating the WorkContext
      public static final String CONTEXT_SETUP_FAILED = "3";

      // Indicates that the container cannot support
      // recreating the context
      public static final String
            CONTEXT_SETUP_UNSUPPORTED = "4";
}
```

```
public interface WorkContextLifecycleListener {
      // indicates that the WorkContext was set successfully
      void contextSetupComplete();

      // Indicates that the WorkContext setup failed
      void contextSetupFailed(String errorCode);
}
```

11.4   `WorkContextProvider` and `WorkContext` Interface

The `WorkContext` interface illustrates execution context information of a particular type. This specification standardizes two `WorkContext` types: the `TransactionContext` class and `SecurityContext` class, to represent the transaction and security context with which the `Work` instance must be executed respectively. For more information on these classes, see Section 11.5 "TransactionContext Class" and Section 16.4, "SecurityContext Class" on page 16-7.

The `getName()` and `getDescription()` methods may be used by the resource adapter developer and the application server for debugging purposes.

```
package javax.resource.spi.work;
public interface WorkContext extends Serializable{
      String getName();
      String getDescription();
}
```

Additional work contexts, based on specific EIS integration scenarios could be supported by an application server and the resource adapter may use them.

The `WorkContextProvider` interface is an optional interface implemented by a `Work` instance to indicate to the `WorkManager`, or its equivalent in other thread pooling implementations, that the task encapsulated as the `Work` instance requires to be run with a specialized execution context.

```
package javax.resource.spi.work;
public interface WorkContextProvider extends Serializable {
      List<WorkContext> getWorkContexts();
}
```

When a resource adapter is required to control the execution context in which a
`Work` instance is executed, it creates a `Work` instance that implements
`WorkContextProvider`. The `Work` instance provides an implementation of the
`getWorkContexts` method to return a List of `WorkContexts` that the `Work`
instance requires established as its execution context prior to execution.

When a `Work` that implements `WorkContextProvider` is submitted to the
`WorkManager` for execution, one of the free threads in the thread pooling
implementation of the application server picks up the `Work` for execution. The
`WorkManager` makes a call to `getWorkContexts` to obtain the `WorkContexts` that
is required to be set as the execution context for the `Work` instance, iterates through
the returned List of `WorkContexts`, and sets them up as the execution context in
which the `Work` instance is executed in.

If the resource adapter returns a null or an empty List when the WorkManager
makes a call to the getWorkContexts method, the WorkManager must treat it as if no
additional execution contexts are associated with that Work instance and must
continue with the Work processing.

When the container's thread has completed the handling of the `Work` instance, it
must cleanup all the contextual information associated with that `Work` instance so
that when the thread is reused for another `Work` instance, the previous contextual
information is not established for the new `Work` instance.

The resource adapter must not make any changes to the state of a `WorkContext`
after the `Work` instance that is associated with that `WorkContext` has been
submitted to the `WorkManager`.

Because nested `Work` submissions are allowed in the Connector `WorkManager`, the
Connector `WorkManager` must support nested contexts unless the `WorkContext`
type prohibits them. See Section 10.3.3, "WorkManager Interface" on page 10-9 for
more information on nested `Work` submission related requirements.

FIGURE 11-3 WorkContext establishment during Work submission(Sequence Diagram)

ResourceAdapter (from adapter)	WorkManager (from app server)	Work (from adapter)	TransactionContext (from adapter)

1. create an instance

2. `scheduleWork()` [`startWork()` or `doWork()` may be used as well].

3. when `Work` is accepted, dispatch a free thread and establish execution context

opt

[if `Work` implements `WorkContextProvider`]

4. `getWorkContexts()` from `Work`

loop

[for each `WorkContext`]

5. get context information and recreate imported transaction in application server

6. after establishing the execution context, call run()

11.4.1   Indicating Support for a `WorkContext` Type

A resource adapter provider can declare that it requires a list of `WorkContext` types to be supported by the application server through the `required-work-context` element in the deployment descriptor of the resource adapter (see Section 20.4.1, "Resource Adapter Provider" on page 20-7) or by way of the Connector annotation (see Section 18.4, "@Connector" on page 18-6).

The application server must check whether all of the `WorkContext` types declared by the resource adapter are supported by the application server during resource adapter deployment. The application server must employ an exact type equality check (by using `java.lang.Class.equals(java.lang.Class)`) to check for the support.

If the application server cannot support one or more of the `WorkContext` types declared in `required-work-context` elements, it must fail deployment of the resource adapter.

11.4.2   Checking Support for a `WorkContext` Type

A resource adapter can check an application server's support for a particular `WorkContext` type through the `isContextSupported()` method in the `BootstrapContext` implementation provided by the application server. This mechanism enables a resource adapter developer to dynamically change the `WorkContexts` based on the support provided by the application server. Fore more information, see Section 5.3.1, "ResourceAdapter JavaBean and Bootstrapping a Resource Adapter Instance" on page 5-4.

```
public interface BootstrapContext {
    // ... other operations
    boolean isContextSupported(
            Class<? extends WorkContext> workContextClass);
}
```

The application server must employ an exact type equality check (by using `java.lang.Class.equals(java.lang.Class)`) in `isContextSupported`, to check whether it supports the `WorkContext` type provided by the resource adapter. This method must be idempotent, that is, all calls to this method by a resource adapter for a particular `WorkContext` type must return the same Boolean value throughout the lifecycle of that resource adapter instance.

This exact type check in `isContextSupported` enables a resource adapter to decide whether the application server supports the contexts that the resource adapter is attempting to establish for a `Work` instance. If a particular `WorkContext` class is not supported by the application server a resource adapter may then either choose to fall back to a superclass that is supported by the application server (again ascertained by way of the `isContextSupported` method) or fail the `Work` submission.

For `WorkContext` classes that are defined as abstract classes, such as `SecurityContext`, the resource adapter must use the abstract class while invoking the `isContextSupported` method and not its implementation class. For more information on `SecurityContext` class, see Section 16.4, "SecurityContext Class" on page 16-7

For custom extensions of the standard `WorkContexts`, the resource adapter must always check support for the most specific `WorkContext` first. It may then go up the inheritance hierarchy in order to find the most specific `WorkContext` type supported by the application server.

11.4.3   Handling Errors During Context Assignment

As specified in Section 10.3.4, "WorkListener Interface and WorkEvent Class" on page 10-16, the `WorkManager` must catch any exception thrown during Work processing, which includes execution context setup (including Section 11.4.2 "Checking Support for a WorkContext Type"), and wrap it with a `WorkCompletedException` set to an appropriate error code defined in `WorkContextErrorCodes`, which indicates the nature of the error condition.

```
public class WorkContextErrorCodes {
    // Indicates an unsupported context type
    public static final String UNSUPPORTED_CONTEXT_TYPE = "1";

    // Indicates more than one contexts of the same type passed
    // in for Work
    public static final String DUPLICATE_CONTEXTS = "2";

    // Indicates failure in recreating the WorkContext
    public static final String CONTEXT_SETUP_FAILED = "3";

    // Indicates that the container cannot support recreating
    // the context
    public static final String CONTEXT_SETUP_UNSUPPORTED = "4";
}
```

The application server must make the following checks during context assignment

- Because not all `WorkContext` instances provided by the resource adapter might be supported by the application server, the application server must ensure that the `WorkContexts` provided by the resource adapter are supported by the application server.
- The application server must also ensure that the `WorkContexts` provided by the resource adapter do not have duplicates. For instance, a resource adapter must not be able to submit two instances of the `TransactionContext` class. The application server must ensure that only one `WorkContext` provided by the resource adapter implements the same `WorkContext` type supported by the application server. If duplicates are detected, the application server must fail the Work submission with a `WorkCompletedException` set to the `DUPLICATE_CONTEXTS` error code.

The check for support and duplicates during context assignment listed above, must be less strict than the checks described in Section 11.4.1 "Indicating Support for a WorkContext Type" and Section 11.4.2 "Checking Support for a WorkContext Type". The application server must employ a `java.lang.Class.isAssignable(java.lang.Class)` style check. Specifically, this method must check whether a WorkContext class that is supported by the application server can be converted to the type provided by the resource adapter, by way of an identity conversion or a widening reference conversion.

If a particular `WorkContext` type provided by the resource adapter is supported by the application server, the application server must use the `WorkContext` as-is and not attempt to use it as a supported parent type. That is, an application server must use the most specific `WorkContext` type it supports.

If a particular `WorkContext` type provided by the resource adapter is not supported by the application server, the application server should be able to safely fallback to a superclass (excluding the `WorkContext` interface) that is supported by it.

If the above conditions are not met, the application server must fail the `Work` processing with a `WorkCompletedException` with an appropriate error code to indicate the nature of the error condition. Because the `WorkCompletedException` might not provide a resource adapter with adequate information about the actual failure during context assignment, the resource adapter may implement the `WorkContextLifecycleListener` to interpret the reasons why a context assignment of a particular `WorkContext` instance failed. For more information, see Section 11.7 "WorkContextLifecycleListener Interface"

11.5   TransactionContext Class

The TransactionContext class extends the ExecutionContext class, as described in Section 10.3.5, "ExecutionContext Class" on page 10-18. It represents the standard interface a resource adapter can use to propagate transaction context information from the EIS to the application server. The Work instance and any message deliveries to MessageEndpoints in that Work instance must all be carried out in the transaction context provided by the TransactionContext class.

```
public class TransactionContext
               extends ExecutionContext
               implements WorkContext {
    public TransactionContext(Xid xid) {..}
    public TransactionContext(Xid xid, long timeout) {..}

    public String getDescription() {
        return "Transaction Context";
    }

    public String getName() {
        return "TransactionContext";
    }
}
```

For a resource adapter, using the WorkContextProvider interface to effect transaction inflow is optional but recommended. A resource adapter could still continue to use the existing Work submission approach with an ExecutionContext and an application server must support this model as well.

A resource adapter must not submit a Work instance that implements WorkContextProvider along with a valid ExecutionContext to a Connector WorkManager. When such a Work instance is submitted to the Connector WorkManager for execution, the application server must detect this scenario and throw a WorkRejectedException to indicate this error scenario. A resource adapter however, could choose to use a null value for the ExecutionContext parameter in Connector WorkManager methods that takes an ExecutionContext as an argument.

11.6   `HintsContext` Interface

An application server's `WorkManager` implementation may allow a `Work` instance to provide, during `Work` submission, application-server specific hints to control the quality-of-service (QoS) characteristics afforded to it by the `WorkManager`. These hints provide guidelines to the `WorkManager` about how the `Work` instance is to be distributed or processed.

The `HintsContext` is a standard `WorkContext` defined in this specification. It provides a mechanism for the resource adapter to pass quality-of-service metadata to the `WorkManager` during the submission of a `Work` instance. The application server may then use the specified hints to control the execution of the `Work` instance.

```java
public class HintsContext implements WorkContext {
    protected String description = "Hints Context";
    protected String name = "HintsContext";

    public String getDescription() {
        return description;
    }

    public String getName() {
        return name;
    }

    public void setDescription(String description){
        this.description = description;
    }

    public void setName(String name){
        this.name = name;
    }

    Map<String, Serializable> hints
            = new HashMap<String, Serializable>();

    public void setHint(String hintName, Serializable value) {
        hints.put(hintName, value);
    }

    public Map<String, Serializable> getHints() {
        return hints;
    }
}
```

The resource adapter may use an instance of the standard `HintsContext` class to specify to the `WorkManager` the hints that need to be used during the processing of the `Work` instance.

The resource adapter may use the `setHint` method to set a hint in the context. It must use a non-null `hintName` while calling the `setHint` method.

This specification defines only a limited set of standard quality-of-service attributes (that is, hint names) in Section 11.6.1 "Standard Hints". The application server is not required to support the standard hint names.

The specification reserves the right to use names with the prefix `javax.resource.` in future versions of the specification. Resource adapters and application servers must not use names with the `javax.resource.` prefix for their custom requirements. The specification also recommends that resource adapter providers choose `hintNames` using the same rules that they use for `Class` names.

The `WorkManager` must reject the establishment of the `HintsContext` if the values provided for the hints are not valid. The `WorkManager` must ignore any unknown hint names submitted by a resource adapter instance. Configuration tools provided by the application server implementation may be used by the resource adapter deployer to override or map the hint name-value pairs provided by the resource adapter developer.

11.6.1   Standard Hints

11.6.1.1   `Work` Name Hint

The resource adapter may use the string `javax.resource.Name`, defined as a constant in `HintsContext.NAME_HINT`, as the `hintName` to indicate a name for a `Work` instance. This `hintName` may be used by the resource adapter and the application server for enhanced logging and debugging purposes. The value for the hint must be a valid `java.lang.String`.

11.6.1.2   Long-running `Work` instance Hint

The resource adapter may use the String `javax.resource.LongRunning`, defined as a constant in `HintsContext.LONGRUNNING_HINT`, as the `hintName` to indicate that a `Work` instance might run for a long period of time (typically lasting throughout the lifecycle of the resource adapter instance) compared to regular tasks that have a shorter execution lifecycle. The value of the hint must be a valid `boolean` value (`true` or `false`).

For example, the resource adapter might employ this hint for a `Work` instance that maintains network connectivity to the EIS instance throughout the lifecycle of the resource adapter.

A `WorkManager` that supports this `hintName` may handle such long running tasks in a separate thread pool or manage and monitor such tasks in a different fashion compared to regular short running tasks. This type of `WorkManager` must provide the same `Work` submission and processing semantics to `Work` instances submitted with or without this hint.

11.7 WorkContextLifecycleListener Interface

A `WorkContext` implementation may implement the `WorkContextLifecycleListener` interface to get fine-grained notifications (along with error codes, if any) while the `WorkManager` sets up the execution context for a `Work` instance.

```
public interface WorkContextLifecycleListener {
    // Indicates that the WorkContext was set successfully
    void contextSetupComplete();

    // Indicates that the WorkContext setup failed
    void contextSetupFailed(String errorCode);
}
```

When a `WorkManager` sets up the execution context of a `Work` instance that implements `WorkContextProvider`, the `WorkManager` must make the relevant lifecycle notifications if a `WorkContext` instance implements this interface. The possible error conditions that might occur while associating a `WorkContext` with a `Work` instance is captured in `WorkContextErrorCodes`. The `WorkManager` must call the `contextSetupFailed` method with the appropriate error code in `WorkContextErrorCodes`.

When a `Work` instance is submitted to the Connector `WorkManager` using one of the methods that passes in a `WorkListener` as a parameter, the `WorkManager` must send `Work` related notifications to the `WorkListener` and `WorkContext` setup-related notifications to the `WorkContextLifecycleListener` interface.

The `WorkManager` must make the notifications related to `Work` accepted and started events prior to calling the `WorkContext` setup related notifications. The order of setup-related notifications of `WorkContext` types within a list of work contexts of a `Work` instance is undefined. The `WorkManager` must make the notifications related to the `Work` completed events after the `WorkContext` setup related notifications.

FIGURE 11-4 Generic Work Context Lifecycle listener callback (Sequence Diagram)
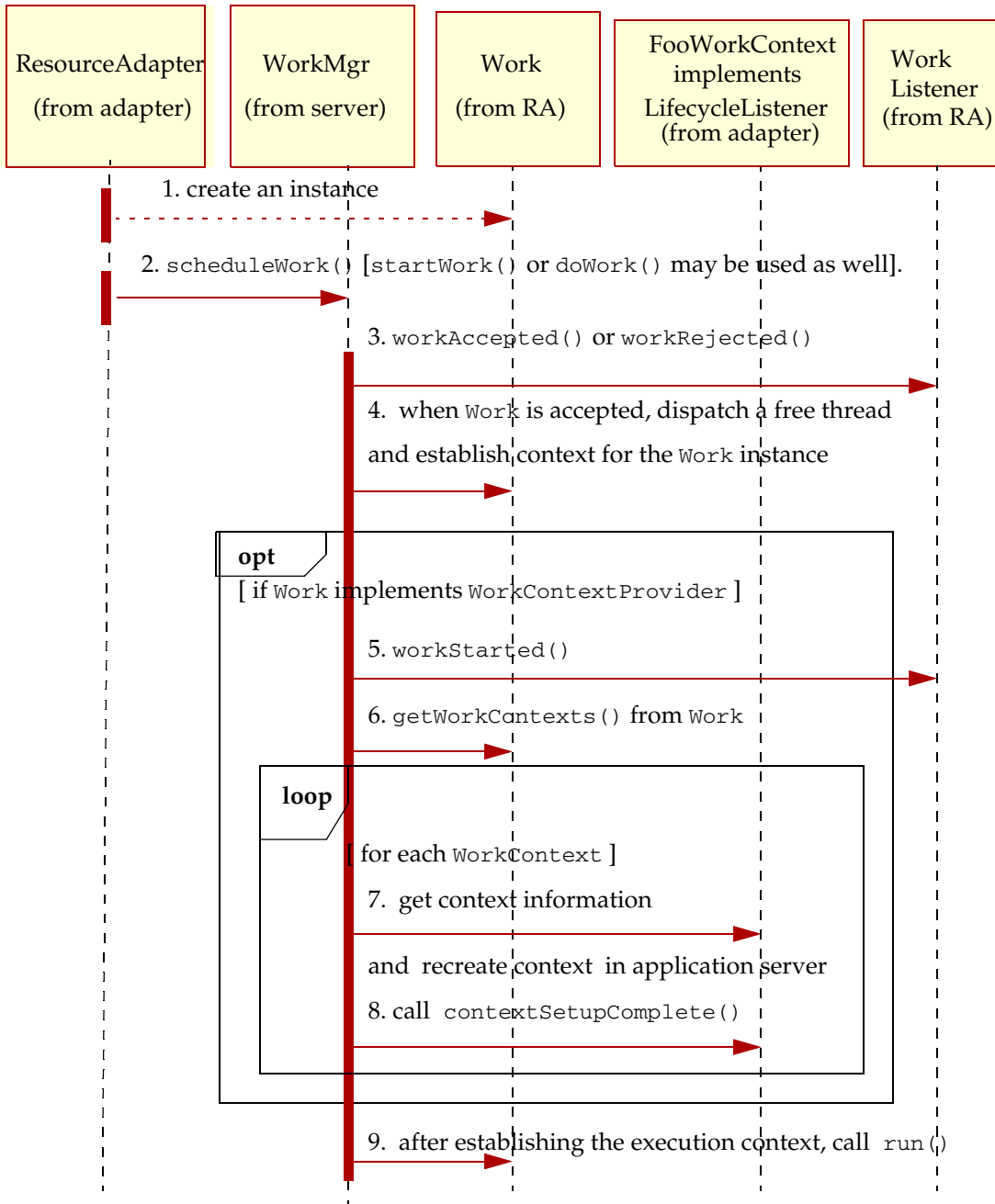
11.8   Illustrative Example

Section 15.3, "Use Case Scenario" on page 15-2, provides details on use case
scenarios where the Transaction Inflow contracts defined in Chapter 15, "Transaction
Inflow" are employed. As an example implementing one of the use cases listed
there, let's consider Wombat Systems, a finance company that has a variety of
software systems as part of its enterprise infrastructure. The software systems
include databases, messaging middleware, and mainframe systems, as well as
several Java EE application servers that host business logic written as EJBs (session,
entity, and message-driven beans).

In order to integrate the various disparate software systems, and to allow them to
communicate with each other, Wombat Systems did the following:

- Used the application servers to hold the integration as well as business logic,
  developed as EJBs
- Purchased or built resource adapters and deployed them on the application
  servers in order to provide bidirectional connectivity between the applications
  residing on the application servers and the various software systems

A particular situation at Wombat Systems requires that the work done by the
application components during a message inflow be automatically enlisted as part of
the imported transaction. The resource adapter developer then leverages the
interfaces defined in the Transaction Inflow portion of the Connector specification,
and achieves the flow-in of transactional context from the EIS to the application
server.

The resource adapter constructs a `Work` instance that is expected to do work as part
of the transactional message. It also creates an `ExecutionContext` instance
containing the constructed `Xid`, as detailed in Section 15.4.1, "Processing of
Transactional Calls" on page 15-4. However, because the resource adapter has to
execute the `Work` instance with other Work contexts as well, it uses a `Work`
implementation that implements the `WorkContextProvider` interface, as shown
below.

```
public class MyResourceAdapterImpl implements ResourceAdapter {
   ...
   public void start(BootstrapContext ctx) {
   bootstrapCtx = ctx;
   }
   ...
   {
      WorkManager workManager = myRA.bootstrapCtx.getWorkManager();
      workManager.scheduleWork(new MyWork());
      ...
```

```
        }
    }

    public class MyWork implements Work, WorkContextProvider {

        void release(){ ..}

        List<WorkContext> getWorkContexts() {
                TransactionContext txIn
                    = new TransactionContext(xid);
                List<WorkContext> icList = new ArrayList<WorkContext>();
                icList.add(txIn);
                // Add additional WorkContexts
                return icList;
        }

        void run(){
                // Deliver message to MessageEndpoint;
           }
        }
```

When this instance of `MyWork` that implements `WorkContextProvider` is
submitted to the `WorkManager` for execution, one of the free threads in the thread-
pooling implementation of the application server picks up the `Work` for execution.
The `WorkManager` then obtains the `WorkContexts` (through a call to
`getWorkContexts` method) that need to be set as the execution context for the
`Work` instance, iterates through the returned `WorkContexts`, and sets them up as the
execution context in which the `Work` instance is executed in.

Because an instance of `TransactionContext` is set, the application server's
`WorkManager` accepts the submitted `Work` instance, and re-creates the transaction
execution context. That is, the work to be done is enlisted as part of the imported
transaction. It then calls the `run` method on the `Work` object. When the `Work`'s `run`
method is called, all deliveries to the `MessageEndpoint` runs under the transaction
context of the `Work` instance, depending on the transaction preference of the bean
method that is being invoked.

Inbound Communicaton

This chapter provides a high level description of the inbound communication model; that is, the communication from an EIS to an application residing in an application server's EJB container through a resource adapter. This also introduces concepts used in subsequent chapters related to inbound communication: Message Inflow (Chapter 13, "Message Inflow"), EJB Invocation (Chapter 14, "EJB Invocation"), and Transaction Inflow (Chapter 15, "Transaction Inflow").

12.1   Overview

In the inbound communication model, the EIS initiates all communication to an application. In this case, the application may be composed of EJBs (session, entity and message-driven beans) and resides in an EJB container.

FIGURE 12-1 Inbound Communication Model
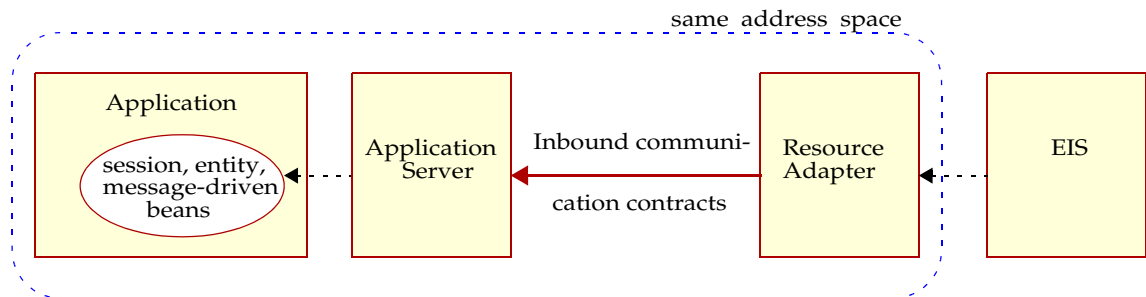
In order to enable inbound communication, a mechanism to invoke EJBs (session, entity and message-driven beans) from a resource adapter is necessary. Further, a mechanism is needed to propagate transaction information from an EIS to an application residing in an EJB container.

Chapter 13, "Message Inflow" describes a mechanism to invoke message-driven beans from a resource adapter. Chapter 15, "Transaction Inflow" provides a mechanism to import transaction information from an EIS into an EJB container.

12.2 An Illustrative Use Case

Wombat Systems is a finance company which has a variety of software systems as part of its enterprise infrastructure. The software systems include databases, enterprise resource planning (ERP) and customer relationship management (CRM) systems, messaging middleware, mainframe systems, as well as several Java EE

application servers which host business logic written as EJBs (session, entity and message-driven beans). Further, there are web service interactions that occur as part of the overall corporate workflow.

FIGURE 12-2 Inbound Communication Model (an Illustrative Use Case)
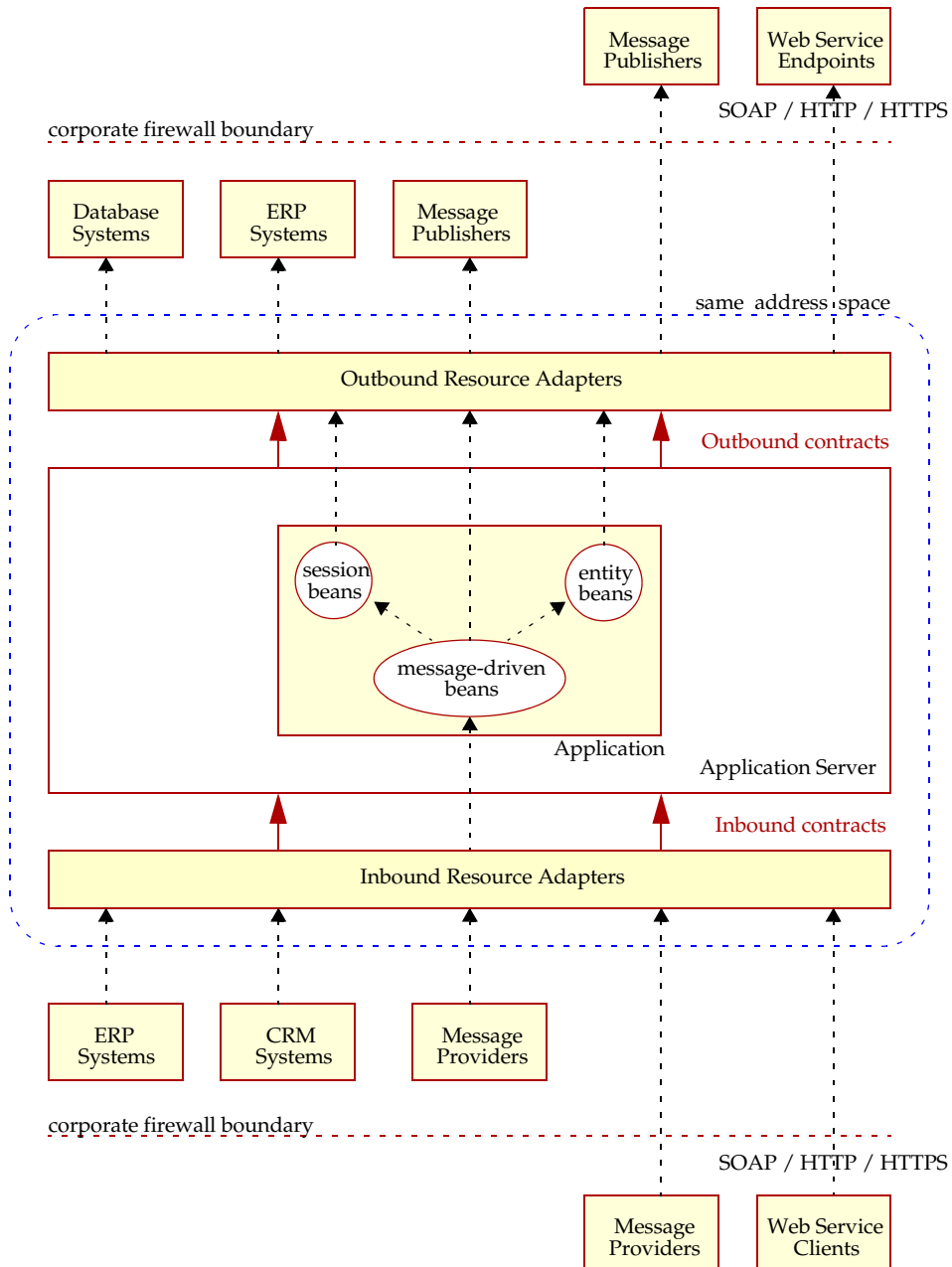
In order to integrate the various disparate software systems, and to allow them to communicate with each other, Wombat Systems did the following:

- Used the application servers to hold the integration as well as business logic, developed as EJBs.
- Purchased resource adapters and deployed them on the application servers, in order to provide bi-directional connectivity between the applications residing on the application servers and the various software systems.

Thus, using the resource adapter as a connectivity enabler, Wombat Systems was able to integrate the disparate software systems in its enterprise infrastructure.

Message Inflow

This chapter specifies a standard, generic contract between an application server and a resource adapter that allows a resource adapter to asynchronously deliver messages to message endpoints residing in the application server 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 wide range of message providers to be plugged into any Java EE compatible application server through a resource adapter.

Note that the usage of the term "Endpoint" in this chapter refers to a message endpoint (for example, a message-driven application).

13.1 Overview

Asynchronous message delivery or event notification is a widely used application communication paradigm. Some of the characteristics of the asynchronous message-based communication paradigm are:

- The message producer may not be directly aware of message consumers. There may be one or more consumers interested in the message.
- Message delivery is solicited; that is, a message consumer has to express interest in receiving messages.
- The messaging infrastructure is type-agnostic; that is, it treats messages as a Binary Large Object (BLOB). It stores and routes messages reliably, to interested messsage consumers, depending on Quality-of-Service (QoS) capabilities.
- The interaction is inherently loosely coupled. The message producer and the consumer do not share any execution context.
- The message producer generally is not interested in the outcome of message processing by consumers. However, it is possible that the provider may care to detect if the message has been consumed or not.

- The message delivery always involves a message routing infrastructure, which offers varying QoS capabilities for storing (persistence) and routing messages reliably.

The Java EE application programming model offers a rich set of components: EJBs (session, entity and message-driven beans), JSPs, and servlets for applications to use. The message-driven bean is an asynchronous message consumer, or message endpoint.

Java EE applications may use two different patterns to interact with a message provider:

- It may directly use specific messaging APIs, such as Java Messaging Service (JMS), to send and synchronously receive messages. This is achieved using the standard connector contracts for connection management. See Chapter 6, "Connection Management". Any message provider may provide a connector resource adapter that supplies connection objects for use by applications to send and synchronously receive messages using the specific messaging API.
- It may use message-driven beans to asynchronously receive messages through a message provider. The EJB specification (see "Enterprise JavaBeans (EJB) Specification, version 3.2" on page E-1) describes the message-driven bean component contract in detail.

While the above patterns allow a Java EE application to send and receive messages, they do not provide a standard system-level contract to plugin message providers to an application server and to deliver messages to message endpoints, or message-driven beans, residing in the application server. Without a standard pluggability contract, an application server would have to use special contracts to interact with various message providers, and a message provider has to do the same to interact with different application servers, which is an *m x n* problem.

FIGURE 13-1 Message Inflow Contract
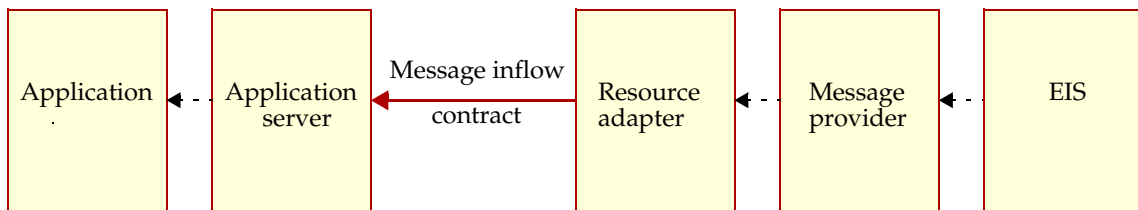
Thus, there is a need for a standard, generic contract between an application server and a message provider which allows a message provider to deliver messages to message endpoints (message-driven beans) residing in the application server independent of the specific messaging style, messaging semantics, and messaging infrastructure used to deliver messages. Such a contract also serves as the standard message provider pluggability contract which allows a wide range of message providers to be plugged into any Java EE compatible application server by way of a resource adapter.

13.2   Goals

- Provide a standard, generic mechanism to plug in a wide range of message providers, including JMS, into a Java EE compatible application server through a resource adapter and dispatch messages to message endpoints. This will allow Java EE components to act as consumers of messages with no required changes to the client programming models. Further, the Java EE components will be able to access messages with no awareness that a resource adapter is delivering the message.
- This generic contract must be capable of supporting various messaging delivery guarantees provided by different messaging styles, as well as allow concurrent delivery of messages.

13.3   Message Inflow Model

FIGURE 13-2 Message Inflow Contract (Object Diagram)
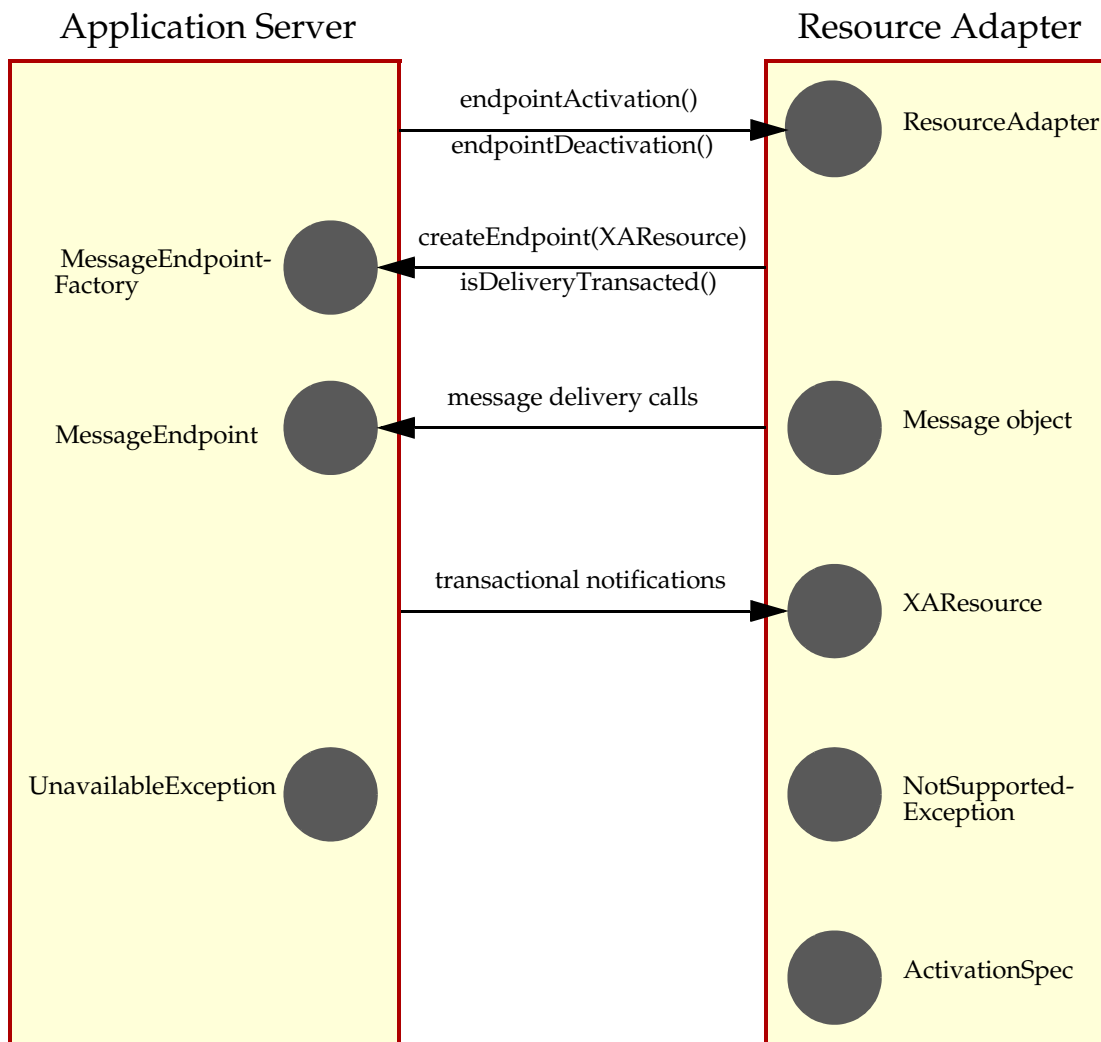
FIGURE 13-3 Message Inflow Contract (Interfaces)

javax.resource.spi

ResourceAdapter
(from adapter)

---

endpointActivation()
endpointDeactivation()
getXAResources()

ActivationSpec
(from adapter)

---

validate()

InvalidPropertyException
(from adapter)

---

getInvalidPropertyDescriptors()
setInvalidPropertyDescriptors()

UnavailableException
(from app server)

---

javax.resource.spi.endpoint

MessageEndpointFactory
(from app server)

---

createEndpoint()
isDeliveryTransacted()

MessageEndpoint
(from app server)

---

beforeDelivery()
afterDelivery()
release()

---

javax.resource

NotSupportedException
(from adapter)

CODE EXAMPLE 13-1 `javax.resource.spi`

```
package javax.resource.spi;

import java.beans.PropertyDescriptor;
import javax.resource.NotSupportedException;
import javax.resource.spi.endpoint.MessageEndpointFactory;

public interface ResourceAdapter {
    ... // other methods
    void endpointActivation(MessageEndpointFactory,
        ActivationSpec) throws ResourceException;
    void endpointDeactivation(MessageEndpointFactory,
        ActivationSpec);
    XAResource[] getXAResources(ActivationSpec[] specs)
        throws ResourceException;
}

public interface ActivationSpec { // JavaBean
    void validate() throws InvalidPropertyException;
}

public class InvalidPropertyException extends ResourceException {
    public InvalidPropertyException() { ... }
    public InvalidPropertyException(String message) { ... }
    public InvalidPropertyException(String message,
        String errorCode) { ... }
    public void setInvalidPropertyDescriptors(
        PropertyDescriptor[] invalidProperties) { ... }
    public PropertyDescriptor[] getInvalidPropertyDescriptors() {
        ... }
}

public class UnavailableException extends ResourceException {
    public UnavailableException() { ... }
    public UnavailableException(String message) { ... }
    public UnavailableException(Throwable cause) { ... }
    public UnavailableException(String message, Throwable cause) {
        ... }
}

public class RetryableUnavailableException
    extends UnavailableException
    implements javax.resource.spi.RetryableException {}
```

CODE EXAMPLE 13-2 `javax.resource.spi.endpoint`

```
package javax.resource.spi.endpoint;

import java.lang.Exception;
import java.lang.Throwable;
import java.lang.NoSuchMethodException;
import javax.transaction.xa.XAResource;
import javax.resource.ResourceException;
import javax.resource.spi.UnavailableException;

public interface MessageEndpointFactory {
   MessageEndpoint createEndpoint(XAResource)
      throws UnavailableException;
   MessageEndpoint createEndpoint(XAResource, long)
      throws UnavailableException;
   String getActivationName();
   Class<?> getEndpointClass();
   boolean isDeliveryTransacted(java.lang.reflect.Method)
      throws NoSuchMethodException;
}

public interface MessageEndpoint {
   void beforeDelivery(java.lang.reflect.Method)
      throws NoSuchMethodException, ResourceException;
   void afterDelivery() throws ResourceException;
   void release();
}
```

The `ResourceAdapter` interface supports methods used for endpoint activations
and deactivations. The `endpointActivation` method is called by the application
server when a message endpoint is activated and the `endpointDeactivation`
method is called by the application server when a message endpoint is deactivated.
The resource adapter is supplied a `MessageEndpointFactory` instance and a
configured `ActivationSpec` instance during endpoint activations and
deactivations. The resource adapter may reject an activation by throwing a
`NotSupportedException`, if the activation information is incorrect.

The resource adapter uses the `MessageEndpointFactory` instance to obtain
message endpoint instances for delivering messages either serially or concurrently.
The `MessageEndpointFactory` may be used for obtaining any number of message
endpoint instances. The `createEndpoint` method call may throw an
`UnavailableException` for several reasons:

- The application server has not completed endpoint activation.
- The application server may decide to limit concurrent message deliveries.
- The application server is about to shutdown.
- The application server may have encountered an internal error condition.

In some cases where the offending condition is temporary, the application server may decide to block the `createEndpoint` method call instead of throwing an `UnavailableException`.

In cases where the `MessageEndpointFactory` may require the rejection of the creation of the `MessageEndpoint` and where the failure to create an endpoint is temporary, the `MessageEndpointFactory` may use the `RetryableUnavailableException`. A resource adapter could then consider the offending condition as transient, and may then retry the `MessageEndpoint` creation process later.

The `MessageEndpointFactory` may also be used to find out whether message deliveries to a target method on a message listener interface that is implemented by a message endpoint or a target method in the `Class` returned by the `getEndpointClass` method, will be transacted or not through the `isDeliveryTransacted` method. The message delivery preferences must not change during the lifetime of a message endpoint.

The `MessageEndpointFactory` also provides a unique name for the message endpoint deployment that it represents. If the message endpoint has been deployed into a clustered application server, then the application server must provide the same name for that message endpoint's activation in each application server instance. It is recommended that this name be human-readable, and is unchanged even in cases when the application server is restarted or the message endpoint redeployed.

The `MessageEndpointFactory` allows a resource adapter to get the `Class` object corresponding to the message endpoint. The resource adapter may use the `Class` object to discover annotations, interfaces implemented, etc. and modify the message delivery behavior of the resource adapter accordingly. In the case of message driven beans, the `Class` object returned is the bean class provided by the application component developer. Refer to the Enterprise JavaBeans specification (see "Enterprise JavaBeans (EJB) Specification, version 3.2" on page E-1) for more details on the requirements for message driven beans with no-methods listener interface. The `MessageEndpointFactory` must return `null` if the `MessageEndpoint` does not implement the business methods of the message endpoint.

A resource adapter capable of message delivery to message endpoints must provide an `ActivationSpec` JavaBean class for each supported endpoint message listener type. The `ActivationSpec` JavaBean has a set of configurable properties specific to the messaging style and the message provider. An instance of the `ActivationSpec` JavaBean is configured by a message endpoint, or application, deployer to setup the necessary configuration information for the endpoint activation, and passed on to the resource adapter by way of the application server during endpoint deployment.

The resource adapter is expected to detect the endpoint message listener type, either by using the `ActivationSpec` JavaBean contents or based on the `ActivationSpec` JavaBean class, and deliver messages to the endpoint. The resource adapter may optionally pass an `XAResource` instance while creating a message endpoint in order to receive transactional notifications when a message delivery is transacted.

The following steps in sequential order represent the various stages in the message endpoint lifecycle, during which message inflow contracts are used:

1. Endpoint deployment

2. Message delivery (transacted and non-transacted)

3. Endpoint undeployment

13.4　Endpoint Deployment

FIGURE 13-4　Endpoint (Message-Driven Bean) Deployment (Actors)
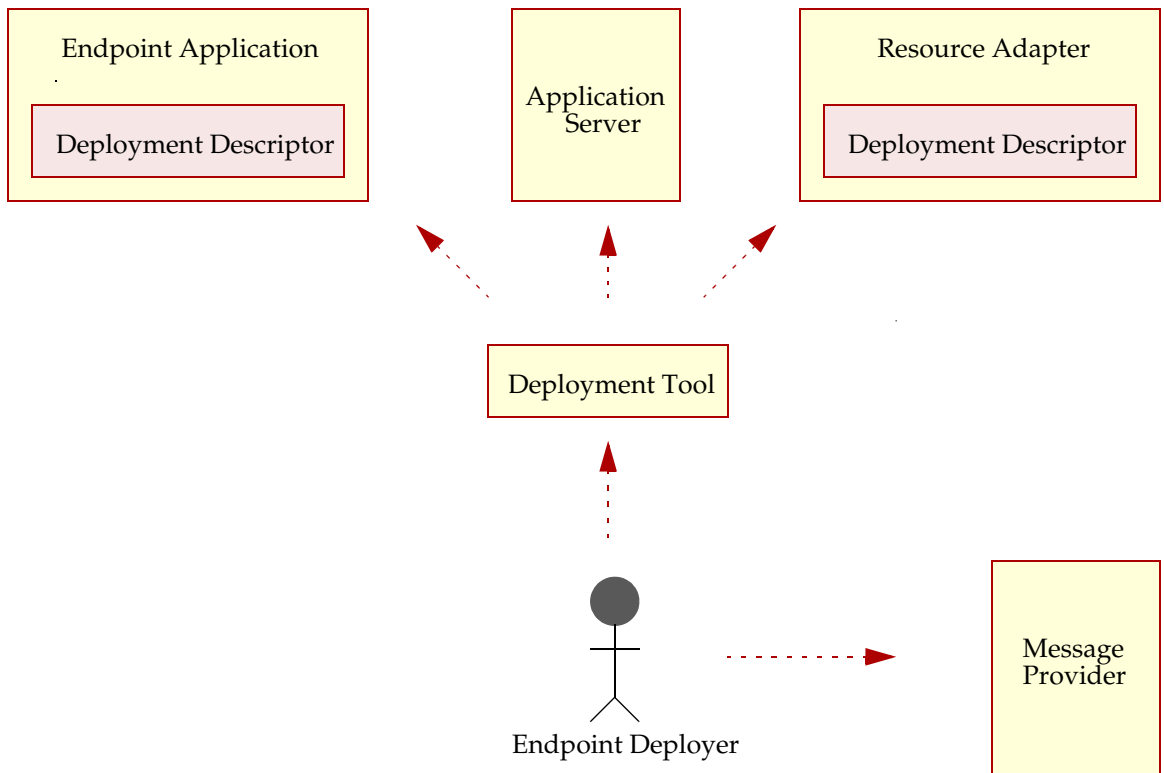
There are several actors involved in the deployment of a message endpoint:

- A message endpoint that is to be deployed on an application server.
- A resource adapter capable of message delivery. The resource adapter is typically provided by a message provider or a third-party, and is used to plug an external message provider into an application server. The resource adapter may be standalone that may be shared by different applications or it may be packaged along with an endpoint application.
- An application server that provides the runtime environment for the application.
- A deployer of the application, a human, who understands the application's needs, and is also aware of the details of the runtime environment in which the application will be deployed.
- A message provider, or messaging infrastructure, that is the source for messages. A message provider may provide special tools that can be used by the deployer to setup the message provider for message delivery.

The roles and responsibilities of the various actors are as follows:

13.4.1 Message Endpoint

The message endpoint is typically a message-driven bean application which is to be deployed on the application server. A MessageEndpoint may be implemented as other implementation specific objects as well. It asynchronously consumes messages from a message provider. It is also possible for the application to send and synchronously receive messages by directly using messaging-style specific APIs.

The message-driven bean developer provides activation configuration information in the message-driven bean deployment descriptor or by way of metadata annotations (MessageDriven annotation when the message-driven bean is realized as MDBs). This includes messaging style specific configuration details, and possibly message provider-specific details as well, which is used by the message-driven bean deployer to setup the activation.

The EJB specification (see "Enterprise JavaBeans (EJB) Specification, version 3.2" on page E-1) has more details on the message-driven bean deployment descriptor element activation-config used to hold the activation configuration information. For example, the deployment descriptor of a message-driven bean which consumes from a JMS resource adapter may contain:

CODE EXAMPLE 13-3   Message-Driven Bean Deployment Descriptor

```
<!-- message-driven bean deployment descriptor -->
   ...
   <activation-config>
     <activation-config-property>
        <activation-config-property-name>
           destinationType
```

CODE EXAMPLE 13-3   Message-Driven Bean Deployment Descriptor

```
              </activation-config-property-name>
              <activation-config-property-value>
                  javax.jms.Topic
              </activation-config-property-value>
          </activation-config-property>
          <activation-config-property>
              <activation-config-property-name>
                  SubscriptionDurability
              </activation-config-property-name>
              <activation-config-property-value>
                  Durable
              </activation-config-property-value>
          </activation-config-property>
          <activation-config-property>
              <activation-config-property-name>
                  MessageSelector
              </activation-config-property-name>
              <activation-config-property-value>
                  JMSType = 'car' AND color = 'blue'
              </activation-config-property-value>
          </activation-config-property>
          ...
      </activation-config>
...
```

The EJB specification does not specify messaging style-specific descriptor elements contained within the `activation-config` element. It is the responsibility of each individual messaging specification or product to specify the standard descriptor elements specific to the messaging style for representing the activation configuration information.

13.4.2   Resource Adapter

The resource adapter is a system component located in the application server's address space (that is, it has already been deployed) that provides connectivity to message providers and is capable of delivering messages to message endpoints residing in the application server. The resource adapter is typically provided by a message provider or a third-party, and is used to plug an external message provider into an application server. The resource adapter may be standalone, shared by different applications, or may be packaged along with an endpoint application.

The resource adapter provides the following information by way of the resource adapter deployment descriptor or through metadata annotations described in Section 18.7, "@Activation" on page 18-14, that is used by the endpoint deployer to setup endpoint activation:

13.4.2.1   List of Supported Message Listener Types

The resource adapter provides a list of endpoint message listener types it supports. Each type is represented as a name of the Java type of the message listener interface.

13.4.2.2   `ActivationSpec` JavaBean

The resource adapter provides the Java class name of an `ActivationSpec` JavaBean, one for each supported message listener type, containing a set of configurable properties that is used to specify endpoint activation configuration information during endpoint deployment. Refer to Section 19.3, "JavaBean Requirements" on page 19-2. An `ActivationSpec` JavaBean instance is created during endpoint deployment, and the instance is configured by the endpoint deployer.

During configuration, an `ActivationSpec` JavaBean instance may check the validity of the configuration settings provided by the endpoint deployer. The `ActivationSpec` has a `validate` method which may be used during endpoint deployment to validate the overall activation configuration information provided by the endpoint deployer. This helps to catch activation configuration errors earlier on without having to wait until endpoint activation time for configuration validation. The implementation of this self-validation check behavior is optional.

The `ActivationSpec` JavaBean implementation is recommended to use the JavaBean validation mechanisms described in Section 5.3.7.5, "JavaBean Validation" on page 5-15 instead of the `validate` method to request validation by the container. If the application server provides an implementation of the BeanValidation specification (see "Bean Validation Specification, version 1.1" on page E-2), the application server must check the validity of the configuration settings provided by the deployer for a JavaBean, using the capabilities provided by the Bean Validation specification before calling the `validate` method.

Note, the `ActivationSpec` JavaBean instance must not make any assumptions about the availability of a live resource adapter instance.

The resource adapter may also provide in its deployment descriptor, using the `required-config-property` element, an optional list of configuration property names required for each activation specification. This information may be used during deployment to ensure that the required configuration properties are specified. An endpoint activation should fail if the required property values are not specified.

The usage of the `required-config-property` element to require the specification of a configuration property during deployment is deprecated. Instead, the `ActivationSpec` JavaBean is recommended to use the JavaBean Validation facilities described in Section 5.3.7.5, "JavaBean Validation" on page 5-15. The `ActivationSpec` JavaBean may annotate the field or the JavaBeans-compliant

accessor method corresponding to the configuration property with the `@NotNull` constraint (or the corresponding XML validation descriptor equivalent), to indicate that the configuartion property must be specified during activation specification.

The resource adapter may also provide in its deployment descriptor, using the `config-property` element, a list of configuration property names for the activation specification.

In the case of JMS message providers, the `destination` property value (refer to Appendix B, "Activation Configuration for Message Inflow to JMS Endpoints") may also be an object that implements the `javax.jms.Destination` interface. In such a case, the resource adapter should provide an administered object (refer to Section 13.4.2.3, "Administered Objects" on page 13-13) that implements the `javax.jms.Destination` interface. The specific type of the JMS destination is specified by the `destinationType` property value. The JMS `ActivationSpec` JavaBean properties should be standardized by the JMS community.

13.4.2.3   Administered Objects

The resource adapter may provide the Java class name and the interface type of an optional set of JavaBean classes representing various administered objects. Refer to Section 19.3, "JavaBean Requirements" on page 19-2. Administered objects are specific to a messaging style or message provider.

For example, some messaging styles may need applications to use special administered objects for sending and synchronously receiving messages through connection objects using messaging-style specific APIs. It is also possible that administered objects may be used to perform transformations on an asynchronously received message in a message provider-specific way.

Note, administered objects are not used for setting up asynchronous message deliveries to message endpoints. The `ActivationSpec` JavaBean is used to hold all the necessary activation information needed for asynchronous message delivery setup.

An administered object may implement the `ResourceAdapterAssociation` interface to associate a resource adapter instance with the administered object.The `ResourceAdapterAssociation` interface specifies the methods to associate a administered object JavaBean with a `ResourceAdapter` JavaBean.

Prior to using the administered object, the application server must create an association between the administered object and a `ResourceAdapter` JavaBean, by calling the `setResourceAdapter` method on the administered object. A successful association is established only when the `setResourceAdapter` method on the administered object returns without throwing an exception.

An administered object instance, that implements `ResourceAdapterAssociation` interface must ensure that its Java class and the interface type implements `javax.resource.Referenceable` and `java.io.Serializable` interfaces. This enables an application server to store the administered object instance in the JNDI naming environment. Refer to Section 20.6.3, "Scenario: Referenceable" on page 20-19 for details on the JNDI reference mechanism.

During deserialization of the administered object, the application server must establish the association between the administered object and the resource adapter instance by calling setResourceAdapter.

13.4.2.4   Configuring Administered Objects

- Create an administered object JavaBean instance. This will initialize the instance with the defaults specified through the JavaBean mechanism.
- Apply the administered object class configuration properties specified in the resource adapter deployment descriptor, on the administered object instance. This may override some of the default values specified by way of the JavaBean mechanism.
- The application server is required to merge values specified by way of annotations and deployment descriptors as specified in Section 18.3, "Deployment Descriptors and Annotations" on page 18-2, before applying the administed object class configuration properties.
- The deployer may further override the values of the administered object before deployment.

13.4.3   Endpoint Deployer

The endpoint deployer is a human who has the responsibility to deploy the message endpoint, or application, on an application server. The deployer is expected to know the requirements of the application and be aware of the details of the runtime environment in which the application will be deployed.

The deployer selects a suitable resource adapter that matches the requirements of the application depending on endpoint message listener type, QoS capabilities, and so on. The deployer configures an `ActivationSpec` JavaBean instance based on the information provided by the application developer or assembler, which is contained in the endpoint deployment descriptor or by way of metadata annotations described in Section 18.7, "@Activation" on page 18-14. The deployer may also use additional message provider-specific information to configure the `ActivationSpec` JavaBean instance.

The deployer also configures a set of administered objects, if necessary. The resource adapter provides the JavaBean classes for such administered objects. The deployer may also interact with a message provider to do the necessary setup for message delivery.

Then the deployer deploys the application on the application server. As part of the deployment procedure, the deployer provides all the configured JavaBean instances to the application server, and also specifies the chosen resource adapter instance to be used for message delivery.

13.4.4   Application Server

The application server provides the runtime environment for the message endpoint. It activates message endpoints when they are deployed. All such deployed endpoints are automatically reactivated when an application server restarts after a normal shutdown or system crash. When an application is undeployed, the application server deactivates the endpoint.

When an endpoint is activated, the application server calls the chosen resource adapter by way of the `endpointActivation` method and passes on a `MessageEndpointFactory` instance and the `ActivationSpec` JavaBean, which was configured by the endpoint deployer. The application server does not interpret the contents of the `ActivationSpec` JavaBean and treats it as an opaque entity. The resource adapter may reject an endpoint activation by throwing a `NotSupportedException` during the `endpointActivation` method call. This is due to incorrect activation information.

The application server must make the application component environment namespace of the endpoint (that is being activated), available to the resource adapter during the call to the `endpointActivation` and `endpointDeactivation` methods. The resource adapter may use this JNDI context to access other resources.

The resource adapter uses the `MessageEndpointFactory` to create message endpoint instances to deliver messages either serially or concurrently. There is no limit to the number of such endpoints that may be created to deliver messages. However, in practice the application server may decide to limit concurrency by rejecting attempts to create new endpoints by throwing an `UnavailableException`. The application server may also attempt to block a message delivery method call in order to limit concurrency and perform flow control.

Note, a resource adapter may attempt to deliver messages during the `endpointActivation` method call. It is up to the application server to decide whether to allow message delivery before activation is completed. If the application

server chooses to prevent message delivery during endpoint activation, it may block the `createEndpoint` method call until the activation is completed or throw an `UnavailableException`.

The resource adapter may pass an `XAResource` instance while creating a message endpoint in order to receive transactional notifications when a message delivery is transacted. The application server must notify the resource adapter through the `XAResource` instance if a message delivery is transacted.

During endpoint deployment, the application server places the configured administered objects, if any, supplied by the endpoint deployer in the component namespace `java:comp/env`. The endpoint deployer specifies a location in the component namespace where each administered object should reside. The configured administered objects residing in the component namespace are used by the endpoint application in a messaging style-specific manner.

When an endpoint is deactivated, the application server notifies the resource adapter through the `endpointDeactivation` method call. The application server must pass the same `MessageEndpointFactory` instance and the `ActivationSpec` JavaBean instance that was used during endpoint activation.

13.4.5   Message Provider

A message provider, or messaging infrastructure, is typically an external system that is the source for messages. Message providers may vary in their QoS capabilities. A message provider may provide special tools that can be directly used by the endpoint deployer to setup the message provider for message delivery.

FIGURE 13-5 Endpoint (Message-Driven Bean) Deployment Steps

Application
Server

10. Place administered objects (if any)
    in the component namespace.

11. Endpoint activation: Passes a handle
    to a MessageEndpointFactory and
    the ActivationSpec JavaBean.

9. Supply the ActivationSpec JavaBean
   and administered objects (if any), and
   specify the chosen resource adapter.

Resource Adapter

Deployment Descriptor

i. Message listener types
   supported.

ii. An ActivationSpec class
    (JavaBean) for each
    supported message listener.

iii. JavaBean classes for
     Administered objects,
     if any.

Endpoint Application

Deployment Descriptor

i. Activation configuration
   information (messaging
   style specific). May also have
   message provider specific
   details.

4. Get endpoint activation configuration
   information.

2. Query resource adapter about
   message listener types, details on
   ActivationSpec, admin
   objects and QoS capabilities.

Deployment Tool

1. Choose a suitable resource adapter.

3. Get activation config details from endpoint.

6. Create a ActivationSpec JavaBean instance
   from the chosen resource adapter and
   configure the JavaBean properties.

7. Configure necessary administered objects.

8. Deploy application.

5. setup for message delivery.

Message
Provider

Endpoint Deployer

13.4.6 Endpoint Deployment Steps

The sequence of steps involved in endpoint deployment involving the various actors is as follows:

1. The endpoint deployer obtains a list of resource adapters capable of delivering messages to the message endpoint, and chooses a suitable one. The decision is based on the message listener type supported by the resource adapter and its QoS capabilities. However, it is possible that the message endpoint application may already contain a suitable resource adapter. In such a case, the resource adapter is deployed along with the endpoint application and is used to deliver messages to the specific endpoint application.

2. The deployer obtains the activation configuration information provided by the endpoint developer available by way of metadata annotations or in the endpoint deployment descriptor.

3. The deployer may need to setup the message provider for message delivery to the endpoint. This may be done using a message provider specific tool.

4. The deployer obtains an `ActivationSpec` JavaBean from the selected resource adapter and configures it. The configuration information is messaging style-specific and may include message provider specific details.

5. The deployer configures the JavaBean instances of administered objects, if any are necessary.

6. The deployer provides the configured JavaBean instances to the application server, and also specifies the resource adapter chosen for message delivery. Note, the contract between a deployment tool and an application server is out of scope for this specification.

7. The application server places the administered objects, if any, in the `java:comp/env` component namespace for use by the message endpoint.

8. The application server activates the message endpoint by calling the chosen resource adapter through the `endpointActivation` method and passes a `MessageEndpointFactory` instance and the configured `ActivationSpec` JavaBean instance provided by the deployer. The resource adapter may reject the endpoint activation by throwing a `NotSupportedException`, which is due to incorrect activation information.

13.4.7   Requirements

- A resource adapter that is capable of delivering messages to message endpoints must provide a list of endpoint message listener types it supports, and also must provide an `ActivationSpec` JavaBean class for each message listener type it supports. This information must be part of the resource adapter deployment descriptor.
- `ActivationSpec` and all administered objects must be JavaBeans.
- A resource adapter must allow an application server to make concurrent `endpointActivation` method or `endpointDeactivation` method calls.
- The endpoint application's `activation-config` properties, specified in the endpoint deployment descriptor or through the message endpoint's annotation, should be a subset of the `ActivationSpec` JavaBean's properties. There must be a one-to-one correspondence between the `activation-config` property names and the `ActivationSpec` JavaBean's property names. This allows automatic merging of the `activation-config` properties with an `ActivationSpec` JavaBean instance during endpoint deployment. Any specified `activation-config` property which does not have a matching property in the `ActivationSpec` JavaBean should be treated as an error.
- When an application server notifies a resource adapter during endpoint deactivation, it must pass the same `MessageEndpointFactory` instance and the `ActivationSpec` JavaBean instance that was used during endpoint activation.
- Any exception thrown by the `endpointDeactivation` method call must be ignored. After this method call the endpoint is deemed inactive.
- All deployed endpoints must be automatically reactivated by the application server when it restarts after a normal shutdown or system crash.
- Before a resource adapter is undeployed, the application server must deactivate all active endpoints consuming messages from that specific resource adapter.

13.4.8   Structure of a Message Listener Interface

A message listener interface implemented by a message endpoint, a message-driven bean, is allowed to have multiple methods. Each method of a message listener interface is allowed to have multiple arguments, a return value, and throw checked application exceptions or unchecked system exceptions.

Checked exceptions are thrown only by a message listener implementation. The message-driven bean container must propagate to the resource adapter any checked exception that occurs during message dispatch.

Unchecked exceptions (`java.lang.RuntimeException` and `java.lang.Error`) may be thrown by either the message listener implementation or may be thrown by the application server code during message dispatch. The application server must

wrap such an unchecked exception within a `javax.ejb.EJBException`, which is a `java.lang.RuntimeException`, and throw the `javax.ejb.EJBException` to the resource adapter.

The EJB specification describes in detail the structural requirements of a message listener interface implemented by a message-driven bean.

13.4.9 Multiple Endpoint Activations With Similar Activation Configuration

Since multiple endpoints, that is, different applications, with similar activation configuration may be deployed in a single application server, the application server may call the `endpointActivation` method on a resource adapter instance multiple times with similar activation configuration. The resource adapter must treat multiple endpoint activations with similar activation configuration separately. When messages start arriving, the resource adapter must, for each active endpoint, deliver a separate copy of incoming messages, even if there are multiple endpoints with similar activation configuration.

13.4.9.1 Requirements

- The application server must supply a unique `MessageEndpointFactory` instance for each activation.
- Refer to Section 19.4, "Equality Constraints" on page 19-3 for equality constraints on `MessageEndpointFactory` and `ActivationSpec` implementations.
- The resource adapter must treat multiple endpoints with similar activation configuration separately and guarantee message delivery semantics.
- The resource adapter must treat each `ActivationSpec` JavaBean uniquely irrespective of its contents. That is, the resource adapter must not treat two separate `ActivationSpec` JavaBeans as equals.

FIGURE 13-6 Endpoint Deployment (Sequence Diagram)

Endpoint deployer	Java EE app server	ResourceAdapter (from adapter)	MessageEndpointFactory (from app server)	Message provider

ActivationSpec (from adapter)

The endpoint deployer chooses a suitable resource adapter based on supported message listener type and QoS capabilities , configures an `ActivationSpec` JavaBean instance obtained from the resource adapter. The configuration includes message style-specific information and may include message provider specific information.

1. create and configure an ActivationSpec JavaBean

2. deploy endpoint application (pass the configured ActivationSpec JavaBean)

3. create an instance

4. endpointActivation(MessageEndpointFactory, ActivationSpec)

5. setup for message delivery (private contract)

13.5   Message Delivery

Once endpoints are activated, they are ready to receive messages. When messages arrive, the resource adapter uses the `MessageEndpointFactory` to create an endpoint instance. The resource adapter narrows the endpoint instance to the actual message listener type (it knows the endpoint type from the `ActivationSpec`), and delivers the message to the endpoint instance. The EJB specification (see "Enterprise JavaBeans (EJB) Specification, version 3.2" on page E-1) prescribes the rules for the message listener interface structure. The same endpoint instance may be used again to deliver subsequent messages serially, but it must not use the same endpoint instance concurrently.

Note that the endpoint instance supplied by the `createEndpoint` method call is a proxy which implements the endpoint message listener type and the `MessageEndpoint` interface and it is not the actual endpoint. This is necessary because the application server may need to intercept the message delivery in order to inject transactions, depending on the actual endpoint preferences, and to perform other checks.

The proxy endpoint instance is implemented by the application server and is used to intercept the message delivery, performs checks, inject transactions, and so on, and to route the message to the actual message endpoint instance. The resource adapter does not have to make this distinction, and should treat the proxy endpoint instance as the actual endpoint instance.

The resource adapter may use a proxy endpoint instance to serially deliver messages. The resource adapter must not use a proxy endpoint instance concurrently from different threads. The proxy endpoint may throw a `java.lang.IllegalStateException` when invoked concurrently. However, a proxy endpoint instance may be used by different threads in a serial fashion.

The resource adapter may call the `release` method on the proxy endpoint instance to indicate that it no longer requires the proxy instance. This hint may be used by the application server for proxy endpoint pooling decisions. This method call frees the state of the proxy instance. The released proxy instance may be reused for further proxy endpoint requests from the same resource adapter. A proxy endpoint instance must not be reused across multiple resource adapter instances.

Between the time a proxy endpoint instance is released and before it is granted back to the same resource adapter (by way of a `createEndpoint` method call), the proxy endpoint instance is considered to be in a free and available state. Any attempted use of a free proxy must result in a `java.lang.IllegalStateException` thrown by the application server.

The application server may start a transaction before delivering the message to the actual endpoint depending on the endpoint preferences. The resource adapter may optionally pass an `XAResource` instance through the `createEndpoint` method in order to receive transaction notifications for those transactions started by an application server before message delivery.

13.5.1 Sample Resource Adapter Code To Illustrate Message Delivery

CODE EXAMPLE 13-4   Message Delivery in a Resource Adapter

```
// Reader Thread(s)
{
   // 1. Strip off msg header and parse message description
   // 2. Choose a set of endpoints which match message description
   // 3. Place message in appropriate buffer / queue
   // 4. Notify worker threads
}

// Worker Thread(s)
{
   // 1. Wake up on notification (message arrival)
   // 2. Pick up the message and locate the MessageEndpointFactory
   // associated with the subscription
   Message msg = ...;
   MessageEndpointFactory endpointFactory = ...;
   MyXAResource xaResource = ...; // for transacted delivery

   // 4. Obtain a message endpoint and narrow it to the
   // correct type.

   // ActivationSpec has endpoint message listener type
   // information.
   Object obj = endpointFactory.createEndpoint(xaResource);
   javax.jms.MessageListener endpoint =
      (javax.jms.MessageListener) obj;

   // 5. Link the XAResource with the endpoint. This allows the
   // XAResource object to know which endpoint/message delivery
   // is transacted when it receives transaction notifications.
   // This may be unnecessary depending on the implementation.
   xaResource.setEndpoint(endpoint);

   // Note: It may be possible to make the XAResource object
   // thread-safe/reentrant and reuse the same object for receiving
   // transaction notifications for different endpoints.
   // The XAResource object may use thread-local storage to
   // remember state, and thus avoid creating multiple
   // XAResource objects.

   // 6. Deliver the message.
   endpoint.onMessage(msg);

   // 7. Wait for notification of incoming messages
```

CODE EXAMPLE 13-4 Message Delivery in a Resource Adapter

```
    // and repeat the above steps on message arrival.
}

package com.wombat.ra;

import javax.transaction.xa.*;

class MyXAResource implements javax.transaction.xa.XAResource {

   public void start(Xid xid) throws XAException {
      // Associate the message delivery with the transaction id.
      // That is, create the tuple (msg id, transaction id) in
      // memory.
   }

   public int prepare(Xid xid) throws XAException {
      // Forward the tuple (message id, transaction id) to the
      // message provider. The provider must persist this
      // information, which is used during crash recovery by the
      // application server. During crash recovery,
      // the application calls the message provider, via the
      // recover method on an XAResource object, queries for
      // a list of in-doubt transactions and completes them.
      // Upon successful completion, return "ready_to_commit"
      // vote, else return "rollback_only" vote.
   }

   public void commit(Xid xid, boolean onePhase)
      throws XAException {
      // forward the transaction ID to the message provider. This
      // serves as the acknowledgement that a message was
      // delivered.
   }

   public void rollback(Xid xid) throws XAException {
      // forward the transaction ID to the message provider. This
      // indicates to the provider that the message was not
      // delivered.
   }

   ...; // other methods
}
```

13.5.1.1 Requirements

- The application server's proxy endpoint instance must implement the endpoint message listener type and the `MessageEndpoint` interface.
- The application server must pass by reference all the method parameter objects passed by the resource adapter during a message delivery method call on a proxy endpoint. The application server must not copy or clone the passed method parameter objects during message delivery to the actual endpoint instance.
- If the application server starts a new transaction, depending on endpoint preferences, before delivering a message to an endpoint instance, it must send all transaction notifications to the `XAResource` instance optionally supplied by the resource adapter as part of the `createEndpoint` method call.
- A resource adapter must not attempt to deliver messages concurrently to a single endpoint instance. The application server must reject concurrent usage of an endpoint instance.

13.5.2 Message Redelivery Upon Crash Recovery

An application server may crash during message delivery. In the case of message deliveries transacted by the application server, the application server must notify the commit decision to the message provider during crash recovery.

During crash recovery:

1. The application server must first restart resource adapter instances by calling the `start` method on each persisted `ResourceAdapter` JavaBean, each corresponding to a resource adapter instance that was active prior to the crash.

2. The application server must call the `getXAResources` method on each `ResourceAdapter` JavaBean, and pass in an array of `ActivationSpec` JavaBeans, each of which corresponds to a deployed endpoint application that was active prior to the system crash. This method need not be called if there were no endpoint applications that were active prior to the crash.

3. Upon being called by the application server during crash recovery through the `getXAResources` method, the resource adapter must return an array of `XAResource` objects, each of which represents a unique resource manager.

   The resource adapter may return `null` if it does not implement the `XAResource` interface. Otherwise, it must return an array of `XAResource` objects, each of which represents a unique resource manager that was used by the endpoint applications. The resource adapter may throw a `ResourceException` if it encounters an error condition.

   Since each returned `XAResource` object represents a unique resource manager, the number of returned `XAResource` objects must be less than or equal to the number of `ActivationSpec` instances specified.

4. Since it is possible that multiple resource adapters may use the same resource manager, there may be more than one `XAResource` object in the collection representing a resource manager. The application server may still need to narrow the collection of `XAResource` objects to a unique set of resource managers by using the `isSameRM` method on the `XAResource` object.

5. The application server must use the `XAResource` objects to query each resource manager for a list of in-doubt in an already prepared state awaiting a commit decision transactions. Then, it must complete each pending transaction by sending the commit decision to the participating resource managers. Note, it is possible that a resource manager may not have pending in-doubt transactions.

The crash recovery procedure ensures that the message provider gets notified about the outcome of all message deliveries that were in an in-doubt transaction state at the time of the crash. Upon such notification, the message provider, depending on the delivery outcome, may decide to redeliver the undelivered messages to the various endpoints when they are reactivated.

13.5.3   Durable Message Delivery Setup

Once message endpoints are activated, they are ready to receive messages from a message provider. Message delivery setup may either be durable or non-durable.

In the case of non-durable message deliveries, messages are lost during application server downtime. When the application server becomes functional again, it automatically reactivates all message endpoints that were previously deployed, and message delivery starts again. But the messages that were produced during the downtime are lost. This is because messages are not persisted by the message provider and redelivered when the message endpoints are reactivated.

In the case of durable activations, messages are not lost during application server downtime. When the application server becomes functional again, it automatically reactivates all message endpoints that were previously deployed, and message delivery starts again. The messages that were produced during the downtime are persisted by the message provider and redelivered when the message endpoints are reactivated. It is the responsibility of the message provider to persist undelivered messages and redeliver them when the endpoint is available; that is, when the endpoint is reactivated by the app server.

Durability of message delivery may be an attribute of the activation setup, and thus it must be captured as part of the endpoint activation information. No additional contracts are required to support durable activations. Activation durability can be specified by a endpoint deployer by way of the `ActivationSpec` JavaBean. Note, some message providers may not support durable message deliveries and hence it is a QoS capability offered by the message provider.

13.5.4 Concurrent Delivery of Messages

During message endpoint activation, the application server supplies a
`MessageEndpointFactory` to the resource adapter. The
`MessageEndpointFactory` is used to get an endpoint instance through the
`createEndpoint` method call. Each call results in a new or an unused endpoint
instance that may be used to deliver messages concurrently; that is, for each active
message endpoint, there may be multiple endpoint instances consuming messages
concurrently.

Thus, for each message endpoint, depending on traffic, the resource adapter may
choose to deliver messages serially using a single endpoint instance or concurrently
using multiple endpoint instances.

There is no limit to the number of such endpoint instances that may be created,
although the application server may limit the concurrency by either throwing an
`UnavailableException` or by blocking the `createEndpoint` method call.

The application server may also attempt to block a message delivery method call in
order to limit concurrency and perform flow control.

13.5.4.1 Requirements

The application server must return a new or an unused endpoint instance for every
`createEndpoint` method call on a `MessageEndpointFactory`.

13.5.5 Delivery Semantics and Acknowledgement

When the resource adapter delivers a message to an endpoint instance, which is
really a proxy endpoint instance, the application server intercepts the message
delivery to perform checks, inject transactions, and so on, and routes the message to
the actual message endpoint instance.

The application server may start a transaction before delivering the message to the
actual endpoint depending on the endpoint preferences. In the case of a transacted
delivery, the resource adapter may use the transaction notifications received through
the `XAResource` object to send back an acknowledgement to its message provider.

In the case of non-transacted delivery, that is, the application server does not start a
transaction, the resource adapter has to rely on the successful completion of the
message delivery call in order to send back an acknowledgement to its provider.

13.5.6   Transacted Delivery (Using Container-Managed Transaction)

Depending on the endpoint preferences, the application server brackets the message delivery to an endpoint instance with a Java Transaction API (JTA) transaction.

- This ensures that all the work done by the endpoint instance is enlisted as part of the transaction.
- This provides atomic message delivery/message consumption; that is, if the transaction were to be aborted by the application server due to an exceptional condition, all the work done by the endpoint instance is aborted, and the delivery is undone. If this does not occur, the transaction is committed, all the work done by the endpoint instance is committed and the delivery is completed.

The application server notifies the resource adapter while beginning and completing transactions by using the `XAResource` instance optionally supplied through the `createEndpoint` method call.

- This allows the adapter to detect the outcome of a transacted delivery, and also influence the outcome of the transaction via through `prepare` method call on the `XAResource` instance.
- This allows the adapter to send back an acknowledgement to its message provider contingent on successful delivery; that is, when notified through the `commit` method call on the `XAResource` instance.
- This also allows the adapter to be notified of the correct delivery outcome upon failure recovery processing; that is, if the system crashes when the transaction is in-doubt, that is, when the transaction has already been prepared, the application server upon recovery correctly completes the transaction and notifies the adapter of the outcome of the transaction. Thus, the adapter can send back an acknowledgement to its message provider after failure recovery, if the message had been successfully delivered.

A resource adapter may optionally provide an `XAResource` instance through the `createEndpoint` method call in order to receive transactional notifications for those transactions started by an application server before message delivery. The resource adapter may find out whether message deliveries to a target method on a message endpoint will be transacted or not through the `isDeliveryTransacted` method in the `MessageEndPointFactory` instance, and decide whether to provide an `XAResource` instance through the `createEndpoint` method. Note, this does not require the resource adapter to support the transaction inflow contract (see Chapter 15, "Transaction Inflow").

There are two delivery options available to the resource adapter for transacted deliveries:

- **Option A, traditional XA style.** The resource adapter optionally provides an `XAResource` instance through the `createEndpoint` method in order to receive XA transaction notifications for transacted message deliveries. In this case, the

application server fully controls the transaction boundaries and the resource adapter is merely a participant (the XAResource Resource Manager (RM)). See FIGURE 13-7.

■ **Option B, beforeDelivery/afterDelivery.** The resource adapter still optionally provides an XAResource instance through the createEndpoint method in order to receive XA transaction notifications for transacted message deliveries. But the resource adapter controls the transaction boundaries through the beforeDelivery/afterDelivery calls, in spite of being only a participant (an XAResource RM).

During the beforeDelivery call from the resource adapter, depending on the transactional preferences of the intended target method (specified through the java.lang.reflect.Method method parameter), the application server starts a transaction and enlists the XAResource instance in the transaction. The processing (by the application server) of the actual message delivery method call on the endpoint must be independent of the class loader associated with the descriptive method object (parameter).

During the afterDelivery call from the resource adapter, the application server completes the transaction and sends transaction completion notifications to the XAResource instance. The actual message delivery happens in between the beforeDelivery and afterDelivery calls.

In this case, the resource adapter controls when the transaction is started and completed by the application server, even though the application server decides on the outcome of the transaction. This allows resource adapters more flexibility in handling message deliveries. For example, the resource adapter may choose to dequeue a message from within the container-managed transaction so that the message dequeue is automatically undone if the container-managed transaction aborts.

There must not be more than one message delivery in-between a single beforeDelivery and afterDelivery method call pair. The application server must reject beforeDelivery or afterDelivery calls that are out of sequence by throwing an IllegalStateException.

The application server must also allow a resource adapter not to perform any message delivery in-between a single beforeDelivery and afterDelivery method call pair. This scenario arises, for instance, when a resource adapter first chooses to deliver a message and calls beforeDelivery, but later is unable to deliver the message (for example in the case of JMS resource adapters, the resource adapter may abort the message delivery and transfer the message to a Dead Message Queue). The resource adapter must be able to call afterDelivery and complete the delivery cycle. The application server must perform any possible cleanup of actions that occurred in between the beforeDelivery and afterDelivery method calls.

The beforeDelivery and afterDelivery method calls are considered part of a single message delivery call. For each message delivery, the beforeDelivery, afterDelivery methods and the actual message delivery method, must be called from a single thread of control.

Further, the application server must set the thread context class loader to the endpoint application class loader during the beforeDelivery call and must reset it during a corresponding afterDelivery call. This allows a resource adapter to use the application class loader to load application specific classes while deserializing, or reconstructing, a message object. Note, setting of the thread context class loader during the beforeDelivery call is independent of whether an XAResource instance is provided by the resource adapter.

For each message delivery to an endpoint instance, the application server must match an afterDelivery call with a corresponding beforeDelivery call; that is, for each message delivery to an endpoint instance, beforeDelivery and afterDelivery calls are treated as a pair. See FIGURE 13-8.

Thus, in the case of transacted deliveries:

- If a resource adapter does not provide an XAResource instance, it does not get XA transaction notifications.
- If a resource adapter provides an XAResource instance, it gets XA transaction notifications.
- If a resource adapter calls beforeDelivery and afterDelivery methods in addition to providing an XAResource instance, it not only receives XA transaction notifications but also gains control over when the transaction is started and completed. The beforeDelivery and afterDelivery calls have no effect when the resource adapter does not provide an XAResource instance or when the delivery is not transacted.

These various delivery options provide more choices to the resource adapter and allow a wide range of resource adapter and messaging provider implementations to be plugged-in. The application server must support both delivery options, option A and option B.

The release method call on a proxy endpoint instance releases the state of the proxy instance and makes it available for reuse. If the release method is called while a message delivery is in-progress, the application server must throw a java.lang.IllegalStateException, since concurrent calls on a proxy endpoint instance is disallowed. In the case of option B, if the release method is called in-between beforeDelivery and afterDelivery method calls, any transaction started during the corresponding beforeDelivery method call must be aborted by the application server.

13.5.7 Non-Transacted Delivery

1. The application server does not bracket the message delivery to an endpoint instance within a JTA transaction.

2. The resource adapter relies on the successful return of the message delivery call on the endpoint instance for delivery confirmation and may send out an acknowledgement to its message provider if appropriate.

3. Any exception thrown by an endpoint instance during message delivery is taken as a failed delivery. The application server must propagate any exception thrown during message delivery to the resource adapter.

4. The application server does not notify the resource adapter about the delivery outcome upon crash recovery. Note, system failures may happen before the application server calls the actual endpoint instance, or while the actual endpoint instance is doing work, or after the endpoint has completed its work but before the message delivery on the endpoint returns.

The application server does not have delivery status information available during failure recovery, nor does it detect what state the actual endpoint instance was in when the failure happened. Consequently, it is hard to model exactly once delivery semantics for non-transacted dispatches.

13.5.8 Transacted Delivery Using an Imported Transaction

It is possible that a resource adapter may attempt message delivery to an endpoint instance with a transaction initiated by a message source, or message provider; that is, the message source initiates a transaction, and pushes a message to the resource adapter from within the transaction. The resource adapter in turn imports the transaction and attempts message delivery on an endpoint instance from within the source managed transaction.

The resource adapter must use the transaction inflow contract (see Chapter 15, "Transaction Inflow") to import transactions initiated by a message source.

It must be possible to serially deliver one or more messages to one or more endpoint instances belonging to one or more endpoint applications within a single transaction, and be able to commit or abort the transaction as a single unit.

That is, it must be possible for a resource adapter to serially submit one or more `Work` objects (associated with a single transaction) that deliver messages to one or more endpoint instances belonging to one or more endpoint applications. If the

enclosing transaction successfully commits, the messages are deemed to have been successfully delivered. If the enclosing transaction aborts, the messages that were delivered as part of the transaction are canceled.

13.5.9   Requirements

An application server must implement the following behavior for transacted and non-transacted message delivery to an endpoint instance. Before invoking the actual endpoint instance the application server must do the checks prescribed in Table 13-1 shown below, depending on the endpoint transaction preferences and the presence of a source managed transaction:

TABLE 13-1   Application Server Behavior for Transacted and Non-transacted Message Delivery

	Source managed transaction	No source managed transaction
Endpoint instance requires transacted message delivery	Use the source managed transaction. Ignore the `XAResource` instance supplied by the resource adapter, if any.	Start a new transaction. Notify the `XAResource` instance supplied by the resource adapter, if any.
Endpoint does not need transacted message delivery	Suspend the source managed transaction. Ignore the `XAResource` instance supplied by the resource adapter, if any.	No action. Ignore the `XAResource` instance supplied by the resource adapter, if any.

The application server must propagate any exception thrown during a message delivery to the resource adapter irrespective of whether the delivery is transacted or not.

For transacted deliveries, the application server must support both delivery options, option A and option B.

13.6 Endpoint Undeployment

- When a message endpoint is undeployed, the application server notifies the resource adapter through the `endpointDeactivation` method. The application server must pass the same `MessageEndpointFactory` instance and the `ActivationSpec` JavaBean instance that was used during the endpoint activation.
- The resource adapter removes the endpoint information from its internal state and in turn may notify the message provider.

FIGURE 13-7 Transacted Message Delivery: Option A(Sequence Diagram)
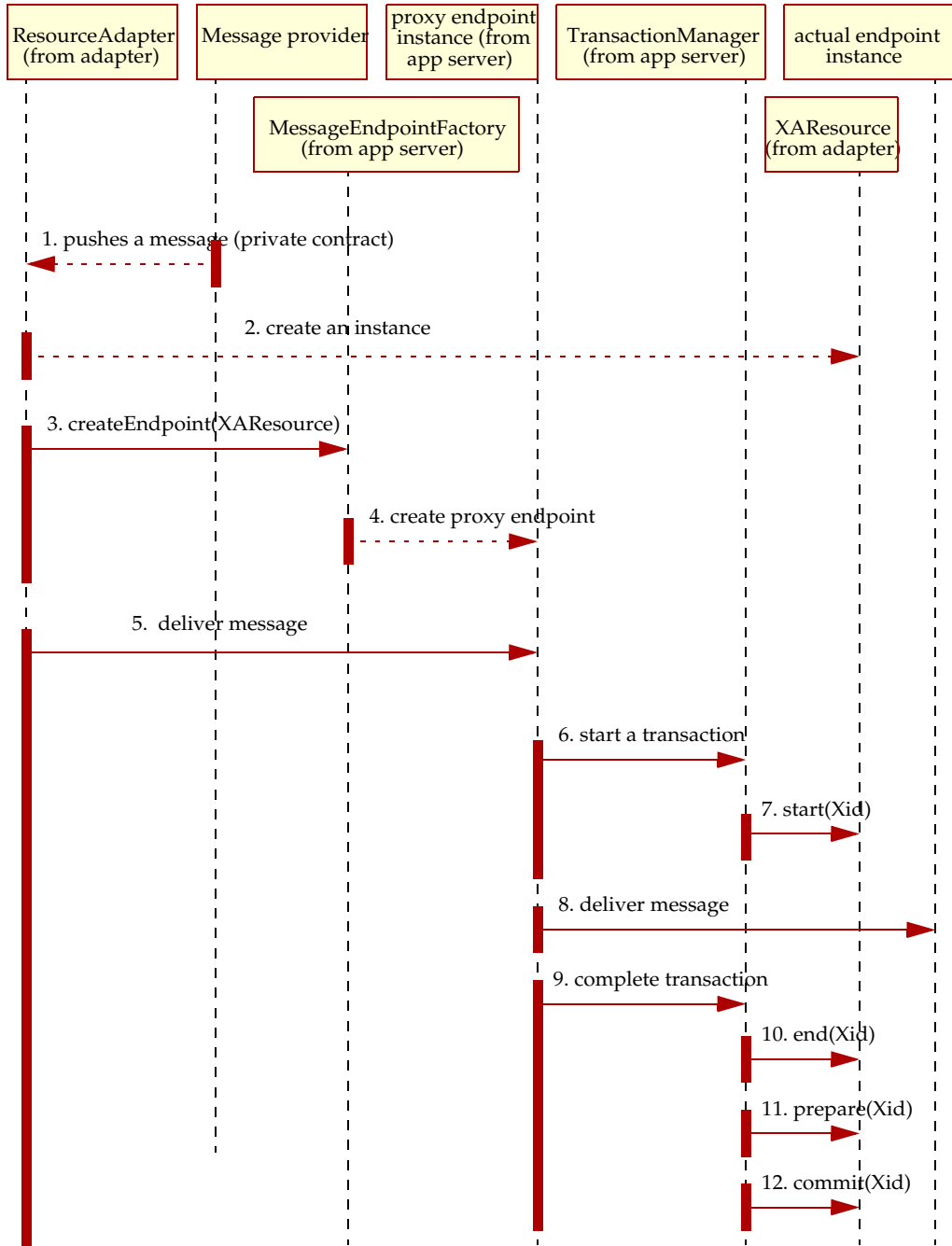
FIGURE 13-8 Transacted Message Delivery: Option B (Sequence Diagram)

FIGURE 13-9 Non-transacted Message Delivery (Sequence Diagram)
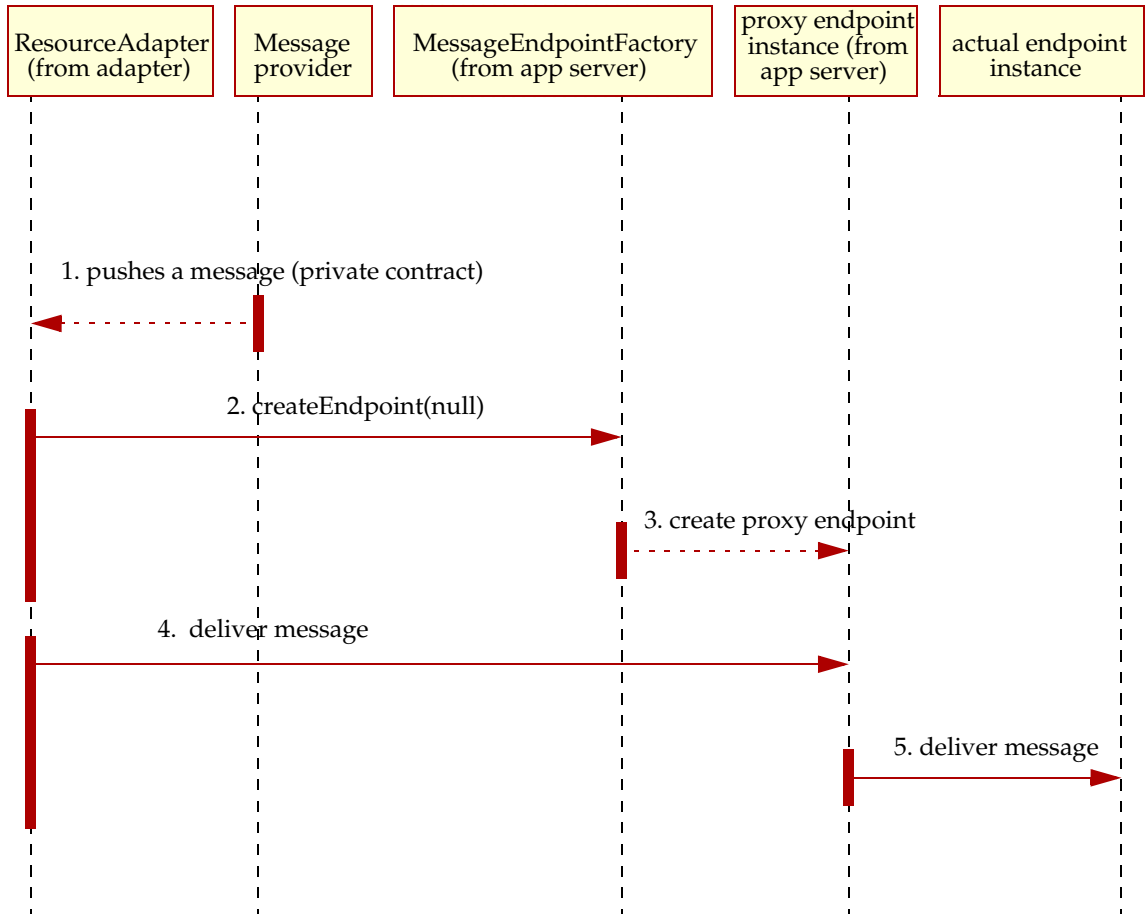
FIGURE 13-10 Endpoint Undeployment (Sequence Diagram)
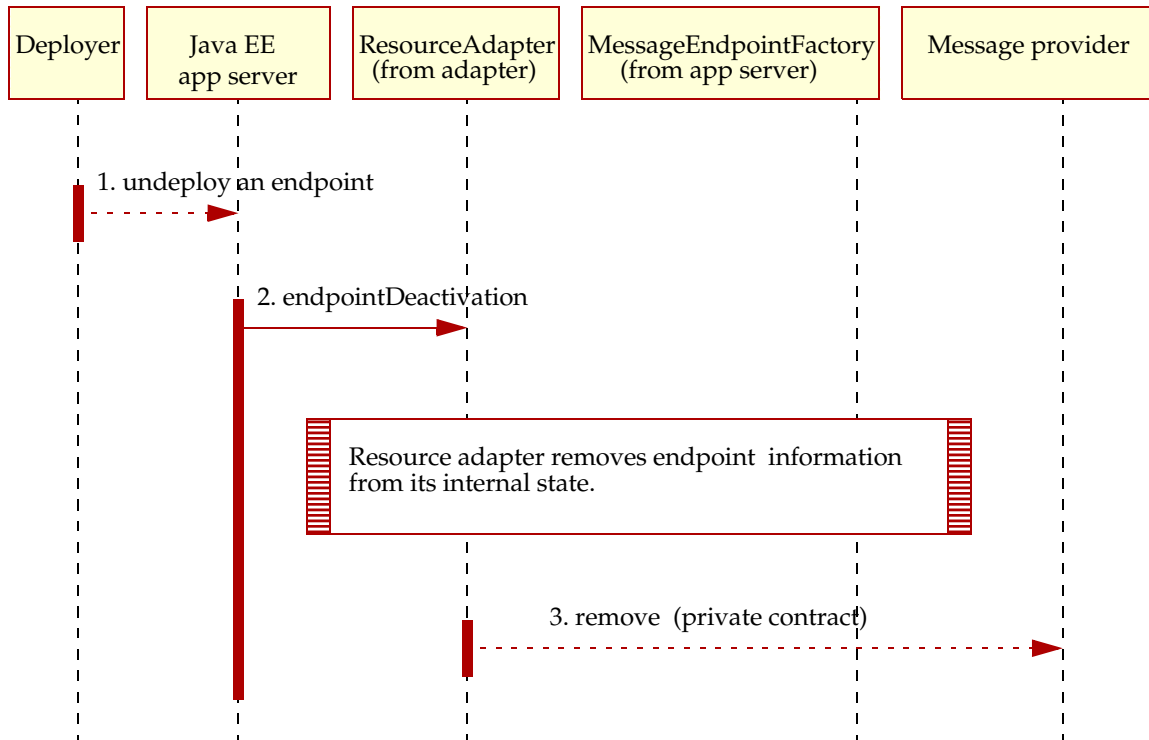
13.7   Java Message Service (JMS) Use Case

For illustration purposes, a JMS use case involving a JMS resource adapter is discussed. The intent of this use case is to show the following:

■ The JMS resource adapter uses the generic message inflow contract and asynchronously delivers messages to message-driven beans through the `onMessage` method on the `javax.jms.MessageListener` interface.
■ The JMS resource adapter is used by an EJB application to send and synchronously receive messages through a `javax.jms.Connection` object.

This use case is shown for illustration purposes only and an application server may or may not achieve all message deliveries to message-driven beans using the generic message inflow contract.

This illustrates how a JMS provider is plugged into a Java EE application server using the standard connector contracts.

CODE EXAMPLE 13-5   Sample JMS Resource Adapter Deployment Descriptor

```xml
<?xml version="1.0" encoding="UTF-8"?>

<connector xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.5">
   <display-name>Wombat-JMSAdapter</display-name>
   <vendor-name>Wombat Software Systems</vendor-name>
   <eis-type>JMS Provider</eis-type>
   <resourceadapter-version>1.0</resourceadapter-version>

   <resourceadapter>
      <resourceadapter-class>
         com.wombat.connector.jms.JMSAdapterImpl
      </resourceadapter-class>

      <!-- ResourceAdapter default configuration properties -->
      <config-property>
         <config-property-name>ServerName</config-property-name>
         <config-property-type>java.lang.String</config-property-type>
         <config-property-value>WombatServer</config-property-value>
      </config-property>
      <config-property>
         <config-property-name>PortNumber</config-property-name>
         <config-property-type>java.lang.String</config-property-type>
         <config-property-value>1050</config-property-value>
      </config-property>
      <config-property>
         <config-property-name>OperationalMode</config-property-name>
         <config-property-type>java.lang.String</config-property-type>
         <config-property-value>Managed</config-property-value>
      </config-property>
      <config-property>
         <config-property-name>ContainerType</config-property-name>
         <config-property-type>java.lang.String</config-property-type>
         <config-property-value>EJB-WEB</config-property-value>
      </config-property>

      <outbound-resourceadapter>

         <connection-definition>
            <managedconnectionfactory-class>
               com.wombat.connector.jms.QueueManagedConnectionFactoryImpl
            </managedconnectionfactory-class>

            <!-- ManagedConnectionFactory default configuration properties -->
```

CODE EXAMPLE 13-5   Sample JMS Resource Adapter Deployment Descriptor

```
              <config-property>
                 <config-property-name>ServerName</config-property-name>
                 <config-property-type>java.lang.String</config-property-type>
                 <config-property-value>
                    WombatQueueServer
                 </config-property-value>

              <config-property>
                 <config-property-name>PortNumber</config-property-name>
                 <config-property-type>java.lang.String</config-property-type>
                 <config-property-value>1051</config-property-value>
              </config-property>

              <connectionfactory-interface>
                 javax.jms.QueueConnectionFactory
              </connectionfactory-interface>
              <connectionfactory-impl-class>
                 com.wombat.connector.jms.QueueConnectionFactoryImpl
              </connectionfactory-impl-class>

              <connection-interface>
                 java.jms.QueueConnection
              </connection-interface>
              <connection-impl-class>
                 com.wombat.connector.jms.QueueConnectionImpl
              </connection-impl-class>
           </connection-definition>

           <connection-definition>
              <managedconnectionfactory-class>
                 com.wombat.connector.jms.TopicManagedConnectionFactoryImpl
              </managedconnectionfactory-class>

              <!-- ManagedConnectionFactory default configuration properties -->
              <config-property>
                 <config-property-name>ServerName</config-property-name>
                 <config-property-type>java.lang.String</config-property-type>
                 <config-property-value>
                    WombatTopicServer
                 </config-property-value>
              </config-property>
              <config-property>
                 <config-property-name>PortNumber</config-property-name>
                 <config-property-type>java.lang.String</config-property-type>
                 <config-property-value>1052</config-property-value>
              </config-property>
```

CODE EXAMPLE 13-5   Sample JMS Resource Adapter Deployment Descriptor

```
            <connectionfactory-interface>
                javax.jms.TopicConnectionFactory
            </connectionfactory-interface>
            <connectionfactory-impl-class>
                com.wombat.connector.jms.TopicConnectionFactoryImpl
            </connectionfactory-impl-class>

            <connection-interface>
                java.jms.TopicConnection
            </connection-interface>
            <connection-impl-class>
                com.wombat.connector.jms.TopicConnectionImpl
            </connection-impl-class>
        </connection-definition>

        <connection-definition>
            <managedconnectionfactory-class>
                com.wombat.connector.jms.ManagedConnectionFactoryImpl
            </managedconnectionfactory-class>

            <!--
            This ManagedConnectionFactory JavaBean inherits the ResourceAdapter
            JavaBean configuration properties, and does not override any
            of the global defaults.
            -->

            <connectionfactory-interface>
                javax.jms.ConnectionFactory
            </connectionfactory-interface>
            <connectionfactory-impl-class>
                com.wombat.connector.jms.ConnectionFactoryImpl
            </connectionfactory-impl-class>

            <connection-interface>
                java.jms.Connection
            </connection-interface>
            <connection-impl-class>
                com.wombat.connector.jms.ConnectionImpl
            </connection-impl-class>
        </connection-definition>

        <transaction-support>XATransaction</transaction-support>
        <reauthentication-support>false</reauthentication-support>

    </outbound-resourceadapter>

    <inbound-resourceadapter>
```

CODE EXAMPLE 13-5   Sample JMS Resource Adapter Deployment Descriptor

```
        <messageadapter>
            <messagelistener>
                <messagelistener-type>
                    javax.jms.MessageListener
                </messagelistener-type>

                <activationspec>
                    <activationspec-class>
                        com.wombat.connector.jms.ActivationSpecImpl
                    </activationspec-class>

                    <!--
                    The endpoint deployer configures the ActivationSpec JavaBean
                    and may override some of the global defaults inherited from
                    the ResourceAdapter JavaBean. For example, the ServerName
                    and the PortNumber properties.
                    -->

                    <!-- required config property names for ActivationSpec -->
                    <required-config-property>
                        <config-property-name>Destination</config-property-name>
                    </required-config-property>

                </activationspec>

            </messagelistener>
        </messageadapter>
    </inbound-resourceadapter>

    <adminobject>
        <adminobject-interface>javax.jms.Queue</adminobject-interface>
        <adminobject-class>
            com.wombat.connector.jms.QueueImpl
        </adminobject-class>
    </adminobject>
    <adminobject>
        <adminobject-interface>javax.jms.Topic</adminobject-interface>
        <adminobject-class>
            com.wombat.connector.jms.TopicImpl
        </adminobject-class>
    </adminobject>
    <adminobject>
        <adminobject-interface>
            javax.jms.Destination
        </adminobject-interface>
        <adminobject-class>
            com.wombat.connector.jms.DestinationImpl
```

CODE EXAMPLE 13-5   Sample JMS Resource Adapter Deployment Descriptor

```
        </adminobject-class>
      </adminobject>

   </resourceadapter>
</connector>
```

CODE EXAMPLE 13-6   A Sample JMS ActivationSpec Implementation

```java
package com.wombat.connector.jms;

import java.io.Serializable;
import javax.resource.spi.ActivationSpec;
import javax.resource.spi.InvalidPropertyException;

public class ActivationSpecImpl implements ActivationSpec,
Serializable {

    public setAcknowledgeMode(String mode) { ... }
    public String getAcknowledgeMode() { ... }

    public setSubscriptionDurability(String durability) { ... }
    public String getSubscriptionDurability() { ... }

    public setMessageSelector(String selector) { ... }
    public String getMessageSelector() { ... }

    public setDestinationType(String destType) { ... }
    public String getDestinationType() { ... }

    public setDestination(String dest) { ... }
    public String getDestination() { ... }

    public setSubscriptionName(String name) { ... }
    public String getSubscriptionName() { ... }

    public setClientId(String id) { ... }
    public String getClientId() { ... }

   public void validate() throws InvalidPropertyException { ... }
}
```

CODE EXAMPLE 13-7 A Sample EJB 2.0 Message-Driven Bean Deployment Descriptor

```
<?xml version="1.0" encoding="US-ASCII"?>

<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD
Enterprise
JavaBeans 2.0//EN' 'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>

<ejb-jar>
  <display-name>MDB1</display-name>
  <enterprise-beans>
    <message-driven>
     <display-name>MDB_DURABLE_CMT</display-name>
      <ejb-name>MDB_DURABLE_CMT</ejb-name>
     <ejb-class>msgbean.MsgBean</ejb-class>
     <transaction-type>Container</transaction-type>
     <message-driven-destination>
        <destination-type>javax.jms.Topic</destination-type>
      <subscription-durability>Durable</subscription-durability>
     </message-driven-destination>
     <message-selector>JMSType = 'car' AND color = 'blue'
     </message-selector>
    </message-driven>
  </enterprise-beans>
</ejb-jar>
```

CODE EXAMPLE 13-8 A Sample EJB 2.1 Message-Driven Bean Deployment Descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns="http://java.sun.com/xml/ns/j2ee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      version="2.1">
   <display-name>MDB1</display-name>
   <enterprise-beans>
     <message-driven>
        <display-name>MDB_DURABLE_CMT</display-name>
        <ejb-name>MDB_DURABLE_CMT</ejb-name>
        <ejb-class>msgbean.MsgBean</ejb-class>
        <messaging-type>javax.jms.MessageListener
             </messaging-type>
        <transaction-type>Container</transaction-type>
        <activation-config>
           <activation-config-property>
              <activation-config-property-name>
                 destinationType
              </activation-config-property-name>
              <activation-config-property-value>
                 javax.jms.Topic
```

CODE EXAMPLE 13-8   A Sample EJB 2.1 Message-Driven Bean Deployment Descriptor

```
                                 </activation-config-property-value>
                             </activation-config-property>
                             <activation-config-property>
                                 <activation-config-property-name>
                                     SubscriptionDurability
                                 </activation-config-property-name>
                                 <activation-config-property-value>
                                     Durable
                                 </activation-config-property-value>
                             </activation-config-property>
                             <activation-config-property>
                                 <activation-config-property-name>
                                     MessageSelector
                                 </activation-config-property-name>
                                 <activation-config-property-value>
                                     JMSType = 'car' AND color = 'blue'
                                 </activation-config-property-value>
                             </activation-config-property>
                         </activation-config>
                     </message-driven>
                 </enterprise-beans>
            </ejb-jar>
```

CODE EXAMPLE 13-9   Sample EJB 2.1 Deployment Descriptor

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar xmlns="http://java.sun.com/xml/ns/j2ee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      version="2.1">
   <display-name>Ejb1</display-name>
   <enterprise-beans>
      <session>
         <display-name>JMSBean</display-name>
         <ejb-name>JMSBean</ejb-name>
         <home>jms.JMSHome</home>
         <remote>jms.JMS</remote>
         <ejb-class>jms.JMSEJB</ejb-class>
         <session-type>Stateful</session-type>
         <transaction-type>Container</transaction-type>
         <resource-ref>
            <description>description</description>
            <res-ref-name>jms/MyQueueConnectionFactory</res-ref-name>
            <res-type>javax.jms.QueueConnectionFactory</res-type>
            <res-auth>Application</res-auth>
            <res-sharing-scope>Shareable</res-sharing-scope>
         </resource-ref>
         <resource-env-ref>
```

```
                <resource-env-ref-name>jms/QueueName</resource-env-ref-name>
                <resource-env-ref-type>javax.jms.Queue</resource-env-ref-type>
            </resource-env-ref>
        </session>
    </enterprise-beans>
</ejb-jar>
```

13.7.1   Message-Driven Bean Asynchronously Receiving Messages

13.7.1.1   Message-Driven Bean Deployment

- A message-driven bean application developer or assembler supplies a deployment descriptor, or annotated application component, which specifies a destination type, message selector, and subscription durability information needed to setup subscription to a certain destination, Queue or Topic. Note, this information is a hint which is used by the message-driven bean deployer to setup the subscription.
- The message-driven bean deployer selects an appropriate JMS resource adapter based on the quality-of-service and creates an ActivationSpec JavaBean instance and configures the required property "Destination" as well as other properties related to the JMS messaging style and the specific resource adapter.
- The endpoint deployer may need to interact with the JMS provider to setup an appropriate "Destination" and other steps necessary to complete message-driven bean deployment.
- The deployer deploys the message-driven bean application. During deployment, the deployer provides the configured ActivationSpec JavaBean to the application server, along with information about the chosen JMS resource adapter.
- The application server calls the endpointActivation method on the JMS resource adapter and passes the configured ActivationSpec JavaBean instance and a MessageEndpointFactory instance. During the endpointActivation method call the JMS adapter interacts with its provider to setup message delivery to the message-driven bean. This completes the endpoint activation, and the message-driven bean is ready to receive messages.

13.7.1.2   Message Delivery

- When messages start arriving, the JMS adapter uses the MessageEndpointFactory instance to get an endpoint instance and delivers messages to the endpoint through the javax.jms.MessageListener.onMessage method.

- The application server interposes the message delivery and injects transactions based on the message-driven bean preferences, container-managed transaction or bean-managed transaction, before delivering the message to a message-driven bean instance.
- When a delivery is transacted, the application server notifies the JMS resource adapter using the `XAResource` object. The JMS resource adapter may use the notifications to send acknowledgements to its message provider.
- The JMS resource adapter, depending on the traffic, may attempt concurrent delivery of messages by using multiple endpoint instances obtained through `MessageEndpointFactory`. The application server appropriately handles concurrent message deliveries and dispatches messages to separate message-driven bean instances.

13.7.1.3 Message-Driven Bean Undeployment

- When the message-driven bean is undeployed, the application server calls the `endpointDeactivation` method on the JMS resource adapter to deactivate the message endpoint.
- The JMS adapter in turn notifies its message provider.

13.7.2 EJB Using JMS API to Send and Synchronously Receive Messages Via a JMS Resource Adapter

- The JMS resource adapter provides `javax.jms.Connection` objects which expose the JMS API to the EJB application. The EJB directly uses the JMS API to send and synchronously receive messages. The `javax.jms.Connection` objects are obtained from a `ConnectionFactory` supplied by the JMS resource adapter.
- Based on the EJB deployment descriptor information (`resource-ref`'s and `resource-env-ref`'s) or resource reference injection annotations (Resource annotation defined in "Common annotations for the Java Platform Specification, version 1.2" on page E-2), the EJB deployer configures the appropriate `ConnectionFactory` objects (`resource-ref`'s) in the component name space (`java:comp/env`). The application deployer also configures the necessary `Queue` or `Topic` administered objects (`resource-env-ref`'s) in the component name space. The JMS resource adapter provides the implementation of the various `ConnectionFactory` and administered objects.
- At runtime, the component does a JNDI lookup of a `ConnectionFactory` object from its component name space (`java:comp/env`), and uses it to create a `javax.jms.Connection` object which is used for sending and synchronously receiving messages. Similarly, the component uses the JNDI lookup mechanism to obtain the configured `Queue` or `Topic` administered objects.

13.7.2.1 Using JMS API to Send Messages

CODE EXAMPLE 13-10 Sending Messages Using the JMS API

```
// get JNDI handle
Context jndiContext = new InitialContext();

// get connection factory
ConnectionFactory connectionFactory = (ConnectionFactory)
      jndiContext.lookup("QueueConnectionFactory");

// get connection from factory
Connection connection = connectionFactory.getConnection();

// get session from connection
Session session = connection.createSession(true,
      AUTO_ACKNOWLEDGE);

// get destination from JNDI
Queue stockQueue = (Queue) jndiContext.lookup("StockQueue");

// create a message producer
MessageProducer sender = session.createProducer(stockQueue);

// create a message
TextMessage message = session.createTextMessage();
message.setText(msgData);

// send the message
sender.send(message);
```

13.7.2.2 Java EE Component Using JMS API to Synchronously Receive Messages

CODE EXAMPLE 13-11 Synchronously Receiving Messages in a Java EE Component

```
// get JNDI handle
Context jndiContext = new InitialContext();

// get connection factory
ConnectionFactory connectionFactory = (ConnectionFactory)
      jndiContext.lookup("QueueConnectionFactory");

// get connection from factory
Connection connection = connectionFactory.getConnection();

// get session from connection
```

CODE EXAMPLE 13-11 Synchronously Receiving Messages in a Java EE Component

```
Session session = connection.createSession(true,
      AUTO_ACKNOWLEDGE);

// get destination from JNDI
Queue stockQueue = (Queue) jndiContext.lookup("StockQueue");

// create a message consumer
MessageConsumer receiver = session.createConsumer(stockQueue);

// enable connection to receive messages
connection.start();

// synchronously receive the message
TextMessage message = (TextMessage) receiver.receive(message);
```

13.8   A Non-JMS Use Case

This illustration is intended to show that it is possible to plug a wide range of message providers into a Java EE application server by way of the standard connector contracts, such that it is possible for an application to either asynchronously receive messages through the message inflow contract or to use a connection object to send and synchronously receive messages.

13.8.1   Resource Adapter Deployment Descriptor

This is an example deployment descriptor for a resource adapter that can provide both inbound and outbound communication with a particular EIS.

On the inbound side, it can deliver messages to a message-driven bean that implements a com.kangaroo.MessageListener. Note, the deployment descriptor has a messagelistener-type element with the value com.kangaroo.MessageListener. The activationspec-class is of type com.kangaroo.MyEISActivationSpecImpl. This ActivationSpec JavaBean has a single required property PortNumber, that is required to establish a connection to the remote EIS. When the EIS data is received, the resource adapter will convert it to a com.kangaroo.Message and deliver it to a message-driven bean instance.

The resource adapter also provides a `ManagedConnectionFactory`
implementation for outbound communication to the EIS. This also takes a single
configuration parameter called `PortNumber`.

CODE EXAMPLE 13-12 Deployment Descriptor for a Resource Adapter

```xml
<?xml version="1.0" encoding="UTF-8"?>

<connector xmlns="http://java.sun.com/xml/ns/j2ee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      version="1.5">

   <display-name>KangarooAdapter</display-name>
   <vendor-name>Kangaroo Software Inc.</vendor-name>
   <eis-type>JMS Provider</eis-type>
   <resourceadapter-version>1.0</resourceadapter-version>

   <resourceadapter>
      <resourceadapter-class>
         com.kangaroo.MyEISAdapterImpl
      </resourceadapter-class>

      <!-- ResourceAdapter default configuration properties -->
      <config-property>
         <description>URL for EIS instance</description>
         <config-property-name>EIS_URL</config-property-name>
         <config-property-type>java.lang.String</config-property-type>
         <config-property-value>TBD</config-property-value>
      </config-property>

      <outbound-resourceadapter>
         <connection-definition>
            <managedconnectionfactory-class>
               com.Kangaroo.MyManagedConnectionFactoryImpl
            </managedconnectionfactory-class>

            <!-- ManagedConnectionFactory default configuration properties -->
            <config-property>
               <config-property-name>PortNumber</config-property-name>
               <config-property-type>java.lang.String</config-property-type>
               <config-property-value>1051</config-property-value>
            </config-property>

            <connectionfactory-interface>
               javax.resource.cci.ConnectionFactory
            </connectionfactory-interface>
            <connectionfactory-impl-class>
               com.Kangaroo.MyConnectionFactoryImpl
            </connectionfactory-impl-class>
```

CODE EXAMPLE 13-12 Deployment Descriptor for a Resource Adapter

```
                <connection-interface>
                    javax.resource.cci.Connection
                </connection-interface>
                <connection-impl-class>
                    com.Kangaroo.MyConnectionImpl
                </connection-impl-class>
            </connection-definition>

            <transaction-support>NoTransaction</transaction-support>
            <reauthentication-support>false</reauthentication-support>
        </outbound-resourceadapter>

        <inbound-resourceadapter>
            <messageadapter>
                <messagelistener>
                    <messagelistener-type>
                        com.kangaroo.MessageListener
                    </messagelistener-type>
                    <activationspec>
                        <activationspec-class>
                            com.Kangaroo.MyEISActivationSpecImpl
                        </activationspec-class>
                        <required-config-property>
                            <config-property-name>PortNumber</config-property-name>
                        </required-config-property>
                    </activationspec>
                </messagelistener>
            </messageadapter>
        </inbound-resourceadapter>

    </resourceadapter>
</connector>
```

13.8.2   Resource Adapter Deployment

Before use, the resource adapter is required to be deployed on the application server. During resource adapter deployment, the deployer configures a `ResourceAdapter` JavaBean instance and deploys the resource adapter.

13.8.3 Message-Driven Bean Asynchronously Receiving Notifications From an EIS

13.8.3.1 The Message-Driven Bean Deployment Descriptor

CODE EXAMPLE 13-13 Deployment Descriptor for a Message-Driven Bean

```xml
<?xml version="1.0" encoding="US-ASCII"?>

<!DOCTYPE ejb-jar PUBLIC ?-//Sun Microsystems, Inc.//DTD
      Enterprise JavaBeans 2.1//EN? ?http://java.sun.com/dtd/ejb-jar_2_1.dtd?>

<ejb-jar>
    <display-name>Ejb1</display-name>
    <enterprise-beans>
       <message-driven>
          <display-name>EIS Receiver Bean</display-name>
          <ejb-name>EISReceiver</ejb-name>
          <ejb-class>myapp.EISReceiverBean</ejb-class>
          <messaging-type>com.kangaroo.MessageListener</messaging-type>
          <transaction-type>Container</transaction-type>
          <activation-config>
             <activation-config-property>
                <activation-config-property-name>
                   functionName
                </activation-config-property-name>
                <activation-config-property-value>
                   CustomerChangeNotification
                </activation-config-property-value>
             </activation-config-property>
             <activation-config-property>
                <activation-config-property-name>
                   CustomerName
                </activation-config-property-name>
                <activation-config-property-value>
                   Wombat Systems
                </activation-config-property-value>
             </activation-config-property>
          </activation-config>
       </message-driven>
    </enterprise-beans>
</ejb-jar>
```

When the message-driven bean is deployed, the bean deployer chooses an appropriate resource adapter based on the type of the message listener it supports. In this case, the deployer chooses the resource adapter with the ResourceAdapter

class `com.kangaroo.MyEISAdapterImpl` since it supports the `com.kangaroo.MessageListener` type.

Then the deployer creates an instance of `com.kangaroo.MyEISActivationSpecImpl` and populates it with values. The `ActivationSpec` JavaBean instance will also contain values of properties that are set in the `activation-config` section of the bean's deployment descriptor. In the example above, the properties are `FunctionName`, with the value `CustomerChangeNotification`, and `CustomerName`, with the value `Wombat Systems`, which the deployer may choose to override.

Finally, the deployer provides the configured `ActivationSpec` JavaBean instance to the application server.

13.8.4 Message-Driven Bean and Resource Adapter Activation

When the application server is started, it will activate the resource adapter by calling its `start` method. The application server will create an instance of the message-driven bean with class name `myapp.EISReceiverBean`. Then the application server will call the `endpointActivation` method on the resource adapter instance and pass in the configured `ActivationSpec` instance associated with the deployed message-driven bean, and a `MessageEndpointFactory` instance. The resource adapter will use the information in the `ActivationSpec` to establish a subscription to the requested data from the EIS.

13.8.5 Message Delivery

When a notification arrives from the EIS, the resource adapter has the responsibility of converting its data to a `com.kangaroo.Message` if it is not already in this format. The resource adapter will then use the `MessageEndpointFactory` to deliver the notification to the message-driven bean. Note that, rather than calling the `MessageEndpointFactory` directly, the resource adapter is likely to instantiate a `Work` object, and pass it to the application server through the `WorkManager` interface. When the `doWork` method of the `WorkManager` is called the dispatch will occur. This will allow the resource adapter to continue to process incoming messages without blocking until message-driven bean dispatch has completed.

EJB Invocation

This chapter describes how to invoke session and entity beans from a resource adapter.

14.1   Overview

A resource adapter may need to call session or entity beans for several reasons:

- To dispatch calls from an EIS to a bean in order to execute business logic
- To use EJB container-managed persistence (CMP) mechanism for persistence

In order to dispatch calls to a session or entity bean, the resource adapter is required to detect the target bean type, the method name, and the method parameters. Upon receiving a request from the EIS by way of a remote protocol, the resource adapter's dispatch logic is required to:

- Choose an appropriate bean and a target method name based on the request received from the EIS.
- Unmarshall, that is, deserialize, the request parameters received from the EIS and call the target bean method.

14.2 EJB Invocation Model

FIGURE 14-1 EJB Invocation Model
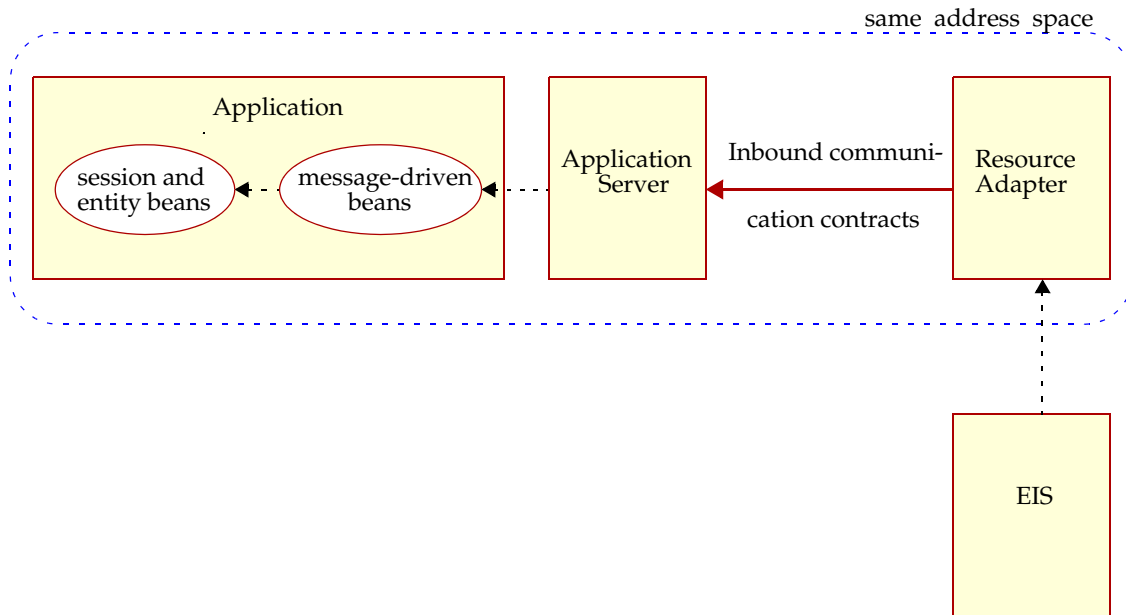
For session or entity bean invocations, the resource adapter's bean dispatch logic can use the bean client view model by way of a message-driven bean. The EJB specification (see "Enterprise JavaBeans (EJB) Specification, version 3.2" on page E-1) defines the EJB client view, and describes how the client view is used to access session or entity beans. The EJB client view is available to a message-driven bean.

The resource adapter could structure its code such that its bean dispatch logic is written as a message-driven bean. The message-driven bean chooses an appropriate session or entity bean and a target method, unmarshalls the request parameters and invokes the chosen bean based on the request information received from the EIS.

The resource adapter can use the message inflow contract to call a message-driven bean, and use the message-driven bean to dispatch calls to session and entity beans using the EJB client view model. The EJB specification allows a request-response style message-driven bean call which could be used for synchronous RPC-style calls. The message-driven bean could be packaged either with the resource adapter or separately.

Thus, the message-driven bean could be used as a replaceable unit of the resource adapter which serves the job of a bean dispatcher. The message inflow contract allows the creation of multiple endpoint instances (message-driven beans) at runtime, and hence it is possible to do concurrent bean dispatches.

Further, the transaction inflow mechanism (described in Chapter 15, "Transaction Inflow") allows the resource adapter to use the transaction information obtained from the EIS for bean invocations. Note, however, the application server may suspend the imported transaction depending on the transaction preference of the target bean method.

14.3   An Illustrative Use Case

Wombat Systems is a resource adapter vendor. The resource adapter supports inbound communication from an EIS to application components residing in an application server container. The resource adapter uses the message inflow contract to call message-driven beans which serve as a dispatcher for session and entity bean invocations. In this case, Wombat Systems supplies both the resource adapter and the message-driven beans, even though these could be supplied by different vendors.

The EIS uses multiple concurrent conversations in its interactions with the resource adapter. Each conversation may involve multiple serial requests. The resource adapter has a set of `Work` objects (threads), each of which is used for carrying on a specific conversation. The resource adapter manages the multiple concurrent conversations, and calls a message-driven bean instance whenever a request message arrives as part of a conversation.

The following code sample shows a possible message-driven bean implementation:

CODE EXAMPLE 14-1   A Message-Driven Bean Implementation

```
package com.wombat.ra;

import javax.ejb.MessageDrivenBean;
import javax.naming.InitialContext;

public class WombatMDB
        implements MessageDrivenBean, WombatMessageListener {

public static int CONV_START = 0;
    public static int CONV_CONTINUE = 1;
    public static int CONV_END = 2;

    private Context jndiContext = null;
```

CODE EXAMPLE 14-1   A Message-Driven Bean Implementation

```
    private ConvBeanHome chome = null;

    public void ejbCreate() {
        jndiContext = new InitialContext();
        chome = (ConvBeanHome)
                jndi.lookup("java:comp/env/ConvBeanHome");
    }

    ConvResponse onMessage(ConvRequest requestMsg) {
        // get conversation id and request type from the request
        // message
        int convId = ...;
        int type = ...;

        if (type == CONV_START) {
            // create entity EJB for holding the specific
            // conversation state
            ConvBean cbean = chome.create(convId);
        } else if (type == CONV_CONTINUE) {
            ConvBean cbean = chome.findByPrimaryKey(convId);

            // unmarshall EJB method parameters
            ...;

            // invoke EJB and return response
            Object resp = cbean.myBusinessMethod(params);
            ConvResponse cresp = Utility.convert(resp);
            return cresponse;
        } else if (type == CONV_END) {
            cbean.remove();
        }

        return null;

    }

    public void ejbRemove() {
        jndiContext = null;
        chome = null;
    }
}
```

The resource adapter uses the message-driven bean as a generic dispatcher for session and entity bean invocations, and relies on the application server to efficiently pool message-driven bean instances. Each message-driven bean call should be just as efficient as a method call on a resource adapter local object.

14.3.0.1   Message-Driven Bean Dispatcher Pattern

When a worker thread from a resource adapter accesses a message-driven bean method, the JNDI context of the bean is available to the thread, although only during the method call on the bean.

The resource adapter may take advantage of this, and use the bean as a dispatcher. That is, the resource adapter may park the thread within the bean method inside a while loop, and use it to process resource adapter specific data structures passed into the bean method as method parameters, and also use the JNDI context of the bean to access resources and other components.

In this case, the bean becomes a special Java object that has access to JNDI context, which the resource adapter may use. This usage pattern illustrates a tight coupling between the resource adapter and the message-driven bean, and it is likely that the resource adapter would provide the bean implementation as well.

Transaction Inflow

This chapter specifies a contract between an application server and a resource adapter that allows a resource adapter to propagate an imported transaction to an application server, so that the application server and subsequent participants can do work as part of the imported transaction. This contract also allows a resource adapter to flow-in transaction completion and crash recovery calls initiated by an EIS, and ensures that the atomicity, consistency, isolation and durability (ACID) properties of the imported transaction are preserved.

15.1 Overview

A resource adapter may need to import an incoming external transaction context obtained from a remote protocol message and do work as part of the imported transaction. The work done by the resource adapter as part of the imported transaction may involve interactions with the application server and the application components.

The resource adapter is expected to process the wire protocol and the transaction context format and be able to import an incoming transaction in an EIS-specific way. The resource adapter is required to propagate the imported transaction to the application server and also flow-in transaction completion and crash recovery calls initiated by the EIS. In order for the resource adapter to accomplish this, it requires the following:

- A standard form to represent the transaction context imported by the resource adapter.
- A mechanism to associate the work done by the resource adapter as part of the imported transaction.

- A mechanism to treat the application server like a resource manager in order to make it participate in the two-phase commit and crash recovery flows initiated by the external transaction originator, the EIS.

FIGURE 15-1 Transaction Inflow Contract
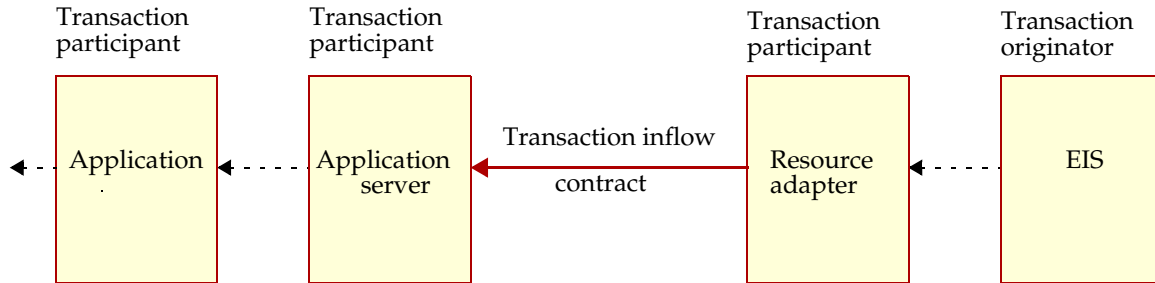
15.2   Goals

- Provide a standard mechanism for a resource adapter to propagate an imported transaction to an application server.
- Provide a standard mechanism for a resource adapter to flow-in transaction completion and crash recovery calls from an EIS.
- Ensure that the ACID properties of the transaction imported by a resource adapter are preserved.

15.3   Use Case Scenario

An EIS initiates a transaction and calls application components residing in an application server through a resource adapter. The EIS propagates the transaction context as part of each call to the resource adapter, which is used by the application server to recreate the transaction context before the application components are called. The work done by the application components is automatically enlisted as part of the imported transaction. When the EIS completes the transaction, the transaction completion notifications flow to the application server through the resource adapter, and the transaction is completed.

The transaction inflow contract may be used in various situations. For example:

- A message provider may use the contract to push messages to a resource adapter for delivery to application components.

- The contract may be leveraged to make the application components do work as part of a transaction initiated by a legacy EIS.

Note that application components may not always do work as part of an EIS-initiated transaction, for example, when the declarative transaction attribute of an enterprise bean's method is `RequiresNew`, `Never`, `NotSupported`, or if bean-managed transaction demarcation is used.

FIGURE 15-2 Transaction Inflow Contract (Object Diagram)
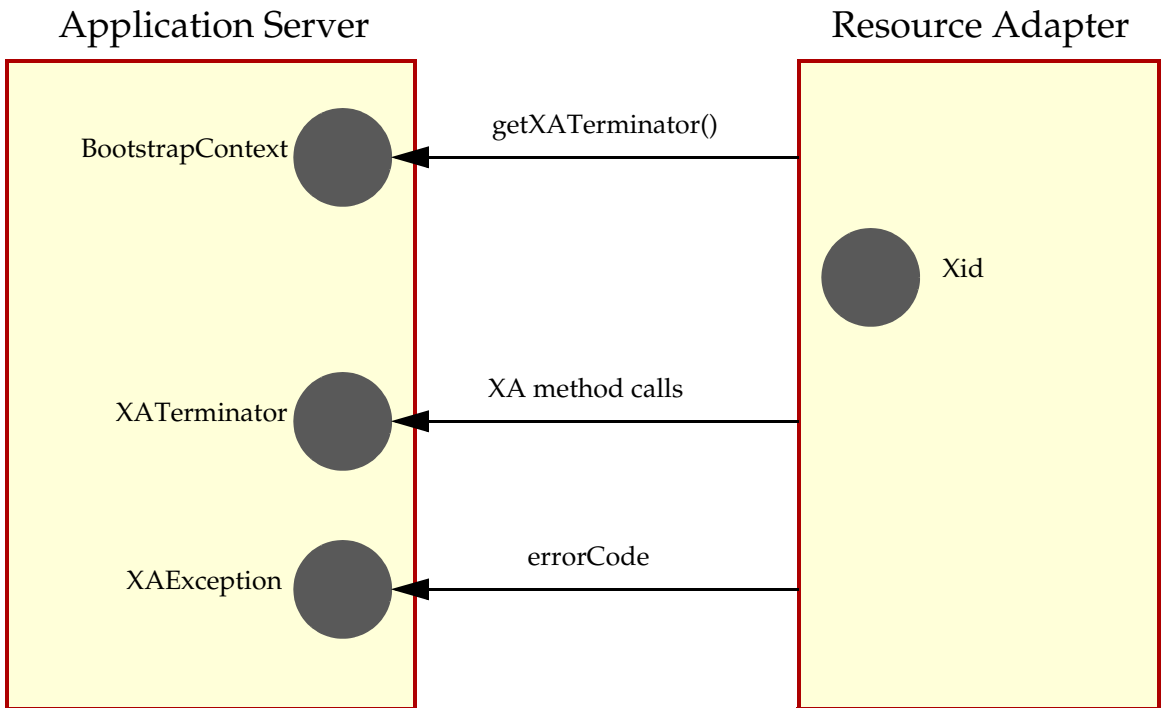
15.4 Transaction Inflow Model

```
package javax.resource.spi;

import javax.transaction.xa.Xid;
import java.transaction.xa.XAException;
import javax.resource.spi.work.WorkManager;

public interface BootstrapContext {
    WorkManager getWorkManager();
    XATerminator getXATerminator();
    ... // other methods
}

public interface XATerminator {
    public void commit(Xid xid, boolean onePhase) throws
XAException;
    public void forget(Xid xid) throws XAException;
    public int prepare(Xid xid) throws XAException;
    public Xid[] recover(int flag) throws XAException;
    public void rollback(Xid xid) throws XAException;
}
```

The `BootstrapContext` interface allows the resource adapter to obtain a `WorkManager` instance and an `XATerminator` instance. These instances are not required to be unique. The resource adapter uses the `WorkManager` instance to submit `Work` instances for execution, and uses the `XATerminator` instance for transaction completion and crash recovery flows.

15.4.1 Processing of Transactional Calls

The steps involved in propagating an imported transaction from a resource adapter to the application server in order to do transactional work is as follows:

1. The EIS makes a transactional call to the resource adapter. The resource adapter is expected to process the EIS-specific transaction context structure and the wire protocol. The resource adapter imports the transaction context that arrived along with the incoming message.

2. The resource adapter represents the imported transaction context in a standard form using the `javax.transaction.xa.Xid` instance.

3. The resource adapter constructs a `Work` instance, which is expected to do work as part of the transactional message, and also creates an `ExecutionContext` instance containing the constructed `Xid`. It then submits the `Work` instance along with the `ExecutionContext` instance to the application server's `WorkManager` for execution. Version 1.6 of the Connector Architecture defines a standard class, `TransactionContext`, for the propagation of transaction context information from the EIS to the application server. Resource adapters may use this instead of `ExecutionContext`. See Section 11.5, "TransactionContext Class" on page 11-14 for more information on the `TransactionContext` class.

4. The application server's `WorkManager` accepts the submitted `Work` instance and recreates the execution context for the `Work` instance. That is, the work to be done is enlisted as part of the imported transaction. It then calls the `run` method on the `Work` object.
Note, however, all the work done by the `Work` object may not be part of the transaction. For example, the application server may suspend the imported transaction depending on the transaction preference of the bean method that may be invoked.

The above steps may be repeated any number of times for a particular transaction from any resource adapter. However, the application server must disallow transactional `Work` submissions with a `WorkCompletedException` set to an appropriate error code, irrespective of which resource adapter it comes from, under the following circumstances:

■ If a `Work` instance associated with the transaction is already present. That is, concurrent work within an imported transaction is disallowed. The error code to indicate this is `WorkException.TX_CONCURRENT_WORK_DISALLOWED`.
■ The application server is unable to recreate the transaction. That is, it fails in its attempt to enlist the `Work` instance with the transaction. The error code to indicate this is `WorkException.TX_RECREATE_FAILED`.

15.4.2   Transaction Completion Processing

The steps involved in completing of the imported transaction initiated by the external EIS are as follows:

1. The EIS sends a prepare message for a particular transaction.

2. The resource adapter obtains an `XATerminator` instance from the application server through the `getXATerminator` method of the `BootstrapContext` instance. Note, this step may be done at any time, and the obtained `XATerminator` instance may be used for transaction completion flows across multiple imported transactions. The `XATerminator` implementation should be thread-safe and re-entrant.

3. The resource adapter calls the `prepare` method of the XATerminator instance with an appropriate `Xid` instance, and returns the outcome of the prepare operation to the EIS.

4. **When the EIS sends a commit message for the transaction, the resource adapter calls the `commit` method of the `XATerminator` instance with an appropriate `Xid` instance. Note, it is possible for the `commit` method to be called without a prior `prepare` method call in the case of one-phase commit.**

FIGURE 15-3 Transactional Calls and Transaction Completion Flow (Sequence Diagram)
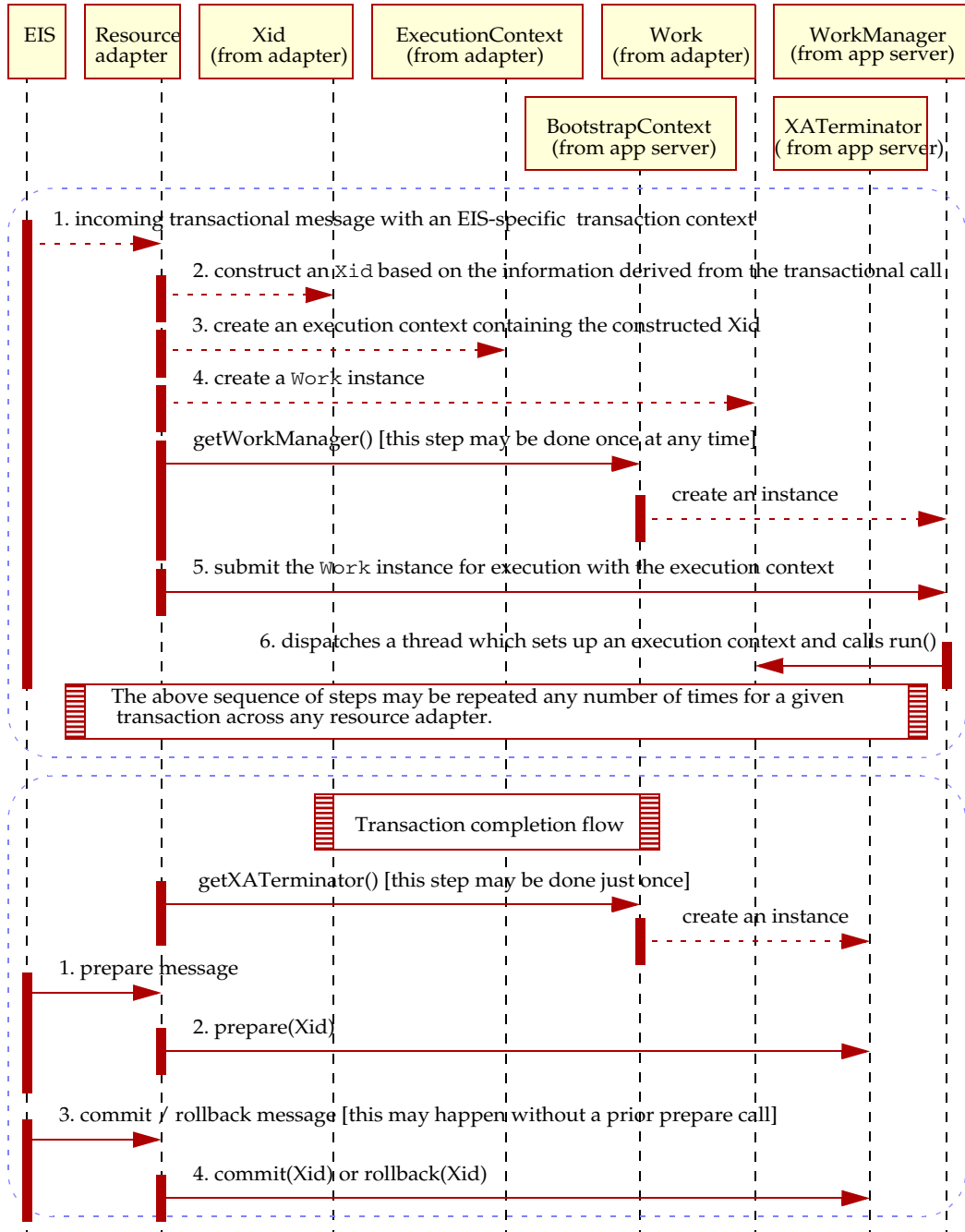
15.4.3   Crash Recovery Processing

- If the EIS detects the failure of the application server while the transaction is active (that is, transaction completion has not begun), it does not wait for the application server to recover or do any recovery processing. The transactional work done by the application server site is presumed aborted.
- If the EIS detects the failure of the application server while the transaction is in-doubt (that is, the transaction has already been prepared), the EIS retries completion by attempting to re-establish network communication until it succeeds. When the application server recovers from the crash, it should recover the state of all transactions that were successfully prepared before the crash, and complete them upon receiving a `commit` method or `rollback` method call from the EIS through the resource adapter.
- If the resource adapter detects the failure of the EIS while the transaction is active (that is, transaction completion has not begun), it aborts all active transactions that originated from the EIS. The resource adapter should keep a list of active transactions and abort them upon EIS failure.
- If the resource adapter detects the failure of the EIS while the transaction is in-doubt (that is, the transaction has already been prepared), it waits for the EIS to recover. When the EIS recovers, it re-establishes network communication with the resource adapter, and queries it for a list of in-doubt transactions. It then completes the in-doubt transactions.

FIGURE 15-4 Crash Recovery Flows When Application Server Crashes (Sequence Diagram)

| EIS | Resource adapter | Xid (from adapter) | BootstrapContext (from app server) | XATerminator (from app server) |

Application server crashes when a transaction is in an active state

When the EIS detects the failure of the application server site while the transaction is active (that is, transaction completion has not begun), it does not wait for the application server to recover or do any recovery processing. The transactional work done by the application server site is presumed aborted.

Application server crashes when a transaction is in an in-doubt state

When the EIS detects the failure of the application server site while the transaction is in-doubt (that is, the transaction has already been prepared), the EIS retries completion by attempting to re-establish the network communication until it succeeds. When the application server recovers from the crash, it should recover the state of all transactions that were successfully prepared before the crash, and complete them upon receiving a commit(Xid) or rollback(Xid) from the EIS via the resource adapter.

1. commit or rollback message [EIS retries completion]

2. getXATerminator()

3. create an instance

4. construct an Xid

5. commit(Xid) or rollback(Xid)

FIGURE 15-5 Crash Recovery Flows When EIS Crashes (Sequence Diagram)

| EIS | Resource adapter | Xid (from adapter) | XATerminator (from app server) |

EIS crashes when a transaction is in an active state

When the resource adapter detects the failure of the EIS site while the transaction is active (that is, transaction completion has not begun), it aborts all active transactions which originated from the EIS. The resource adapter must keep a list of active transactions and abort them upon EIS failure.

1. create an Xid for each active transaction

2. rollback(Xid) for each active transaction

EIS crashes when a transaction is in an in-doubt state

When the resource adapter detects the failure of the EIS site while the transaction is in-doubt (that is, the transaction has already been prepared), it waits for the EIS to recover. When the EIS recovers, it re-establishes network communication with the resource adapter, and queries it for a list of in-doubt transactions. It then completes the in-doubt transactions.

1. EIS recovers and queries for a list of in-doubt transactions

2. recover() [returns a list of in-doubt transactions]

4. send commit or rollback to the in-doubt transactions

5. construct an Xid for each in-doubt transaction

6. commit(Xid) or rollback(Xid) the in-doubt transaction

15.4.4   Requirements

■ An application server must implement the transaction inflow contract. That is, it must allow `Work` submissions with a transaction context, an `Xid`, and provide a valid `XATerminator` instance when called through the `getXATerminator` method of the `BootstrapContext` instance.

■ A resource adapter may optionally choose to use the transaction inflow contract. But, a resource adapter that uses the transaction inflow contract to import an EIS transaction and do transactional work must implement the prescribed transaction inflow contract.

■ The `XATerminator` instance provided by the application server must be thread-safe and re-entrant. The resource adapter may use an `XATerminator` instance across different transactions concurrently.

■ When the application server is unable to recreate the transaction context, if any, specified for a `Work` instance, it must throw a `WorkCompletedException` set to the error code `WorkException.TX_RECREATE_FAILED`.

■ For a particular imported transaction, at any given time, there must be at most one `Work` instance associated with the transaction. The associated `Work` instance may be in any state, that is, waiting for execution to begin or already executing. However, it must be possible for several `Work` instances to do work on a transaction as long as there is at most one `Work` instance associated with the transaction at any time. It must also be possible for different resource adapters to participate in the same transaction.

The application server must disallow `Work` submissions with a `WorkCompletedException` set to the error code `WorkException.TX_CONCURRENT_WORK_DISALLOWED`, if there is already a `Work` instance associated with the transaction, irrespective of which resource adapter is involved in the `Work` submission. This must be done using the `getGlobalTransactionId` method of the `Xid` object present in the execution context of the submitted `Work` instance. The `Xid`'s branch identifier must be ignored.

The application server must not try to serialize `Work` processing based on transaction information.

■ The application server must reject `Work` submissions for a transaction whose completion is in-progress, with a `WorkCompletedException` set to the error code `WorkException.TX_CONCURRENT_WORK_DISALLOWED`.

■ The application server must reject transaction completion or crash recovery calls for a specific transaction with a `javax.transaction.xa.XAException`, when a `Work` instance associated with the transaction is present. The application server must not block or serialize transaction completion or crash recovery calls waiting for a `Work` instance associated with the transaction to complete.

■ The application server must reject multiple transaction completion or crash recovery calls for the same transaction with a `javax.transaction.xa.XAException`.

- The application server must reject transaction completion or crash recovery calls with a `javax.transaction.xa.XAException` upon any errors.
- The application server should recover the state of all in-doubt transactions upon failure recovery.

15.4.5 Non-Requirements

- The application server is not responsible for ensuring transaction IDs of the imported transactions from different EISs are unique. Each EIS is expected to use unique transaction IDs.
- It is possible that a rogue resource adapter or EIS may provide non-unique `Xids`, or attempt to complete transactions that it does not own. The application server is not required to detect the above cases. It is also not required to detect transactional, transaction completion, or crash recovery calls from a rogue EIS.

15.4.6 Recommendations

The resource adapter should keep a list of active transactions and abort them upon detecting EIS failure.

15.4.7 Transaction Inflow in a Non-Managed Environment

Though the transaction inflow contract is primarily intended for a managed environment, it may be used in a non-managed environment provided the application that bootstraps a resource adapter instance is capable of functioning as a resource manager.

In a non-managed environment, support for the transaction inflow contract is not required. That is, the `getXATerminator` method of the `BootstrapContext` instance may return a null instance.

CHAPTER **16**

Security Inflow

This chapter specifies a standard, generic security contract between the EIS/resource adapter and the application server that enables a resource adapter to establish security information while submiting a `Work` instance for execution to a `WorkManager` and while delivering messages to message endpoints residing in the application server.

16.1 Overview

It is critical, in EIS integration scenarios, that all interactions between an application server and resource adapter are secure and unauthorized access to application components deployed in an application server be prevented. The security architecture for integration of EISs into the application server is detailed in Chapter 8, "Security Architecture" and the security contract in related to connection establishment with the EIS is discussed in this chapter

Resource adapters, typically employ transport and message level security for connecting to, and receiving messages from an EIS. To achieve end-to-end application security, it is important that all activities that a `Work` instance performs, including delivering messages to a `MessageEndpoint` happens in the context of an established identity.

This chapter references the following chapters and documents:

- Security Architecture specified in Chapter 8, "Security Architecture"
- Security scenarios based on the connector architecture (Refer to Appendix D "Security Scenarios" on page C-1)
- "Java Authentication Service Provider Interface for Containers Specification, version 1.4" on page E-2 (JSR-196)

FIGURE 16-1 Security Inflow Contract
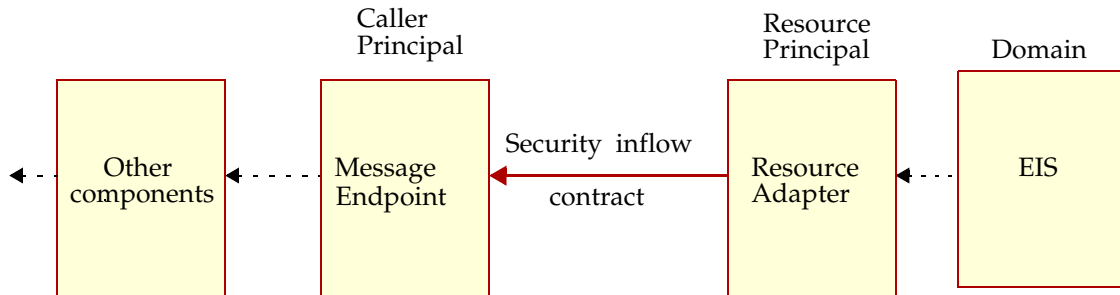
16.2 Goals

The Security Inflow contract is designed to meet the following goals:

- Enable an end-to-end security model for Java EE applications, to support integration with EIS based on the Connector architecture
- Support the execution of a `Work` instance in the context of an established identity
- Support the propagation of user information/`Principal` information from an EIS to a `MessageEndpoint` during Message Inflow
- Ensure that the security inflow contract is transparent to an application component provider
- Enable a `WorkManager` to make authorization checks based on the security context information that is provided with the submitted `Work` instance
- Enable an application component container to authorize and control access during Message Inflow to message endpoints residing in the application component container
- Allow `MessageEndpoints` to be portable across multiple EISs that use different security mechanisms
- Map security identities in foreign domains into corresponding identities in the receiving container or context
- Ensure backward compatibility with the existing "Security Management contract", defined in Chapter-17 of "Enterprise JavaBeans (EJB) Specification, version 3.2" on page E-1 Core Contracts and Requirements specification, for security context inflow to `MessageEndpoints` realized as message-driven beans

16.3   Security Inflow Model

This chapter uses the terminology described in Section 8.3, "Terminology" on page 8-2 and does not dictate any specific security policy or technology. The resource adapter performs EIS sign-on and secure association with the EIS in an EIS specific manner. No specific scheme or configuration to perform sign-on and establish such secure associations is mandated by this security inflow model. For more details on EIS sign-on, see Section 8.5.1, "Authentication Mechanism" on page 8-6 and for details on secure association with the EIS, see Section 8.5.4, "Secure Association" on page 8-8.

The security inflow contract between the resource adapter and the application server leverages the Generic Work Context mechanism (described in Chapter 11, "Generic Work Context") by describing a standard `WorkContext`, `SecurityContext` that may be provided by the resource adapter while submitting a `Work` for execution.

Submitting a `Work` instance without specifying the security contextual information, in which the Work has to be executed in, has the following drawbacks related to application security:

- When a `Work` instance is submitted by a resource adapter to a `WorkManager` for execution:
  - The `Work` instance is always executed in an unknown security context, or a default security context set by the application server's `WorkManager`
  - In the absence of inflow of security identities during `Work` submission, the container cannot ensure the task has been granted fine-grained permissions/access-control based on authenticated user identities but is limited to making access decisions based on code-based identity information.
- When a message is delivered asynchronously to `MessageEndpoints` residing in the application server, the resource adapter cannot establish the security identity of the caller of the `MessageEndpoint`, that is, the value returned when `EJBContext.getCallerPrincipal()` is called is unknown. (when the `MessageEndpoint` is realized as a message-driven bean).

The `SecurityContext` provides a portable mechanism for the resource adapter to pass security context information to the application server. This work context enables an EIS/resource adapter to flow-in security context information while submitting a `Work` to a `WorkManager` for execution. All activities that happen as part of the `Work` instance, including message deliveries to `MessageEndpoints` then occurs in the context of an established identity. This avoids the drawbacks listed above and extends the end-to-end security model for Java EE applications to include the `Work` execution and Message Inflow aspects.

A resource adapter submits a `Work` instance, that implements `WorkContextProvider` to a `WorkManager`. The resource adapter includes a concrete implementation of `SecurityContext` as one of the work contexts it requires to be established as the execution context of the `Work` instance.

When one of the free threads from the application server's thread pool picks up the `Work` instance for execution, as described in the Generic Work Context Model (see Section 11.3, "Generic Work Context Model" on page 11-2), the application server establishes the security context information described in the `SecurityContext`, before executing the `Work` instance.

When a message is delivered to a `MessageEndpoint` instance by the resource adapter in the context of a `Work` instance, the security context that is set up for that message delivery is inherited from the security context set in the `Work` instance. In other words, as in Transaction Inflow (see Chapter 15, "Transaction Inflow"), all message deliveries that are delivered to endpoints within a single `Work` instance are processed under the same security identity.

FIGURE 16-2 Security Inflow Conmtext (Object Diagram)
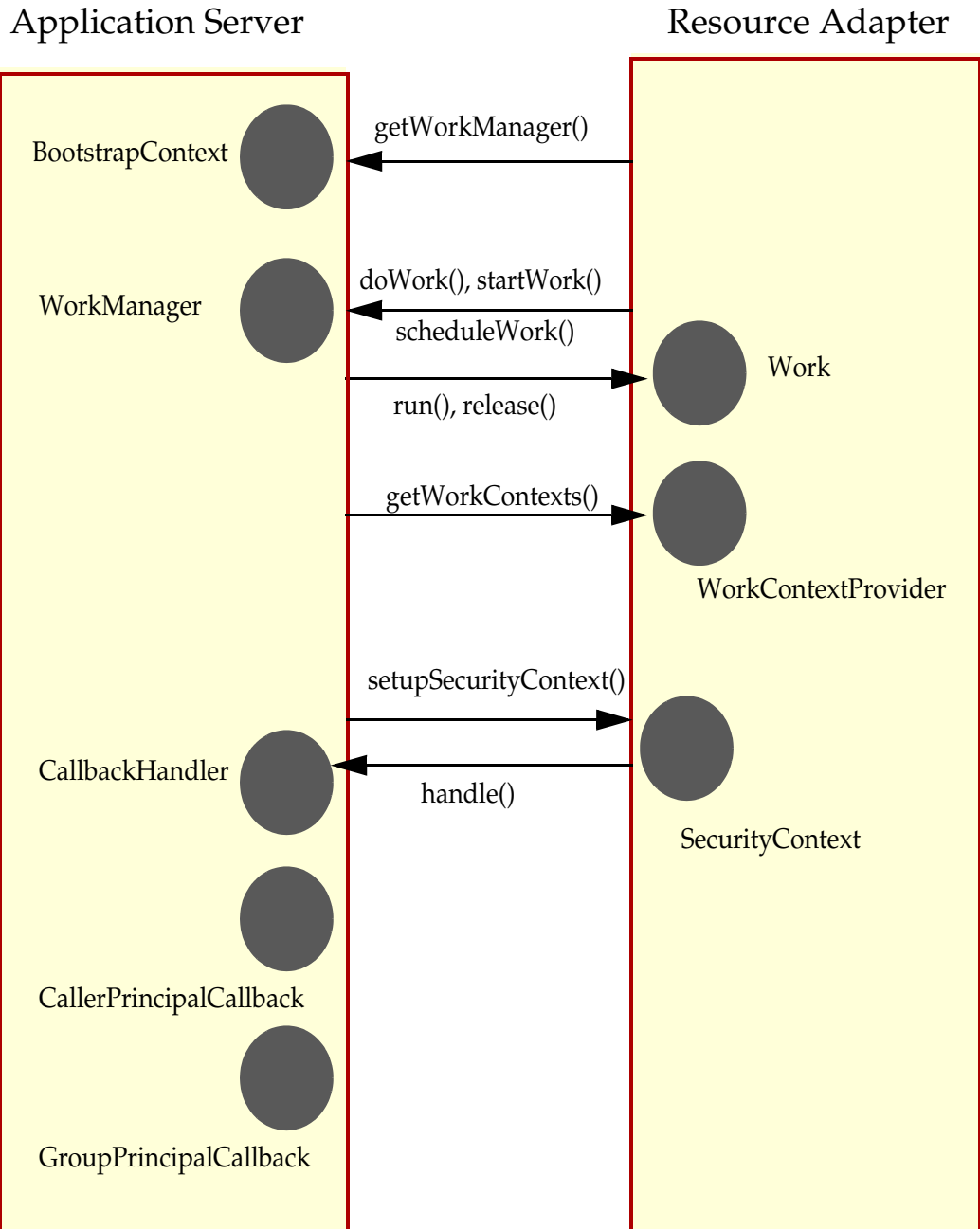
CODE EXAMPLE 16-1 SecurityContext

```
package javax.resource.spi.work;

import javax.security.auth.Subject;
import javax.security.auth.callback.CallbackHandler;

public abstract class SecurityContext
                    implements WorkContext {

    public String getDescription() {
        return "Security Context";
    }

    public String getName() {
        return "SecurityContext";
    }

    public abstract void setupSecurityContext(
                            CallbackHandler handler,
                            Subject executionSubject,
                            Subject serviceSubject);
}
```

When a resource adapter flows-in an identity to be used by the application server,
the identity may or may not belong to the EIS security domain and the caller
principal to be established for a message-driven bean (or a `MessageEndpoint`) is
required to be an identity of the application server's security domain. Therefore the
EIS integration scenario has two choices related to establishing the Caller identity:

■ **Case 1: Resource adapter flows-in an identity in the application server's
  security policy domain.** In this case, the application server may just use the
  initiating principal, flown-in from the resource adapter, as the caller principal in
  the security context the `Work` instance executes as.
■ **Case 2: Resource adapter flows-in an identity belonging to the EIS' security
  domain**. The resource adapter establishes a connection to the EIS and requires to
  execute a `Work` instance in the context of an EIS identity. In this case, the
  initiating or caller principal does not exist in the application server's security
  domain. A translation from one domain to the other is required to be performed.

For more information on these two cases, see Section 16.4.3 "Case 1: Identity in the
Container Security Domain" and Section 16.4.4 "Case 2: Identity Translated Between
Security Domains".

The `WorkManager` may enforce any security policies, as per its configuration, while establishing the security context for the `Work` instance. The application server may also enforce additional security policies, based on the configuration of the `MessageEndpoints`, during message delivery. See Section 13.5, "Message Delivery" on page 13-22.

16.4   `SecurityContext` Class

The `SecurityContext` is one of the standard `WorkContext`s defined in this specification. It enables a `Work` instance to propagate security related context information from an EIS to the application server. It is modelled as an abstract class that implements the `WorkContext` interface.

The resource adapter provides a custom concrete implementation of the `SecurityContext` abstract class and provides an implementation for the method `setupSecurityContext` to set up security context for the `Work` instance being submitted. A resource adapter indicates to the `WorkManager`, that a `Work` instance is required to be run in a specified security execution context by submitting a `Work` instance that implements `WorkContextProvider` interface and ensuring that the `List` of `WorkContext`s for that `Work` instance contains an instance of `SecurityContext`.

FIGURE 16-3 Security Context Establishment During Work Submission(Sequence Diagram)
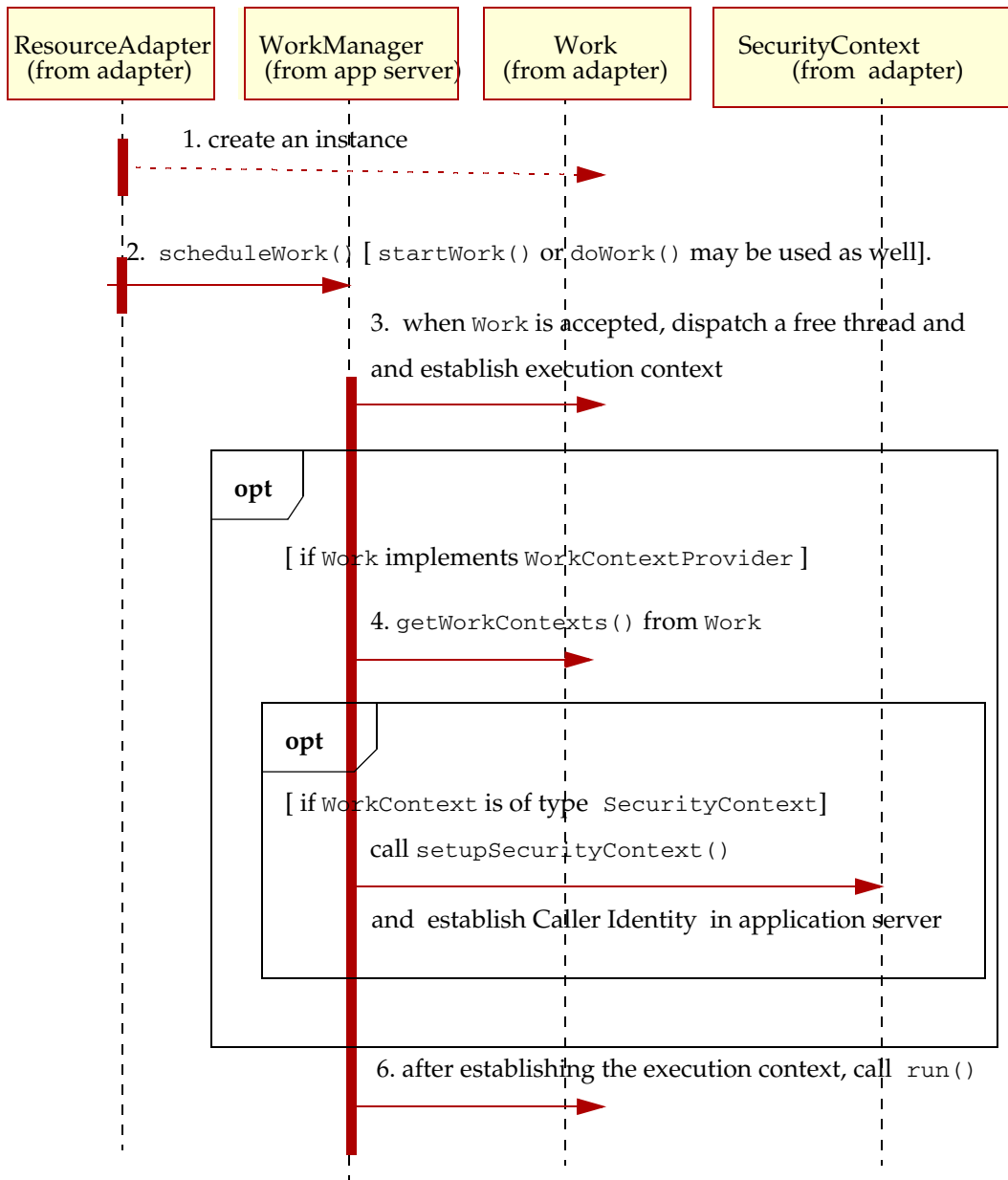
16.4.1 Establishing the Security Context

While setting the execution context of a `Work` instance, the `WorkManager` establishes the security context for the `Work` instance when it encounters a `WorkContext` instance implementing `SecurityContext`.

For setting the security context of a `Work` instance, the application server calls the `setupSecurityContext` method of the `SecurityContext` implementation provided by the resource adapter. The following conditions are applicable to the application server provider while calling the `setupSecurityContext` method:

- The `handler` argument must not be null, and the `CallbackHandler` implementation passed as the argument `handler` to `setupSecurityContext` must support the following `Callback`s defined in "Java Authentication Service Provider Interface for Containers Specification, version 1.4" on page E-2:

  - `CallerPrincipalCallback`
  - `GroupPrincipalCallback`
  - `PasswordValidationCallback`

  The following `Callback`s are recommended to be supported by the `CallbackHandler` implementation:

  - `CertStoreCallback`
  - `PrivateKeyCallback`
  - `SecretKeyCallback`
  - `TrustStoreCallback`
- The `executionSubject` argument must be non-null and it must not be read-only. It is expected that the resource adapter `Work` implementation will populate this `executionSubject` with `Principal` and credentials that would be flown into the application server.
- The `serviceSubject` argument may be null. If it is not null, it must not be read-only. It represents the application server's credentials and it may be used by the `Work` implementation to retrieve Principals and credentials necessary to establish a connection to the EIS (in the cause of mutual-auth like scenarios). The `serviceSubject` may contain the credentials of the application server or the `SecurityContext` implementation may collect the service credentials, as necessary, by using the `CallbackHandler` passed to it.

When the `setupSecurityContext` method is called by the application server container, the resource adapter may perform the following steps to establish caller identity information for a `Work` instance:

1. Identify the security context that is required to be flown-in to the application server to serve as the execution context of the `Work` instance.

2. Populate the `executionSubject` with the EIS `Principal` and Credentials that should serve as the security context for the `Work` instance to be executed in.

3. Add instances of the necessary `Callbacks` (Section 16.4.2 "Callbacks for Information from the Application Server" describes when a particular `Callback` is required to be employed by the resource adapter), usually a subset of the ones listed above, to an array and invokes the `handle()` method in the container's `CallbackHandler` implementation by passing the array of `Callback` instances.

4. On sucessful return from the `CallbackHandler.handle()` method the `setupSecurityContext()` returns.

On successful return of `setupSecurityContext`, the container must use the "modified" `executionSubject` (modified as a result of handling the various `Callbacks`) to establish the caller identity of the `Work` instance

On successful return from `setupSecurityContext`, the `WorkManager` must ensure that the `Work` is set up to be executed with the established security identity. Any subsequent `MessageEndpoint` deliveries in that `Work` instance (to message-driven beans for instance) should have the security context established appropriately. When message-driven beans are the `MessageEndpoints`, `MessageDrivenContext.getCallerPrinicipal()` must return the principal corresponding to the established security identity, and `MessageDrivenContext.isCallerInRole()` must return the result of testing the established security identity for role membership.

As detailed in Section 11.4, "WorkContextProvider and WorkContext Interface" on page 11-8, a Connector `WorkManager` must support nested `Work` submissions. One or more `Work` instances in such a nested `Work` submission may include a `SecurityContext`. The Connector `WorkManager` must restrict the security context, established by way of the `SecurityContext` of a `Work` instance, to that `Work` instance alone. When a nested `Work` instance is submitted without a `SecurityContext`, the Connector `WorkManager` must not inherit the Security Context information of the parent `Work` instance. It must establish the equivalent of an unauthenticated caller principal for the nested `Work` instance.

FIGURE 16-4 Establish the Security Context (Sequence Diagram)
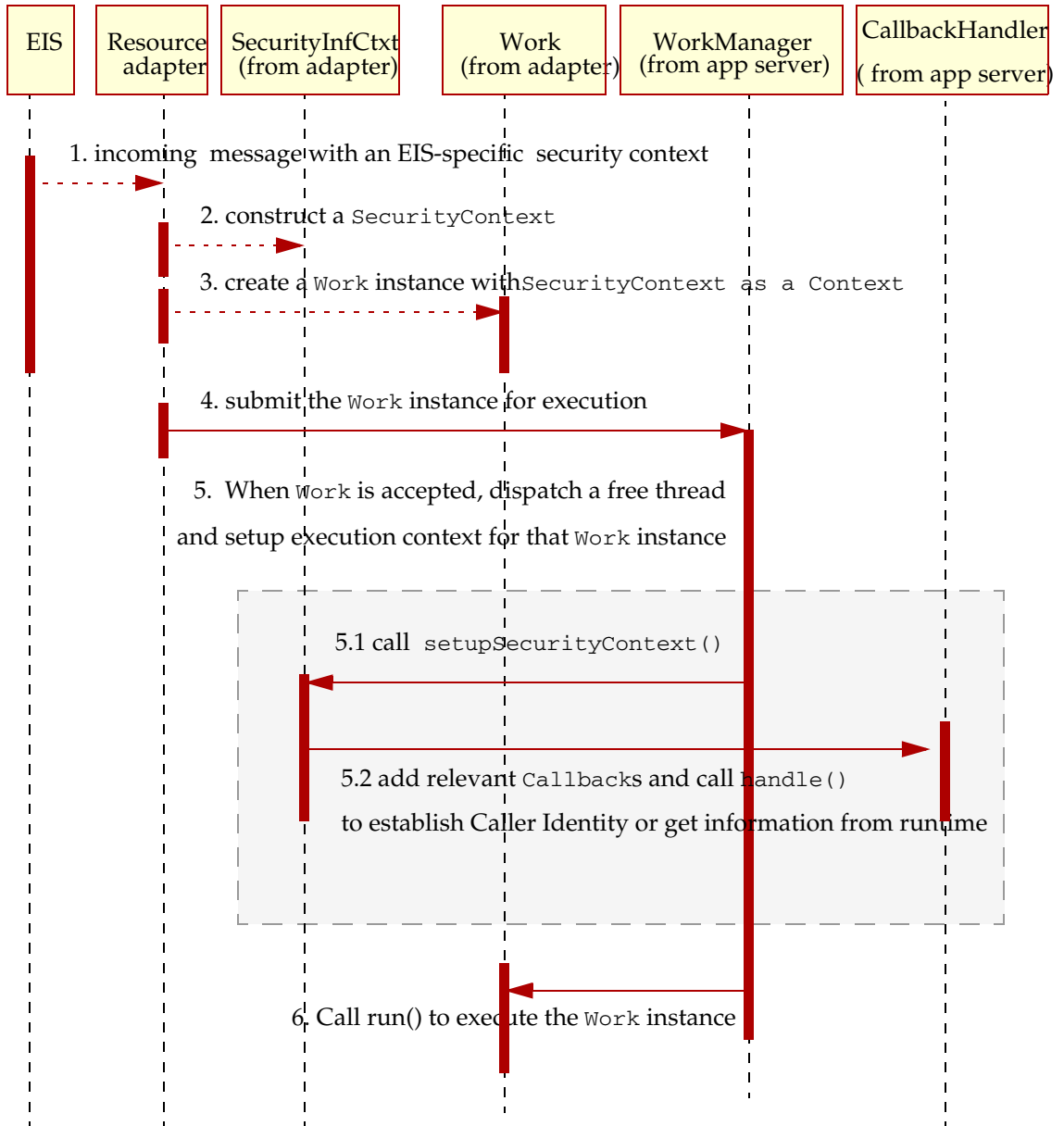
16.4.2 Callbacks for Information from the Application Server

As part of step 3 described in the section above, the following `Callbacks` may be employed by a resource adapter. The descriptions of the `Callbacks` below have been taken from the "Java Authentication Service Provider Interface for Containers Specification, version 1.4" on page E-2 specification. For detailed information, refer to the "Java Authentication Service Provider Interface for Containers Specification, version 1.4" on page E-2 and the Java API documentation of the `Callbacks` defined in the `javax.security.auth.message.callback` package of that specification:

- A resource adapter may use the `CallerPrincipalCallback` to set the container's representation of the caller principal. The `CallbackHandler` must establish the caller principal associated with the invocation being processed by the container. When the argument `Principal` is null, the handler will establish the container's representation of the unauthenticated caller principal.
- A resource adapter might use the `GroupPrincipalCallback` to establish the container's representation of the corresponding group principals within the Subject. When a null value is passed to the groups argument, the handler will establish the container's representation of no group principals within the Subject. Otherwise, the handler's processing of this callback is additive, yielding the union (without duplicates) of the principals existing within the `Subject`, and those created with the names occuring within the argument array. The `CallbackHandler` will define the type of the created principals.
- A resource adapter might use the `PasswordValidationCallback` to employ the password validation facilities of its containing runtime. Since a resource adapter employing the `PasswordValidationCallback` makes an assumption of access to the password validation facilities (and thereby access to the security domain), it can be deployed in Case #1 scenarios only
  (For more information on Case #1 scenario, see Section 16.4.3 "Case 1: Identity in the Container Security Domain"). The resource adapter must pass this information to th e deployer through an out-of-band mechanism.

16.4.3 Case 1: Identity in the Container Security Domain

As explained in Section 16.3 "Security Inflow Model", the EIS integration scenario may result in the resource adapter reusing the application server security policy domain. In such cases, when the resource adapter flows-in an identity through the security context inflow model described in Section 16.4.1 "Establishing the Security Context", the identity belongs to the application server's security domain already. Therefore, the application server may use the `Principals` used in `CallerPrincipalCallback` and `GroupPrincipalCallback` without any translation to the application server security policy domain.

FIGURE 16-5 Case 1: Identity in Container's Security Domain (Sequence Diagram)
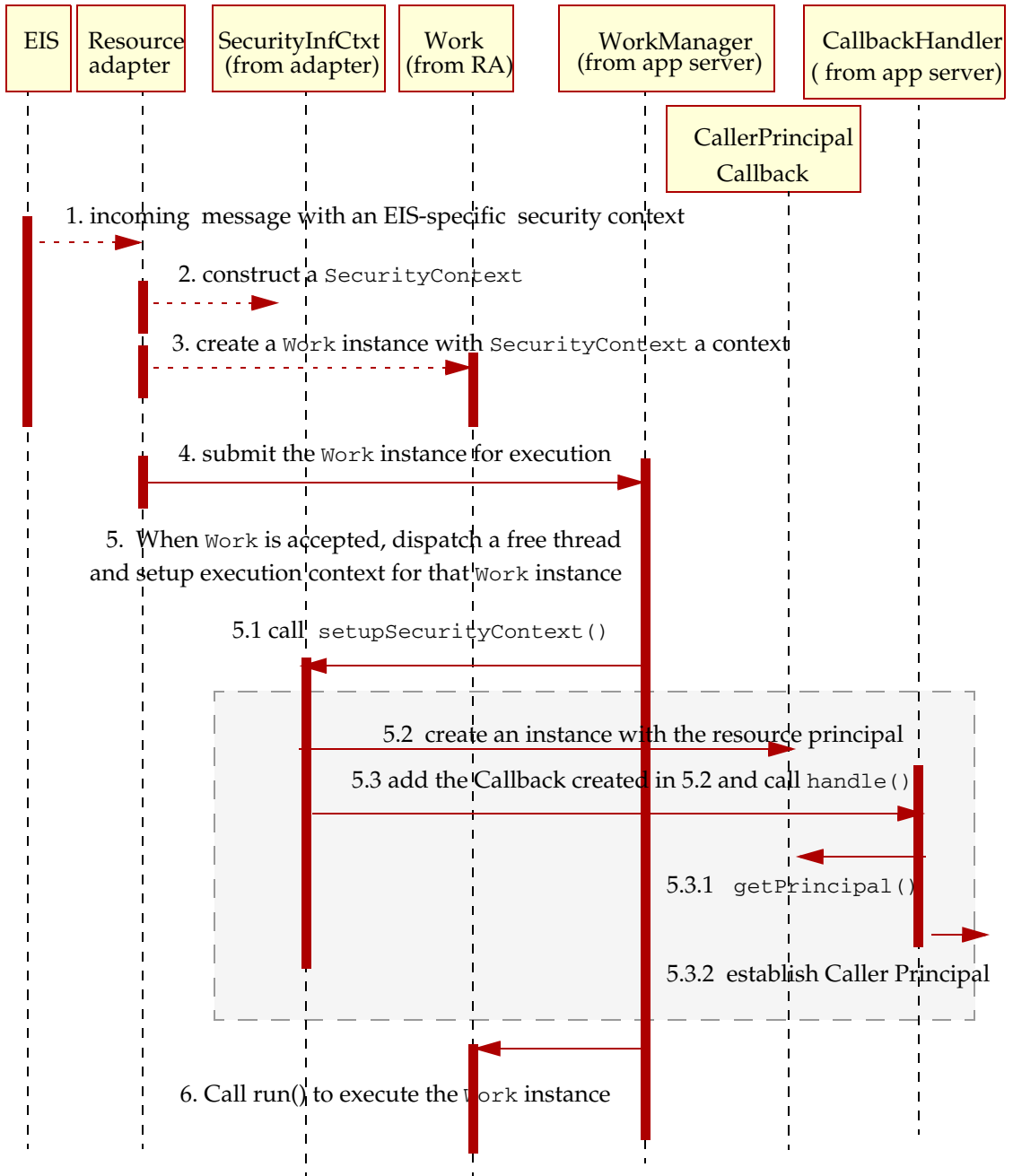
16.4.4 Case 2: Identity Translated Between Security Domains

When the resource adapter, on the other hand, connects to an EIS that uses a different security policy domain, it requires that the `Work` instance be executed in the context of the container identity mapped from the EIS identity. To handle such a case, these `Principals` and groups, available in the resource adapter, would need to be mapped to `Principals` and groups as relevant in the `MessageEndpoint` container's security policy domain.

During the inflow of the EIS `Subject`, the mapping of one or more principals on the path may be required before delivering to the `MessageEndpoint`/message-driven bean. These translations from the identity of initiating/caller resource principal to an application server principal could be one of the following types (see Section 8.5.2, "Resource Principal" on page 8-6)

- Configured Identity
- Principal Mapping
- Caller Impersonation
- Credentials Mapping

For example, in the case of Principal Mapping, an employee may be identified by a userid and password (basic authentication) in an EIS. The resource principal may need to be mapped to a Kerberos principal, that is relevant in the application server security domain, before delivering the method invocation to the message-driven bean. In the case of `MessageEndpoints` realized as message-driven beans `MessageDrivenContext.getCallerPrincipal` method then, returns the principal that is the result of the mapping and not the original EIS principal. In this example, `getCallerPrincipal` would return the Kerberos principal.

The management of the security infrastructure, to enable principal mapping or other schemes listed above, is performed by the System Administrator role and the mechanism through which a container enables this mapping is beyond the scope of the Connector specification.

The application server must provide tools to set up Caller Identity information for a Work/Message Endpoint container. This includes support for mapping of EIS/resource principals to Caller Principals in the application server security domain.

To handle Principal Mapping scenarios described above, the application server must provide a `CallbackHandler` that can be configured to perform Principal Mapping during its handling of the `CallerPrincipalCallback` and

`GroupPrincipalCallbacks`. This specification does not define interfaces for Principal Mapping service and `CallbackHandler` configuration. The deployer must use application server specific tools and techniques to enable this mapping.

FIGURE 16-6 Case 2: Identity Translated Between Security Domains (Sequence Diagram)
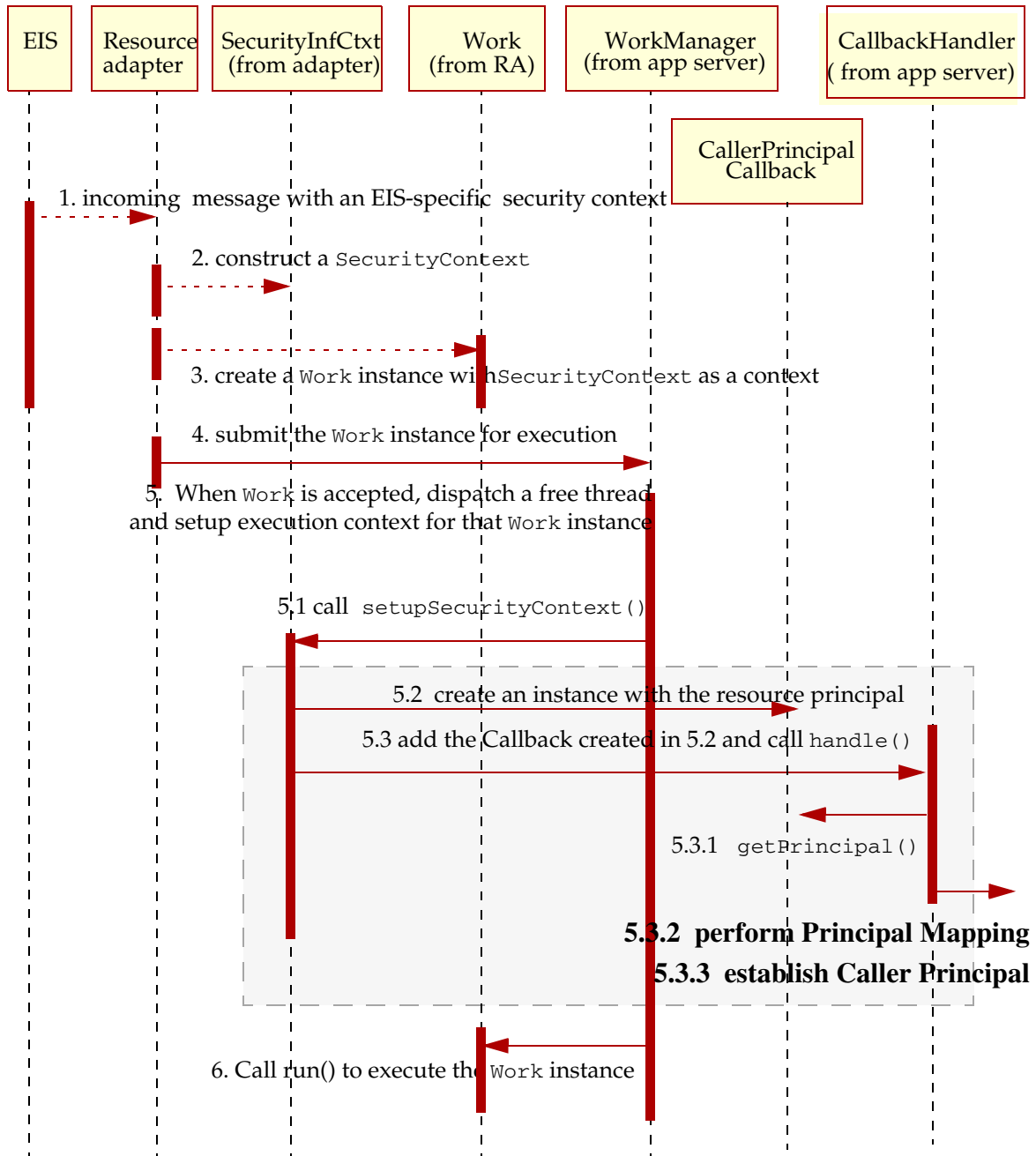
16.4.5   Establising a `Principal` as the Caller Identity

Prior to returning to the container, `setupSecurityContext` must use the container provided `CallbackHandler` to handle a `CallerPrincipalCallback`, unless either of the following conditions are met at the time the method returns to the container:

- **Case A.** The resource adapter intends to establish an authenticated caller identity, and the principal `Set` of the `executionSubject` contains exactly the one `Principal` that would otherwise have been used to construct the `CallerPrincipalCallback`
- **Case B.** The resource adapter intends to establish the unauthenticated caller identity, and the principal `Set` of the `executionSubject` is empty.

The resource adapter, in the two cases above, is not required to use the `Callbacks` listed in Section 16.4.2 "Callbacks for Information from the Application Server".

16.4.5.1   Case A: Establishing a Single `Principal` as the Caller Identity

The resource adapter can add the one `Principal` it requires to be set as the Caller identity in the `Principal Set` of the `executionSubject`. See FIGURE 16-7 for a sequence diagram depicting this case. Note that the resource adapter must be configured to have the necessary security permissions to add a Principal to the executionSubject.

On return from `setupSecurityContext`, the container must determine whether or not it handled the `CallerPrincipalCallback`. If it determines that it did not handle the `Callback`, the container must transform the contents of the `executionSubject` and of any related authentication state to be equivalent to that which would have resulted had it handled the `Callback` on behalf of the resource adapter. This transformation also includes all security identity translation requirements detailed in Section 16.4.4 "Case 2: Identity Translated Between Security Domains".

16.4.5.2   Case B: Establishing an Unauthenticated Security Context

If a resource adapter requires to establish an unauthenticated security context (which may or may not have an associated `Principal`) for the `Work` instance, the resource adapter may perform either of the following operations when `setupSecurityContext` is called:

- It may use the `CallbackHandler` to handle a `CallerPrincipalCallback` with a null `Principal` or name
- Or, if it uses the simplification described in Section 16.4.5 "Establishing a Principal as the Caller Identity", it may return an empty `executionSubject`.

The `WorkManager` must detect that the handler was not called and establish the container's representation of the unauthenticated identity for that `Work` instance.

FIGURE 16-7 Case A: Establishing a Single Principal as the Caller Identity (Seq. Diagram)
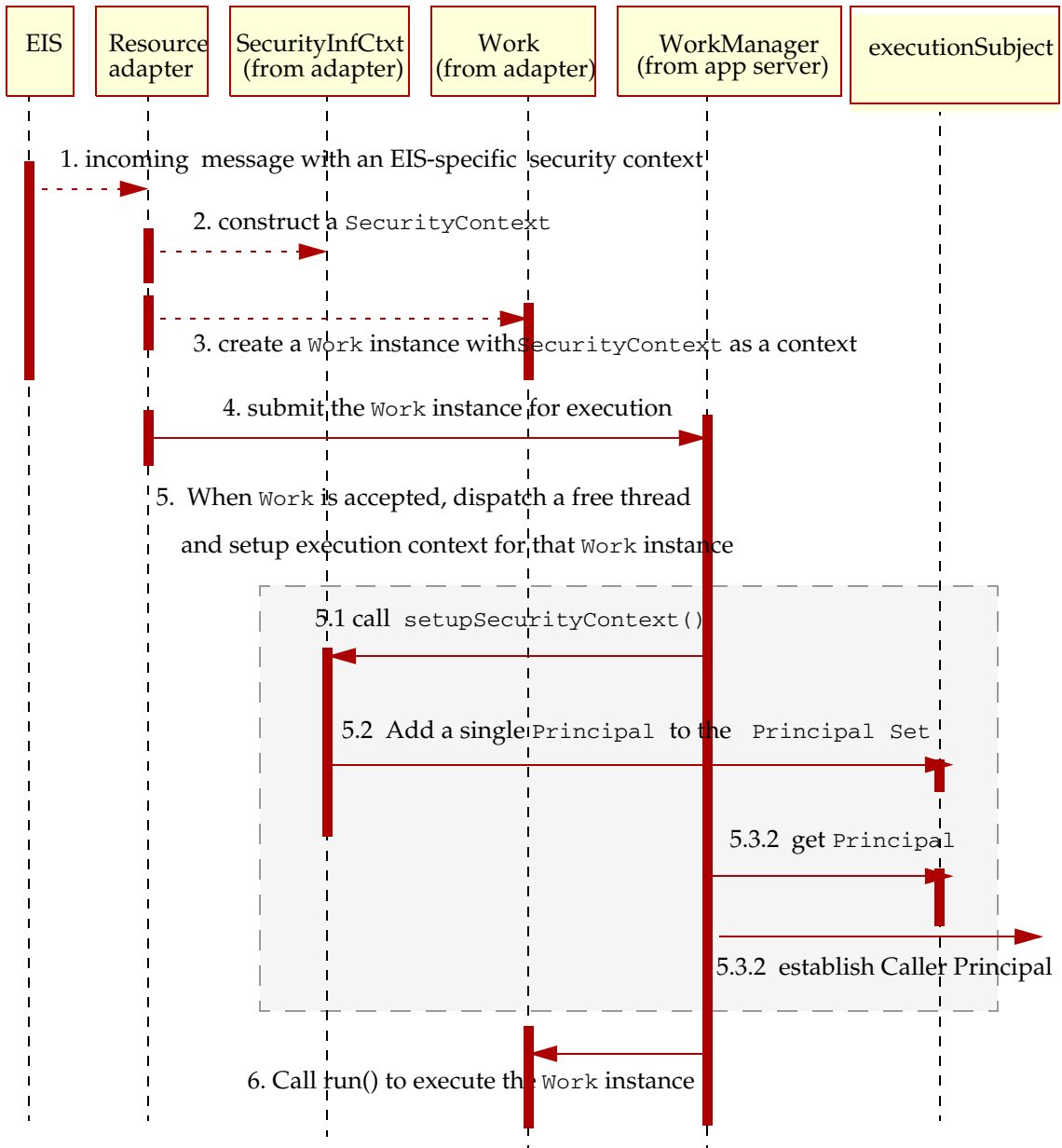
16.4.6 Security Configuration Responsibilities

The system administrator, deployer and application component (`MessageEndpoint` provider) have particular responsiblities in the assignment of security roles, security domain and realm assignment.

When `MessageEndpoints` are realized as message-driven beans, the EJB Core Contracts and Requirements of "Enterprise JavaBeans (EJB) Specification, version 3.2" on page E-1, states the following responsibilities:

- Deployer: (section 17.4.2) : The Deployer assigns principals and/or groups of principals (such as individual users or user groups) used for managing security in the operational environment to the security roles defined by means of the `DeclareRoles` and `RolesAllowed` metadata annotations and/or `security-role` elements of the deployment descriptor. ... the process of assigning the logical security roles defined in the application's deployment descriptor to the operational environment's security concepts is specific to that operational environment. Typically, the deployment process consists of assigning to each security role one or more user groups (or individual users) defined in the operational environment.
- EJB container provider (Section 17.6.7): Principal Mapping If the application requires that its clients are deployed in a different security domain, or if multiple applications deployed across multiple security domains need to interoperate, the EJB Container Provider is responsible for the mechanism and tools that allow mapping of principals. The tools are used by the System Administrator to configure the security for the application?s environment.
- System Administrator (Section 17.7.2): Principal Mapping : If the client is in a different security domain than the target enterprise bean, the System Administrator is responsible for mapping the principals used by the client to the principals defined for the enterprise bean. The result of the mapping is available to the Deployer. The specification of principal mapping techniques is beyond the scope of the EJB architecture.

16.4.7 Requirements

- The application server must support the `SecurityContext` interface. It must also satisfy all the requirements stated in Section 16.4.1 "Establishing the Security Context"
- The application server must support resource adapters that employ Case 1 or 2 style integration mode. Cases 1 and 2 are detailed in Section 16.4.3 "Case 1: Identity in the Container Security Domain" and Section 16.4.4 "Case 2: Identity Translated Between Security Domains" respectively.
- The application server must provide configuration tools to establish Caller Identity information for a `MessageEndpoint` or `Work` instance as stated in Section Section 16.4.4 "Case 2: Identity Translated Between Security Domains". In

other words, the container must provide support for configuring principal mapping. The application server must also support the simplifications detailed in Section 16.4.5 "Establising a Principal as the Caller Identity".

- The application server must support the security role assignments relevant to the MessageEndpoint implementation as stated in Section 16.4.6 "Security Configuration Responsibilities"

16.5 Illustrative Example

16.5.1 Case 1: Identity in the Container Security Domain

The Case #1 scenario enables resource adapters that work closely with the application server and could authenticate the credentials with the application server's security domain directly. For example, consider an EIS that is tightly plugged in with the application server or container, like say, for illustration purposes, an XMPP (Extensible Messaging and Presence Protocol) server.

In this case the XMPP resource adapter, could leverage the application server's security domain directly for managing and authenticating users instead of having its own security domain. In this scenario, the XMPP resource adapter requires the delivery of an XMPP "exchange message" that was sent by a user *JoeUser* (*JoeUser* was authenticated in the AS security domain by the XMPP resource adapter, through out-of-band implementation-specific schemes) to a `MessageEndpoint`. Since the security identity is in the application server's security domain, there isn't a need to translate the known identity to an identity in the application server's security domain. The XMPP resource adapter already has the user name *JoeUser* and the necessary authentication data and could use *JoeUser*/authentication data to establish the security context of the `Work` instance.

In order to support the propagation of user information/`Principal` information from the EIS (XMPP server in this case) to a `MessageEndpoint` during message inflow, the XMPP resource adapter uses a `Work` instance to deliver the XMPP exchange message and provides a `SecurityContext` as one of the `WorkContexts` for the `Work` instance.

After the container successfully processes the security work context information, the application server will ensure that the `Work` is set up to be executed with the established security identity. All subsequent `MessageEndpoint` deliveries in that `Work` instance (to message-driven beans for instance) will have the security context established appropriately. When message-driven beans are the `MessageEndpoints`, `MessageDrivenContext.getCallerPrinicipal()` must return the principal

corresponding to the established security identity, and
`MessageDrivenContext.isCallerInRole()` must return the result of testing the
established security identity for role membership.

```
public class XMPPResourceAdapterImpl implements ResourceAdapter {
...
    public void start(BootstrapContext ctx) {
        bootstrapCtx = ctx;
    }

    ...
    {
        WorkManager workManager =
            myRA.bootstrapCtx.getWorkManager();
        workManager.scheduleWork(new XMPPMessageDeliveryWork());
        ...
    }
}


public class XMPPMessageDeliveryWork
            implements Work, WorkContextProvider {

    void release(){ ..}

    List<WorkContext> getWorkContexts() {
        List<WorkContext> l = new ArrayList<WorkContext>();
        SecurityContext scIn = new XMPPSecurityContext();
        l.add(scIn);
        return l;
    }

    void run(){
        // deliver "exchange message" from the user
        // to MessageEndpoint;
    }
}


public class XMPPSecurityContext extends SecurityContext {

    @Override
    public void setupSecurityContext(CallbackHandler handler,
            Subject executionSubject, Subject serviceSubject) {
        // Get username, password from client's response
        // to XMPP register message
        // Note: PasswordValidationCallback usage is required
        // only if the RA requires authentication.
```

```
        PasswordValidationCallback pwdCallback =
        new PasswordValidationCallback(
            executionSubject, username, pwd);
    CallerPrincipalCallback cpCallback =
        new CallerPrincipalCallback(
            executionSubject, username);
    handler.handle(new Callback[] { pwdCallback, cpCallback });

    if (pwdCallback.getResult()) {
        return; // login success.
    } else {
    // login failure. Failure while setting Security Context
    }
  }
}
```

16.5.2   Case 2: Identity Translated Between Security Domains

The Case #2 scenario enables resource adapters that are aware of only the EIS Principal to execute `Work` instances under a security context that is mapped to the application server's security domain. As an illustrative example, consider the case where integration with an EIS, an XMPP (Extensible Messaging and Presence Protocol) server is established through the deployment of a third-party XMPP resource adapter. The XMPP resource adapter is only aware of the identities/security domain of the XMPP server(EIS) and is unaware of the details of the security domain where it gets deployed onto.

In this case the XMPP resource adapter, the deployer/system administrator uses application server specific tools to effect a mapping fom the XMPP server security domain `Principal` to an equivalent `Principal` in the application server's security domain.

In this scenario, the XMPP resource adapter needs to flow in an XMPP exchange message that was sent by a user *JoeUser_EISPrincipal* to a `MessageEndpoint`. *JoeUser_EISPrincipal* was authenticated in the XMPP server's security domain by the XMPP server runtime and the RA through out-of-band implementation-specific schemes. The XMPP resource adapter is unaware about the security identities or configuration of the application server's security domain.

The deployer/system administrator, using application server configuration tools provides a mapping between the XMPP security domain and the application server's security domain. For example let's assume the *JoeUser_EISPrincipal* `Principal` is mapped to a *JoeUser_ASPrincipal*. When the XMPP resource adapter executes the `CallerPrincipalCallback` with *JoeUser_EISPrincipal*, the `CallbackHandler`

implementation, aware of the configured mapping rules, maps the
JoeUser_EISPrincipal to *JoeUser_ASPrincipal* and establishes *JoeUser_ASPrincipal* as the
Caller Principal.

The resource adapter implementation source code remains the same as in Section
16.5.1 "Case 1: Identity in the Container Security Domain".

Common Client Interface

This chapter specifies the Common Client Interface (CCI).

17.1   Overview

The CCI defines a standard client API for application components. The CCI enables application components and Enterprise Application Integration (EAI) frameworks to drive interactions across heterogeneous EISs using a common client API. The following figure shows a high-level view of the CCI and its relationship to other application components.

FIGURE 17-1 Common Client Interface
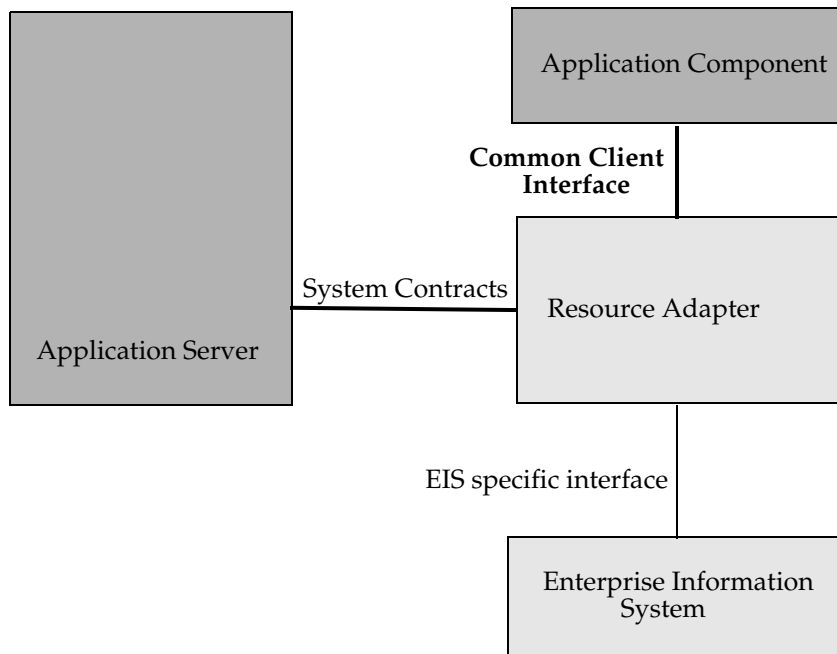
17.2   Goals

The CCI is designed with the following goals:

- It defines a remote function-call interface that focuses on executing functions on an EIS and retrieving the results. The CCI can form a base level API for EIS access on which higher level functionality can be built.
- It is targeted primarily towards application development tools and EAI frameworks.

- Although it is simple, it has sufficient functionality and an extensible application programming model.
- It provides an API that both leverages and is consistent with various facilities defined by the Java SE and Java EE platforms.
- It is independent of a specific EIS. For example, it does not use data types specific to an EIS. However, the CCI can be capable of being driven by EIS-specific metadata from a repository.

An important goal for the CCI is to complement existing standard JDBC API and not to replace this API. The CCI defines a common client API that is parallel to the JDBC for EISs that are not relational databases.

Since the CCI is targeted primarily towards application development tools and EAI vendors, it is not intended to discourage the use of JDBC APIs by these vendors. For example, an EAI vendor will typically combine JDBC with CCI by using the JDBC API to access relational databases and using CCI to access other EISs.

17.3 Scenarios

The following scenarios illustrate the use of CCI by enterprise tools and Enterprise Application Integration (EAI) vendors:

17.3.1 Enterprise Application Integration Framework

The EAI vendor uses the Common Client Interface as a standard way to plug-in resource adapters for heterogeneous EISs. The vendor provides an application integration framework on top of the functionality provided by the resource adapters. The framework uses the standard CCI interfaces to drive interactions with the connected EISs.

The following figure also shows the use of JDBC by the EAI framework for connecting to and accessing relational databases.

FIGURE 17-2 Scenario: EAI Framework
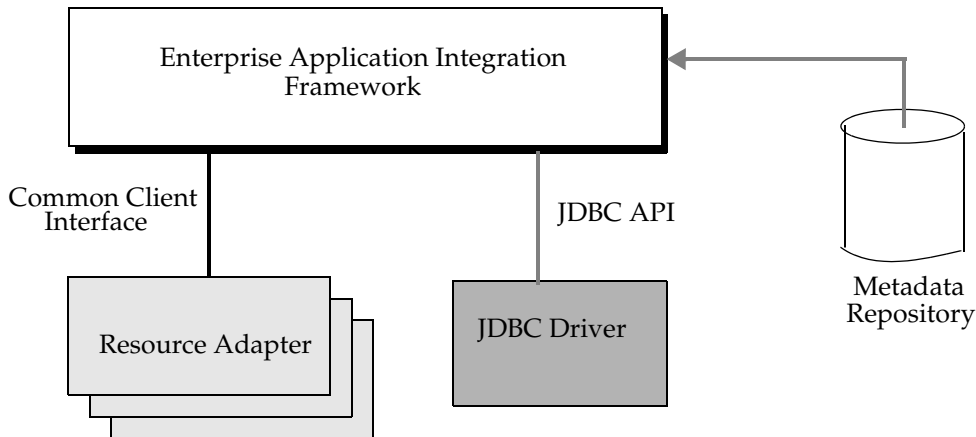
17.3.2 Metadata Repository and API

An EAI or application development tool uses a metadata repository to drive CCI-based interactions with heterogeneous EISs. See FIGURE 17-2 and FIGURE 17-3 for illustrative examples. A repository may maintain meta information about functions, with type mapping information and data structures for the invocation parameters, existing on an EIS system.

Note – The specification of a standard repository API and metadata format is outside the scope of the current version of the connector architecture.

17.3.3 Enterprise Application Development Tool

The CCI functions as a plug-in contract for an application development tool that develops additional functionality around a resource adapter.

The application development tool generates Java classes based on the meta information accessed from a metadata repository. These Java classes encapsulate CCI-based interactions and expose a simple application programming model, typically based on the JavaBeans framework, to the application developers. An application component uses the generated Java classes for EIS access.

An application development tool can also compose or generate an application component that uses the generated Java classes for EIS access.

FIGURE 17-3 Scenario: Enterprise Application Development Tool
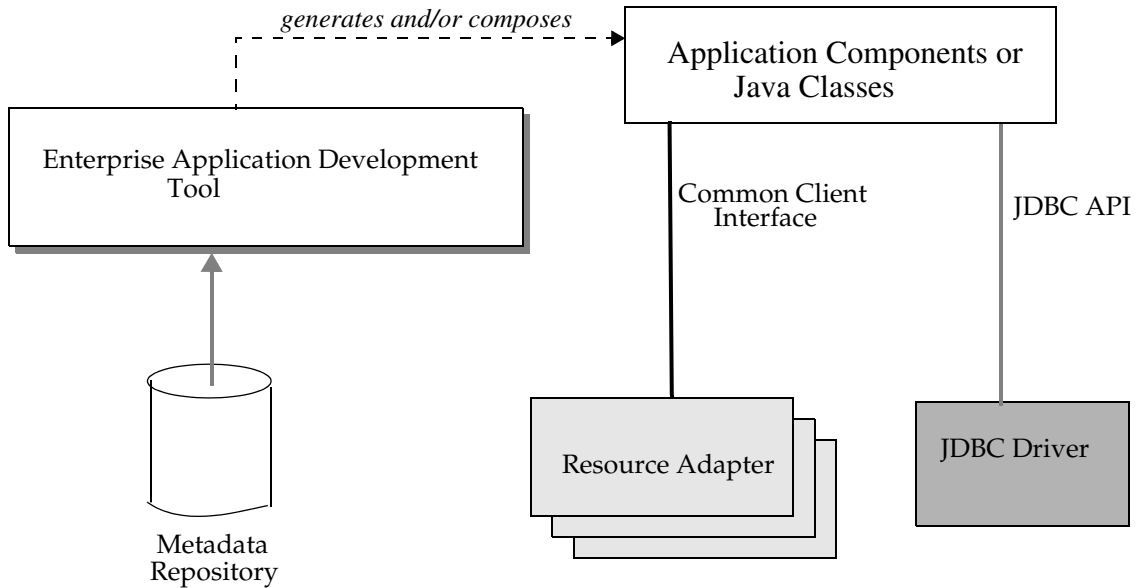
17.4   Common Client Interface

The CCI is divided in to the following parts:

- Connection-related interfaces that represent a connection factory and an application level connection:

  - `javax.resource.cci.ConnectionFactory`
  - `javax.resource.cci.Connection`
  - `javax.resource.cci.ConnectionSpec`
  - `javax.resource.cci.LocalTransaction`
- Interaction-related interfaces that enable a component to drive an interaction, specified through an `InteractionSpec`, with an EIS instance:

  - `javax.resource.cci.Interaction`
  - `javax.resource.cci.InteractionSpec`
- Service endpoint message listener interface:

  - `javax.resource.cci.MessageListener`

- Data representation-related interfaces that are used to represent data structures involved in an interaction with an EIS instance:
  - `javax.resource.cci.Record`
  - `javax.resource.cci.MappedRecord`
  - `javax.resource.cci.IndexedRecord`
  - `javax.resource.cci.RecordFactory`
  - `javax.resource.cci.Streamable`
  - `javax.resource.cci.ResultSet`
  - `java.sql.ResultSetMetaData`
- Metadata related-interfaces that provide basic meta information about a resource adapter implementation and an EIS connection:
  - `javax.resource.cci.ConnectionMetaData`
  - `javax.resource.cci.ResourceAdapterMetaData`
  - `javax.resource.cci.ResultSetInfo`
- Additional classes:
  - `javax.resource.ResourceException`
  - `javax.resource.cci.ResourceWarning`

See FIGURE 17-4 for the class diagram for CCI.

17.4.1 Requirements

A resource adapter provider provides an implementation of the CCI interfaces as part of its resource adapter implementation. The connector architecture does not mandate that a resource adapter support the CCI interfaces as its client API.

Note – A resource adapter is allowed to support a client API specific to its underlying EIS. An example of an EIS-specific client APIs is JDBC API for relational databases.

The connector architecture also allows a third-party vendor to provide an implementation of CCI interfaces above a resource adapter. For example, a base resource adapter supports the system contracts and provides an EIS specific client API. A third-party tools vendor may provide the CCI implementation above this base resource adapter.

The connector architecture also allows a resource adapter implementation to support all interfaces except the data representation-related interfaces. In this case, a third-party vendor provides both the development-time and runtime aspects of data structures required to drive interactions wthird-partyith an EIS instance. The section on the `Record` interface specification describes this case in more detail.

FIGURE 17-4 Class Diagram: Common Client Interface
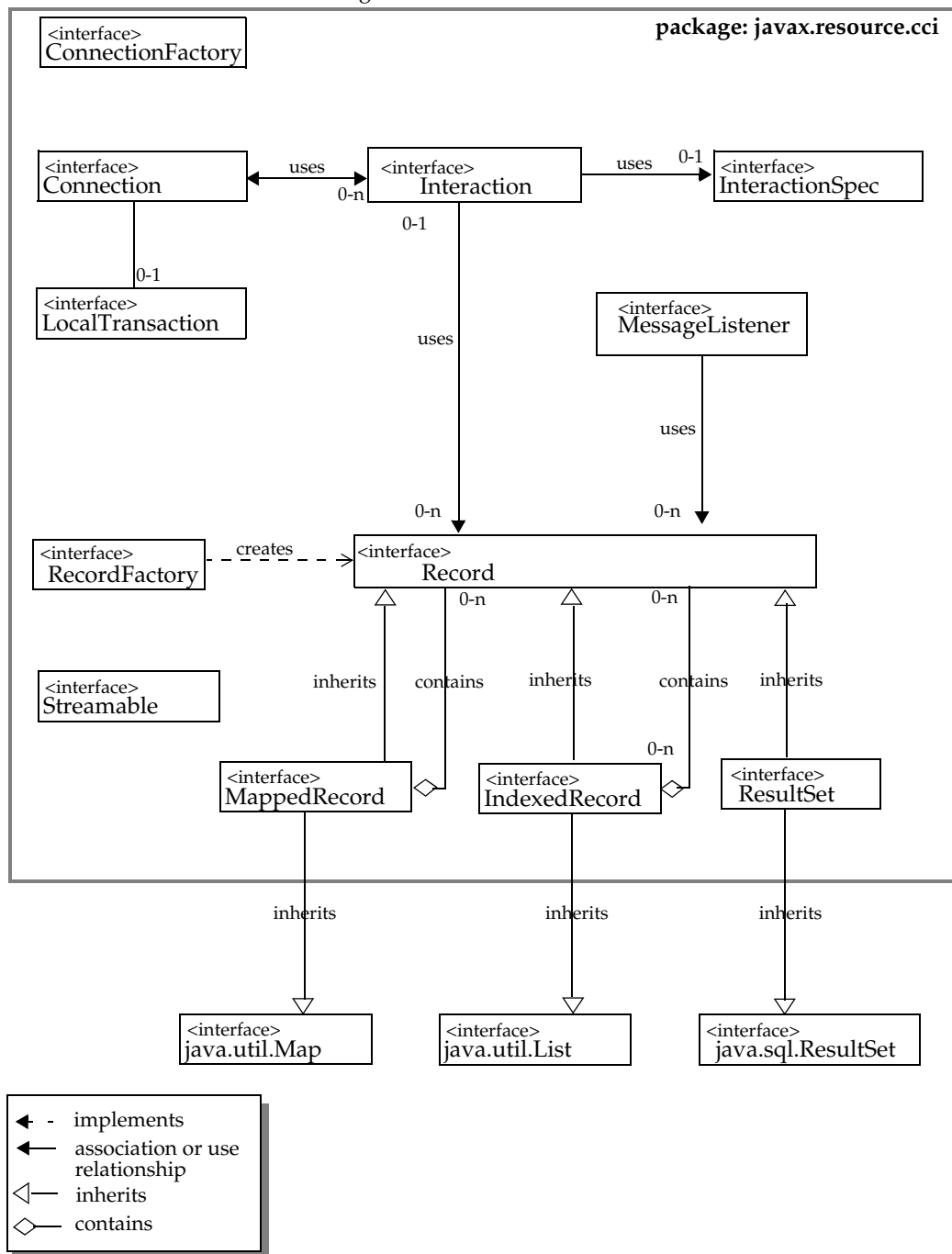
17.5 Connection Interfaces

This section specifies interfaces for the connection factory and application level connection.

17.5.1 `ConnectionFactory`

The `javax.resource.cci.ConnectionFactory` provides an interface for getting a connection to an EIS instance. A component looks up a `ConnectionFactory` instance from the JNDI namespace and then uses it to get a connection to the EIS instance.

The following code extract shows the `ConnectionFactory` interface:

```
public interface javax.resource.cci.ConnectionFactory
        extends java.io.Serializable,
        javax.resource.Referenceable {

    public RecordFactory getRecordFactory()
            throws ResourceException;

    public Connection getConnection()
            throws ResourceException;
    public Connection getConnection(
        javax.resource.cci.ConnectionSpec properties)
            throws ResourceException;

    public ResourceAdapterMetaData getMetaData()
            throws ResourceException;

}
```

The `getConnection` method gets a connection to an EIS instance. The `getConnection` variant with no parameters is used when a component requires the container to manage EIS sign-on. In this case of the container-managed sign-on, the component does not pass any security information.

A component may also use the `getConnection` variant with a `javax.resource.cci.ConnectionSpec` parameter, if any resource adapter specific security information and connection parameters is required to be passed. In

the component-managed sign-on case, an application component passes security information, such as username and password, through the `ConnectionSpec` instance.

It is important to note that the properties passed through the `getConnection` method should be client-specific, such as username, password, and language, and not be related to the configuration of a target EIS instance, such as port number or server name. The `ManagedConnectionFactory` instance is configured with a complete set of properties required for the creation of a connection to an EIS instance. Configured properties on a `ManagedConnectionFactory` can be overridden by client-specific properties passed by an application component through the `getConnection` method. Refer to Section 20.5.2, "ManagedConnectionFactory" on page 20-13 for configuration of a `ManagedConnectionFactory`.

Note that in a managed environment, the `getConnection` method with no parameters is the recommended model for getting a connection. The container manages the EIS sign-on in this case.

The `ConnectionFactory` interface also provides a method to get a `RecordFactory` instance. The `ConnectionFactory` implementation class may throw a `javax.resource.NotSupportedException` from the method `getRecordFactory`.

17.5.1.1   Requirements

An implementation class for `ConnectionFactory` must implement the `java.io.Serializable` interface to support JNDI registration. A `ConnectionFactory` implementation class is also required to implement `javax.resource.Referenceable`. Note that the `javax.resource.Referenceable` interface extends the `javax.naming.Referenceable` interface. Refer to Section 20.6, "JNDI Configuration and Lookup" on page 20-15 for more details on JNDI based requirements for the `ConnectionFactory` implementation.

An implementation class for `ConnectionFactory` must provide a default constructor.

17.5.2   `ConnectionSpec`

The interface `javax.resource.cci.ConnectionSpec` is used by an application component to pass connection request-specific properties to the `getConnection` method.

The `ConnectionSpec` interface has been introduced to increase the toolability of the CCI. The `ConnectionSpec` interface must be implemented as a JavaBean. Refer to Section 19.3, "JavaBean Requirements" on page 19-2.

The following code extract shows the `ConnectionSpec` interface.

```
public interface javax.resource.cci.ConnectionSpec {
}
```

The CCI specification defines a set of standard properties for a `ConnectionSpec`. The properties are defined either on a derived interface or an implementation class of an empty `ConnectionSpec` interface. In addition, a resource adapter may define additional properties specific to its underlying EIS.

The following standard properties are defined by the CCI specification for `ConnectionSpec`:

TABLE 17-1 Standard Properties for ConnectionSpec

Property	Description
UserName	The name of the user establishing a connection to an EIS instance.
Password	The password for the user establishing a connection.

An important point to note is about the relationship between `ConnectionSpec` and `ConnectionRequestInfo`. The `ConnectionSpec` is used at the application level and is defined under the scope of CCI while `ConnectionRequestInfo` is defined as part of the system contracts. Separate interfaces have been defined to ensure the separation between CCI interfaces and system contracts. `ConnectionRequestInfo` has no explicit dependency on CCI. Note that a resource adapter may not implement CCI but it must implement system contracts. The specification of a standard repository API and metadata format is outside the scope of the current version of the connector architecture. The mapping between CCI's `ConnectionSpec` and `ConnectionRequestInfo` is achieved in an implementation-specific manner by a resource adapter.

17.5.3 `Connection`

A `javax.resource.cci.Connection` represents an application level connection handle that is used by a component to access an EIS instance. The actual physical connection associated with a `Connection` instance is represented by a `ManagedConnection`.

A component gets a Connection instance by using the getConnection method of a ConnectionFactory instance. A Connection instance may be associated with zero or more Interaction instances.

The following code extract shows the Connection interface:

```
public interface javax.resource.cci.Connection {
        public Interaction createInteraction() throws
                ResourceException;

        public ConnectionMetaData getMetaData() throws
                ResourceException;
        public ResultSetInfo getResultSetInfo() throws
                ResourceException;

        public LocalTransaction getLocalTransaction()
                    throws ResourceException;

        public void close() throws ResourceException;
}
```

The createInteraction method creates an Interaction instance associated with the Connection instance. An Interaction enables a component to access EIS data and functions.

The getMetaData method returns information about the EIS instance associated with a Connection instance. The EIS instance-specific information is represented by the ConnectionMetaData interface.

The getResultSetInfo method returns information on the result set functionality supported by the connected EIS instance. If the CCI implementation does not support result set functionality, then the method getResultSetInfo must throw a NotSupportedException.

The close method initiates a close of the connection. The OID in FIGURE 6-6 on page 38 describes the resulting behavior of such an application level connection close.

The getLocalTransaction method returns a LocalTransaction instance that enables a component to demarcate resource manager local transactions. If a resource adapter does not allow a component to demarcate local transactions using the LocalTransaction interface, the getLocalTransaction method must throw a NotSupportedException.

17.5.3.1 Auto Commit

When a `Connection` is in an auto-commit mode, an `Interaction`, associated with the `Connection`, automatically commits after it has been executed. The auto-commit mode must be turned off if multiple interactions have to be grouped in a single transaction and committed or rolled back as a unit.

CCI does not provide explicit `set/getAutoCommit` methods in the `Connection` interface. This simplifies the application programming model for the transaction management.

A resource adapter must manage the auto-commit mode as follows:

- A transactional resource adapter either at the `XATransaction` or `LocalTransaction` level must set the auto-commit mode of `Connection` instances participating in a transaction to off within the transaction. This requirement holds for true both container-managed and bean-managed transaction demarcation.
- A transactional resource adapter must set the auto-commit mode of `Connection` instances to on when used outside a transaction.

These requirements are independent of whether a transaction is managed as a local or XA transaction. A transactional resource adapter should implement this requirement in an implementation-specific manner.

A non-transactional resource adapter at the `NoTransaction` level, is not required to support the auto-commit mode for `Connection`.

17.6 Interaction Interfaces

This section specifies interfaces that enable a component to drive an interaction with an EIS instance and to demarcate resource manager local transactions.

17.6.1 Interaction

The `javax.resource.cci.Interaction` enables a component to execute EIS functions. An `Interaction` instance supports the following interactions with an EIS instance:

- An `execute` method that takes an input `Record`, output `Record`, and an `InteractionSpec`. This method executes the EIS function represented by the `InteractionSpec` and updates the output `Record`.

- An `execute` method that takes an input `Record` and an `InteractionSpec`. This method implementation executes the EIS function represented by the `InteractionSpec` and produces the output `Record` as a return value.

If an `Interaction` implementation does not support a variant of the `execute` method, the method must throw a `javax.resource.NotSupportedException`.

Refer to Section 17.10.2 "Interaction and Record" for details on how input and output records are created and used in the above variants of the `execute` method.

An `Interaction` instance is created from a `Connection` and must maintain its association with the `Connection` instance. The `close` method releases all resources maintained by the resource adapter for the `Interaction`. The `close` of an `Interaction` instance should not close the associated `Connection` instance.

The following code extract shows the `Interaction` interface:

```
public interface javax.resource.cci.Interaction {

      public Connection getConnection();

      public void close() throws ResourceException;

      public boolean execute(InteractionSpec ispec,
              Record input,
              Record output) throws ResourceException;

      public Record execute(InteractionSpec ispec,
              Record input) throws ResourceException;
      ...
}
```

17.6.2   InteractionSpec

A `javax.resource.cci.InteractionSpec` holds properties for driving an `Interaction` with an EIS instance. An `InteractionSpec` uses an `Interaction` to execute the specified function on an underlying EIS.

The CCI specification defines a set of standard properties for an `InteractionSpec`. The properties are defined either on a derived interface or an implementation class of an empty `InteractionSpec` interface. The following code extract shows the `InteractionSpec` interface.

```
public interface javax.resource.cci.InteractionSpec
          extends java.io.Serializable {

      // Standard Interaction Verbs
      public static final int SYNC_SEND = 0;
      public static final int SYNC_SEND_RECEIVE = 1;
      public static final int SYNC_RECEIVE = 2;


}
```

An `InteractionSpec` implementation is not required to support a standard property if that property does not apply to its underlying EIS. The `InteractionSpec` implementation class must provide getter and setter methods for each of its supported properties. The getter and setter methods convention should be based on the JavaBeans design pattern.

17.6.2.1   Standard Properties

The standard properties are as follows:

- **FunctionName.** A string representing the name of an EIS function. Some examples are the name of a transaction program in a CICS system or the name of a business object or function module in an ERP system. The format of the name is specific to an EIS and is outside the scope of the CCI specification.
- **InteractionVerb.** An integer representing the mode of interaction with an EIS instance as specified by the `InteractionSpec`. The values of the interaction verb may be one of the following:

  - `SYNC_SEND`. The execution of an `Interaction` does only a send to the target EIS instance. The input record is sent to the EIS instance without any synchronous response in terms of an output `Record` or `ResultSet`.
  - `SYNC_SEND_RECEIVE`. The execution of an `Interaction` sends a request to the EIS instance and receives a response synchronously. The input record is sent to the EIS instance with the output received either as `Record` or a `ResultSet`.
  - `SYNC_RECEIVE`. The execution of an `Interaction` results in a synchronous receive of an output `Record`. For instance, a session bean gets a method invocation and it uses this `SYNC_RECEIVE` form of interaction to retrieve messages that have been delivered to a message queue.

The default `InteractionVerb` property is `SYNC_SEND_RECEIVE`.

If the `InteractionVerb` property is not defined for an `InteractionSpec`, the default mode for an interaction is `SYNC_SEND_RECEIVE`.

Other forms of interaction verbs are outside the scope of the CCI specification.

The CCI does not support asynchronous delivery of messages to the component instances. The message inflow contract should be used for asynchronous delivery of messages.

- **ExecutionTimeout.** An integer representing the number of milliseconds an `Interaction` waits for an EIS to execute the specified function.

17.6.2.2 `ResultSet` Properties

The following standard properties give hints to an `Interaction` instance about the `ResultSet` requirements:

- **FetchSize.** An integer representing the number of rows that should be fetched from an EIS when more rows are needed for a result set. If the value is zero, the hint is ignored. The default value is zero.
- **FetchDirection.** An integer representing the direction in which the rows in a result set are processed. The valid integer values are defined in the `java.sql.ResultSet` interface. The default value is `ResultSet.FETCH_FORWARD`.
- **MaxFieldSize.** An integer representing the maximum number of bytes allowed for any value in a column of a result set or a value in a `Record`.
- **ResultSetType.** An integer representing the type of the result set produced by an execution of the `InteractionSpec`. The `java.sql.ResultSet` interface defines the result set types.
- **ResultSetConcurrency.** An integer representing the concurrency type of the result set produced by the execution of the `InteractionSpec`. The `java.sql.ResultSet` interface defines the concurrency types for a result set.

Note that if a CCI implementation cannot support the specified requirements for a result set, it should choose an appropriate alternative and raise a `SQLWarning` from the `ResultSet` methods to indicate this condition. Refer to Section 17.11 "ResultSet" for more details.

A component can determine the actual scrolling ability and concurrency type of a result set by invoking the `getType` and `getConcurrencyType` methods of the `ResultSet` interface.

17.6.2.3 Additional Properties

An `InteractionSpec` implementation may define additional properties besides the standard properties. Note that the format and type of the additional properties is specific to an EIS and is outside the scope of the CCI specification.

17.6.2.4   Implementation

The `InteractionSpec` interface must be implemented as a JavaBean to support tools. The properties on the `InteractionSpec` implementation class must be defined through the getter and setter methods design pattern.

The CCI implementation may, though is not required to, provide a `BeanInfo` class for the `InteractionSpec` implementation. This class provides explicit information about the properties supported by the `InteractionSpec`.

An implementation class for the `InteractionSpec` interface must implement the `java.io.Serializable` interface.

The specified properties must be implemented as either bound or constrained properties. Refer to the JavaBeans specification (refer to http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html) for details on bound and constrained properties.

17.6.2.5   Administered Object

An `InteractionSpec` instance may be, though it is not required to be, registered as an administered object in the JNDI namespace. This enables a component provider to access `InteractionSpec` instances using logical names, called resource environment references. Resource environment references are special entries in the component's environment. The deployer binds a resource environment reference to an `InteractionSpec` administered object in the operational environment.

The EJB specification (see "Enterprise JavaBeans (EJB) Specification, version 3.2" on page E-1) specifies resource environment references in more detail.

17.6.2.6   Illustrative Scenario

The development tool introspects the `InteractionSpec` implementation class and shows a property sheet with all the configurable properties. The developer then configures the properties for an `InteractionSpec` instance.

At runtime, the configured `InteractionSpec` instance is used to specify properties for the execution of an `Interaction`. The runtime environment may lookup an `InteractionSpec` instance using a logical name from the JNDI namespace.

17.6.3   `LocalTransaction`

The `javax.resource.cci.LocalTransaction` defines a transaction demarcation interface for resource manager local transactions. An application component uses the `LocalTransaction` interface to demarcate local transactions. Refer to Chapter 7, "Transaction Management" for more details on local transactions.

Note that this interface is used for local transaction demarcation at the application level, while the `javax.resource.spi.LocalTransaction` interface is defined as part of the system contracts and is used by a container for local transaction management.

The following code extract shows the `LocalTransaction` interface:

```
public interface javax.resource.cci.LocalTransaction {
      public void begin() throws ResourceException;
      public void commit() throws ResourceException;
      public void rollback() throws ResourceException;
}
```

17.6.3.1   Requirements

A CCI implementation may, though is not required to, implement the `LocalTransaction` interface.

If the `LocalTransaction` interface is supported by a CCI implementation, the `Connection.getLocalTransaction` method must return a `LocalTransaction` instance. A component may then use the returned `LocalTransaction` to demarcate a resource manager local transaction on the underlying EIS instance.

A resource adapter is allowed to implement the `javax.resource.spi.LocalTransaction` interface without implementing the application-level `javax.resource.cci.LocalTransaction` interface. In this case, a container uses the system contract-level `LocalTransaction` interface for managing local transactions. Refer to Section 7.7, "Local Transaction Management Contract" on page 7-28 for more details on local transaction management.

17.7   Basic Metadata Interfaces

This section specifies the interfaces that provide basic meta information about a resource adapter implementation and an EIS connection.

17.7.1   ConnectionMetaData

The `javax.resource.cci.ConnectionMetaData` interface provides information about an EIS instance connected through a `Connection` instance. A component calls the `Connection.getMetaData` method to get a `ConnectionMetaData` instance.

The following code extract shows the `ConnectionMetaData` interface:

```
public interface javax.resource.cci.ConnectionMetaData {
      public String getEISProductName() throws ResourceException;
      public String getEISProductVersion() throws
          ResourceException;
      publicString getUserName() throws ResourceException;

}
```

The `getEISProductName` and `getEISProductVersion` methods return information about the EIS instance.

The `getUserName` method returns the user name for an active connection as known to the underlying EIS instance. The name corresponds the resource principal under whose security context a connection to the EIS instance has been established.

17.7.1.1   Implementation

A CCI implementation must provide an implementation class for the `ConnectionMetaData` interface.

A resource adapter provider or third-party vendor may extend the `ConnectionMetaData` interface to provide additional information. Note that the format and type of the additional information is specific to an EIS and is outside the scope of the CCI specification.

17.7.2   ResourceAdapterMetaData

The `javax.resource.cci.ResourceAdapterMetaData` interface provides information about the capabilities of a resource adapter implementation. Note that this interface does not provide information about an EIS instance that is connected through a resource adapter.

A component uses the `ConnectionFactory.getMetaData` method to get metadata information about a resource adapter. The `getMetaData` method does not require that an active connection to an EIS instance be established.

The following code extract shows the `ResourceAdapterMetaData` interface:

```
public interface javax.resource.cci.ResourceAdapterMetaData {
       public String getAdapterVersion();
       public String getAdapterVendorName();
       public String getAdapterName();
       public String getAdapterShortDescription();

       public String getSpecVersion();

       public String[] getInteractionSpecsSupported();
       public boolean supportsExecuteWithInputAndOutputRecord();
       public boolean supportsExecuteWithInputRecordOnly();

       public boolean supportsLocalTransactionDemarcation();
}
```

The `getSpecVersion` method returns a string representation of the version of the connector architecture specification that is supported by the resource adapter.

The `getInteractionSpecsSupported` method returns an array of fully-qualified names of `InteractionSpec` types supported by the CCI implementation for this resource adapter. Note that the fully-qualified class name is for the implementation class of an `InteractionSpec`. This method may be used by tools vendors to find information on the supported `InteractionSpec` types. The method should return an array of length 0 if the CCI implementation does not define specific `InteractionSpec` types.

The `supportsExecuteWithInputAndOutputRecord` and `supportsExecuteWithInputRecordOnly` methods are used by tools vendors to find information about the `Interaction` implementation. It is important to note that the `Interaction` implementation must support at least one variant of the `execute` methods.

The `supportsExecuteWithInputAndOutputRecord` method returns `true` if the implementation class for the `Interaction` interface implements the `public boolean execute(InteractionSpec ispec, Record input, Record output)` method. If not, the method returns `false`.

The `supportsExecuteWithInputRecordOnly` method returns `true` if the implementation class for the `Interaction` interface implements the `public Record execute(InteractionSpec ispec, Record input)` method. If not, the method returns `false`.

The `supportsLocalTransactionDemarcation` method returns `true` if the resource adapter implements the `LocalTransaction` interface and supports local transaction demarcation on the underlying EIS instance through the `LocalTransaction` interface.

The `ResourceAdapterMetaData` interface may be extended to provide more information specific to a resource adapter implementation.

17.8   Service Endpoint Message Listener Interface

The `MessageListener` interface serves as a request-response message listener type that message endpoints (refer to Chapter 13, "Message Inflow") may implement. This allows an EIS to communicate with an endpoint using a request-response style.

```
interface javax.resource.cci.MessageListener {
    Record onMessage(Record inputData) throws ResourceException;
}
```

17.9   Exception Interfaces

This section specifies `ResourceException` class defined by the CCI.

17.9.1   ResourceException

The `javax.resource.ResourceException` class is used as the root of the exception hierarchy for CCI. A `ResourceException` provides the following information:

- A resource adapter-specific string describing the error. This string is a standard Java exception message and is available through the `getMessage` method.
- A resource adapter-specific error code.
- A reference to another exception. A `ResourceException` is often the result of a lower level problem. If appropriate, this lower level exception, a `java.lang.Exception` or its derived exception type, can be linked to a `ResourceException` instance. Note, this has been deprecated in favor of the J2SE release 1.4 exception chaining facility.

A CCI implementation can extend the `ResourceException` interface to throw more specific exceptions. It may also chain instances of `java.lang.Exception` or its subtypes to a `ResourceException`.

17.9.2   `ResourceWarning`

The `javax.resource.cci.ResourceWarning` class provides information on the warnings related to interactions with EIS. A `ResourceWarning` is silently chained to an `Interaction` instance that has caused the warning to be reported.

The `Interaction.getWarnings` method enables a component to access the first `ResourceWarning` in a chain of warnings. Other `ResourceWarning` instances are chained to the first returned `ResourceWarning` instance.

17.10   `Record`

A `Record` is the Java representation of a data structure used as input or output to an EIS function.

A `Record` has both development-time and runtime aspects. See the following figure for an illustration of this. An implementation of a `Record` is either:

- A custom `Record` implementation that gets generated at the development time by a tool. The generation of a custom implementation is based on the meta information accessed by the tool from a metadata repository. The type mapping and data representatiothird-partyn is generated as part of the custom `Record` implementation. So the custom `Record` implementation typically does not need to access the metadata repository at runtime.
- A generic `Record` implementation that uses a metadata repository at runtime for meta information. For example, a generic type of `Record` may access the type mapping information from the repository at runtime.

Note: The specification of a standard repository API and metadata format is outside the scope of the current version of the connector architecture.

FIGURE 17-5 Record at Development-time and Runtime
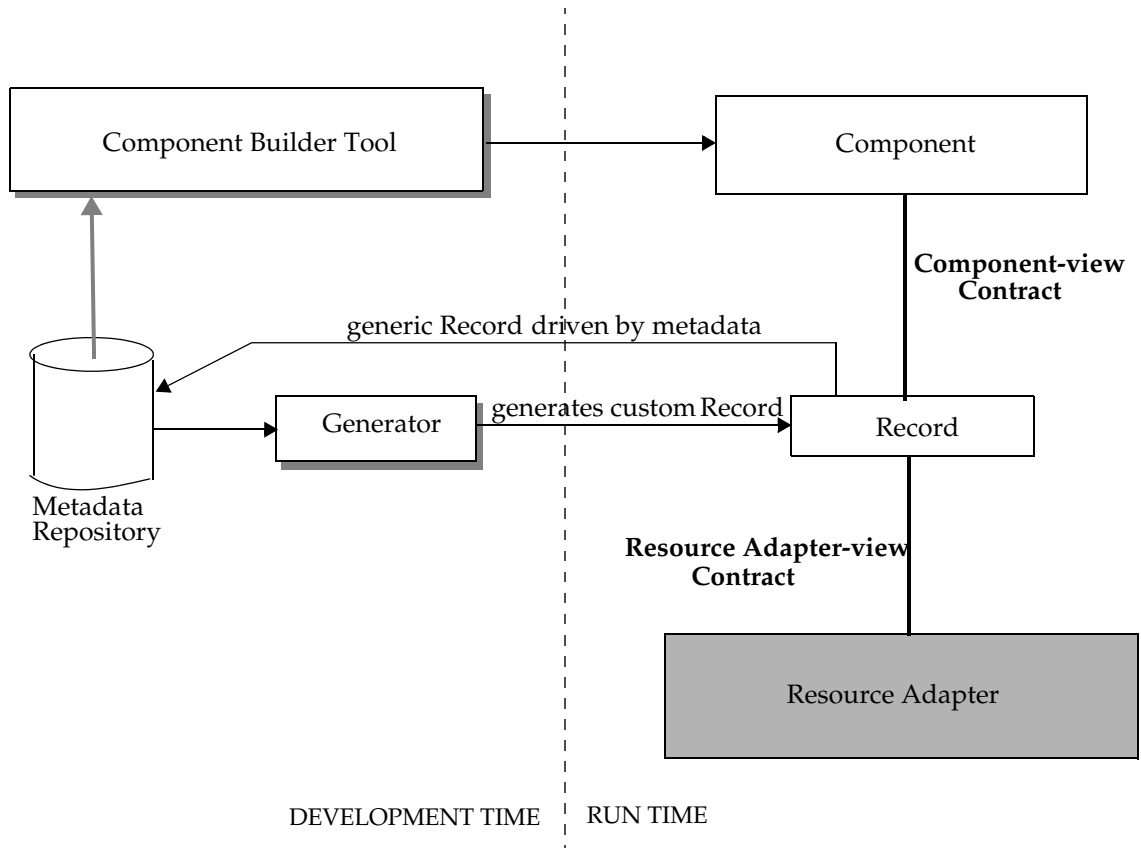
The meta information used in a `Record` representation and type mapping may be available in a metadata repository as:

- Meta information expressed in an EIS-specific format. For example, an ERP system has its own descriptive format for its meta information.
- Formatted in structures based on the programming language that has been used for writing the target EIS function, such as, `COBOL` structures used by CICS transaction programs.
- A standard representation of data structures as required for EIS functions. The standard representation is typically aggregated in a metadata repository based on the meta information extracted from multiple EISs.

A resource adapter may provide an implementation of all CCI interfaces except the data representation-related interfaces, namely, `Record` and `RecordFactory`. In this case, a third-party vendor provides both development-time and runtime support for

the `Record` and `RecordFactory` interfaces. This requires that a `Record` implementation must support both component-view and resource adapter-view contracts, as specified in the following subsections.

17.10.1 Component-View Contract

The component-view contract provides a standard contract for using a `Record` for components and component building tools. A `Record` implementation must support the component-view contract.

The application programming model for a `Record` is as follows:
- A component creates an instance of a generated implementation class for a custom record. The implementation class represents an EIS-specific data structure.
- A component uses the `RecordFactory` interface to create an instance of the generic `Record` implementation class. The implementation class of a generic `Record` is independent of any EIS-specific data structure.

A related CCI issue is the level of support in the CCI data representation interfaces (namely, `Record`, `MappedRecord`, and `IndexedRecord`) for the type mapping facility. The issue has to be addressed based on the following parameters:

- There is no standardized mapping across various type systems. For example, the existing type systems range from Java, CORBA, COM, COBOL and many more. It is difficult to standardize the type specification and mappings across such a diverse set of type systems within the connector architecture scope.
- Building a limited type mapping facility into the CCI data representation interfaces will constrain the use of CCI data representation interfaces across different types of EISs. For example, it may be difficult to support EISs that have complex structured types with a limited type mapping support.
- Building an extensive type mapping facility into the current version of CCI data representation interfaces will limit the future extensibility of these interfaces. This applies specifically to the support for standards that are emerging for XML-based data representation. An important goal for CCI data representation interfaces is to support XML-based facilities. This goal is difficult to achieve in the current scope of the connector architecture.

This specification proposes that the type mapping support for the CCI be kept open for future versions. A future version of this specification may standardize type mappings.

17.10.1.1 Type Mapping

Type mapping for EIS-specific types to Java types is not directly exposed to an application component. For example in the case of a custom `Record` implementation, the getter and setter methods, defined in a `Record` and exposed to

an application component, return the correct Java types for the values extracted from the `Record`. The custom `Record` implementation internally handles all the type mapping.

In the case of a generic `Record` implementation, the type mapping is done in the generic `Record` by means of the type mapping information obtained from the metadata repository. Since the component uses generic methods on the `Record` interface, the component code does the required type casting.

The compatibility of Java types and EIS types should be based on a type mapping that is defined specific to a class of EISs. For example, an ERP system from vendor X specifies a type mapping specific to its own EIS. Another example is type mapping between Java and COBOL types. Note that the JDBC specification specifies a standard mapping of SQL data types to Java types specific to relational databases.

In cases of both custom and generic `Records`, the type mapping information is provided by a metadata repository either at development-time or runtime.

17.10.1.2 `Record` Interface

The `javax.resource.cci.Record` interface is the base interface for the representation of a record. A `Record` instance is used as an input or output to the `execute` methods defined in an `Interaction`.

FIGURE 17-6 Component-view Contract
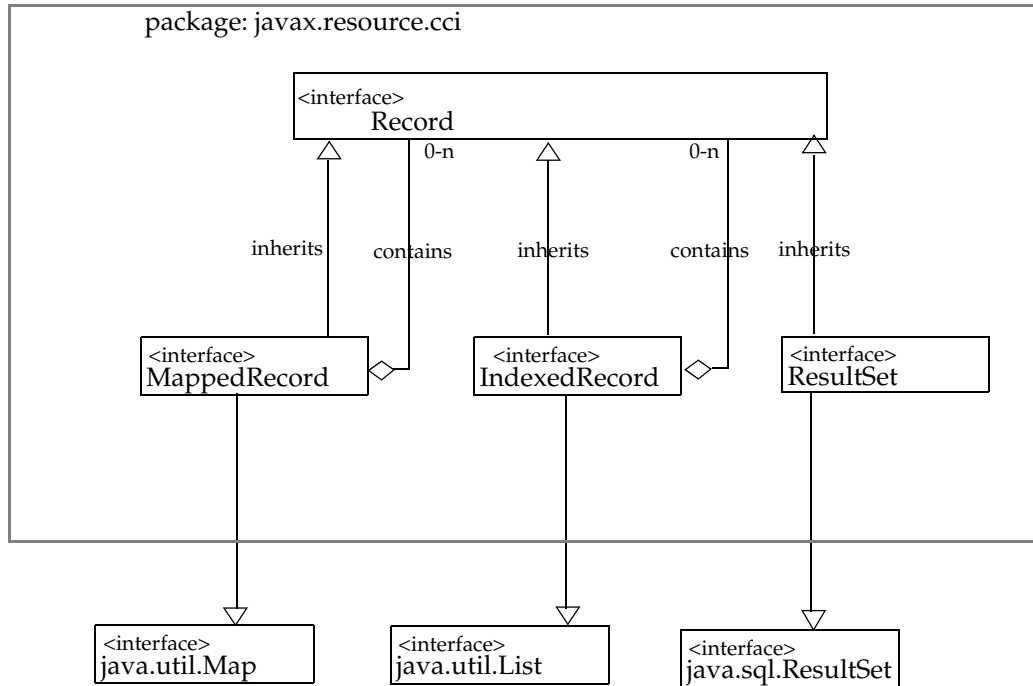
The `Record` interface may be extended to form one of the following representations:

- `javax.resource.cci.MappedRecord`: A key-value pair based collection represents a record. This interface is based on `java.util.Map`.
- `javax.resource.cci.IndexedRecord`: An ordered and indexed collection represents a record. This interface is based on `java.util.List`.
- `javax.resource.cci.ResultSet`: This interface extends both `java.sql.ResultSet` and `javax.resource.cci.Record`. A result set represents tabular data. Section 17.11 "ResultSet" specifies the requirements for the `ResultSet` interface in detail.
- A JavaBean based representation of an EIS data structure: An example is a custom record generated to represent a purchase order in an ERP system or an invoice in a mainframe TP system.

Refer to Section 17.12 "Code Samples" for code samples that illustrate the use of record.

`MappedRecord` or `IndexedRecord` may contain another `Record`. This means that `MappedRecord` and `IndexedRecord` can be used to create a hierarchical structure of any arbitrary depth.

`MappedRecord` and `IndexedRecord` can be used to represent either a generic or custom record.

A basic Java type is used as the leaf element of a hierarchical structure represented by a `MappedRecord` or `IndexedRecord`.

A generated custom `Record` may also contain other records to form a hierarchical structure.

The following code extract shows the `Record` interface:

```
public interface javax.resource.cci.Record
          extends java.lang.Cloneable, java.io.Serializable {

      public String getRecordName();
      public void setRecordName(String name);

      public void setRecordShortDescription(String description);
      public String getRecordShortDescription();

      public boolean equals(Object other);
      public int hashCode();

      public Object clone() throws CloneNotSupportedException;
}
```

The `Record` interface defines the following set of standard properties:

- `Name` of a `Record`: Note that the CCI does not define a standard format for naming a `Record`. The name format is specific to an EIS type.
- `Description` of a `Record`: This property is used primarily by tools to show a description of a `Record` instance.

17.10.1.3 `MappedRecord` and `IndexedRecord` Interfaces

The `javax.resource.cci.MappedRecord` interface is used for representing a key-value map based collection of record elements. The `MappedRecord` interface extends both the `Record` and `java.util.Map` interface.

```
public interface javax.resource.cci.MappedRecord
            extends Record, java.util.Map,
            java.io.Serializable {
}
```

The `javax.resource.cci.IndexedRecord` interface represents an ordered collection of record elements based on the `java.util.List` interface. This interface allows a component to access record elements by their integer index, position in the list, and search for elements in the list.

```
public interface javax.resource.cci.IndexedRecord
            extends Record, java.util.List,
            java.io.Serializable {
}
```

17.10.1.4 `RecordFactory`

The `javax.resource.cci.RecordFactory` interface is used for creating `MappedRecord` and `IndexedRecord` instances. Note that the `RecordFactory` is only used for creating generic record instances. A CCI implementation provides an implementation class for the `RecordFactory` interface.

The following code extract shows the `RecordFactory` interface:

```
public interface javax.resource.cci.RecordFactory {
      public MappedRecord createMappedRecord(String recordName)
            throws ResourceException;

      public IndexedRecord createIndexedRecord(String
            recordName) throws ResourceException;

}
```

The methods `createMappedRecord` and `createIndexedRecord` take the name of the record that is to be created by the `RecordFactory`. The name of the record acts as a pointer to the meta information stored in the metadata repository for a specific record type. The format of the name is outside the scope of the CCI specification and specific to a CCI implementation and/or metadata repository.

A `RecordFactory` implementation should be capable of using the name of the desired `Record` and accessing meta information for the creation of the `Record`.

17.10.2   `Interaction` and `Record`

Records should be used as follows for the two variants of the `execute` method of the `Interaction` interface:

```
boolean execute(InteractionSpec, Record input, Record output)
```

- A custom record instance is used as an input or output to the `execute` method. A custom record implementation class is generated by an application development tool or EAI framework based on the meta information.
- The `RecordFactory` interface is used to create a generic `MappedRecord` or `IndexedRecord` instance. The generic record is used as input or output to the `execute` method.

```
Record execute(InteractionSpec, Record input)
```

- The input record can be either a custom or generic record.
- The returned record is a generic record instance created by the implementation of the `execute` method. The generic record instance may represent a `ResultSet` or a hierarchical structure as represented through the `MappedRecord` and `IndexedRecord` interfaces.

When the `Interaction.execute` method is called, a generic record instance may use the connection associated with the `Interaction` instance to access the metadata from the underlying EIS. If there is a separate metadata repository, then the generic record gets the metadata from the repository. The generic record implementation may use the above illustrative mechanism to achieve the necessary type mapping.

The generic record implementation encapsulates the above behavior and interacts with `Interaction` implementation in the `execute` method to get the active connection, if so needed. The contract between the generic record and `Interaction` implementation classes is specific to a CCI implementation.

17.10.3   Resource Adapter-view Contract

A resource adapter views the data represented by a `Record` either as:

- A stream of bytes through the `Streamable` interface, or,

- A format specific to a resource adapter. For example, a resource adapter may extract or set the data for a `Record` using an interface defined specifically for the resource adapter.

A resource adapter-specific interface for viewing the `Record` representation is outside the scope of the CCI specification. A resource adapter must describe the resource adapter-specific interface to the users, typically tools vendors, of the resource adapter-view contract.

17.10.3.1   `Streamable` Interface

The `javax.resource.cci.Streamable` interface enables a resource adapter to extract data from an input `Record` or set data into an output `Record` as a stream of bytes. See the following figure.

FIGURE 17-7 Streamable Interface
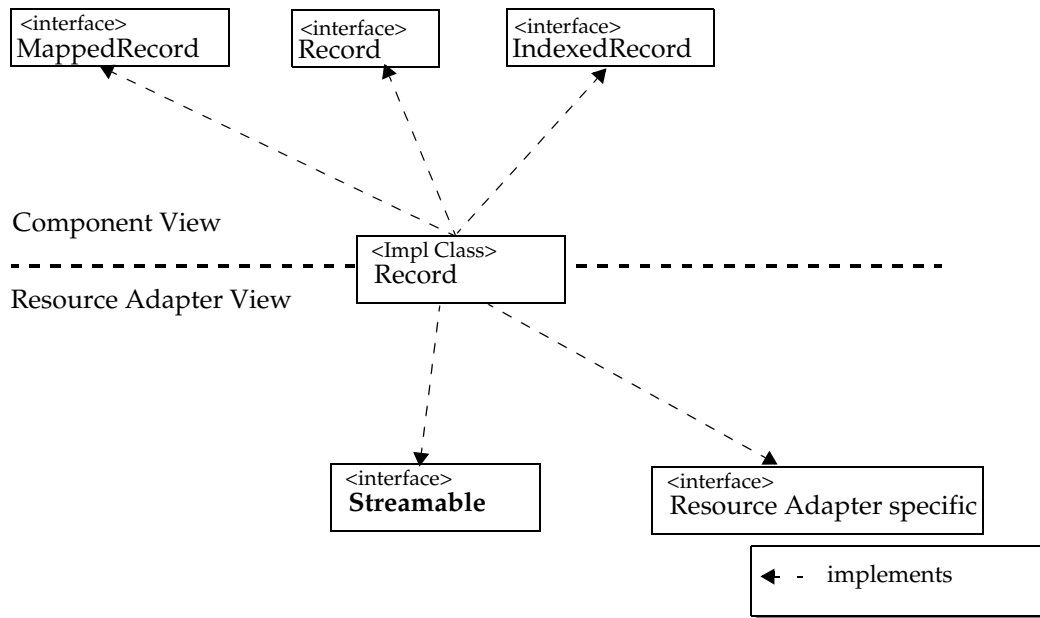
The `Streamable` interface provides a resource adapter's view of the data set in a `Record` instance by a component. A component uses `Record` or any derived interfaces to manage records.

A component does not directly use the `Streamable` interface. The interface is used by a resource adapter implementation.

The following code extract shows the `Streamable` interface:

```
public interface javax.resource.cci.Streamable {
      public void read(InputStream istream) throws IOException;
      public void write(OutputStream ostream) throws IOException;
}
```

The `read` extracts method data from an `InputStream` and initializes fields of a `Streamable` object. The `write` method writes fields of a `Streamable` object to an `OutputStream`. The implementations of both the `read` and `write` methods for a `Streamable` object must call the `read` and `write` methods respectively on the super class if there is one.

An implementation class of `Record` may choose to implement the `Streamable` interface or support a resource adapter-specific interface to manage record data.

17.11 ResultSet

A result set represents tabular data that is retrieved from an EIS instance by the execution of an interaction. The `execute` method on the `Interaction` interface can return a `ResultSet` instance.

The CCI `ResultSet` interface is based on the JDBC `ResultSet` interface. The `ResultSet` extends the `java.sql.ResultSet` and `javax.resource.cci.-Record` interfaces.

The following code extract shows the `ResultSet` interface:

```
public interface javax.resource.cci.ResultSet
            extends Record, java.sql.ResultSet {
}
```

FIGURE 17-8 ResultSet Interface
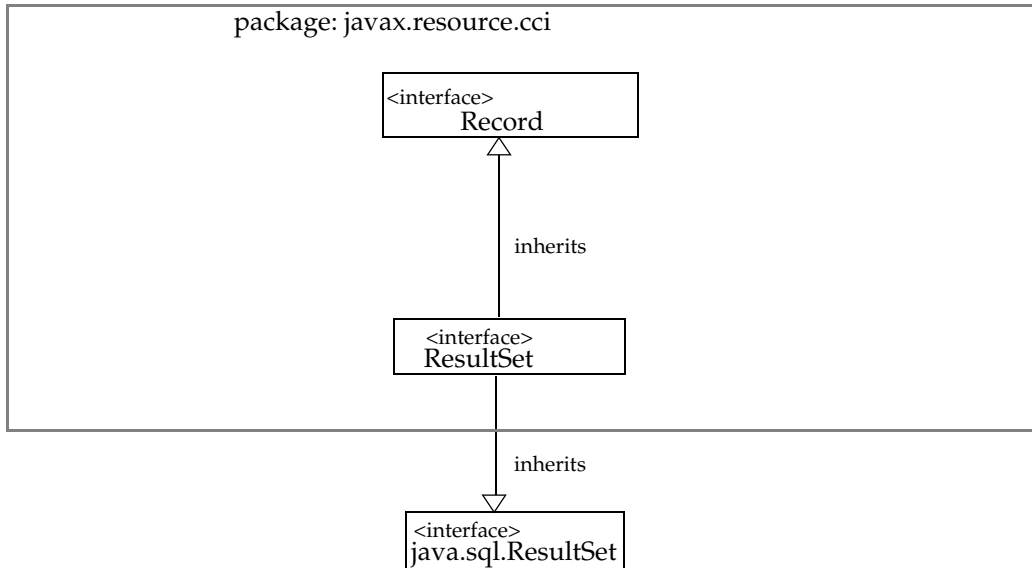
The following section specifies the requirements for a CCI `ResultSet` implementation.

Refer to the JDBC (see "JDBC API Specification, version 4.1" on page E-1) specification and Java docs for more details on the `java.sql.ResultSet` interface. The following section specifies only a brief outline of the `ResultSet` interface. It focuses on the differences between the implementation requirements set by the CCI and JDBC. Note that the JDBC semantics for a `ResultSet` hold for the cases that are not explicitly mentioned in the following section.

CCI uses the JDBC `ResultSet` interface because:

- JDBC `ResultSet` is a standard, established, and well-documented interface for accessing and updating tabular data.
- JDBC `ResultSet` interface is defined in the core `java.sql` package. An introduction of an independent CCI-specific `ResultSet` interface (that is, different from the JDBC `ResultSet` interface) may create confusion in terms of differences in the programming model and functionality.

- The use of the JDBC `ResultSet` interface enables a tool or EAI vendor to leverage existing facilities that have been for the JDBC `ResultSet`.

Note – A CCI implementation is not required to support the `javax.resource.cci.ResultSet` interface. If a CCI implementation does not support result set functionality, it should not support interfaces and methods that are associated with the result set functionality. An example is the `java.sql.ResultSetMetaData` interface.

17.11.1 `ResultSet` Interface

The `ResultSet` interface provides a set of getter methods for retrieving column values from the current row. A column value can be retrieved using either the index number of the column or the name of the column. The columns are numbered starting at one. For maximum portability, result set columns within each row should be read left-to-right, and each column should be read only once.

The `ResultSet` interface also defines a set of update*XXX* methods for updating the column values of the current row.

17.11.1.1 Type Mapping

A `ResultSet` implementation should attempt to convert the underlying EIS-specific data type to the Java type as specified in the *XXX* part of the get*XXX* method and return a suitable Java value.

A `ResultSet` implementation must establish a type mapping between the EIS specific data types and Java types. The type mapping is specific to an EIS.

The CCI specification does not specify standard type mappings specific to each type of EIS.

17.11.1.2 `ResultSet` Types

The CCI `ResultSet`, similar to the JDBC `ResultSet`, supports the following types of result set: `forward-only`, `scroll-insensitive`, and `scroll-sensitive`.

A forward-only result set is non-scrollable; its cursor moves only forward, from top to bottom. The view of data in the result set depends on whether the EIS instance materializes results incrementally.

A scroll-insensitive result set is scrollable; its cursor can move forward or backward and can be moved to a particular row or to a row whose position is relative to the current row. This type of result set is not sensitive to any changes made by another transaction or result sets in the same transaction that are made while the result set is open. This type of result set provides a static view of the underlying data with respect to changes made by other result sets. The order and values of rows are set at the time of the creation of a scroll-insensitive result set.

A scroll-sensitive result set is scrollable. It is sensitive to changes that are made while the result set is open. This type of result set provides a more dynamic view of the underlying data.

A component can use the `ownUpdatesAreVisible`, `ownDeletesAreVisible`, and `ownInsertsAreVisible` methods of the `ResultSetInfo` interface to determine whether a result set can "see" its own changes while the result set is open. For example, a result set's own changes are visible if the updated column values can be retrieved by calling the get*XXX* method after the corresponding update*XXX* method. Refer to the JDBC (see "JDBC API Specification, version 4.1" on page E-1) specification for more details on this feature.

17.11.1.3 Scrolling

The CCI `ResultSet` supports the same scrolling ability as the JDBC `ResultSet`.

If a resource adapter implements the cursor movement methods, its result sets are scrollable. A scrollable result set created by executing an `Interaction` can move through its contents in both a forward (first-to-last) or backward (last-to-first) direction. A scrollable result set also supports relative and absolute positioning.

The CCI `ResultSet`, similar to the JDBC `ResultSet`, maintains a cursor that indicates the row in the result set that is currently being accessed. The cursor maintained on a `forward-only` result set can only move forward through the contents of the result set. The rows are accessed in a first-to-last order. A scrollable result set can also be moved in a backward direction (last-to-first) and to a particular row.

Note that a CCI `ResultSet` implementation should only provide support for scrollable result sets if the underlying EIS supports such a facility.

17.11.1.4 Concurrency Types

A component can set the concurrency type of a CCI `ResultSet` to be either read-only or updatable. These types are consistent with the concurrency types defined by the JDBC `ResultSet`.

A result set that uses read-only concurrency does not allow updates of its content, while an updatable result set allows updates to its contents. An updatable result set may hold a write lock on the underlying data item and thus reduce concurrency.

Refer to the JDBC specification (see "JDBC API Specification, version 4.1" on page E-1) for detailed information and examples.

17.11.1.5   Updatability

A result set of concurrency type `CONCUR_UPDATABLE` supports the ability to update, insert, and delete its rows. The CCI support for this type of result set is similar to the JDBC `ResultSet`.

The methods update*XXX* on the `ResultSet` interface are used to modify the values of an individual column in the current row. These methods do not update the underlying EIS. The `updateRow` method must be called to update data on the underlying EIS. A resource adapter may discard changes made by a component if the component moves the cursor from the current row before calling the method `updateRow`.

Refer to the JDBC specification (see "JDBC API Specification, version 4.1" on page E-1) for more information.

17.11.1.6   Persistence of Java Objects

The `ResultSet` interface provides the `getObject` method to enable a component to retrieve column values as Java objects. The type of the Java object returned from the `getObject` method is compatible with the type mapping supported by a resource adapter-specific to its underlying EIS. The `updateObject` method enables a component to update a column value using a Java object.

17.11.1.7   Support for SQL Types

It is optional for a CCI `ResultSet` to support the SQL type `JAVA_OBJECT` as defined in `java.sql.Types`.The JDBC specification specifies the JDBC support for persistence of Java objects.

The support for the following SQL types as defined in `java.sql.Types` is optional for a CCI `ResultSet` implementation:

- Binary large object (`BLOB`)
- Character large object (`CLOB`)
- SQL `ARRAY` type
- SQL `REF` type
- SQL `DISTINCT` type

■ SQL STRUCT type

If an implementation of the CCI ResultSet interface does not support these types, it must throw a java.sql.SQLException indicating that the method is not supported, or java.lang.UnsupportedOperationException from the following methods:

■ getBlob
■ getClob
■ getArray
■ getRef

17.11.1.8 Support for Customized SQL Type Mapping

The CCI is not required to support customized mapping of SQL structured and distinct types to Java classes. The JDBC API defines support for such customization mechanisms.

The CCI ResultSet should throw a java.sql.SQLException indicating that the method is not supported or java.lang.UnsupportedOperationException from the getObject method that takes a java.util.Map parameter.

17.11.2 ResultSetMetaData

The java.sql.ResultSetMetaData interface provides information about the columns in a ResultSet instance. A component uses ResultSet.getMetaData method to get information about a ResultSet.

Refer to the JDBC API documentation for a detailed specification of the ResultSetMetaData interface.

17.11.3 ResultSetInfo

The javax.resource.cci.ResultSetInfo interface provides information on the support provided for ResultSet functionality by a connected EIS instance. A component calls the Connection.getResultInfo method to get the ResultSetInfo instance.

A CCI implementation is not required to support the javax.resource.cci.ResultSetInfo interface. The implementation of this interface is provided only if the CCI supports the ResultSet facility.

The following code extract shows the `ResultSetInfo` interface:

```
public interface javax.resource.cci.ResultSetInfo {
        public boolean updatesAreDetected(int type)
                    throws ResourceException;
        public boolean insertsAreDetected(int type)
                    throws ResourceException;
        public boolean deletesAreDetected(int type)
                    throws ResourceException;

        public boolean supportsResultSetType(int type)
                    throws ResourceException;
        public boolean supportsResultTypeConcurrency(int type,
                    int concurrency)
                    throws ResourceException;

        public boolean ownUpdatesAreVisible(int type)
                    throws ResourceException;
        public boolean ownInsertsAreVisible(int type)
                    throws ResourceException;
        public boolean ownDeletesAreVisible(int type)
                    throws ResourceException;


        public boolean othersUpdatesAreVisible(int type)
                    throws ResourceException;
        public boolean othersInsertsAreVisible(int type)
                    throws ResourceException;
        public boolean othersDeletesAreVisible(int type)
                    throws ResourceException;
}
```

The type parameter to the above methods represents the type of the `ResultSet`, defined as TYPE_*XXX* in the `ResultSet` interface.

Note that these methods should throw a `ResourceException` in the following cases:

- A resource adapter and the connected EIS instance cannot provide any meaningful values for these properties.
- The CCI implementation does not support the `ResultSet` functionality. In this case, a `NotSupportedException` should be thrown from invocations on the above methods.

A component uses the `rowUpdated`, `rowInserted`, and `rowDeleted` methods of the `ResultSet` interface to determine whether a row has been affected by a visible update, insert, or delete is the result set is open. The `updatesAreDetected`, `insertsAreDetected` and `deletesAreDetected` methods enable a component to find out whether or not changes to a `ResultSet` are detected.

A component uses the `ownUpdatesAreVisible`, `ownDeletesAreVisible` and `ownInsertsAreVisible` methods to determine whether a `ResultSet` can "see" its own changes when the result set is open.

A component uses the `supportsResultSetType` method to check the `ResultSet` types supported by a resource adapter and its underlying EIS instance.

The `supportsResultSetTypeConcurency` method provides information on the `ResultSet` concurrency types supported by a resource adapter and its underlying EIS instance.

17.12 Code Samples

The following code extracts illustrate the application programming model based on the CCI.

An application development tool or EAI framework normally hides all the CCI-based programming details from an application developer. For example, an application development tool generates a set of Java classes that abstract the CCI-based application programming model and offers a simple programming model to an application developer.

17.12.1 `Connection`

1. Get a `Connection` to an EIS instance after a lookup of a `ConnectionFactory` instance from the JNDI namespace. In this case, the component allows the container to manage the EIS sign-on.

```
javax.naming.Context nc = new InitialContext();
javax.resource.cci.ConnectionFactory cf =
              (ConnectionFactory)nc.lookup(
              "java:comp/env/eis/ConnectionFactory");
javax.resource.cci.Connection cx = cf.getConnection();
```

2. Create an `Interaction` instance:

```
javax.resource.cci.Interaction ix = cx.createInteraction();
```

17.12.2   InteractionSpec

- Create a new instance of the respective InteractionSpec class or look up a pre-configured InteractionSpec in the runtime environment using JNDI.

```
com.wombat.cci.InteractionSpecImpl ixSpec = // ...

ixSpec.setFunctionName("<EIS_SPECIFIC_FUNCTION_NAME>");
ixSpec.setInteractionVerb(InteractionSpec.SYNC_SEND_RECEIVE);
...
```

17.12.3   Mapped Record

- Get a RecordFactory instance:

```
javax.resource.cci.RecordFactory rf = // ... get a RecordFactory
```

- Create a generic MappedRecord using the RecordFactory instance. This record instance acts as an input to the execution of an interaction. The name of the Record acts as a pointer to the meta information, stored in the metadata repository, for a specific record type.

```
javax.resource.cci.MappedRecord input =
      rf.createMappedRecord("<NAME_OF_RECORD>");
```

- Populate the generic MappedRecord instance with input values. The component code adds the values based on the meta information it has accessed from the metadata repository.

```
input.put("<key: element1>", new String("<VALUE>"));
input.put("<key: element2>", ...);
...
```

- Create a generic IndexedRecord to hold the output values that are set by the execution of the interaction.

```
javax.resource.cci.IndexedRecord output =
    rf.createIndexedRecord("<NAME_OF_RECORD>");
```

- Execute the `Interaction`:

```
boolean ret = ix.execute(ixSpec, input, output);
```

- Extract data from the output `IndexedRecord`. Note that the type mapping is done in the generic `IndexedRecord` by means of the type mapping information in the metadata repository. Since the component uses generic methods on the `IndexedRecord`, the component code does the required type casting.

```
java.util.Iterator iterator = output.iterator();
while (iterator.hasNext()) {
        // Get a record element and extract value
}
```

17.12.4   ResultSet

- Set the requirements for the `ResultSet` returned by the execution of an `Interaction`. This step is optional. Default values are used if the requirements are not explicitly set:

```
com.wombat.cci.InteractionSpecImpl ixSpec = .. // get an
                                       // InteractionSpec;

ixSpec.setFetchSize(20);
ixSpec.setResultSetType(ResultSet.TYPE_SCROLL_INSENSITIVE);
```

- Execute an `Interaction` that returns a `ResultSet`:

```
javax.resource.cci.ResultSet rs = (javax.resource.cci.ResultSet)
                     ix.execute(ixSpec, input);
```

- Iterate over the `ResultSet`. The example here positions the cursor on the first row and then iterates forward through the contents of the `ResultSet`. The get*XXX* methods are used to retrieve column values:

```
rs.beforeFirst();
while (rs.next()) {
        // get the column values for the current row using getXXX
        // method
}
```

- The following example shows a backward iteration through the `ResultSet`:

```
rs.afterLast();
while (rs.previous()) {
        // get the column values for the current row using getXXX
        // method
}
```

17.12.5   Custom `Record`

- Extend the `Record` interface to represent an EIS-specific custom Record. The `CustomerRecord` interface supports a simple getter-setter design pattern for its field values. A development tool generates the implementation class of the `CustomerRecord`.

```
public interface CustomerRecord extends
javax.resource.cci.Record,
                javax.resource.cci.Streamable {

        public void setName(String name);
        public void setId(String custId);
        public void setAddress(String address);

        public String getName();
        public String getId();
        public String getAddress();
}
```

- Create an empty `CustomerRecord` instance to hold output from the execution of an `Interaction`.

CustomerRecord customer = ... // create an instance

- Create a `PurchaseOrderRecord` instance as an input to the `Interaction` and set the properties on this instance. The `PurchaseOrderRecord` is another example of a custom `Record`.

```
PurchaseOrderRecord purchaseOrder = ... // create an instance
purchaseOrder.setProductName("...");
purchaseOrder.setQuantity("...");
...
```

■ Execute an `Interaction` that populates the output `CustomerRecord` instance.

```
// Execute the Interaction
boolean ret = ix.execute(ixSpec, purchaseOrder, customer);

// Check the CustomerRecord
System.out.println( customer.getName() + ":" +
                    customer.getId() + ":" +
                    customer.getAddress());
```

Metadata Annotations

This chapter defines a simplified API for development of resource adapters. The goal of the API is to simplify the development of resource adapter implementations for programmers who are just starting with resource adapters, or developing resource adapters of small to medium complexity. The existing Connector APIs remain available for use in resource adapters that require them and resource adapter implementations written to those APIs may be used in conjunction with components written to the new Connector 1.6 APIs.

18.1   Overview

The simplified API makes extensive use of Java language annotations, that was introduced in Java SE 5.0 (see "A Metadata Facility for the Java Programming Language" on page E-2). The purpose of the API is to improve the existing Connector architecture by reducing its complexity from a resource adapter developer's point of view.

The use of annotations reduces or completely eliminates the need to deal with a deployment descriptor in many cases. The use of annotations also reduces the need to keep the deployment descriptor synchronized with changes to source code.

Other component specifications in the Java EE platform, like Enterprise JavaBeans and Web Services through the annotations defined in the Web Services Metadata specification have already brought such ease of development simplifications to the developer.

18.2 Goals

The simplified API is designed with the following goals:

- Define Java language metadata annotations that can be used to annotate resource adapter artifacts. These annotations may reduce the need for implementing certain interfaces and assist in reducing the number of classes required to build a resource adapter implementation.
- Reduce the need to write redundant code by leveraging existing facilities in the Java EE platform.
- Limit and reduce the need for a deployment descriptor for common scenarios by defining related metadata annotations
- Define "programmatic defaults" to reduce the need for a resource adapter developer to develop code to represent common scenarios.

18.3 Deployment Descriptors and Annotations

Deployment descriptors are considered as an alternative to metadata annotations or as a mechanism for the overriding of metadata annotations (for example to permit the further customization of an application for a particular development environment at a later stage of the development or application assembly etc).

For additional requirements on annotations discovery and processing by the application server, see the Deployment Section of the "Application Assembly and Deployment" chapter of "Java Platform, Enterprise Edition (Java EE) Specification, version 7" on page E-1.

Deployment descriptors may be "sparse", unlike the full deployment descriptors required as part of the J2EE Connector 1.5 specification.

The specification allows a resource adapter to be developed in mixed-mode form, that is the ability for a resource adapter developer to utillize the metadata annotations defined in this chapter and the deployment descriptors in their application. When such a combination is used, if the specification does not define a particular behavior, the rules for the use of deployment descriptors as an overriding mechanism apply.

18.3.1 `metadata-complete` Deployment Descriptor Element

A new attribute, `metadata-complete`, is introduced in the Connector 1.6 deployment descriptor (`ra.xml`). The `metadata-complete` attribute defines whether the deployment descriptor for the resource adapter module is complete, or whether the class files available to the module and packaged with the resource adapter should be examined for annotations that specify deployment information.

If `metadata-complete` is set to "true", the deployment tool of the application server must ignore any annotations that specify deployment information, which might be present in the class files of the application. If `metadata-complete` is not specified or is set to "false", the deployment tool must examine the class files of the application for annotations, as specified by this specification. If the deployment descriptor is not included or is included but not marked `metadata-complete`, the deployment tool will process annotations.

Application servers must assume that `metadata-complete` is true for resource adapter modules with deployment descriptor version lower than 1.6. The following table describes the requirements for determining when to process annotations on the classes in a resource adapter archive.

TABLE 18-1 Annotation Processing Requirements for a Resource Adapter Archive

Deployment Descriptor	metadata-complete?	process annotations
Connector 1.5 ra.xml or earlier	Not applicable	No
Connector 1.6 ra.xml	True	No
Connector 1.6 ra.xml	False or Unspecified	Yes
No ra.xml bundled with the RAR module archive	Not Applicable	Yes

18.3.2 Merging Annotations and Deployment Descriptor

An application assembler or deployer may use the deployment descriptor to override the metadata annotations specified by the resource adapter developer. See the chapter titled "Resources, Naming, and Injection" in the Java EE Platform specification (see "Java Platform, Enterprise Edition (Java EE) Specification, version 7" on page E-1) for general rules on annotations and injection and override behavior. The rules below complement the rules specified in that section.

An application assembler or deployer is recommended not to override certain information specified through annotations, such as transaction support, authentication and security requirements of the resource adapter module, using the deployment descriptor. The resource adapter developer specifies this information considering the capabilities of the resource adapter, and altering these values using the deployment descriptor may not be valid or appropriate. For instance, if a resource adapter developer marks a resource adapter's transaction support level as `LocalTransaction` through annotations, since the resource adapter implementation only supports the `LocalTransaction` interface and not the `XATransaction` interface, it is incorrect and prohibited to override the transaction support level through the deployment descriptor to `XATransaction`.

When `metadata-complete` is specified as `false` or if the `metadata-complete` attribute is unspecified in the deployment descriptor, the deployment tool must examine the classes of the resource adapter for annotations. The deployment tool must follow the annotation discovery and processing requirements specified in the "Deployment Section" of the "Application Assembly and Deployment" chapter of the Java EE Platform Specification (see "Java Platform, Enterprise Edition (Java EE) Specification, version 7" on page E-1).

The information provided by the annotations must be merged with the deployment descriptor packaged along with the resource adapter module. The general rule is that uniqueness constraints specified in the deployment descriptor schema (see Section 20.7, "Resource Adapter XML Schema Definition" on page 20-24) specify what combinations of annotations and their corresponding deployment descriptor elements are allowed.

While merging the information present in the annotations and the deployment descriptor, the application server must satisfy the following requirements:

1. If a deployment descriptor element and one or more annotations specify information for the same unique identity (as specified by the XML schema), the information provided in the deployment descriptor overrides the value specified in the annotation.

2. If there is no match between the identity of the annotations and the deployment descriptor, and as long as the XML schema allow the combination of these identities, the information provided in the deployment descriptor must be considered in addition to the annotations.

3. It is an error, either by way of annotations alone or as a result of the combination of annotation and deployment descriptor, to specify combinations of identities that do not satisfy the uniqueness constraints in the deployment descriptor schema.

The application server must consider the following exceptions to the third rule above:

- If a resource adapter module specifies the fully qualified Java class name of the resource adapter class in the deployment descriptor through the `resourceadapter-class` element, the application server must ignore any `Connector` annotations in the resource adapter module's annotation discovery scope.
- If the JavaBean class specified in the `resourceadapter-class` element is annotated with the `Connector` annotation, the application server must use the information in the deployment descriptor to override the values specified in the annotation.

18.3.3 Annotation Processing Requirements of Superclasses

The following JavaBeans are permitted to have superclasses that are themselves of the same type:

- `ResourceAdapter`
- `ManagedConnectionFactory`
- `ActivationSpec`
- Administered Object

For instance, a `ResourceAdapter` JavaBean is permitted to have a superclass that is itself a `ResourceAdapter` JavaBean. See Section 5.3.1 "ResourceAdapter JavaBean and Bootstrapping a Resource Adapter Instance" for more information on `ResourceAdapter` JavaBean.

However there are no rules for processing of annotations or the deployment descriptor in these cases. For the purposes of processing the particular JavaBean, all superclass processing is identical regardless of whether the superclasses are themselves JavaBean types listed above.

In this regard, the use of JavaBean types as superclasses merely represents a convenient use of *implementation inheritance*, but does not have *component inheritance* semantics. Therefore, if a class is annotated with the `Connector` annotation, its subclass is not considered a `ResourceAdapter` JavaBean unless the subclass is also annotated with the `Connector` annotation.

However, the application server is required to process `ConfigProperty` annotations placed on the superclasses while processing the configuration properties of a JavaBean. As an example, if a subclass `MySubClass`, is annotated with the `Connector` annotation, and the application server is processing `ConfigProperty` annotations placed in the field or setter methods in the subclass, the container must also process any `ConfigProperty` annotations placed on the fields or setter methods of all superclasses of the subclass.

All the metadata annotations described in this chapter are in the
`javax.resource.spi` package unless otherwise specified. The following sections
will describe the metadata annotations that are required to be supported by the
application server.

18.4 @Connector

The `Connector` annotation is a component-defining annotation and it can be used
by the resource adapter developer to specify that the JavaBean is a resource adapter
JavaBean. The `Connector` annotation is applied to the JavaBean class and the
JavaBean class must implement the `ResourceAdapter` interface. It is recommended
that the resource adapter developer annotate at most one JavaBean with the
`Connector` annotation within the valid annotation discovery scope as defined in
Section 18.3 "Deployment Descriptors and Annotations".

If more than one JavaBean is annotated with the `Connector` annotation, the
application server must use the JavaBean class specified in the deployment
descriptor through the `resourceadapter-class` element. It is an error to provide
a resource adapter module with more than one JavaBean class annotated with the
`Connector` annotation and not providing a deployment descriptor.

CODE EXAMPLE 18-1 Connector Annotation

```
package javax.resource.spi;
@Documented
@Retention(RUNTIME)
@Target(TYPE)
public @interface Connector {
    String[] description() default {};
    String[] displayName() default {};
    String[] smallIcon() default {};
    String[] largeIcon() default {};

    String vendorName() default "";
    String eisType() default "";
    String version() default "";

    String[] licenseDescription() default {};
    boolean licenseRequired() default false;

    AuthenticationMechanism[] authMechanisms() default {};
    boolean reauthenticationSupport() default false;
```

```
        SecurityPermission[] securityPermissions() default {};

    TransactionSupport.TransactionSupportLevel transactionSupport()
 default TransactionSupport.TransactionSupportLevel.NoTransaction;

    Class<? extends WorkContext>[] requiredWorkContexts()
                                        default {};
}
```

The `smallIcon` and `largeIcon` annotation elements specifies file names for small
and a large GIF or JPEG icon images that are used to represent the resource adapter
in a GUI tool. Each `smallIcon` must be associated with a `largeIcon` element and
the application server must use the ordinal value in their respective arrays to find
the related pairs of icons.

The `vendorName` annotation element specifies the name of the resource adapter
provider vendor. The `eisType` annotation element contains information about the
type of EIS. For example, the type of an EIS can be product name of EIS independent
of any version info. This helps in identifying EIS instances that can be used with this
resource adapter.

The `licenseDescription` and `licenseRequired` annotation elements specify
licensing requirements for the resource adapter module. This type specifies whether
a license is required to deploy and use this resource adapter, and an optional
description of the licensing terms.

The `authMechanisms` element specifies the authentication mechanisms supported
by the resource adapter. See Section 18.4.3 "@AuthenticationMechanism" for more
information on the `AuthenticationMechanism` annotation. The annotation
element `reauthenticationSupport` specifies whether the resource adapter
implementation supports re-authentication of existing ManagedConnection instance.
Note that this information is for the resource adapter implementation and not for
theunderlying EIS instance.

The `securityPermissions` annotation element specifies the extended security
permissions required to be provided for the operation of the resource adapter
module. See Section 18.4.4 "@SecurityPermission" for more information on the
`SecurityPermission` annotation.

The `transactionSupport` annotation element specifies the level of transaction
support provided by the resource adapter.

The `requiredWorkContexts` annotation element specifies a list of fully qualified
classes that implements the `WorkContext` interface that a resource adapter requires
the application server to support.

18.4.1 Implementing the `ResourceAdapter` Interface

It is optional for a resource adapter implementation to bundle a JavaBean class implementing the `javax.resource.spi.ResourceAdapter` interface (see Section 5.3.1, "ResourceAdapter JavaBean and Bootstrapping a Resource Adapter Instance" on page 5-4). In particular, a resource adapter implementation that only performs outbound communication to the EIS might not provide a JavaBean that implements the `ResourceAdapter` interface or a JavaBean annotated with the `Connector` annotation.

However, if a resource adapter requires to perform tasks that uses the facilities provided by the application server through the `ResourceAdapter` interface (for example obtain a reference to the `BootstrapContext`, get lifecycle callbacks, or perform inbound message delivery), the resource adapter implementation must provide a JavaBean that implements the `ResourceAdapter` interface. The resource adapter developer may, in this case, use the `Connector` annotation or the deployment descriptor (see Section 20.4.1, "Resource Adapter Provider" on page 20-7) to specify the resource adapter JavaBean. A JavaBean that is annotated with the `Connector` annotation must implement the `ResourceAdapter` interface and must satisfy the requirements listed in Section 5.3.1, "ResourceAdapter JavaBean and Bootstrapping a Resource Adapter Instance" on page 5-4.

18.4.2 Example

A simple resource adapter JavaBean, that does not support transactions, could be defined as follows by the resource adapter provider.

CODE EXAMPLE 18-2   Connector Annotation Usage Example

```
@Connector()
public class MyResourceAdapter implements ResourceAdapter{
    // Define common configuration properties.
    ...
}
```

18.4.3 `@AuthenticationMechanism`

The `AuthenticationMechanism` annotation can be used by the developer, as part of the `Connector` annotation, to specify the authentication mechanism supported by the resource adapter (see Section 8.5.1, "Authentication Mechanism" on page 8-6).

CODE EXAMPLE 18-3   AuthenticationMechanism Annotation

```
package javax.resource.spi;

@Documented
@Retention(RUNTIME)
@Target({})
public @interface AuthenticationMechanism {
    public enum CredentialInterface {
        PasswordCredential, GSSCredential, GenericCredential
    }

    String authMechanism() default  "BasicPassword";
    String[] description() default {};
    CredentialInterface credentialInterface()
        default CredentialInterface.PasswordCredential;
}
```

The `authMechanism` annotation element specifies an authentication mechanism supported by the resource adapter. Note that this authentication support is for the resource adapter and not for the underlying EIS instance. The `CredentialInterface` enumeration is used to represent the various credential interfaces that can be used by the resource adaper to support the representation of credentials and the `credentialInterface` annotation element is used to specify the credential interface supported by the resource adapter.

18.4.4   @SecurityPermission

The `SecurityPermission` annotation can be used by the developer, as part of the `Connector` annotation, to specify the extended security permissions required by the resource adapter (see Section 20.4.1, "Resource Adapter Provider" on page 20-7).

CODE EXAMPLE 18-4   SecurityPermission annotation
```
package javax.resource.spi;

@Documented
@Retention(RUNTIME)
@Target({})
public @interface SecurityPermission {
    String[] description() default {};
    String permissionSpec() default "";
}
```

The `description` element is used to provide an optional description to mention any specific reason that a resource requires a given security permission.

The `permissionSpec` element specifies a security permission based on the Security policy file syntax. These security permissions are different from those required by the default permission set as specified in Section 21.2, "Security Permissions" on page 21-2.

18.5 @ConfigProperty

The `ConfigProperty` annotation can be used by the developer on JavaBeans listed below to indicate to the application server, that a specific JavaBean property is a configuration property for that JavaBean. A configuration property may be used by the deployer and resource adapter provider to provide additional configuration information. The `ConfigProperty` annotation may be placed on a property mutator method (the setter method) or the field corresponding to the JavaBean property.

CODE EXAMPLE 18-5   ConfigProperty Annotation

```
package javax.resource.spi;

@Documented
@Retention(RUNTIME)
@Target({FIELD, METHOD})
public @interface ConfigProperty {
    Class type() default Object.class;
    String[] description() default {};
    String defaultValue() default "";
    boolean ignore() default false;
    boolean supportsDynamicUpdates() default false;
    boolean confidential() default false;
}
```

The `type` element defines the Java type of the configuration property and the `defaultValue` element specifies the default value for the property.

When the `ConfigProperty` annotation is applied on a field, the default value of the `type` element is the type of the field. When applied on a method, the default value is the type of the JavaBean property.

For field based annotation, if the `type` element is not specified by the developer, the application server must infer its value by looking at the field declaration itself. If the defaultValue annotation element is not specified, the application server must use the value assigned to the field, if any, as the default value of the configuration property. It is an error if the value of the `type` annotation element specified by the developer in the `ConfigProperty` annotation, and the type of the field are not equal.

For setter method based annotations, if the `type` annotation element is not specified by the developer, the application server must infer its value by inspecting the method declartion. The property setter methods must follow the standard JavaBeans convention (as defined by the JavaBeans `Introspector` class). It is an error if the `type` specified by the developer in the `ConfigProperty` annotation and the type of the setter method's parameter are not equal.

The valid values of the `type` element, whether inferred or explicitly specified, must be limited to the values detailed in the documentation of the config-property-typeType element in the resource adapter XML Schema (see Section 20.7, "Resource Adapter XML Schema Definition" on page 20-24).

The `ignore` annotation element is used to indicate that the configuration tools must ignore considering the configuration property during auto-discovery of Configuration properties. (see Section 18.5.1 "Discovery of Configuration Properties").

The `supportsDynamicUpdates` and the `confidential` annotation elements provide additional metadata about the configuration property to the application server. See Section 5.3.7.6, "Configuration Property Attributes" on page 5-16 for more information on these configuration property attributes.

The application server is required to process `ConfigProperty` annotations specified in the field or setter method declaration of the following JavaBeans:

- **ResourceAdapter**. A JavaBean implementing the `ResourceAdapter` interface or a JavaBean annotated with the `Connector` annotation
- **ManagedConnectionFactory**. A JavaBean implementing the `ManagedConnectionFactory` interface or a JavaBean annotated with the `ConnectionDefinition` annotation
- **AdministeredObject**. A JavaBean annotated with the `AdministeredObject` annotation or a JavaBean specified as an administered object's implementation class using the deployment descriptor
- **ActivationSpec**. A JavaBean implementing the `ActivationSpec` interface or a JavaBean annotated with the `Activation` annotation.

These JavaBeans are still required to satisfy the JavaBean requirements listed in Section 19.3, "JavaBean Requirements" on page 19-2. The application server is required to process ConfigProperty annotations in the JavaBeans listed above irrespective of whether the JavaBeans are specified by way of deployment descriptor elements or metadata annotations.

18.5.1   Discovery of Configuration Properties

Configuration tools provided by the container must introspect the JavaBeans listed in Section 18.5 "@ConfigProperty" above for Connector 1.6 resource adapters to automatically discover the configuration properties of a JavaBean through JavaBeans introspection.

The resource adapter developer is, therefore, not required to annotate every property of the JavaBean with the `ConfigProperty` annotation. The resource adapter developer may only annotate specific properties of a JavaBean with the `ConfigProperty` annotation to specify non-default values. For example, when a property is required to be hidden from a configuration tool, the resource adapter provider need only annotate that property in the JavaBean with the `ConfigProperty` annotation and specifying the `ignore` annotation element as true.

18.6   `@ConnectionDefinition` and `@ConnectionDefinitions`

The `ConnectionDefinition` and `ConnectionDefinitions` annotations are applied to the JavaBean class and are restricted to be applied only on JavaBean classes that implement the `ManagedConnectionFactory` interface (see Section 5.3.2, "ManagedConnectionFactory JavaBean and Outbound Communication" on page 5-8).

The `ConnectionDefinition` annotation defines a set of connection interfaces and classes pertaining to a particular connection type (identical to the role played by the connection-definition element in the deployment descriptor).

The `ConnectionDefinitions` annotation can be used by the developer to specify a set of connection definitions that a `ManagedConnectionFactory` JavaBean is a part of. (see Section 20.4.1, "Resource Adapter Provider" on page 20-7 and Section 20.7, "Resource Adapter XML Schema Definition" on page 20-24 for a discussion on connection-definition).

If a `ManagedConnectionFactory` JavaBean is part of only one connection-definition, the developer may annotate that JavaBean with a `ConnectionDefinition` annotation. The `ConnectionDefinitions` annotation is intended for `ManagedConnectionFactory` JavaBeans that are part of more than one connection-definitions. It is an error to annotate a JavaBean that does not implement the `ManagedConnectionFactory` interface with either ot these two annotations.

CODE EXAMPLE 18-6   ConnectionDefinitions Annotation

```
package javax.resource.spi;

@Documented
@Retention(RUNTIME)
@Target(TYPE)

public @interface ConnectionDefinitions {
    ConnectionDefinition[] value();
}
```

CODE EXAMPLE 18-7   ConnectionDefinition Annotation

```
package javax.resource.spi;
@Documented
@Retention(RUNTIME)
@Target(TYPE)
public @interface ConnectionDefinition {
    Class connectionFactory();
    Class connectionFactoryImpl();
    Class connection();
    Class connectionImpl();
}
```

The `connectionFactory` and `connectionFactoryImpl` annotation elements are used by the developer to specify the fully qualified Java interface and implementation class for the connection factory that is supported by the resource adapter as part of the connectionDefinition.

The `connection` and `connectionImpl` annotation elements are used by the developer to specify the fully qualified Java interface and implementation class for the connection that is supported by the resource adapter as part of the connectionDefinition.

18.6.1   Example

A simple `ManagedConnectionFactory` implementation that is part of a connection definition, could be defined as follows

CODE EXAMPLE 18-8   ConnectionDefinition Annotation Usage Example

```
@ConnectionDefinition(connectionFactory=com.wombat.ra.CF.class,
                      connectionFactoryImpl=
                                    com.wombat.ra.CFImpl.class,
```

```
                    connection=com.wombat.ra.Conn.class,
                    connectionImpl=com.wombat.ra.ConnImpl.class)
public class ManagedConnectionFactoryImpl
        implements ManagedConnectionFactory {
    ...
}
```

18.7 @Activation

The `ActivationSpec` JavaBean contains the configuration information pertaining to inbound connectivity from an EIS instance. A resource adapter capable of message delivery to message endpoints must provide a JavaBean class that implements the `javax.resource.spi.ActivationSpec` interface (see Section 5.3.3, "ActivationSpec JavaBean and Inbound Communication" on page 5-9) or annotate a JavaBean with the `Activation` annotation for each supported endpoint message listener type.

The `Activation` annotation can be used by a resource adapter provider to designate a JavaBean as an `ActivationSpec` JavaBean (see Section 5.3.3 "ActivationSpec JavaBean and Inbound Communication"). The `Activation` annotation is applied to the JavaBean class. The resource adapter provider may annotate one or more JavaBeans with the `Activation` annotation. The JavaBean is required to implement the `javax.resource.spi.ActivationSpec` interface even if the JavaBean is annotated with the `Activation` annotation.

CODE EXAMPLE 18-9 Activation Annotation

```
package javax.resource.spi;

@Documented
@Retention(RUNTIME)
@Target(TYPE)
public @interface Activation {
    Class[] messageListeners();
}
```

The `messageListeners` annotation element indicates the message listener type(s) supported with the ActivationSpec JavaBean. Together with the `messageListeners` annotation element, this annotation specifies information about the specific message listener types supported by the messaging resource adapter.

18.7.1   Example

An `ActivationSpec` JavaBean that is associated with the `MyMessageListener`
message listener type, uses Bean Validation annotations and the `validate` method
to validate the state of the JavaBean could be defined as follows by the resource
adapter provider.

CODE EXAMPLE 18-10 Activation Annotation Usage Example

```
@Activation(messageListeners =
                   {com.wombat.ra.MyMessageListener.class})
public class MyActivationSpec implements ActivationSpec{
    //Use of Bean Validation annotations to express
    //validation requirements
    @Size(min=5, max=5)
    private int length;

    //... other methods
    //Use of validate() method is also allowed
    public void validate() throws InvalidPropertyException {
        //custom validation logic
    }
}
```

18.8   @AdministeredObject

The `AdministeredObject` annotation can be used by the resource adapter provider
to designate a JavaBean as an administered object (see Section 13.4.2.3,
"Administered Objects" on page 13-13). Administered objects are specific to a
messaging style or message provider.

The `AdministeredObject` annotation is applied to the JavaBean class. A resource
adapter implementation that supports inbound communication may annotate one or
more JavaBeans with the `AdministeredObject` annotation.

CODE EXAMPLE 18-11 AdministeredObject Annotation

```
package javax.resource.spi;

@Documented
@Retention(RUNTIME)
@Target(TYPE)
public @interface AdministeredObject {
    Class[] adminObjectInterfaces() default {};
```

```
    }
```

The `adminObjectInterfaces` annotation element specifies the Java type of the interface implemented by the administered object. This annotation element is optional and when this value is not provided by the resource adapter provider, the application server must use the following rules to determine the Java interfaces of the administered object:

- The following interfaces must be excluded while determining the Java interfaces of the administered object:
  - `java.io.Serializable`
  - `java.io.Externalizable`
  - `javax.resource.spi.ResourceAdapterAssociation`
- If the JavaBean implements only one interface, that interface is chosen as the Java Interface implemented by the administered object
- If the JavaBean class implements more than one Java interface, the resource adapter provider must explicitly state the interfaces supported by the administered object either through the `adminObjectInterfaces` annotation element or through the deployment descriptor. It is an error if the resource adapter provider does not use either of the two schemes to specify the Java types of the interfaces supported by the administered object.

18.9   Resource Definition Annotations

Resource definition annotations allow an application to be deployed into a Java EE environment with less administrative configuration. Refer to the section titled "Resource Definition and Configuration" in the "Resources, Naming and Injection" chapter of the Java EE Platform Specification (see "Java Platform, Enterprise Edition (Java EE) Specification, version 7" on page E-1) for an overview of resource definition annotations.

The `ConnectionFactoryDefinition` and `AdministeredObjectDefinition` annotations described below are resource definition annotations that aid the application developer in defining and configuring resource adapter related resources needed for the operational environment.

These resource definition annotations refer to a resource adapter by name, from which the resources needs to be created. The name of a resource adapter is decided as per the rules defined in the sections titled "Deploying a Stand-Alone Java EE Module" and "Deploying a Java EE Application" in the "Application Assembly and Deployment" chapter of the Java EE Platform Specification.

When a resource adapter RAR packaged within a Java EE application EAR needs to be referenced, the resource adapter name may be prefixed with a "#" character to portably refer to the embedded resource adapter within the EAR. As an example, a Servlet bundled in an enterprise archive EAR, may access the embedded resource adapter foo.rar in the EAR, by using the name "*#foo*".

These resource definition annotations must only be defined in modules that have access to the resource adapter as per the rules defined in Section 20.2.0.4, "Requirements" on page 20-5.

These resource definition annotations must be supported in all products that support the deployment process as defined by the Java EE Platform Specification, and that support the Connector API. For example, a product that includes support for both the Connector API and the Servlet API must support the use of these resource definition annotations in web applications.

It is not required to support the placement of these resource definitions in classes packaged in resource adapter modules.

18.9.1 @ConnectionFactoryDefinition

The `ConnectionFactoryDefinition` annotation is a resource definition annotation that is used to define a connector connection factory and have it registered in JNDI. See the section titled "Connection Factory Definition" in the "Resources, Naming, and Injection" chapter of the Java EE Platform Specification for more details on the connection factory resource definition annotation.

The section titled "Annotations and Deployment Descriptors" of the "Resources, Naming, and Injection" chapter of the Java EE Platform Specification describes how environment entries created by these annotations may be specified or overridden using deployment descriptor elements. The deployment descriptor element `connection-factory` that may be used to define or override the values defined in the `ConnectionFactoryDefinition` annotation is described in the section titled "Common Java EE 7 XML Schema definitions" of the Java EE Platform Specification.

CODE EXAMPLE 18-12 ConnectionFactoryDefinition Annotation

```
package javax.resource;
import java.lang.annotation.Target;
import java.lang.annotation.Retention;
import java.lang.annotation.ElementType;
import java.lang.annotation.RetentionPolicy;

@Documented
@Target({ElementType.TYPE})
```

```
@Retention(RetentionPolicy.RUNTIME)
public @interface ConnectionFactoryDefinition {
    String name();
    String description() default "";

    String resourceAdapter();
    String interfaceName();

    TransactionSupport.TransactionSupportLevel transactionSupport()
        default
        TransactionSupport.TransactionSupportLevel.NoTransaction;
    int maxPoolSize() default -1;
    int minPoolSize() default -1;
    String[] properties() default {};
}
```

The connection factory will be registered in JNDI under the name specified in the mandatory name annotation element. It may be defined to be in any valid Java EE namespace, and the namespace will determine the accessibility of the connection factory from other components. The optional description element specifies a description of the connection factory.

The name of the resource adapter that the connection factory must be created from must be indicated by the resourceAdapter element. The resource adapter must be available at runtime prior to any attempt to access the connection factory.

The mandatory interfaceName specifies the fully qualified name of the connection factory interface class. The interfaceName annotation element is used by the container to choose the appropriate connection definition included in the resource adapter, and identify the ManagedConnectionFactory that is used to create this connection factory.

The transactionSupport annotation element specifies the level of transaction support the connection factory needs to support. If a transaction support level is specified, it must be a level of transaction support whose ordinal value in the TransactionSupport.TransactionSupportLevel enum is equal to or lesser than the resource adapter's transaction support classification.

The minPoolSize annotation element specifies the minimum number of connections that should be allocated for a connection pool that backs this connnection factory resource. The maxPoolSize annotation element specifies the maximum number of connections that should be allocated for a connection pool that backs this connnection factory resource. The defaults for these attributes are vendor specific (See the section titled "Resource Definition and Configuration" in the "Resources, Naming, and Injection" of the Java EE Platform Specification for more details on these default values).

The connection factory may be configured by setting the annotation elements for the commonly used connection factory properties as indicated above. Additional properties required by the `ManagedConnectionFactory`, that is associated with the connection factory being defined, are specified through the `properties` element. Properties, if specified, that do not match configuration property names of the `ManagedConnectionFactory` or cannot be mapped to vendor-specific properties may be ignored.

18.9.1.1   Example

A XA-capable connection factory resource may be defined in a Servlet as follows:

CODE EXAMPLE 18-13 ConnectionFactoryDefinition Annotation Definition Example

```
@ConnectionFactoryDefinition(name="java:comp/eis/MyEISCF",
    interfaceName="com.eis.ConnectionFactory",
    resourceAdapter="MyEISRA",
    transactionSupport=
        TransactionSupport.TransactionSupportLevel.XATransaction)
```

Once defined, a connector connection factory resource may be referenced by a component, that has the standalone `MyEISRA` resource adapter visible to it as per the rules defined in Section 20.2.0.4, "Requirements" on page 20-5, using the `resource-ref` deployment descriptor element or the `Resource` annotation. For example, the above connection factory could be referenced as follows in a Stateless Session Bean in the same enterprise application archive.

CODE EXAMPLE 18-14 ConnectionFactoryDefinition Annotation Usage Example

```
    @Stateless
    public class MySessionBean {
        @Resource(lookup = "java:comp/eis/MyEISCF")
        com.eis.ConnectionFactory myCF;
        ...
    }
```

18.9.2 @ConnectionFactoryDefinitions

The `ConnectionFactoryDefinition` annotation is a resource definition annotation that is used to define a connector connection factory and have it registered in JNDI. Since repeated annotations are not allowed, the `ConnectionFactoryDefinitions` annotation acts as a container for multiple connector connection factory definitions.

CODE EXAMPLE 18-15 ConnectionFactoryDefinitions Annotation

```
package javax.resource;
import java.lang.annotation.Target;
import java.lang.annotation.Retention;
import java.lang.annotation.ElementType;
import java.lang.annotation.RetentionPolicy;

@Documented
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface ConnectionFactoryDefinitions {
    ConnectionFactoryDefinition[] value();
}
```

The `value` annotation element contains the multiple connector connection factory definitions.

18.9.2.1 Example

Multiple connector connection factory definitions may be declared in a Servlet as follows:

CODE EXAMPLE 18-16 ConnectionFactoryDefinitions Annotation Definition Example

```
@ConnectionFactoryDefinitions({
    @ConnectionFactoryDefinition(name="java:comp/eis/MyXACF",
        interfaceName="com.eis.FooConnectionFactory",
        resourceAdapter="MyEISRA1",
        transactionSupport=XATransaction),
    @ConnectionFactoryDefinition(name="java:comp/eis/MyNoTXCF",
        interfaceName="com.eis.BarConnectionFactory",
        resourceAdapter="MyEISRA2",
        transactionSupport=NoTransaction)
})
```

Once defined, the connector connection factory resources may be referenced by a component, that has the standalone `MyEISRA1` and `MyEISRA2` resource adapters visible to it as per the rules defined in Section 20.2.0.4, "Requirements" on page 20-5, using the `resource-ref` deployment descriptor element or the `Resource` annotation. For example, the above connection factories could be referenced as follows in a Stateless Session Bean in the same enterprise application archive.

CODE EXAMPLE 18-17 ConnectionFactoryDefinitions Annotation Usage Example

```
@Stateless
public class MySessionBean {
    @Resource(lookup = "java:comp/eis/MyXACF")
    com.eis.FooConnectionFactory xacf;
    ...
    @Resource(lookup = "java:comp/eis/MyNoTXCF")
    com.eis.BarConnectionFactory notxcf;
    ...
}
```

18.9.3 @AdministeredObjectDefinition

The `AdministeredObjectDefinition` annotation is a resource definition annotation that is used to define an administered object and have it registered in JNDI. See the section titled "Connector Administered Object Definition" in the "Resources, Naming, and Injection" chapter of the Java EE Platform for more details on the administered object definition annotation.

The section titled "Annotations and Deployment Descriptors" of the "Resources, Naming, and Injection" chapter of the Java EE Platform Specification describes how environment entries created by these annotations may be specified or overridden using deployment descriptor elements. The deployment descriptor element `administered-object` that may be used to define or override the values defined in the `AdministeredObjectDefinition` annotation, is described in the section titled "Common Java EE 7 XML Schema definitions" of the Java EE Platform Specification.

CODE EXAMPLE 18-18 AdministeredObjectDefinition Annotation

```
package javax.resource;
import java.lang.annotation.Target;
import java.lang.annotation.Retention;
import java.lang.annotation.ElementType;
import java.lang.annotation.RetentionPolicy;

@Documented
@Target({ElementType.TYPE})
```

```
@Retention(RetentionPolicy.RUNTIME)
public @interface AdministeredObjectDefinition {
    String name();
    String description() default "";

    String resourceAdapter();
    String className();
    String interfaceName() default "";

    String[] properties() default {};
}
```

The administered object will be registered in JNDI under the name specified in the mandatory `name` annotation element. It may be defined to be in any valid Java EE namespace, and the namespace will determine the accessibility of the administered object from other components. The optional `description` element specifies a description of the administered object.

The name of the resource adapter that the administered object must be created from must be indicated by the `resourceAdapter` element. The resource adapter must be available at runtime prior to any attempt to access the administered object.

The mandatory fully qualified name of the administered object's class must be indicated by the `className` element. The fully qualified name of the administered object's interface must be indicated by the `interfaceName` element, only if the class indicated in the `className` element implements more than one interface and the application server cannot determine the unique Java interface of the administered object according the rules defined in Section 18.8, "@AdministeredObject" on page 18-15.

Additional properties required to be configured in the administered object are specified through the `properties` element. Properties, if specified, that do not match configuration property names of the `AdministeredObject` JavaBean or cannot be mapped to vendor-specific properties may be ignored.

18.9.3.1   Example

A Queue Administered Object resource of an embedded JMS resource adapter may be defined in a Servlet as follows:

CODE EXAMPLE 18-19 AdministeredObjectDefinition Annotation Definition Example

```
@AdministeredObjectDefinition (name="java:comp/eis/MyQueue",
    className="com.wombat.connector.jms.QueueImpl",
    resourceAdapter="#MyJMSRA")
```

Once defined, the Queue resource may be referenced by a component, that has the embedded MyJMSRA resource adapter visible to it as per the rules defined in Section 20.2.0.4, "Requirements" on page 20-5, using the resource-ref deployment descriptor element or the Resource annotation. For example, the above administered object definition could be referenced as follows in a Stateless Session Bean in the same enterprise application EAR archive.

CODE EXAMPLE 18-20 AdministeredObjectDefinition Annotation Usage Example

```
@Stateless
public class MySessionBean {
    @Resource(lookup = "java:comp/eis/MyQueue")
    javax.jms.Queue myQ;
    ...
}
```

18.9.4   @AdministeredObjectDefinitions

The AdministeredObjectDefinition annotation is a resource definition annotation that is used to define an administered object and have it registered in JNDI. Since repeated annotations are not allowed, the AdministeredObjectDefinitions annotation acts as a container for multiple administered object definitions.

CODE EXAMPLE 18-21 AdministeredObjectDefinitions Annotation

```
package javax.resource;
import java.lang.annotation.Target;
import java.lang.annotation.Retention;
import java.lang.annotation.ElementType;
import java.lang.annotation.RetentionPolicy;

@Documented
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface AdministeredObjectDefinitions {
    AdministeredObjectDefinition[] value();
}
```

The value annotation element contains the multiple administered object definitions.

18.9.4.1   Example

Multiple administered object definitions, for instance a Queue and a Topic administered object, may be declared together in a Servlet as follows:

CODE EXAMPLE 18-22 AdministeredObjectDefinitions Annotation Definition Example

```
@AdministeredObjectDefinitions({
    @AdministeredObjectDefinition (name="java:comp/eis/MyQueue",
        className="com.wombat.connector.jms.QueueImpl",
        resourceAdapter="MyJMSRA"),
    @AdministeredObjectDefinition (name="java:comp/eis/MyTopic",
        className="com.wombat.connector.jms.TopicImpl",
        resourceAdapter="MyJMSRA")
})
```

Once defined, the Queue and Topic administered resources may be referenced by a component, that has the standalone `MyJMSRA` resource adapter visible to it as per the rules defined in Section 20.2.0.4, "Requirements" on page 20-5, using the `resource-ref` deployment descriptor element or the `Resource` annotation. For example, the above administered objects could be referenced as follows in a Stateless Session Bean in the same enterprise application archive.

CODE EXAMPLE 18-23 AdministeredObjectDefinitions Annotation Usage Example

```
    @Stateless
    public class MySessionBean {
        @Resource(lookup = "java:comp/eis/MyQueue")
        javax.jms.Queue myQ;
        ...
        @Resource(lookup = "java:comp/eis/MyTopic")
        javax.jms.Queue myT;
        ...
    }
```

CHAPTER **19**

API Requirements

This chapter specifies the API requirements for the resource adapter and application server implementations.

19.1   Requirements of the Application Server

- The application server must support the deployment of a resource adapter in EJB and Web containers.
- The application server must support all the connector architecture API requirements in EJB and Web containers.
- A single resource adapter instance may be shared by both a Web container and an EJB container.
- The application server must support all versions of the resource adapter DTDs (Document Type Definitions) and the resource adapter XML Schema Definition. This ensures that resource adapters written to previous versions of this specification can be deployed on products supporting the current version of this specification.

19.2   Requirements of the Resource adapter

The following matrix specifies the required (+) and optional (?) API requirements on a resource adapter.

LM - Lifecycle management contract

WM - Work management contract

MI - Message Inflow contract

TI - Transaction Inflow contract

CM - Connection management contract

TM- Transaction management contract

SM - Security management contract

CCI - Common Client Interface

TABLE 19-1   Resource Adapter API Requirements

	LM	WM	MI	TI	CM	TM	SM	CCI
Outbound	?	?			+	+	+	?
Inbound	+	?	+	?				
Bi-directional	+	?	+	?	+	+	+	?

Note, the message inflow contract must be supported by an inbound resource adapter.

19.3   JavaBean Requirements

The various JavaBean implementations provided by a resource adapter must adhere to the following rules:

- A JavaBean implementation must contain a null constructor.
- A JavaBean implementation must provide getter and setter methods, to access and modify the public properties of the JavaBean instance.

Note, for JavaBean serialization, implementing the `java.io.Serializable` interface is not necessary. The XML long-term persistence mechanism introduced in J2SE 1.4 can save the state of a JavaBean in an XML format that is resilient to version changes in the implementation of that JavaBean. Refer to Java SE (see "Java Platform, Standard Edition 7API Specification" on page E-1) classes `java.beans.XMLEncoder`, `java.beans.XMLEncoder`, and `java.beans.PersistenceDelegate`.

For details, refer to JavaBeans specification (see "JavaBeans Specification 1.01 Final Release" on page E-2).

19.4 Equality Constraints

This section specifies the equality constraints on object implementations of the various types defined by this specification.

19.4.1 Equality based on Java Object Identity

The candidate objects are implementations of `MessageEndpointFactory`, `ActivationSpec`, `ManagedConnection` types.

These objects, in general, should not override the default `equals` and `hashCode` methods. However, if these methods are overridden, they must preserve the equality constraints based on Java object identity; that is, no two objects are considered equal.

19.4.2 Equality Based on Config Properties and Class Information

The candidate objects are implementations of `ResourceAdapter`, `ManagedConnectionFactory`, `ConnectionRequestInfo`, `java.security.Principal`, `org.ietf.jgss.GSSCredential`, `GenericCredential`, `PasswordCredential`, and `Record` types.

These objects must override the default `equals` and `hashCode` methods, and provide an equality behavior based on the configuration properties and class information. That is, any two objects can be equal only if their configuration properties match and they have the same class implementation.

Packaging Requirements

This chapter specifies requirements for packaging and deploying a resource adapter. These requirements support a modular, portable deployment of a resource adapter into a Java EE compliant application server.

20.1   Overview

A resource adapter provider develops a set of Java interfaces and classes as part of its implementation of a resource adapter. These Java classes implement connector architecture-specified contracts and EIS-specific functionality provided by the resource adapter. The development of a resource adapter may also require the use of native libraries specific to the underlying EIS.

The Java interfaces and classes are packaged together (with required native libraries, help files, documentation, and other resources) with a deployment descriptor to create a resource adapter module. A deployment descriptor defines the contract between a resource adapter provider and a deployer for the deployment of a resource adapter. With the introduction of a simplified API through the use of Java language annotations described in Chapter 18, "Metadata Annotations", it is optional for a resource adapter provider and a deployer to provide a deployment descriptor along with a resource adapter module.

FIGURE 20-1 Packaging and Deployment Lifecycle of a Resource adapter
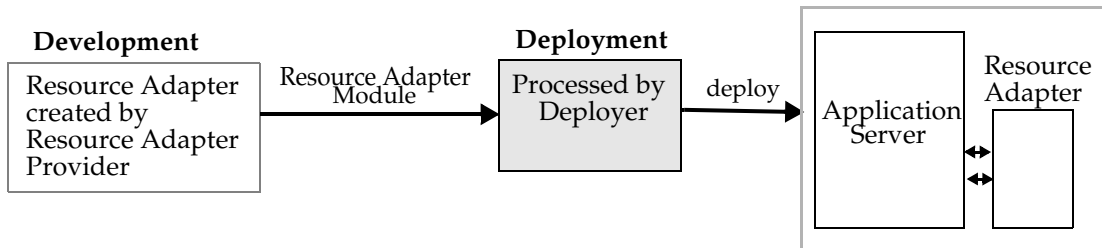
A resource adapter module corresponds to a Java EE module in terms of the Java EE composition hierarchy. Refer to the Java EE Platform specification (see "Java Platform, Enterprise Edition (Java EE) Specification, version 7" on page E-1) for more details on the deployment of Java EE modules and applications. A Java EE module represents the basic unit of composition of a Java EE application. Examples of Java EE modules include EJB modules, application client modules, and web client modules.

A resource adapter module must be deployed either:

- Directly into an application server as a stand-alone unit or,
- Deployed with a Java EE application that consists of one or more Java EE modules in addition to a resource adapter module. The Java EE specification specifies requirements for the assembly and packaging of Java EE applications.

The following figure shows the composition model of a resource adapter module with other Java EE modules.

FIGURE 20-2 Deployment of a Resource Adapter Module
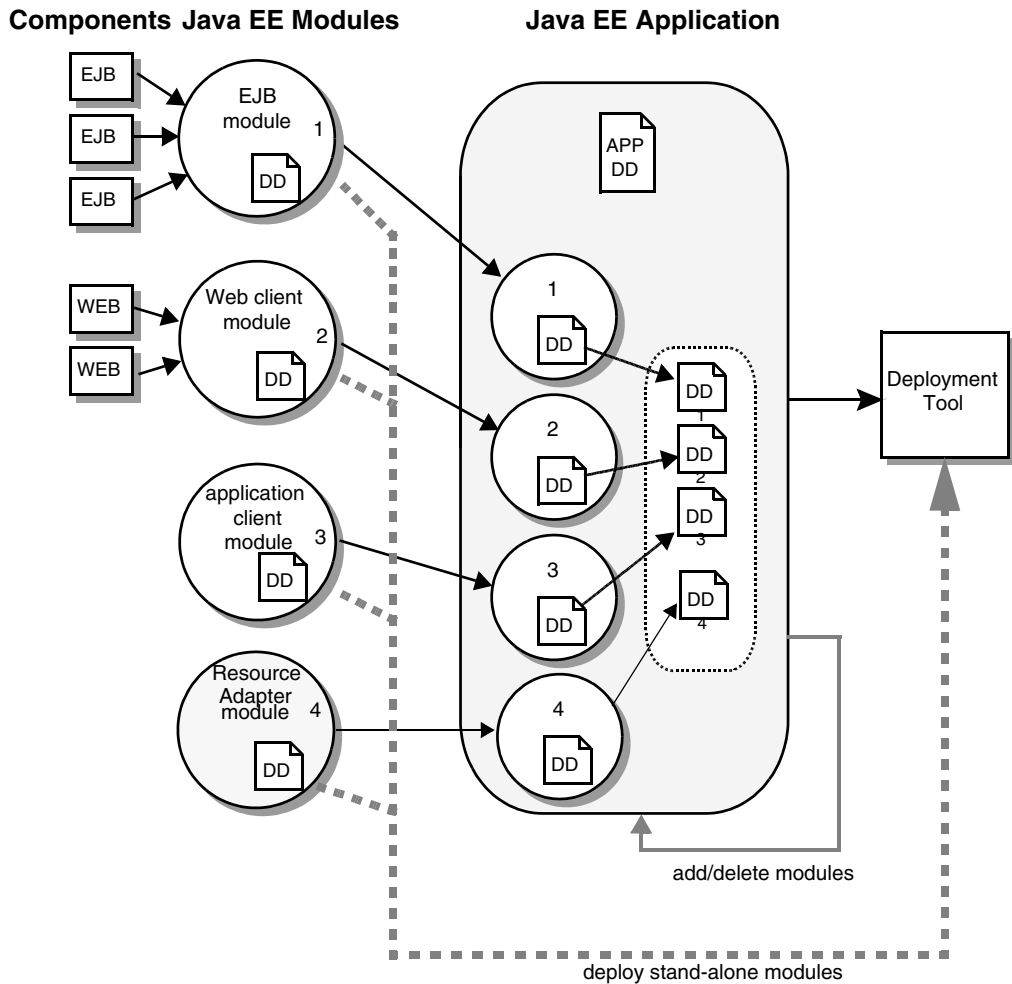
The stand-alone deployment of a resource adapter module into an application server is typically done to support scenarios in which multiple Java EE applications share a single resource adapter module. However, in certain scenarios, a resource adapter

module is required only by components within a single Java EE application. The deployment option of a resource adapter module bundled with a Java EE application supports the latter scenario.

At deployment time, a resource adapter deployer deploys a resource adapter module to an application server.

20.2 Packaging

The file format for a packaged resource adapter module defines the contract between a resource adapter provider and deployer.

A packaged resource adapter includes the following elements:

- Java classes and interfaces that are required for the implementation of both the connector architecture contracts and the functionality of the resource adapter.
- Utility Java classes for the resource adapter.
- Platform-dependent native libraries required by the resource adapter.
- Help files and documentation.
- Descriptive meta information that ties the above elements together.

20.2.0.1 Resource Adapter Archive

A resource adapter must be packaged using the Java Archive (JAR) format in to an RAR (resource adapter archive). For example, a resource adapter for EIS A can be packaged as an archive with a filename eisA.rar.

The RAR file may contain a deployment descriptor based on the format specified in Section 20.6.4 "Requirements". If a resource adapter module chooses to bundle a deployment descriptor, the deployment descriptor must be stored with the name META-INF/ra.xml in the RAR file.

The Java interfaces, implementation, and utility classes required by the resource adapter must be packaged as one or more JAR files as part of the resource adapter module. A JAR file must use the .jar file extension.

The resource adapter may also use the library support mechanisms described in the Java EE Platform Specification to specify library dependencies. See the "Java Platform, Enterprise Edition (Java EE) Specification, version 7" on page E-1 for more information on the Java EE Platform's support for libraries.

The platform-specific libraries required by the resource adapter must be packaged with the resource adapter module.

20.2.0.2   RAR Contents

The following table describes the contents of a RAR file, where each element is located within the RAR file and whether they are required.

TABLE 20-1   Description of RAR File Contents

Contents of RAR file	Requirements	Relative Location Within RAR File
Deployment Descriptor	Optional	`META-INF/ra.xml`
howto.html, image files, locale files, etc.	Optional	Arbitrary (that is, could be at root level or at a sub-level).
JAR files	Optional	Arbitrary
Platform-specific native libraries	Optional	Arbitrary

20.2.0.3   Sample Directory Structure

The following lists the files in a sample resource adapter module:

```
META-INF/ra.xml
howto.html
images/icon.jpg
ra.jar
cci.jar
win.dll
solaris.so
```

In the above example, `ra.xml` is the deployment descriptor. `ra.jar` and `cci.jar` contain Java interfaces and implementation classes for the resource adapter. `win.dll` and `solaris.so` are examples of native libraries.

Note that a resource adapter module can be structured such that various elements are partitioned using subdirectories.

20.2.0.4   Requirements

- When a standalone resource adapter RAR is deployed, the resource adapter may be made available to all Java EE applications in the application server. The application server, however, must make the standalone resource adapter RAR available to applications that meet the requirements listed in Section 20.3 "Class Loading Requirements".

- When a resource adapter `RAR` packaged within a Java EE application `EAR` (also referred to as an "embedded RAR") is deployed, the resource adapter must be made available only to the Java EE application with which it is packaged.

20.3   Class Loading Requirements

This specification does not define the exact arrangement or hierarchy of classloaders that must be used by a container. This section of the specification defines the requirements in terms of what applications must have visibility to a resource adapter RAR.

A resource adapter RAR packaged within a Java EE application EAR, as specified in Section 20.2.0.4 "Requirements" above, must be made available only to the Java EE application with which it is packaged.

The requirements below specify the applications that must have visibility to a standalone resource adapter RAR.

- If an application references a resource using a deployment descriptor entry or a corresponding annotation, and that resource is supplied by a standalone resource adapter, that standalone resource adapter must be made available to the application.
- If an application references an extension using the Extension Mechanism Architecture (see the section titled "Library Support" in the "Application Assembly and Deployment" chapter of the "Java Platform, Enterprise Edition (Java EE) Specification, version 7" on page E-1) and a jar file within a standalone resource adapter supplies that extension, the standalone resource adapter must be made available to the application.
- If a standalone resource adapter is configured to deliver messages to a message-driven bean in an application, the standalone resource adapter must be made available to the application.
- Even lacking such a reference, it must be possible for the Deployer to configure an application so that any particular standalone resource adapter is available to the application.

An application that satisfy the requirements can portably assume the visibility of the corresponding standalone resource adapter RAR. An application server may choose to make all deployed standalone resource adapter RARs available to all applications.

20.4   Deployment

A deployment descriptor defines the contract between a resource adapter provider and a deployer. It captures the declarative information that is intended for the deployer to enable deployment of a resource adapter in a target operational environment. Deployment information may also provided by the metadata annotations described in Chapter 18, "Metadata Annotations". The container is required to follow the rules defined in Section 18.3, "Deployment Descriptors and Annotations" on page 18-2 to derive the final deployment information.

A resource adapter module must be deployed based on the deployment requirements specified by the resource adapter provider in the deployment descriptor and through metadata annotations. Section 20.7 "Resource Adapter XML Schema Definition" specifies the XML Schema for the deployment descriptor for a resource adapter module. See Chapter 18, "Metadata Annotations" for more information on the metadata annotations that can be employed.

The J2EE Deployment API Specification (see "J2EE Deployment API Specification" on page E-2) describes the general deployment procedure in detail.

20.4.1   Resource Adapter Provider

The resource adapter provider is responsible for specifying the deployment descriptor for a resource adapter.

The resource adapter provider may specify the following information in the deployment descriptor or through metadata-annotations:

- **General information:** The resource adapter provider should specify the following general information:
  - Name of the resource adapter.
  - Description of the resource adapter.
  - URI of a UI icon for the resource adapter.
  - Name of the vendor who provides the resource adapter.
  - Licensing requirement and description. Note that the management of licensing is outside the scope of the connector architecture.
  - Type of the EIS system supported. For example, the name of a specific database, ERP system, or mainframe TP system without any versioning information.
  - Version of the connector architecture specification, represented as a string, supported by the resource adapter.
  - Version of the resource adapter represented as a string

- Required `WorkContext` classes: A resource adapter may optionally provide a list of `required-work-context` elements representing a list of `WorkContext` classes that a resource adapter requires the application server to support. . The resource adapter provider must specify the name of a Java `Class` that implements the `javax.resource.spi.work.WorkContext` interface.
- **ResourceAdapter class:** The resource adapter provider must specify, if available, the name of a Java class that implements the `javax.resource.spi.ResourceAdapter` interface. The implementation of this class must be a JavaBean. A `ResourceAdapter` JavaBean is configured by the resource adapter deployer during deployment. The application server must instantiate exactly one `ResourceAdapter` JavaBean per functional resource adapter instance. The application server must create at least one functional resource adapter instance per resource adapter deployment. The configuration properties are specific to a resource adapter.
- **ResourceAdapter class configuration properties:** The resource adapter provider may optionally provide a set of configuration properties for the `ResourceAdapter` instance, which may be used by the resource adapter deployer to configure a `ResourceAdapter` JavaBean instance.
- **Outbound resource adapter information:**
  - `ManagedConnectionFactory` **class**: The resource adapter provider must specify the name of the Java class that implements the `javax.resource.spi.ManagedConnectionFactory` interface. The implementation must be a JavaBean.

    Typically, a `ManagedConnectionFactory` class is used to produce `ConnectionFactory` and `Connection` objects of a particular type. In order to produce objects of different types, a separate `ManagedConnectionFactory` class can be used for each supported type. The deployment descriptor element `connection-definition` can be used to specify different `ManagedConnectionFactory` classes, each pertaining to a particular type.

  - `ConnectionFactory` **interface and implementation class**: The resource adapter provider must specify the fully-qualified name of the Java interface and implementation class for each connection factory supported by the resource adapter.
  - `Connection` **interface and implementation class**: The resource adapter provider must specify the fully-qualified name of the Java interface and implementation class for each connection supported by the resource adapter.
  - **Transactional support**: The resource adapter provider must specify the level of transaction support provided by the resource adapter implementation. The level of transaction support must be any one of the following: `NoTransaction`, `LocalTransaction`, or `XATransaction`. Note that this support is specified for a resource adapter and not for the underlying EIS instance.
    `NoTransaction`: The resource adapter does not support either the resource manager local or JTA transactions. It does not implement either `XAResource` or `LocalTransaction` interfaces.

`LocalTransaction`: The resource adapter supports resource manager local transactions by implementing the `LocalTransaction` interface. The local transaction management contract is specified in Section 7.7, "Local Transaction Management Contract" on page 7-28.

`XATransaction`: The resource adapter supports both resource manager local and JTA transactions by implementing the `LocalTransaction` and `XAResource` interfaces respectively. The requirements for supporting the `XAResource` based contract are specified in Section 7.6, "XAResource-based Transaction Contract" on page 7-15.

- **Configurable properties per** `ManagedConnectionFactory` **instance**: The resource adapter provider specifies the name, type, description, and an optional default value for the properties that have to be configured on a per `ManagedConnectionFactory` instance.

  Each `ManagedConnectionFactory` instance creates connections to a specific EIS instance based on the properties configured on the `ManagedConnectionFactory` instance. The configurable properties are specified only once in the deployment descriptor, even though a resource adapter can be used to configure multiple `ManagedConnnectionFactory` instances that create connections to different instances of the same underlying EIS type.

- **Authentication mechanism**: The resource adapter provider must specify all authentication mechanisms supported by the resource adapter. This includes the support provided by the resource adapter implementation but not by the underlying EIS instance. The standard values are: `BasicPassword` and `Kerbv5`. A resource adapter may support one or more of these authentication mechanisms.

  - `BasicPassword`: user-password based authentication mechanism that is specific to an EIS.
  - `Kerbv5`: Kerberos version 5 based authentication mechanism.

  If no authentication mechanism is specified as part of the deployment descriptor, the resource adapter supports no standard security authentication mechanism as part of the security contract.

- **Reauthentication support**: The resource adapter provider must specify whether a resource adapter supports re-authentication of an existing physical connection.
- **Extended security permissions**: The security permissions listed in the deployment descriptor are different from those required by the default permission set. Refer to Chapter 21, "Runtime Environment" for more details on security permissions.

- Inbound resource adapter information

  - **Message listener type**: The resource adapter provider must specify one or more message listener types supported by a messaging resource adapter. The message listener type is the name of the Java type of a message listener interface.

- ActivationSpec class: The resource adapter provider must specify the Java class name of the activation specification class. The implementation of this class must be a JavaBean. An `ActivationSpec` specifies an activation specification per message listener type. The `ActivationSpec` is configured by a message endpoint deployer during application deployment.
- Required ActivationSpec properties: The resource adapter provider may optionally specify a set of required properties for an `ActivationSpec`. This is useful in validating the `ActivationSpec` during endpoint application deployment.
- ActivationSpec class configuration properties: The resource adapter provider may optionally provide a set of configuration properties for the `ActivationSpec` instance, which may be used by the resource adapter deployer to configure a `ActivationSpec` JavaBean instance

- **Administered objects**: The resource adapter provider must specify the name of the Java type of the interface implemented by an administered object, which must be a JavaBean, and its Java class name. Administered objects are specific to a messaging style or message provider. There may be zero to more administered objects specified. There must be no more than one administered object definition with the same interface and Class name combination in a resource adapter.

The deployment descriptor specified by the resource adapter provider for its resource adapter must be consistent with the XML Schema specified in Section 20.7 "Resource Adapter XML Schema Definition". Metadata annotations are detailed in Chapter 18, "Metadata Annotations".

Note – The connector architecture does not specify standard deployment properties for the configuration of non-Java parts, such as native libraries, of a resource adapter. This applies only to the properties of the non-Java part not exposed through the Java part of the resource adapter. The non-Java part of a resource adapter should be configured using mechanisms specific to a resource adapter.

20.4.2   Deployer

During resource adapter deployment, the deployer is responsible for configuring a resource adapter. The configuration of a resource adapter is based on the properties defined in the deployment descriptor and metadata annotations (see Section 18.3, "Deployment Descriptors and Annotations" on page 18-2) as part of the resource adapter module.

20.4.2.1 Standalone Resource Adapter Module

During deployment, the deployer configures and deploys a resource adapter based on the deployment descriptor information. The deployer may choose to override the information in the deployment descriptor.

20.4.2.2 Resource Adapter Module with Java EE Application

Refer to the Java EE platform specification (see "Java Platform, Enterprise Edition (Java EE) Specification, version 7" on page E-1) for the requirements specified for the deployment of a Java EE application.

20.4.2.3 Configuration

To configure a resource adapter, the deployer must configure a `ResourceAdapter` JavaBean instance. The configuration properties are specific to a resource adapter. In the case of outbound resource adapters, the deployer must do the following tasks:

- Configure one or more property sets (one property set per `ManagedConnectionFactory` instance) for creating connections to various underlying EIS instances. The deployer creates a property set to set valid values for various configurable fields. The configuration of each field is based on the name, type and description of the field specified in the deployment descriptor or metadata annotations described in Chapter 18, "Metadata Annotations".

  Each property set represents a specific configuration to be set on a `Managed-ConnectionFactory` instance for creating connections to a specific EIS instance. Since a resource adapter may be used to create connections to multiple instances of the same EIS, there can be multiple property sets for a single resource adapter, one for each configured `ManagedConnectionFactory` instance.

- Configure application server mechanisms for transaction management based on the level of transaction support specified by the resource adapter.
- Configure security in the target operational environment based on the security requirements specified by the resource adapter in its deployment descriptor or annotations discussed in Section 18.4.4, "@SecurityPermission" on page 18-9.

20.4.2.4 Security Configuration

The security configuration is based on:

- Whether the resource adapter supports a specific authentication mechanism and credentials interface. The deployment descriptor includes an element `authentication-mechanism` that specifies a supported authentication mechanism and the corresponding credentials interface.

- Whether the application server is configured to support a specific mechanism type. For example, if the application server is not configured for the Kerberos mechanism, it is not capable of passing Kerberos credentials to the resource adapter as part of the security contract.

During the deployment, the deployer may, though is not required to, check whether or not an underlying EIS supports the same capabilities, such as transaction support and authentication mechanisms, as the corresponding resource adapter.

For example, if a resource adapter provides implementation support for Kerberos based authentication but the underlying EIS instance does not support Kerberos, the deployer may decide not to configure Kerberos for authentication to this EIS instance. However if the deployer does not perform such checks during deployment, any invalid configurations should lead to runtime exceptions.

20.5   Interfaces/Classes

This section specifies the Java classes and interfaces related to the configuration of a resource adapter in an operational environment.

20.5.1   ResourceAdapter

The Java class which implements the interface `javax.resource.spi.ResourceAdapter` must be a JavaBean. The `ResourceAdapter` JavaBean may also be specified through the `Connector` annotation (see Section 18.4, "@Connector" on page 18-6).

A `ResourceAdapter` JavaBean represents exactly one functional resource adapter unit or instance. The application server must instantiate exactly one `ResourceAdapter` JavaBean per functional resource adapter instance. The application server must create at least one functional resource adapter instance per resource adapter deployment. A `ResourceAdapter` JavaBean instance is configured by the resource adapter deployer during deployment. The configuration properties are specific to a resource adapter.

The resource adapter provider may optionally provide a set of configuration properties, specified in the resource adapter deployment descriptor, for the `ResourceAdapter` instance, which is used by the resource adapter deployer to configure the `ResourceAdapter` JavaBean instance during deployment. The deployer may override the configuration information in the deployment descriptor while configuring the `ResourceAdapter` JavaBean instance.

20.5.1.1 Requirements

The `ResourceAdapter` implementation must be a JavaBean.

20.5.2 `ManagedConnectionFactory`

The class that implements the `ManagedConnectionFactory` interface supports a set of properties. These properties provide information required by the `ManagedConnectionFactory` for the creation of physical connections to the underlying EIS.

A resource adapter must implement the `ManagedConnectionFactory` interface as a JavaBean. As a JavaBean implementor, the resource adapter can also provide a `BeanInfo` class that implements the `java.beans.BeanInfo` interface and provides explicit information about the methods and properties supported by the `ManagedConnectionFactory` implementation class.

The implementation of `ManagedConnectionFactory` as a JavaBean improves the ability of tools that are based on the JavaBeans framework to manage the configuration of `ManagedConnectionFactory` instances.

20.5.2.1 Requirements

The `ManagedConnectionFactory` implementation must be a JavaBean. The `ManagedConnectionFactory` implementation may also be annotated with the `ConnectionDefinition` annotation (see Section 18.6, "@ConnectionDefinition and @ConnectionDefinitions" on page 18-12). Any specified `ManagedConnectionFactory` property in the deployment descriptor which does not have a matching property in the `ManagedConnectionFactory` JavaBean should be treated as an error.

20.5.3 Properties Conventions

The `ManagedConnectionFactory` implementation class must provide getter and setter methods for each of its supported properties. The supported properties must be consistent with the specification of configurable properties specified in the deployment descriptor. With the introduction of metadata annotations for specifying configuration properties, the resource adapter provider is not required to specify the configuration properties through the deployment descriptor and may use the `ConfigProperty` annotation (see Section 18.5, "@ConfigProperty" on page 18-10)

instead. The container is also required to discover configuration properties of a JavaBean. See Section 18.5.1, "Discovery of Configuration Properties" on page 18-12 for more information.

The getter and setter methods convention must be based on the JavaBeans design pattern. These methods are defined in the implementation class and not in the `ManagedConnectionFactory` interface. This requirement keeps the `ManagedConnectionFactory` interface independent of any resource adapter or EIS-specific properties.

20.5.4 Standard Properties

The connector architecture identifies a standard set of properties common across various types of resource adapters and EISs. A resource adapter is not required to support a standard property if that property does not apply to its configuration.

These standard properties are defined as follows:

TABLE 20-2   Standard Properties of the Connector Architecture

Property	Description
ServerName	Name of the server for the EIS instance.
PortNumber	Port number for establishing a connection to an EIS instance.
UserName	Name of the user establishing a connection to an EIS instance.
Password	Password for the user establishing a connection.
ConnectionURL	URL for the EIS instance to which it connects.

In addition to these standard properties, a `ManagedConnectionFactory` implementation class may support properties specific to a resource adapter and its underlying EIS.

All properties are administered by the deployer and are not visible to an application component provider.

The specified properties are required to be implemented as either bound or constrained properties. Refer to the JavaBeans specification (`http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html`) for details on bound and constrained properties.

In the XML deployment descriptor, any bounds or well-defined values of properties should be described in the `description` element. With the support for Bean Validation (see Section 5.3.7.5, "JavaBean Validation" on page 5-15), the resource adapter provider is not required to describe the bounds and constraints of properties and may use the Bean Validation annotations to describe bounds and constraints.

20.6 JNDI Configuration and Lookup

This section specifies requirements for the configuration of the JNDI environment for a resource adapter.

In both managed and non-managed application scenarios, an application component or application client must look up a connection factory instance in the component's environment using the JNDI interface. The application component then uses the connection factory instance to get a connection to the underlying EIS. Section 6.4, "Application Programming Model" on page 6-5 specifies the application programming model in more detail.

The following code extract shows the JNDI lookup of a `javax.resource.cci.ConnectionFactory` instance.

```
// Application Component/Client Code
obtain the initial JNDI context
Context initctx = new InitialContext();

// perform JNDI lookup to obtain connection factory
javax.resource.cci.ConnectionFactory cxf =

(javax.resource.cci.ConnectionFactory)initctx.lookup("java:comp/
env/eis/MyEIS");

javax.resource.cci.Connection cx = cxf.getConnection();
```

20.6.1 Responsibilities

In both managed and non-managed environments, registration of a connection factory instance in the JNDI namespace must use either the JNDI `Reference` or `Serializable` mechanism.

The choice between the two JNDI mechanisms depends on:

- Whether the JNDI provider being used supports a specific mechanism.
- Whether the application server and resource adapter provide the necessary support, specified in the respective requirements.
- Constraints on the size of serialized objects that can be stored in the JNDI namespace. The reference mechanism allows only a reference to the actual object to be stored in the JNDI namespace. This is preferable to the serializable mechanism, which stores the whole serialized object in the namespace.

This section specifies the responsibilities of the roles involved in the JNDI configuration of a resource adapter.

20.6.1.1 Deployer

The deployer is responsible for configuring connection factory instances in the JNDI environment. The deployer should manage the JNDI namespace such that the same programming model, as shown in Section 20.6 "JNDI Configuration and Lookup", for the JNDI-based connection factory lookup is supported in both managed and non-managed environments.

20.6.1.2 Resource Adapter

The implementation class for a connection factory interface must implement both the `java.io.Serializable` and `javax.resource.Referenceable` interfaces to support JNDI registration.

The following code extract shows the `javax.resource.Referenceable` interface:

```
public interface javax.resource.Referenceable
          extends javax.naming.Referenceable {
      public void setReference(Reference ref);
}
```

The `ManagedConnectionFactory` implementation class must implement the `java.io.Serializable` interface.

To support the `Reference` mechanism in a non-managed environment, a resource adapter or a helper class must provide an implementation of the `javax.naming.spi.ObjectFactory` interface.

20.6.1.3 Application Server

The implementation class for `javax.resource.spi.ConnectionManager` must implement the `java.io.Serializable` interface.

An application server must provide an implementation class for the `javax.naming.spi.ObjectFactory` interface to support JNDI `Reference` mechanism-based connection factory lookup. The implementation of this interface is application server-specific.

Section 20.6.3 "Scenario: Referenceable" specifies more details on `Reference` mechanism-based JNDI configuration in a managed environment.

20.6.2   Scenario: `Serializable`

The implementation classes for both the
`javax.resource.cci.ConnectionFactory` and
`javax.resource.spi.ManagedConnectionFactory` interfaces implement the
`java.io.Serializable` interface.

The deployment code retrieves the configuration properties from the XML
deployment descriptor or the metadata annotations (see Section 18.5,
"@ConfigProperty" on page 18-10) for the resource adapter. The deployment code
then creates an instance of the `ManagedConnectionFactory` implementation class
and configures the properties of the instance.

```
// Deployment Code
// Create an instance of the ManagedConnectionFactory class
com.myeis.ManagedConnectionFactoryImpl mcf =
        new com.myeis.ManagedConnectionFactoryImpl();

// Set the properties of the ManagedConnectionFactory instance
// Note: Properties are defined in the implementation class and
// not in the
// javax.resource.spi.ManagedConnectionFactory interface
mcf.setServerName("...");
mcf.setPortNumber("...");
...
```

Note that in a non-managed environment, an application developer writes the
deployment code. In a managed environment, the deployment tool typically hides
the deployment code.

The deployment code uses the `ManagedConnectionFactory` instance to create a connection factory instance. The code then registers the connection factory instance in the JNDI namespace.

```
// Deployment Code
// In a managed environment, create a ConnectionManager specific to
// the application server. Note that in a non-managed environment,
// ConnectionManager will be specific to the resource adapter.
com.wombatserver.ConnectionManager cm =
          new com.wombatserver.ConnectionManager(...);

// Create an instance of a connection factory
Object cxf = mcf.createConnectionFactory(cm);

// Get the JNDI context
javax.naming.Context ctx = new javax.naming.InitialContext(env);

// Bind to the JNDI namespace specifying a factory name
ctx.bind("...", cxf);
```

When an application component does a JNDI lookup of a connection factory instance, the returned connection factory instance should get associated with a configured `ManagedConnectionFactory` instance and a `ConnectionManager` instance. The implementation class for connection factory should achieve the association between these instances in an implementation-specific manner.

The following section illustrates JNDI configuration in a managed environment based on the `Reference` mechanism. This section uses the CCI interfaces `javax.resource.cci.ConnectionFactory` and `javax.resource.cci.Connection` as the connection factory and connection interfaces respectively.

20.6.3 Scenario: `Referenceable`

The implementation class for the `ConnectionFactory` interface implements `javax.resource.Referenceable` shown in the following code extract. Refer to the JNDI specification for more details on the `Referenceable` interface.

```
public class com.myeis.ConnectionFactoryImpl implements
            javax.resource.Referenceable,
            java.io.Serializable,
            javax.resource.cci.ConnectionFactory {

    // Reference to this ConnectionFactory
    javax.naming.Reference reference;

    // setReference is called by the deployment code
    public void setReference(Reference ref) {
        reference = ref;
    }

    // getReference is called by the JNDI provider during
    // Context.bind
    public Reference getReference() throws NamingException {
        return reference;
    }
    ...
}
```

The `getReference` method on the `ConnectionFactory` implementation class must return a non-null value or throw `javax.naming.NamingException`.

20.6.3.1 `ObjectFactory` Implementation

An application server provides a class (in an application server-specific implementation) that implements the `javax.naming.spi.ObjectFactory` interface. Refer to the JNDI specification for more details on the `ObjectFactory` interface.

In the `ObjectFactory.getObjectInstance` method, the information carried by the `Reference` parameter (set in the `ConnectionFactoryImpl.setReference` method) is used to lookup the property set to be configured on the target `ManagedConnectionFactory` instance.

The mapping from a `Reference` instance to multiple configured property sets enables an application server to configure multiple `ManagedConnectionFactory` instances with respective property sets. An application server maintains the property set configuration in an implementation-specific way based on the deployment descriptor specification and metadata annotations.

The implementation and structure of `Reference` is specific to an application server. The following code extract is an illustrative example. It illustrates an implementation of the `ObjectFactory.getObjectInstance` method:

```
public class com.wombatserver.ApplicationServerJNDIHandler
        implements javax.naming.spi.ObjectFactory {
    // ...
    public Object getObjectInstance(Object obj, Name name,
            Context ctx, Hashtable env)
                throws Exception {

        javax.naming.Reference ref =
                (javax.naming.Reference)obj;

        // Using the information carried by the Reference
        // instance,
        // (<referenceName, logicalName> in this example) lookup
        // a configured property set and then configure a
        // ManagedConnectionFactory instance with specified
        // properties.
        ... // [implementation specific]
        //
        // For example, instantiation of the
        // ManagedConnectionFactory
        // class and invocation of its setter method
        // can be done using the Java Reflection mechanism.

        javax.resource.spi.ManagedConnectionFactory mcf = ...

        // Create a Connection Manager instance specific to the
        // application server
        com.wombatserver.ConnectionManager cxManager = ...

        // Create a connection factory instance.
        // The ConnectionManager instance provided by the
        // application
        // server gets associated with the created
        // connection factory instance
        return mcf.createConnectionFactory(cxManager);
    }
    ...
}
```

20.6.3.2 Deployment

The following deployment code shows the registration of a reference to a connection factory instance in the JNDI namespace:

```
// Deployment Code
javax.naming.Context ctx = new javax.naming.InitialContext(env);

// Create an instance of the connection factory
com.myeis.ConnectionFactoryImpl cf =
          new com.myeis.ConnectionFactoryImpl();

// Create a reference for the ConnectionFactory instance
javax.naming.Reference ref = new javax.naming.Reference(
            ConnectionFactoryImpl.class.getName(),
            new javax.naming.StringRefAddr(
              "<referenceName>", "<logicalName>"),
            ApplicationServerJNDIHandler.class.getName(),
            null);

cf.setReference(ref);

// Bind to the JNDI namespace specifying a name for the connection
// factory
ctx.bind("...", cf);
```

Note that the deployment code should be designed as generic, though the above example does not show it that way. The code should dynamically create an instance of a connection factory, create a `Reference` instance, and then set the reference.

The `Context.bind` method registers a `Reference` to the connection factory instance in the JNDI namespace.

20.6.3.3 Scenario: Connection Factory Lookup

The following steps occur when an application component calls the method JNDI `Context.lookup` to lookup a connection factory instance:

1. JNDI passes control to the application server. The `ObjectFactory-.getObjectInstance` method implemented by the application server is called.

2. The application server creates a new instance of the `ManagedConnectionFactory` implementation class provided by the resource adapter. The application server must follow the requirements in Section 5.3.2, "ManagedConnectionFactory JavaBean and Outbound Communication" on page 5-8 and Section 5.3.7.3, "ManagedConnectionFactory JavaBean Instance

Configuration" on page 5-14 while configuring a `ManagedConnectionFactory` JavaBean instance. The application server may use an existing instance of the `ManagedConnectionFactory` implementation class, if available.

3. The application server calls setter methods on the `ManagedConnectionFactory` instance to set various configuration properties of this instance. These properties provide information required by the `ManagedConnectionFactory` instance to create physical connections to the underlying EIS. The application server uses an existing property set configured during the deployment of a resource adapter to set the required properties of the `ManagedConnectionFactory` instance.

4. After the newly created `ManagedConnectionFactory` instance has been configured with its properties set, the application server creates a new `ConnectionManager` instance.

5. The application server calls the `createConnectionFactory` method of the `ManagedConnectionFactory` instance, passing in the `ConnectionManager` instance from the previous step, to get a `ConnectionFactory` instance.

6. The application server returns the connection factory instance to the JNDI provider, so that this instance can be returned as a result of the JNDI lookup. The application component gets the `ConnectionFactory` instance as a result of the JNDI lookup.

FIGURE 20-3 OID:Lookup of a ConnectionFactory Instance from JNDI

Application
Component

JNDI Naming   Application
Context   Server

                    Connection
                    Manager

Resource Adapter

ManagedConnectionFactory
Implementation
Class

Initial configuration of the resource adapter, followed by the
application deployment

The application server maintains the configuration properties of
the ManagedConnectionFactory instance in an implementation-
specific way based on the XML deployment descriptor or
annotation

lookup

JNDI passes control to the
application server

create a new instance or use an existing instance

set properties by calling setter methods

create a new instance

createConnectionFactory(ConnectionManager)

ManagedConnectionFactory-
creates a ConnectionFactory
instance and returns it to the
application server

The application server returns the
connection factory instance.
This instance is the one returned
from JNDI lookup.

return ConnectionFactory

20.6.4   Requirements

The default configuration values for the various JavaBean classes specified in the resource adapter deployment descriptor by way of the config-property element or through the ConfigProperty annotation, override and take precedence over the defaults specified for the same classes by the resource adapter developer through the JavaBean mechanism. Note, a deployer may finally override such default configuration information while configuring the various JavaBean instances.

20.7   Resource Adapter XML Schema Definition

This section specifies the XML Schema Definition (XSD) for the deployment descriptor for a resource adapter. Some of the types used in this XSD are defined in the Java EE platform specification (Section 8., "Java Platform, Enterprise Edition (Java EE) Specification, version 7" on page E-1). The comments in the XSD specify additional requirements for syntax and semantics that cannot be specified by using the XML Schema language. Note, the description-group element defined in javaee_7.xsd allows multiple descriptions, in order to allow specifying the same description for different locales.

CODE EXAMPLE 20-1   Schema Definition for the Deployment Descriptor for a Resource Adapter

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://xmlns.jcp.org/xml/ns/javaee"
        xmlns:javaee="http://xmlns.jcp.org/xml/ns/javaee"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        elementFormDefault="qualified"
        attributeFormDefault="unqualified"
        version="1.7">
  <xsd:annotation>
    <xsd:documentation>
      $Id$
    </xsd:documentation>
  </xsd:annotation>

  <xsd:annotation>
    <xsd:documentation>
      <![CDATA[

    This is the XML Schema for the Connector 1.7 deployment
    descriptor.  The deployment descriptor must be named
```

CODE EXAMPLE 20-1   Schema Definition for the Deployment Descriptor for a Resource Adapter

```
    "META-INF/ra.xml" in the connector's rar file.  All Connector
    deployment descriptors must indicate the connector resource
    adapter schema by using the Java EE namespace:

    http://xmlns.jcp.org/xml/ns/javaee

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

        <connector xmlns="http://xmlns.jcp.org/xml/ns/javaee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
          http://xmlns.jcp.org/xml/ns/javaee/connector_1_7.xsd"
          version="1.7">
          ...
        </connector>

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

    http://xmlns.jcp.org/xml/ns/javaee/connector_1_7.xsd

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

<xsd:annotation>
  <xsd:documentation>

    The following conventions apply to all Java EE
    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="javaee_7.xsd"/>
```

CODE EXAMPLE 20-1   Schema Definition for the Deployment Descriptor for a Resource Adapter

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


  <xsd:element name="connector" type="javaee:connectorType">
    <xsd:annotation>
      <xsd:documentation>

    The connector element is the root element of the deployment
    descriptor for the resource adapter. This element includes
    general information - vendor name, resource adapter version,
    icon - about the resource adapter module. It also includes
    information specific to the implementation of the resource
    adapter library as specified through the element
    resourceadapter.

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

  </xsd:element>

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

  <xsd:complexType name="activationspecType">
    <xsd:annotation>
      <xsd:documentation>

    The activationspecType specifies an activation
    specification.  The information includes
    Java class name of an activation specification and a set of
    required configuration property names.

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

    <xsd:sequence>
      <xsd:element name="activationspec-class"
       type="javaee:fully-qualified-classType">
    <xsd:annotation>
      <xsd:documentation>
        <![CDATA[

          The element activationspec-class specifies the fully
          qualified Java class name of the activation
          specification class. This class must implement the
          javax.resource.spi.ActivationSpec interface. The
          implementation of this class is required to be a
```

CODE EXAMPLE 20-1   Schema Definition for the Deployment Descriptor for a Resource Adapter

```
            JavaBean.

            Example:
        <activationspec-class>com.wombat.ActivationSpecImpl
        </activationspec-class>

            ]]>
        </xsd:documentation>
    </xsd:annotation>
        </xsd:element>
        <xsd:element name="required-config-property"
         type="javaee:required-config-propertyType"
         minOccurs="0"
         maxOccurs="unbounded"/>
        <xsd:element name="config-property"
         type="javaee:config-propertyType"
         minOccurs="0"
         maxOccurs="unbounded">
  <xsd:annotation>
  <xsd:documentation>
     The required-config-property element is deprecated since
     Connector 1.6 specification. The resource adapter
     implementation is recommended to use the @NotNull
     Bean Validation annotation or its XML validation
     descriptor equivalent to indicate that a configuration
     property is required to be specified by the deployer.
     See the Connectors specification for more information.
  </xsd:documentation>
  </xsd:annotation>

        </xsd:element>

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

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

  <xsd:complexType name="adminobjectType">
    <xsd:annotation>
      <xsd:documentation>

     The adminobjectType specifies information about an
     administered object.  Administered objects are specific to a
     messaging style or message provider.  This contains
     information on the Java type of the interface implemented by
     an administered object, its Java class name and its
```

CODE EXAMPLE 20-1   Schema Definition for the Deployment Descriptor for a Resource Adapter

```
    configuration properties.

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

    <xsd:sequence>
      <xsd:element name="adminobject-interface"
       type="javaee:fully-qualified-classType">
  <xsd:annotation>
    <xsd:documentation>
      <![CDATA[

          The element adminobject-interface specifies the
           name of the Java type of the
          interface implemented by an administered object.

          Example:
  <adminobject-interface>javax.jms.Destination
  </adminobject-interface>

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

      </xsd:element>
      <xsd:element name="adminobject-class"
       type="javaee:fully-qualified-classType">
  <xsd:annotation>
    <xsd:documentation>
      <![CDATA[

          The element adminobject-class specifies the fully
          qualified Java class name of an administered object.

          Example:
      <adminobject-class>com.wombat.DestinationImpl
      </adminobject-class>

          ]]>
    </xsd:documentation>
  </xsd:annotation>
      </xsd:element>

      <xsd:element name="config-property"
       type="javaee:config-propertyType"
       minOccurs="0"
       maxOccurs="unbounded"/>
```

CODE EXAMPLE 20-1   Schema Definition for the Deployment Descriptor for a Resource Adapter

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

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

  <xsd:complexType name="authentication-mechanismType">
    <xsd:annotation>
      <xsd:documentation>

   The authentication-mechanismType specifies an authentication
   mechanism supported by the resource adapter. Note that this
   support is for the resource adapter and not for the
   underlying EIS instance. The optional description specifies
   any resource adapter specific requirement for the support of
   security contract and authentication mechanism.

   Note that BasicPassword mechanism type should support the
   javax.resource.spi.security.PasswordCredential interface.
   The Kerbv5 mechanism type should support the
   org.ietf.jgss.GSSCredential interface or the deprecated
   javax.resource.spi.security.GenericCredential interface.

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

    <xsd:sequence>
      <xsd:element name="description"
       type="javaee:descriptionType"
       minOccurs="0"
       maxOccurs="unbounded"/>
      <xsd:element name="authentication-mechanism-type"
       type="javaee:xsdStringType">
    <xsd:annotation>
      <xsd:documentation>
        <![CDATA[

          The element authentication-mechanism-type specifies
          type of an authentication mechanism.

          The example values are:

          <authentication-mechanism-type>BasicPassword
          </authentication-mechanism-type>

          <authentication-mechanism-type>Kerbv5
          </authentication-mechanism-type>
```

CODE EXAMPLE 20-1   Schema Definition for the Deployment Descriptor for a Resource Adapter

```
          Any additional security mechanisms are outside the
          scope of the Connector architecture specification.

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

      </xsd:element>
      <xsd:element name="credential-interface"
       type="javaee:credential-interfaceType"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

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

  <xsd:complexType name="config-property-nameType">
    <xsd:annotation>
      <xsd:documentation>
    <![CDATA[

      The config-property-nameType contains the name of a
      configuration property.

      The connector architecture defines a set of well-defined
      properties all of type java.lang.String. These are as
      follows.

            ServerName
            PortNumber
            UserName
            Password
            ConnectionURL

      A resource adapter provider can extend this property set to
      include properties specific to the resource adapter and its
      underlying EIS.

      Possible values include
      ServerName
      PortNumber
      UserName
      Password
      ConnectionURL

      Example: <config-property-name>ServerName</config-property-name>
```

CODE EXAMPLE 20-1   Schema Definition for the Deployment Descriptor for a Resource Adapter

```
      ]]>
      </xsd:documentation>
    </xsd:annotation>
    <xsd:simpleContent>
      <xsd:restriction base="javaee:xsdStringType"/>
    </xsd:simpleContent>
  </xsd:complexType>

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

  <xsd:complexType name="config-property-typeType">
    <xsd:annotation>
      <xsd:documentation>
    <![CDATA[

      The config-property-typeType contains the fully
      qualified Java type of a configuration property.

      The following are the legal values:
          java.lang.Boolean, java.lang.String, java.lang.Integer,
          java.lang.Double, java.lang.Byte, java.lang.Short,
          java.lang.Long, java.lang.Float, java.lang.Character

      Used in: config-property

      Example:
      <config-property-type>java.lang.String</config-property-type>

      ]]>
      </xsd:documentation>
    </xsd:annotation>
    <xsd:simpleContent>
      <xsd:restriction base="javaee:string">
    <xsd:enumeration value="java.lang.Boolean"/>
    <xsd:enumeration value="java.lang.String"/>
    <xsd:enumeration value="java.lang.Integer"/>
    <xsd:enumeration value="java.lang.Double"/>
    <xsd:enumeration value="java.lang.Byte"/>
    <xsd:enumeration value="java.lang.Short"/>
    <xsd:enumeration value="java.lang.Long"/>
    <xsd:enumeration value="java.lang.Float"/>
    <xsd:enumeration value="java.lang.Character"/>
      </xsd:restriction>
    </xsd:simpleContent>
  </xsd:complexType>
```

CODE EXAMPLE 20-1   Schema Definition for the Deployment Descriptor for a Resource Adapter

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

  <xsd:complexType name="config-propertyType">
   <xsd:annotation>
     <xsd:documentation>

  The config-propertyType contains a declaration of a single
  configuration property that may be used for providing
  configuration information.

  The declaration consists of an optional description, name,
  type and an optional value of the configuration property. If
  the resource adapter provider does not specify a value than
  the deployer is responsible for providing a valid value for
  a configuration property.

  Any bounds or well-defined values of properties should be
  described in the description element.

     </xsd:documentation>
   </xsd:annotation>
   <xsd:sequence>
     <xsd:element name="description"
      type="javaee:descriptionType"
      minOccurs="0"
      maxOccurs="unbounded"/>
     <xsd:element name="config-property-name"
      type="javaee:config-property-nameType"/>
     <xsd:element name="config-property-type"
      type="javaee:config-property-typeType"/>
     <xsd:element name="config-property-value"
      type="javaee:xsdStringType"
      minOccurs="0">
   <xsd:annotation>
     <xsd:documentation>
       <![CDATA[

         The element config-property-value contains the value
         of a configuration entry. Note, it is possible for a
         resource adapter deployer to override this
         configuration information during deployment.

         Example:
         <config-property-value>WombatServer</config-property-value>

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

CODE EXAMPLE 20-1   Schema Definition for the Deployment Descriptor for a Resource Adapter

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

    <xsd:element name="config-property-ignore" type="javaee:true-falseType"
    minOccurs="0" maxOccurs="1">
    <xsd:annotation>
        <xsd:documentation>
        <![CDATA[

          The element config-property-ignore is used to specify
          whether the configuration tools must ignore considering the
          configuration property during auto-discovery of
          Configuration properties. See the Connector specification for
          more details. If unspecified, the container must not ignore
          the configuration property during auto-discovery.
          This element must be one of the following, "true" or "false".

          ]]>
       </xsd:documentation>
    </xsd:annotation>
    </xsd:element>
    <xsd:element name="config-property-supports-dynamic-updates" type
"javaee:true-falseType"
    minOccurs="0" maxOccurs="1">
    <xsd:annotation>
        <xsd:documentation>
        <![CDATA[
         The element config-property-supports-dynamic-updates is used to specify
          whether the configuration property allows its value to be updated, by
          application server's configuration tools, during the lifetime of
          the JavaBean instance. See the Connector specification for
          more details. If unspecified, the container must not dynamically
          reconfigure the property.
          This element must be one of the following, "true" or "false".

          ]]>
       </xsd:documentation>
    </xsd:annotation>
    </xsd:element>

    <xsd:element name="config-property-confidential" type="javaee:true-
falseType"
    minOccurs="0" maxOccurs="1">
    <xsd:annotation>
        <xsd:documentation>
        <![CDATA[
          The element config-property-confidential is used to specify
```

```
                whether the configuration property is confidential and
                recommends application server's configuration tools to use special
                visual aids for editing them. See the Connector specification for
                more details. If unspecified, the container must not treat the
                property as confidential.

                This element must be one of the following, "true" or "false".
                ]]>
         </xsd:documentation>
      </xsd:annotation>
      </xsd:element>



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

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

   <xsd:complexType name="connection-definitionType">
      <xsd:annotation>
         <xsd:documentation>

      The connection-definitionType defines a set of connection
      interfaces and classes pertaining to a particular connection
      type. This also includes configurable properties for
      ManagedConnectionFactory instances that may be produced out
      of this set.

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

      <xsd:sequence>
         <xsd:element name="managedconnectionfactory-class"
          type="javaee:fully-qualified-classType">
      <xsd:annotation>
         <xsd:documentation>
            <![CDATA[

            The element managedconnectionfactory-class specifies
            the  name of the Java class that
            implements the
            javax.resource.spi.ManagedConnectionFactory interface.
            This Java class is provided as part of resource
            adapter's implementation of connector architecture
            specified contracts. The implementation of this
            class is required to be a JavaBean.
```

CODE EXAMPLE 20-1   Schema Definition for the Deployment Descriptor for a Resource Adapter

```
        Example:
        <managedconnectionfactory-class>
  com.wombat.ManagedConnectionFactoryImpl
        </managedconnectionfactory-class>

        ]]>
  </xsd:documentation>
</xsd:annotation>
  </xsd:element>
  <xsd:element name="config-property"
   type="javaee:config-propertyType"
   minOccurs="0"
   maxOccurs="unbounded"/>
  <xsd:element name="connectionfactory-interface"
   type="javaee:fully-qualified-classType">
<xsd:annotation>
  <xsd:documentation>
    <![CDATA[

      The element connectionfactory-interface specifies
      the  name of the ConnectionFactory
      interface supported by the resource adapter.

      Example:
      <connectionfactory-interface>com.wombat.ConnectionFactory
      </connectionfactory-interface>

      OR

      <connectionfactory-interface>javax.resource.cci.ConnectionFactory
      </connectionfactory-interface>

      ]]>
  </xsd:documentation>
</xsd:annotation>
  </xsd:element>

  <xsd:element name="connectionfactory-impl-class"
   type="javaee:fully-qualified-classType">
<xsd:annotation>
  <xsd:documentation>
    <![CDATA[

      The element connectionfactory-impl-class specifies
      the  name of the ConnectionFactory
      class that implements resource adapter
```

CODE EXAMPLE 20-1   Schema Definition for the Deployment Descriptor for a Resource Adapter

```
            specific ConnectionFactory interface.

            Example:

            <connectionfactory-impl-class>com.wombat.ConnectionFactoryImpl
            </connectionfactory-impl-class>

            ]]>
      </xsd:documentation>
    </xsd:annotation>
      </xsd:element>
      <xsd:element name="connection-interface"
       type="javaee:fully-qualified-classType">
    <xsd:annotation>
      <xsd:documentation>
        <![CDATA[

            The connection-interface element specifies the fully
            qualified name of the Connection interface supported
            by the resource adapter.

            Example:

      <connection-interface>javax.resource.cci.Connection
      </connection-interface>

            ]]>
      </xsd:documentation>
    </xsd:annotation>
      </xsd:element>
      <xsd:element name="connection-impl-class"
       type="javaee:fully-qualified-classType">
    <xsd:annotation>
      <xsd:documentation>
        <![CDATA[

            The connection-impl-classType specifies the fully
            qualified name of the Connection class that
            implements resource adapter specific Connection
            interface.  It is used by the connection-impl-class
            elements.

            Example:

      <connection-impl-class>com.wombat.ConnectionImpl
      </connection-impl-class>
```

CODE EXAMPLE 20-1   Schema Definition for the Deployment Descriptor for a Resource Adapter

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

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

  <xsd:complexType name="connectorType">
    <xsd:annotation>
      <xsd:documentation>

    The connectorType defines a resource adapter.

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

    <xsd:sequence>
      <xsd:element name="module-name" type="javaee:string" minOccurs="0">
        <xsd:annotation>
          <xsd:documentation>

            The element module-name specifies the name of the
            resource adapter.

            If there is no module-name specified, the module-name
            is determined as defined in Section EE.8.1.1 and EE.8.1.2
            of the Java Platform, Enterprise Edition (Java EE)
            Specification, version 6.
          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:group ref="javaee:descriptionGroup"/>
      <xsd:element name="vendor-name"
       type="javaee:xsdStringType" minOccurs="0">
    <xsd:annotation>
      <xsd:documentation>

        The element vendor-name specifies the name of
        resource adapter provider vendor.

      If there is no vendor-name specified, the application server
      server must consider the default "" (empty string) as
      the name of the resource adapter provider vendor.
```

CODE EXAMPLE 20-1   Schema Definition for the Deployment Descriptor for a Resource Adapter

```
      </xsd:documentation>
    </xsd:annotation>
    </xsd:element>
    <xsd:element name="eis-type"
     type="javaee:xsdStringType" minOccurs="0">
<xsd:annotation>
  <xsd:documentation>

    The element eis-type contains information about the
    type of the EIS. For example, the type of an EIS can
    be product name of EIS independent of any version
    info.

    This helps in identifying EIS instances that can be
    used with this resource adapter.

 If there is no eis-type specified, the application
 server must consider the default "" (empty string) as
 the type of the EIS.

  </xsd:documentation>
</xsd:annotation>
    </xsd:element>
    <xsd:element name="resourceadapter-version"
     type="javaee:xsdStringType" minOccurs="0">
<xsd:annotation>
  <xsd:documentation>

    The element resourceadapter-version specifies a string-based version
    of the resource adapter from the resource adapter
    provider.

    If there is no resourceadapter-version specified, the application
    server must consider the default "" (empty string) as
    the version of the resource adapter.

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

    </xsd:element>
    <xsd:element name="license"
     type="javaee:licenseType"
     minOccurs="0"/>
    <xsd:element name="resourceadapter"
     type="javaee:resourceadapterType"/>

    <xsd:element name="required-work-context"
```

CODE EXAMPLE 20-1 Schema Definition for the Deployment Descriptor for a Resource Adapter

```
         type="javaee:fully-qualified-classType"
         minOccurs="0"
         maxOccurs="unbounded">
      <xsd:annotation>
       <xsd:documentation>

         The element required-work-context specifies the
          class name that implements WorkContext interface,
         that a resource adapter requires the application server to support.

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

       </xsd:element>

     </xsd:sequence>
     <xsd:attribute name="version"
        type="javaee:dewey-versionType"
        fixed="1.7"
        use="required">
       <xsd:annotation>
     <xsd:documentation>

      The version indicates the version of the schema to be used by the
      deployment tool. This attribute does not have a default, and the
      resource adapter developer/deployer is required to specify it.
      This attribute allows the deployment tool to choose which
      schema to validate the descriptor against.


     </xsd:documentation>
       </xsd:annotation>
     </xsd:attribute>
     <xsd:attribute name="metadata-complete" type="xsd:boolean">
     <xsd:annotation>
        <xsd:documentation>
           The metadata-complete attribute defines whether the deployment
           descriptor for the resource adapter module is complete, or whether
           the class files available to the module and packaged with the resource
           adapter should be examined for annotations that specify deployment
           information.

           If metadata-complete is set to "true", the deployment tool of the
           application server must ignore any annotations that specify
           deployment information, which might be present in the class files of
           the application.If metadata-complete is not specified or is set to
```

```
                "false", the deployment tool must examine the class files of the
             application for annotations, as specified by this specification. If
             the deployment descriptor is not included or is included but not
              marked metadata-complete, the deployment tool will process
             annotations

              Application servers must assume that metadata-complete is true for
              resource adapter modules with deployment descriptor version
              lower than 1.6.
          </xsd:documentation>
      </xsd:annotation>
      </xsd:attribute>

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

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

   <xsd:complexType name="credential-interfaceType">
     <xsd:annotation>
       <xsd:documentation>

     The credential-interfaceType specifies the
     interface that the resource adapter implementation
     supports for the representation of the
     credentials. This element(s) that use this type,
     i.e. credential-interface,  should be used by
     application server to find out the Credential
     interface it should use as part of the security
     contract.

     The possible values are:

     javax.resource.spi.security.PasswordCredential
     org.ietf.jgss.GSSCredential
     javax.resource.spi.security.GenericCredential

       </xsd:documentation>
     </xsd:annotation>
     <xsd:simpleContent>
       <xsd:restriction base="javaee:fully-qualified-classType">
     <xsd:enumeration
         value="javax.resource.spi.security.PasswordCredential"/>
     <xsd:enumeration
         value="org.ietf.jgss.GSSCredential"/>
     <xsd:enumeration
         value="javax.resource.spi.security.GenericCredential"/>
```

CODE EXAMPLE 20-1   Schema Definition for the Deployment Descriptor for a Resource Adapter

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

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

  <xsd:complexType name="inbound-resourceadapterType">
    <xsd:annotation>
      <xsd:documentation>

    The inbound-resourceadapterType specifies information
    about an inbound resource adapter. This contains information
    specific to the implementation of the resource adapter
    library as specified through the messageadapter element.

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

    <xsd:sequence>
      <xsd:element name="messageadapter"
       type="javaee:messageadapterType"
       minOccurs="0">
    <xsd:unique name="messagelistener-type-uniqueness">
      <xsd:annotation>
        <xsd:documentation>

          The messagelistener-type element content must be
          unique in the messageadapter. Several messagelisteners
          can not use the same messagelistener-type.

        </xsd:documentation>
      </xsd:annotation>
      <xsd:selector xpath="javaee:messagelistener"/>
      <xsd:field    xpath="javaee:messagelistener-type"/>
    </xsd:unique>
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

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

  <xsd:complexType name="licenseType">
    <xsd:annotation>
      <xsd:documentation>

    The licenseType specifies licensing requirements for the
```

```
       resource adapter module. This type specifies whether a
       license is required to deploy and use this resource adapter,
       and an optional description of the licensing terms
       (examples: duration of license, number of connection
       restrictions). It is used by the license element.

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

       <xsd:sequence>
         <xsd:element name="description"
          type="javaee:descriptionType"
          minOccurs="0"
          maxOccurs="unbounded"/>
         <xsd:element name="license-required"
          type="javaee:true-falseType">
       <xsd:annotation>
         <xsd:documentation>

           The element license-required specifies whether a
           license is required to deploy and use the
           resource adapter. This element must be one of
           the following, "true" or "false".

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

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

   <xsd:complexType name="messageadapterType">
     <xsd:annotation>
       <xsd:documentation>

       The messageadapterType specifies information about the
       messaging capabilities of the resource adapter. This
       contains information specific to the implementation of the
       resource adapter library as specified through the
       messagelistener element.

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

       <xsd:sequence>
```

CODE EXAMPLE 20-1   Schema Definition for the Deployment Descriptor for a Resource Adapter

```
      <xsd:element name="messagelistener"
       type="javaee:messagelistenerType"
       maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

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

  <xsd:complexType name="messagelistenerType">
    <xsd:annotation>
      <xsd:documentation>

    The messagelistenerType specifies information about a
    specific message listener supported by the messaging
    resource adapter. It contains information on the Java type
    of the message listener interface and an activation
    specification.

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

    <xsd:sequence>
      <xsd:element name="messagelistener-type"
       type="javaee:fully-qualified-classType">
    <xsd:annotation>
      <xsd:documentation>
        <![CDATA[

          The element messagelistener-type specifies the fully
          qualified name of the Java type of a message
          listener interface.

          Example:

    <messagelistener-type>javax.jms.MessageListener
    </messagelistener-type>

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

      </xsd:element>
      <xsd:element name="activationspec"
       type="javaee:activationspecType"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
```

CODE EXAMPLE 20-1   Schema Definition for the Deployment Descriptor for a Resource Adapter

```
   </xsd:complexType>

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

  <xsd:complexType name="outbound-resourceadapterType">
    <xsd:annotation>
      <xsd:documentation>

   The outbound-resourceadapterType specifies information about
   an outbound resource adapter. The information includes fully
   qualified names of classes/interfaces required as part of
   the connector architecture specified contracts for
   connection management, level of transaction support
   provided, one or more authentication mechanisms supported
   and additional required security permissions.

    If any of the outbound resource adapter elements (transaction-support,
    authentication-mechanism, reauthentication-support) is specified through
    this element or metadata annotations, and no  connection-definition is
    specified as part of this element or through annotations, the
    application server must consider this an error and fail deployment.

   If there is no authentication-mechanism specified as part of
   this or metadata annotations, then the resource adapter does not
   support any standard security authentication mechanisms as
   part of security contract. The application server ignores
   the security part of the system contracts in this case.

    If there is no transaction-support specified as part of
    this element, then the application server must
    consider that the resource adapter does not support either
    the resource manager local or JTA transactions and must consider
    the transaction support as NoTransaction. Note that resource adapters may
     specify the level of transaction support to be used at runtime for a
     ManagedConnectionFactory through the TransactionSupport interface.

    If there is no reauthentication-support specified as part of
    this element or metadata annotation, then the application server must
    consider that the resource adapter does not support re-authentication of
    ManagedConnections.

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

    <xsd:sequence>
      <xsd:element name="connection-definition"
       type="javaee:connection-definitionType"
```

CODE EXAMPLE 20-1   Schema Definition for the Deployment Descriptor for a Resource Adapter

```
      maxOccurs="unbounded" minOccurs="0"/>
     <xsd:element name="transaction-support"
      type="javaee:transaction-supportType" minOccurs="0"/>
     <xsd:element name="authentication-mechanism"
      type="javaee:authentication-mechanismType"
      minOccurs="0"
      maxOccurs="unbounded"/>
     <xsd:element name="reauthentication-support"
      type="javaee:true-falseType" minOccurs="0"/>
   <xsd:annotation>
     <xsd:documentation>

       The element reauthentication-support specifies
       whether the resource adapter implementation supports
       re-authentication of existing Managed- Connection
       instance. Note that this information is for the
       resource adapter implementation and not for the
       underlying EIS instance. This element must have
       either a "true" or "false" value.

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

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

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

  <xsd:complexType name="required-config-propertyType">
    <xsd:annotation>
      <xsd:documentation>
    <![CDATA[

      The required-config-propertyType contains a declaration
      of a single configuration property used for specifying a
      required configuration property name. It is used
      by required-config-property elements.


      Usage of this type is deprecated from Connector 1.6 specification.
      Refer to required-config-property element for more information.

      Example:

      <required-config-property>Destination</required-config-property>
```

CODE EXAMPLE 20-1 Schema Definition for the Deployment Descriptor for a Resource Adapter

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

    <xsd:sequence>
      <xsd:element name="description"
       type="javaee:descriptionType"
       minOccurs="0"
       maxOccurs="unbounded"/>
      <xsd:element name="config-property-name"
       type="javaee:config-property-nameType"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

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

  <xsd:complexType name="resourceadapterType">
    <xsd:annotation>
      <xsd:documentation>

    The resourceadapterType specifies information about the
    resource adapter. The information includes
    resource adapter Java class name, configuration properties,
    information specific to the implementation of the resource
    adapter library as specified through the
    outbound-resourceadapter and inbound-resourceadapter
    elements, and an optional set of administered objects.

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

    <xsd:sequence>
      <xsd:element name="resourceadapter-class"
       type="javaee:fully-qualified-classType"
       minOccurs="0">
    <xsd:annotation>
      <xsd:documentation>

        The element resourceadapter-class specifies the
         name of a Java class that implements
        the javax.resource.spi.ResourceAdapter
        interface. This Java class is provided as part of
        resource adapter's implementation of connector
        architecture specified contracts. The implementation
        of this class is required to be a JavaBean.
```

CODE EXAMPLE 20-1   Schema Definition for the Deployment Descriptor for a Resource Adapter

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

   </xsd:element>
   <xsd:element name="config-property"
    type="javaee:config-propertyType"
    minOccurs="0"
    maxOccurs="unbounded"/>
   <xsd:element name="outbound-resourceadapter"
    type="javaee:outbound-resourceadapterType"
    minOccurs="0">
<xsd:unique name="connectionfactory-interface-uniqueness">
   <xsd:annotation>
     <xsd:documentation>


        The connectionfactory-interface element content
        must be unique in the outbound-resourceadapter.
        Multiple connection-definitions can not use the
        same connectionfactory-type.

     </xsd:documentation>
   </xsd:annotation>
   <xsd:selector xpath="javaee:connection-definition"/>
   <xsd:field    xpath="javaee:connectionfactory-interface"/>
</xsd:unique>
   </xsd:element>
   <xsd:element name="inbound-resourceadapter"
    type="javaee:inbound-resourceadapterType"
    minOccurs="0"/>
   <xsd:element name="adminobject"
    type="javaee:adminobjectType"
    minOccurs="0"
    maxOccurs="unbounded"/>

<xsd:unique name="adminobject-type-uniqueness">
   <xsd:annotation>
     <xsd:documentation>

      The adminobject-interface and adminobject-class element content must be
       unique in the resourceadapterType. Several admin objects
       can not use the same adminobject-interface and adminobject-class.

     </xsd:documentation>
   </xsd:annotation>
   <xsd:selector xpath="javaee:adminobject"/>
   <xsd:field    xpath="javaee:adminobject-interface"/>
```

CODE EXAMPLE 20-1   Schema Definition for the Deployment Descriptor for a Resource Adapter

```
      <xsd:field    xpath="javaee:adminobject-class"/>
    </xsd:unique>

      <xsd:element name="security-permission"
       type="javaee:security-permissionType"
       minOccurs="0"
       maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

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

  <xsd:complexType name="security-permissionType">
    <xsd:annotation>
      <xsd:documentation>

    The security-permissionType specifies a security
    permission that is required by the resource adapter code.

    The security permission listed in the deployment descriptor
    are ones that are different from those required by the
    default permission set as specified in the connector
    specification. The optional description can mention specific
    reason that resource adapter requires a given security
    permission.

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

    <xsd:sequence>
      <xsd:element name="description"
       type="javaee:descriptionType"
       minOccurs="0"
       maxOccurs="unbounded"/>
      <xsd:element name="security-permission-spec"
       type="javaee:xsdStringType">
    <xsd:annotation>
      <xsd:documentation>

        The element security-permission-spec specifies a security
        permission based on the Security policy file
        syntax. Refer to the following URL for Sun's
        implementation of the security permission
        specification:

        http://docs.oracle.com/javase/6/docs/technotes/guides/security/PolicyF
```

CODE EXAMPLE 20-1   Schema Definition for the Deployment Descriptor for a Resource Adapter

```
         iles.html
      </xsd:documentation>
    </xsd:annotation>
       </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>


</xsd:schema>
```

Runtime Environment

This chapter focuses on the Java portion of a resource adapter that executes within a Java compatible runtime environment. A Java runtime environment is provided by an application server and its containers.

The chapter specifies the Java APIs that a Java EE-compliant application server and its containers must make available to a resource adapter at runtime. A portable resource adapter can rely on these APIs to be available on all Java EE-compliant application servers.

The chapter also specifies programming restrictions imposed on a resource adapter. These restrictions enable an application server to enforce security and manage a runtime environment with multiple configured resource adapters.

21.1 Programming APIs

A resource adapter provider relies on a Java EE compliant application server to provide the following APIs:

- Java SDK, Standard Edition, version 6.0 that includes the following as part of either the core platform or standard extensions: Java IDL, JNDI Standard Extension, and RMI-IIOP. (see "Java Platform, Standard Edition 7API Specification" on page E-1)
- Required APIs for Java^{TM} Platform, Enterprise Edition, Version 6 as specified in the Java EE platform specification (see "Java Platform, Enterprise Edition (Java EE) Specification, version 7" on page E-1).
- Java Authentication and Authorization Service (JAAS) 1.0 that requires at least Java 2 SDK, Standard Edition, version 1.3 or the Java 2 Runtime Environment version 1.3.

21.2   Security Permissions

An application server must provide a set of security permissions for executing a resource adapter in a managed runtime environment. A resource adapter must be granted explicit permissions to access system resources.

Since the exact set of required security permissions for a resource adapter depends on the overall security policy for an operational environment and the implementation requirements of a resource adapter, the connector architecture does not define a fixed set of permissions.

The following permission set represents the default set of security permissions that a resource adapter should expect from an application server. These security permissions are described in detail in the Java^{TM} Platform, Standard Edition

documentation. Refer to
`http://docs.oracle.com/javase/6/docs/technotes/guides/security/permissions.html`.

TABLE 21-1 Default Security Permission Set

Security Permission	Default Policy	Notes
`java.security.AllPermission`	deny	Extreme care should be taken before granting this permission to a resource adapter. This permission should only be granted if the resource adapter code is completely trusted and when it is prohibitively cumbersome to add necessary permissions to the security policy.
`java.awt.AWTPermission`	deny *	A resource adapter must not use AWT code to interact with display or input devices.
`java.io.FilePermission`	grant read and write <pathname> deny rest	A `java.io.FilePermission` represents access to a file or directory. A `FilePermission` consists of a pathname and a set of actions valid for that pathname. A resource adapter is granted permission to read/write files as specified by the `pathname`, which is specific to a configured operational environment. It is important to consider the implications of granting `Write` permission for `<<ALL FILES>>` because this grants the resource adapter permissions to write to the entire file system. This can allow a malicious resource adapter to mangle system binaries for the JVM environment.
`java.net.NetPermission`	deny *	

TABLE 21-1 Default Security Permission Set

Security Permission	Default Policy	Notes
`java.util.` `PropertyPermission`	grant `read` (allows `System.getProperty` to be called) deny rest	Granting code permission to access certain system properties (`java.home`) can potentially give malevolent code sensitive information about the system environment, such as the Java installation directory.
`java.lang.reflect.Refl ectPermission`	deny *	
`java.lang.` `RuntimePermission`	deny *	By default, `RuntimePermission` is denied to the resource adapter code. A resource adapter should explicitly request `LoadLibrary.{libraryNam e}` to link a dynamic library. The `libraryName` represents a specific library. A resource adapter that manages threads must explicitly request permission to `modifyThread` through its deployment descriptor. A resource adapter should never be granted `exitVM` permission in a managed application server environment.

TABLE 21-1 Default Security Permission Set

Security Permission	Default Policy	Notes
`java.security. SecurityPermission`	deny *	
`java.net. SocketPermission`	grant `connect  *` deny rest	This represents permission to access a network by way of sockets. A `SocketPermission` consists of a host specification and a set of actions specifying ways to connect to that host. A resource adapter is granted permission to connect to any host as indicated by the wildcard *. A resource adapter may be granted permission to accept connections from other hosts by way of a "grant `accept  *`". This may be necessary for resource adapters that support inbound communication.
`java.security. SerializablePermission`	deny *	This ensures that a resource adapter cannot subclass `ObjectOutputStream` or `ObjectInputStream` to override the default serialization or deserialization of objects or to substitute one object for another during serialization or deserialization.

21.3   Requirements

A resource adapter provider must ensure that resource adapter code does not conflict with the default security permission set. By ensuring this, a resource adapter can be deployed and run in any application server without execution or manageability problems.

If a resource adapter requires security permissions other than those specified in the default set, it must describe such requirements in the XML deployment descriptor using the `security-permission` element or through the `SecurityPermission` annotation described in Section 18.4.4, "@SecurityPermission" on page 18-9.

A deployment descriptor-based specification of an extended permission set for a resource adapter allows the deployer to analyze the security implications of the extended permission set and make a deployment decision accordingly. An application server must be capable of deploying a resource adapter with the default permission set.

21.3.1　Example

The resource adapter implementation creates a `java.net.Socket` and retrieves the hostname using the `getHostName` method in `java.net.InetAddress`.

TABLE 21-2　Methods and Security Permissions Required

Method	Security Manager Method Called	Permission
`java.net.Socket Socket(...)`	checkConnect(*host*, *port*)	`java.net.SocketPermission` "*host:port*", `"connect"`
`java.net.InetAddres s` `public String getHostName()`	checkConnect(*host*, -1)	`java.net.SocketPermission` "*host*", "*resolve*"

The default `SocketPermission`, as specified in Table 21-1, is `grant connect` and `deny rest`. This means that if resource adapter uses the default permission set, the first method `Socket(...)` will be allowed while the second method `InetAddress.getHostName` is disallowed.

The resource adapter is required to explicitly request security permission for the `InetAddress.getHostName` method in the `security-permission-spec` element of its XML deployment descriptor or through the `SecurityPermission` annotation described in Section 18.4.4, "@SecurityPermission" on page 18-9. The following is an example of allowing additional security permissions:

```
<security-permission-spec>
      grant {
          permission java.net.SocketPermission *, "resolve";
            };
</security-permission-spec>
```

21.4 Privileged Code

A resource adapter runs in its own protection domain as identified by its code source and security permission set. For the resource adapter to be allowed to perform a secured action, such as writing a file, it must have been granted permission for that particular action.

Resource adapter code is considered system code which may require more security permissions than the calling application component code. For example, when an application component calls a resource adapter method to execute a function call on the underlying EIS instance, the resource adapter code may need more security permissions than allowed to the calling component, such as the ability to create a thread.

The Java security architecture requires that whenever a system resource access or any secured action is attempted, all code traversed by the current execution thread up to that point must have the necessary permissions for the system resource access, unless some code on the thread has been marked as privileged. Refer to `http://docs.oracle.com/javase/6/docs/technotes/guides/security/doprivileged.html`.

To support such scenarios, the resource adapter code should use the `privileged` code feature in the Java security architecture. This enables the resource adapter code to temporarily perform more secured actions than are available directly to the application code calling the resource adapter.

21.4.1 Example

A resource adapter from Wombat Systems packaged in the `wombat.rar` file contains the following permission specification:

```
<security-permission>
   <security-permission-spec>
      grant {
         permission java.io.FilePermission
         "${user.home}${file.separator}trace{file.separator}-",
         "read,write,delete";
      };
   </security-permission-spec>
</security-permission>
```

During resource adapter deployment, the application server processes this `security-permission-spec` and grants the necessary permissions to the `wombat.rar` code base. This is an implementation-specific mechanism and not prescribed by the specification. As an example, the application server may append these permissions to the `java.policy` file or some implementation-specific policy file, and this may involve manual intervention.

```
// application code
...
WombatConnectionFactory wcf =
      (WombatConnectionFactory)
jndi.lookup("WombatConnectionFactory");
WombatConnection wc = wcf.getConnection(..);
doWork(wc); // calls into resource adapter code
// resource adapter implementation of WombatConnection
...
AccessController.doPrivileged(new PrivilegedAction() {
   public Object run() {
      // privileged code goes here, for example:
      File file = File.createNewFile();
      writeTraceInfoToFile(file);
      return null; // nothing to return
   }
});
```

In addition to specifying these required permissions, the resource adapter must also use `doPrivileged` blocks at strategic locations in its code to prevent the permission checking from reaching the application code or the application server code. The `doPrivileged` block allows the `AccessController` to temporarily grant the necessary permissions to the resource adapter code and to stop checking the rest of the call stack. This allows the resource adapter code to be unaffected by the calling application code's security permission restrictions.

21.5   Dependency Injection

A resource adapter archive can be a bean archive (see Chapter 12 of "Contexts and Dependency Injection for the Java EE Platform 1.1" on page E-2). The section titled "Support for Dependency Injection" in the "Resources, Naming and Injection" chapter of the Java EE Platform Specification (see "Java Platform, Enterprise Edition (Java EE) Specification, version 7" on page E-1) provides more details on the dependency injection requirements of an application server.

The following JavaBeans of a resource adapter archive have their lifecycle managed by the application server (see Chapter 5, "Lifecycle Management"):

- `ResourceAdapter`
- `ManagedConnectionFactory`
- `ActivationSpec`
- Administered Objects

These JavaBeans may be used as CDI managed beans if they are annotated with a CDI bean-defining annotation or contained in a bean archive for which CDI is enabled. However, if they are used as CDI managed beans, it must be noted that the instances that are managed by CDI may not be the instances that are managed by the application server. For example:

- If a `ResourceAdapter` class is injected into other component classes like Servlets, the injected `ResourceAdapter` instance may not be the `ResourceAdapter` instance managed by the application server

- If an `ActivationSpec` declares an injection point whose bean type is a `ResourceAdapter` class, the injected `ResourceAdapter` instance may not be the `ResourceAdapter` instance managed by the application server or the one associated with the `ActivationSpec`

Since these JavaBeans may not be portably supported as CDI managed beans, it is recommended to not use these JavaBeans as CDI managed beans. A future version of this specification would address supporting these JavaBeans as CDI managed beans.

Exceptions

This chapter specifies standard exceptions that identify error conditions which may occur as part of the connector architecture.

The connector architecture defines two classes of exceptions:

- **System Exceptions** - Indicate an unexpected error condition that occurs as part of an invocation of a method defined in the system contracts. For example, system exceptions are used to indicate transaction management-related errors. A system exception is targeted for handling by an application server or resource adapter, depending on who threw the exception, and may not be reported in its original form directly to an application component.
- **Application Exceptions** - Thrown when an application component accesses an EIS resource. For example, an application exception may indicate an error in the execution of a function on a target EIS. These exceptions are meant to be handled directly by an application component.

The connector architecture defines the `javax.resource.ResourceException` class as the root of the system exception hierarchy. The `ResourceException` class extends the `java.lang.Exception` class and is a checked exception.

The `javax.resource.ResourceException` is also the root of the application exception hierarchy for CCI. Application level exceptions are specified in more detail in the API documentation for CCI.

Note, an extended implementation of an exception type provided by a resource adapter may override the `getLocalizedMessage` method to provide a localized message.

22.1   `ResourceException`

A `ResourceException` provides the following information:

- A resource adapter-specific string describing the error. This string is a standard Java exception message and is available through the `getMessage` method.
- A resource adapter-specific error code that identifies the error condition represented by the `ResourceException`.
- A reference to another exception. Often a `ResourceException` results from a lower-level problem. If appropriate, a lower-level exception, such as `java.lang.Exception` or any derived exception type, may be linked to a `ResourceException` instance.

22.2 System Exceptions

The connector architecture requires that methods, as part of a system contract implementation, use the checked `ResourceException` and other standard exceptions derived from it to indicate system-level error conditions. Using checked exceptions leads to a strict enforcement of the contract for throwing and catching system exceptions and dealing with error conditions.

In addition, a method implementation may use `java.lang.RuntimeException` or any derived exception to indicate runtime error conditions of varying severity levels. Using unchecked exceptions to indicate important system-level error conditions is not recommended for an implementation of system contracts.

The method should use `java.lang.Error` to indicate a serious error condition that it does not require the caller to catch. A method is not required to declare in its throws clause any subclasses of `Error` that may be thrown but not caught during the execution of the method, since these errors are abnormal conditions that should never occur.

22.2.1 Exception Hierarchy

The `ResourceException` represents a generic form of exception. A derived exception represents a specific class of error conditions. This design enables the method invocation code to catch a class of error conditions based on the exception type and to handle error conditions appropriately.

The following exceptions are derived from `ResourceException` to indicate more specific classes of system error conditions:

- **javax.resource.spi.SecurityException.** A `SecurityException` indicates error conditions related to the security contract between an application server and resource adapter. The common error conditions represented by this exception are:

- Invalid security information, represented by a `Subject` instance, passed across the security contract. For example, credentials may have expired or be in an invalid format.
- Lack of support for a specific security mechanism in an EIS or resource adapter.
- Failure to create a connection to an EIS because of failed authentication or authorization.
- Failure to authenticate a resource principal to an EIS or failure to establish a secure association with an underlying EIS instance.
- Access control exception indicating that a requested access to an EIS resource or a request to create a new connection has been denied.

- **`javax.resource.spi.LocalTransactionException`.** A `LocalTransaction-Exception` represents various error conditions related to the local transaction management contract. The JTA specification specifies the `javax.transaction-.xa.XAException` class for exceptions related to an `XAResource`-based transaction management contract. The `LocalTransactionException` is used for the local transaction management contract to indicate the following types of error conditions:

  - Invalid transaction context when a transaction operation is executed. For example, calling the `LocalTransaction.commit` method without an active local transaction is an error condition.
  - Transaction is rolled back instead of being committed in the `LocalTransaction.commit` method.
  - Attempt to start a local transaction from the same thread on a `ManagedConnection` instance that is already associated with an active local transaction.
  - All resource adapter or resource manager-specific error conditions related to local transaction management. Examples are violation of integrity constraints, deadlock detection, communication failure during transaction completion, or any retry requirement.

- **`javax.resource.spi.ResourceAdapterInternalException`.** This exception indicates all system-level error conditions related to a resource adapter. The common error conditions indicated by this exception type are:

  - Invalid configuration of the `ManagedConnectionFactory` for creating a new physical connection. An example is an invalid server name for a target EIS instance.
  - Failure to create a physical connection to a EIS instance due to a communication protocol error or a resource adapter implementation-specific error.
  - Error conditions internal to a resource adapter implementation.

- **`javax.resource.spi.EISSystemException`.** An `EISSystemException` is used to indicate any EIS-specific system-level error conditions. Examples of common error conditions are failure or inactivity of an EIS instance, communication failure, and an EIS-specific error during the creation of a physical connection.

- **`javax.resource.spi.ApplicationServerInternalException.`** This exception is thrown by an application server to indicate error conditions specific to an application server. Example error conditions are: errors related to an application server configuration or implementation of mechanisms internal to an application server, such as connection pooling and thread management.
- **`javax.resource.spi.ResourceAllocationException.`** This exception is thrown by an application server or resource adapter to indicate a failure to allocate system resources, such as threads and physical connections. An example is an error condition that results when an upper bound is reached for the maximum number of physical connections that can be managed by an application server-specific connection pool.
- **`javax.resource.spi.IllegalStateException.`** This exception is thrown from a method if the invoked code, either the resource adapter or the application server for system contracts, is in an illegal or inappropriate state for the method invocation.
- **`javax.resource.NotSupportedException.`** This exception is thrown to indicate that invoked code, either the resource adapter or the application server for system contracts, cannot execute an operation because the operation is not a supported feature. For example, if the transaction support level for a resource adapter is `NoTransaction`, an invocation of the `ManagedConnection.getXAResource` method throws a `NotSupportedException` exception.
- **`javax.resource.spi.CommException.`** This exception indicates errors related to failed or interrupted communication with an EIS instance. Examples of common error conditions represented by this exception type include communication protocol errors and invalidated connections due to server failure.
- **`javax.resource.spi.InvalidPropertyException.`** This exception is thrown to indicate invalid configuration property settings.
- **`javax.resource.spi.UnavailableException.`** This exception is thrown to indicate that a service is unavailable.

22.3   `Work` Exceptions

These exceptions are thrown by an application server to report error conditions related to the work management contract.

- **`javax.resource.spi.work.WorkException.`** A common base class for all `Work` processing related exceptions.
- **`javax.resource.spi.work.WorkRejectedException.`** This exception is thrown to indicate that a submitted `Work` instance has been rejected. The rejection may be due to internal factors or start timeout expiration.
- **`javax.resource.spi.work.WorkCompletedException.`** This exception is thrown to indicate that a submitted `Work` instance has completed with an exception.

22.4 Additional Exceptions

The JTA specification (see "Java Transaction API (JTA) Specification, version 1.2" on page E-1) specifies the `javax.transaction.xa.XAException` class for exceptions related to the `XAResource`-based transaction management contract.

Compatibility and Migration

This chapter summarizes compatibility and migration issues for the Java EE Connectors specification. For a detailed description on Compatibility and Migration and how they relate to the Java EE Platform in general, refer the chapter on "Compatibility and Migration" in the Java EE platform specification (see "Java Platform, Enterprise Edition (Java EE) Specification, version 7" on page E-1).

23.1   Compatibility

Java EE application servers are compatible with the Java EE Connector Architecture specification if they implement the APIs and behavior required by this specification. Resource adapter modules are compatible with a release of the Java EE Connector Architecture specification if they only depend on APIs and behavior defined by that release of the specification. A new release of the Java EE Connector architecture specification is compatible with previous releases of the specification if all portable resource adapters written to the previous release of the specification will also run unchanged and with identical behavior on the new release of the specification.

23.1.1   Changes Since Connector 1.5 Specification

This section lists the compatibility issues of the Java EE Connector Architecture 1.6 specification vis-a-vis the earlier version of the specification (that is, J2EE Connector Architecture 1.5).

23.1.1.1 Standalone Resource Adapter Visibility to an Application

The Java EE Connector Architecture 1.6 specification defines the classloading requirements for an application server in Section 20.3, "Class Loading Requirements" on page 20-6.

In the earlier version of this specification, the specification required all deployed standalone resource adapters to be made available to all the applications. The new requirements allow an application server to make a standalone resource adapter available to an application, only when there is an explicit reference to the resource adapter. See Section 20.3, "Class Loading Requirements" on page 20-6, for more information on when an application may portably assume the visibility of a deployed standalone resource adapter

23.1.1.2 Deprecation of `required-config-property`

With the availability of validation capabilities of the Bean Validation specification (see "Bean Validation Specification, version 1.1" on page E-2) to a resource adapter JavaBean, as detailed in Section 5.3.7.5, "JavaBean Validation" on page 5-15, the existing mechanism provided to an `ActivationSpec` JavaBean to specify `required-config-property` has been deprecated.

The use of `required-config-property` element, described in Section 13.4.2.2, "ActivationSpec JavaBean" on page 13-12, is deprecated since the Java EE Connector Architecture 1.6 Specification. The resource adapter implementation is recommended to use the `NotNull` Bean Validation annotation or its XML validation descriptor equivalent to indicate that a configuration property is required to be specified by the deployer, as described in Section 13.4.2.2, "ActivationSpec JavaBean" on page 13-12

23.2 Migration

Migration is the act of converting a resource adapter to use new facilities introduced in this release of the specification. Given the strong level of compatibility in this release of the specification, migration is largely an optional exercise. Still, an application may be improved (better performance, simpler to develop, more flexible, etc.) by converting it to use newer facilities of the specification.

23.2.1 Changes Since the Connector 1.5 Specification

23.2.1.1 Annotations

Through the availability of the metadata annotations (see Chapter 18, "Metadata Annotations"), it is simpler to develop a resource adapter module. By using annotations, many resource adapters can avoid the need for deployment descriptors and sometimes avoid the need to implement interfaces, greatly simplifying application development. Developers should consider the use of annotations instead of deployment descriptors.

Projected Items

The following are some of the features planned for future versions of this specification:

■ **Common Client Interface.** The CCI may be extended to include support for XML, type mapping, and metadata facility.

APPENDIX **A**

Previous Version Deployment Descriptors

This appendix contains Document Type Definitions (DTDs) and XML schemas (XSD) for Deployment Descriptors from previous versions of the J2EE Connector Architecture specification. All Java EE products are required to support these DTDs as well as the XML Schema Definition specified in this version of the specification. This ensures that resource adapters written to previous versions of this specification can be deployed on products supporting the current version of this specification.

A.1 J2EE Connector Architecture 1.0 Resource Adapter XML DTD

This section specifies the XML DTD for the 1.0 resource adapter deployment descriptor. The comments in the DTD specify additional requirements for syntax and semantics that cannot be specified by the DTD mechanism.

A resource adapter, or an application server on behalf of a resource adapter, may specify additional deployment information beyond the standard deployment descriptor. The additional information should be stored in a separate file and should refer to the standard deployment descriptor.

A resource adapter is not allowed to add any non-standard information into a standard deployment descriptor.

CODE EXAMPLE A-1   Connector Architecture 1.0 Resource Adapter DTD

```
<!--
This is the XML DTD for the Connector 1.0 deployment descriptor.
All Connector 1.0 deployment descriptors must include a DOCTYPE of
the following form:
```

CODE EXAMPLE A-1   Connector Architecture 1.0 Resource Adapter DTD

```
<!DOCTYPE connector PUBLIC"-//Sun Microsystems, Inc.//DTD
Connector 1.0//EN" "http://java.sun.com/dtd/connector_1_0.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 connector element is the root element of the deployment
descriptor
for the resource adapter. This element includes general information
- vendor
name, version, specification version supported, icon - about the
resource adapter module. It also includes information specific to
the
implementation of the resource adapter library as specified through
the element resourceadapter.
-->
<!ELEMENT connector (display-name?, description?, icon?, vendor-
name,
spec-version, eis-type, version, license?, resourceadapter>

<!--
The element authentication-mechanism specifies an authentication
mechanism
supported by the resource adapter. Note that this support is for
the resource adapter and not for the underlying EIS instance. The
```

CODE EXAMPLE A-1   Connector Architecture 1.0 Resource Adapter DTD

```
optional description specifies any resource adapter specific
requirement
for the support of security contract and authentication mechanism.

Note that BasicPassword mechanism type should support the
javax.resource.spi.security.PasswordCredential interface. The
Kerbv5
mechanism type should support the
javax.resource.spi.security.Generic-
Credential interface.

Used in: resourceadapter
-->
<!ELEMENT authentication-mechanism (
description?, authentication-mechanism-type, credential-
interface)>

<!--
The element authentication-mechanism-type specifies type of an
authentication
mechanism.

The example values are:
<authentication-mechanism-type>BasicPassword
                  </authentication-mechanism-type>
   <authentication-mechanism-type>Kerbv5
                  </authentication-mechanism-type>

Any additional security mechanisms are outside the scope of the
Connector architecture specification.

Used in: authentication-mechanism
-->
<!ELEMENT authentication-mechanism-type (#PCDATA)>

<!--
The element config-property contains a declaration of a single
configuration property for a ManagedConnectionFactory instance.

Each ManagedConnectionFactory instance creates connections to a
specific EIS instance based on the properties configured on the
ManagedConnectionFactory instance. The configurable properties are
specified only once in the deployment descriptor, even though a
resource adapter can be used to configure multiple
ManagedConnnection-
Factory instances (that create connections to different instances
of
```

CODE EXAMPLE A-1   Connector Architecture 1.0 Resource Adapter DTD

```
the same EIS).

The declaration consists of an optional description, name, type
and an optional value of the configuration property. If the
resource
adapter provider does not specify a value than the deployer is
responsible for providing a valid value for a configuration
property.

Any bounds or well-defined values of properties should be described
in the description element.

Used in: resourceadapter
-->
<!ELEMENT config-property (description?, config-property-name,
config-property-type, config-property-value?)>

<!--
The element config-property-name contains the name of a
configuration
property.

The connector architecture defines a set of well-defined properties
all of type java.lang.String. These are as follows:
     <config-property-name>ServerName</config-property-name>
     <config-property-name>PortNumber</config-property-name>
     <config-property-name>UserName</config-property-name>
     <config-property-name>Password</config-property-name>
     <config-property-name>ConnectionURL</config-property-name>

A resource adapter provider can extend this property set to include
properties specific to the resource adapter and its underlying EIS.

Used in: config-property

Example: <config-property-name>ServerName</config-property-name>
-->
<!ELEMENT config-property-name (#PCDATA)>

<!--
The element config-property-type contains the  Java
type of a configuration property as required by ManagedConnection-
Factory instance.

The following are the legal values of config-property-type:
   java.lang.Boolean, java.lang.String, java.lang.Integer,
   java.lang.Double, java.lang.Byte, java.lang.Short,
```

CODE EXAMPLE A-1   Connector Architecture 1.0 Resource Adapter DTD

```
java.lang.Long, java.lang.Float, java.lang.Character

Used in: config-property

Example: <config-property-type>java.lang.String</config-property-
type>
-->
<!ELEMENT config-property-type (#PCDATA)>


<!--
The element config-property-value contains the value of a
configuration
entry.

Used in: config-property

Example: <config-property-value>WombatServer</config-property-
value>
-->
<!ELEMENT config-property-value (#PCDATA)>

<!--
The element connection-impl-class specifies the fully-qualified
name of the Connection class that implements resource adapter
specific Connection interface.

Used in: resourceadapter

Example: <connection-impl-class>com.wombat.ConnectionImpl
        </connection-impl-class>
-->
<!ELEMENT connection-impl-class (#PCDATA)>

<!--
The element connection-interface specifies the fully-qualified
name of the Connection interface supported by the resource
adapter.

Used in: resourceadapter

Example: <connection-interface>javax.resource.cci.Connection
        </connection-interface>
-->
<!ELEMENT connection-interface (#PCDATA)>

<!--
```

CODE EXAMPLE A-1   Connector Architecture 1.0 Resource Adapter DTD

```
The element connectionfactory-impl-class specifies the fully-
qualified
name of the ConnectionFactory class that implements resource
adapter
specific ConnectionFactory interface.

Used in: resourceadapter

Example: <connectionfactory-impl-
class>com.wombat.ConnectionFactoryImpl
        </connectionfactory-impl-class>
-->
<!ELEMENT connectionfactory-impl-class (#PCDATA)>

<!--
The element connectionfactory-interface specifies the fully-
qualified
name of the ConnectionFactory interface supported by the resource
adapter.

Used in: resourceadapter

Example: <connectionfactory-
interface>com.wombat.ConnectionFactory
        </connectionfactory-interface>
OR
<connectionfactory-interface>javax.resource.cci.ConnectionFactory
        </connectionfactory-interface>
-->
<!ELEMENT connectionfactory-interface (#PCDATA)>

<!--
The element credential-interface specifies the interface that the
resource adapter implementation supports for the representation
of the credentials. This element should be used by application
server
to find out the Credential interface it should use as part of the
security contract.

The possible values are:
   <credential-
interface>javax.resource.spi.security.PasswordCredential
        </credential-interface>
   <credential-
interface>javax.resource.spi.security.GenericCredential
        </credential-interface>
```

CODE EXAMPLE A-1   Connector Architecture 1.0 Resource Adapter DTD

```
Used in: authentication-mechanism
-->
<!ELEMENT credential-interface (#PCDATA)>


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

Used in: authentication-mechanism, config-property, connector,
license,
security-permission
-->
<!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: connector

Example:

<display-name>Employee Self Service</display-name>
-->
<!ELEMENT display-name (#PCDATA)>


<!--
The element eis-type contains information about the type of the
EIS. For example, the type of an EIS can be product name of EIS
independent of any version info.

This helps in identifying EIS instances that can be used with
this resource adapter.

Used in: connector
-->
<!ELEMENT eis-type (#PCDATA)>


<!--
```

CODE EXAMPLE A-1   Connector Architecture 1.0 Resource Adapter DTD

```
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 GUI.

Used in: connector
-->
<!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
component'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 element license specifies licensing requirements for the
resource
adapter module. This element specifies whether a license is
required
to deploy and use this resource adapter, and an optional
description
of the licensing terms (examples: duration of license, number of
connection restrictions).

Used in: connector
-->
<!ELEMENT license (description?, license-required)>

<!--
The element license-required specifies whether a license is
required
to deploy and use the resource adapter. This element must be one of
the following:

    <license-required>true</license-required>
    <license-required>false</license-required>
```

```
Used in: license
-->
<!ELEMENT license-required (#PCDATA)>


<!--
The element managedconnectionfactory-class specifies the
name of the Java class that implements the
javax.resource.spi.Managed-
ConnectionFactory interface. This Java class is provided as part of
resource adapter's implementation of connector architecture
specified
contracts.

Used in: resourceadapter

Example:
  <managedconnectionfactory-
class>com.wombat.ManagedConnectionFactoryImpl
  </managedconnectionfactory-class>
-->
<!ELEMENT managedconnectionfactory-class (#PCDATA)>


<!--
The element reauthentication-support specifies whether the
resource
adapter implementation supports re-authentication of existing
Managed-
Connection instance. Note that this information is for the resource
adapter implementation and not for the underlying EIS instance.

This element must be one of the following:
        <reauthentication-support>true</reauthentication-support>
       <reauthentication-support>false</reauthentication-support>
Used in: resourceadapter
-->
<!ELEMENT reauthentication-support (#PCDATA)>


<!--
The element resourceadapter specifies information about the
resource
adapter. The information includes fully-qualified names of
class/interfaces required as part of the connector architecture
specified contracts, level of transaction support provided,
configurable properties for ManagedConnectionFactory instances,
one or more authentication mechanisms supported and additional
required security permissions.
```

CODE EXAMPLE A-1   Connector Architecture 1.0 Resource Adapter DTD

```
If there is no authentication-mechanism specified as part of
resource adapter element then the resource adapter does not
support any standard security authentication mechanisms as part
of security contract. The application server ignores the security
part of the system contracts in this case.

Used in: connector
-->

<!ELEMENT resourceadapter (
managedconnectionfactory-class, connectionfactory-interface,
connectionfactory-impl-class, connection-interface,
connection-impl-class, transaction-support, config-property*,
authentication-mechanism*, reauthentication-support, security-
permission*
)>

<!--
The element security permission specifies a security permission
that
is required by the resource adapter code.

The security permission listed in the deployment descriptor are
ones
that are different from those required by the default permission
set
as specified in the connector specification. The optional
description
can mention specific reason that resource adapter requires a given
security permission.

Used in: resourceadapter
-->
<!ELEMENT security-permission (description?, security-permission-
spec)>

<!--
The element permission-spec specifies a security permission based
on the Security policy file syntax. Refer the following URL for
Sun's implementation of security permission specification:

http://java.sun.com/products/jdk/1.3/docs/guide/security/PolicyFi
les.html#FileSyntax

Used in: security-permission
-->
```

CODE EXAMPLE A-1   Connector Architecture 1.0 Resource Adapter DTD

```
<!ELEMENT security-permission-spec (#PCDATA)>

<!--
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
component'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 element spec-version specifies the version of the connector
architecture specification that is supported by this resource
adapter. This information enables deployer to configure the
resource
adapter to support deployment and runtime requirements of the
corresponding connector architecture specification.

Used in: connector

Example:
  <spec-version>1.0</spec-version>
-->
<!ELEMENT spec-version (#PCDATA)>

<!--
The transaction-support element specifies the level of transaction
support provided by the resource adapter.
The value of transaction-support must be one of the following:
<transaction-support>NoTransaction</transaction-support>
 <transaction-support>LocalTransaction</transaction-support>
 <transaction-support>XATransaction</transaction-support>

Used in: resourceadapter
-->
<!ELEMENT transaction-support (#PCDATA)>

<!--
```

```
The element vendor-name specifies the name of resource adapter
provider
vendor.

Used in: connector

Example:
  <vendor-name>Wombat Systems</vendor-name>
-->
<!ELEMENT vendor-name (#PCDATA)>

<!--
The element version specifies a string-based version of the
resource adapter from the resource adapter provider.

Used in: connector

Example:
  <version>1.0</version>
-->
<!ELEMENT version (#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 authentication-mechanism id ID #IMPLIED>
<!ATTLIST authentication-mechanism-type id ID #IMPLIED>
<!ATTLIST config-property id ID #IMPLIED>
<!ATTLIST config-property-name id ID #IMPLIED>
<!ATTLIST config-property-type id ID #IMPLIED>
<!ATTLIST config-property-value id ID #IMPLIED>
<!ATTLIST connection-impl-class id ID #IMPLIED>
<!ATTLIST connection-interface id ID #IMPLIED>
<!ATTLIST connectionfactory-impl-class id ID #IMPLIED>
<!ATTLIST connectionfactory-interface id ID #IMPLIED>
<!ATTLIST connector id ID #IMPLIED>
<!ATTLIST credential-interface id ID #IMPLIED>
```

CODE EXAMPLE A-1   Connector Architecture 1.0 Resource Adapter DTD

```
<!ATTLIST description id ID #IMPLIED>
<!ATTLIST display-name id ID #IMPLIED>
<!ATTLIST eis-type id ID #IMPLIED>
<!ATTLIST icon id ID #IMPLIED>
<!ATTLIST large-icon id ID #IMPLIED>
<!ATTLIST license id ID #IMPLIED>
<!ATTLIST license-required id ID #IMPLIED>
<!ATTLIST managedconnectionfactory-class id ID #IMPLIED>
<!ATTLIST reauthentication-support id ID #IMPLIED>
<!ATTLIST resourceadapter id ID #IMPLIED>
<!ATTLIST security-permission id ID #IMPLIED>
<!ATTLIST security-permission-spec id ID #IMPLIED>
<!ATTLIST small-icon id ID #IMPLIED>
<!ATTLIST spec-version id ID #IMPLIED>
<!ATTLIST transaction-support id ID #IMPLIED>
<!ATTLIST vendor-name id ID #IMPLIED>
<!ATTLIST version id ID #IMPLIED>
```

A.2   J2EE Connector Architecture 1.5 Resource Adapter XML XSD

This section specifies the XML Schema definition (XSD) for the 1.5 resource adapter deployment descriptor. The comments in the XSD specify additional requirements for syntax and semantics that cannot be specified by the XSD mechanism.

A resource adapter, or an application server on behalf of a resource adapter, may specify additional deployment information beyond the standard deployment descriptor. The additional information should be stored in a separate file and should refer to the standard deployment descriptor.

A resource adapter is not allowed to add any non-standard information into a standard deployment descriptor.

CODE EXAMPLE A-2   Connector Architecture 1.5 Resource Adapter XSD

```
<?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"
```

CODE EXAMPLE A-2   Connector Architecture 1.5 Resource Adapter XSD

```
      attributeFormDefault="unqualified"
      version="1.5">
  <xsd:annotation>
     <xsd:documentation>
     @(#)connector_1_5.xsds    1.27 06/17/03
     </xsd:documentation>
  </xsd:annotation>
  <xsd:annotation>
     <xsd:documentation>
     Copyright 2003 Sun Microsystems, Inc., 901 San Antonio
     Road, Palo Alto, California 94303, U.S.A. All rights
    reserved.

     Sun Microsystems, Inc. has intellectual property rights
     relating to technology described in this document. In
     particular, and without limitation, these intellectual
     property rights may include one or more of the U.S. patents
     listed at http://www.sun.com/patents and one or more
     additional patents or pending patent applications in the
     U.S. and other countries.

     This document and the technology which it describes are
     distributed under licenses restricting their use, copying,
     distribution, and decompilation. No part of this document
     may be reproduced in any form by any means without prior
     written authorization of Sun and its licensors, if any.

     Third-party software, including font technology, is
     copyrighted and licensed from Sun suppliers.

     Sun, Sun Microsystems, the Sun logo, Solaris, Java, J2EE,
     JavaServer Pages, Enterprise JavaBeans and the Java Coffee
     Cup logo are trademarks or registered trademarks of Sun
     Microsystems, Inc. in the U.S. and other countries.

     Federal Acquisitions: Commercial Software - Government Users
     Subject to Standard License Terms and Conditions.
     </xsd:documentation>
  </xsd:annotation>
  <xsd:annotation>
     <xsd:documentation>
        <![CDATA[
        This is the XML Schema for the Connector 1.5 deployment
        descriptor.  The deployment descriptor must be named
        "META-INF/ra.xml" in the connector's rar file.  All
        Connector
```

CODE EXAMPLE A-2   Connector Architecture 1.5 Resource Adapter XSD

```
  deployment descriptors must indicate the connector
 resource
  adapter schema by using the J2EE namespace:

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

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

  <connector 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/connector_1_5.xsd"
  version="1.5">
  ...
  </connector>

  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/connector_1_5.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="connector" type="j2ee:connectorType">
   <xsd:annotation>
```

CODE EXAMPLE A-2   Connector Architecture 1.5 Resource Adapter XSD

```
        <xsd:documentation>
        The connector element is the root element of the
       deployment
        descriptor for the resource adapter. This element
       includes
        general information - vendor name, resource adapter
       version,
        icon - about the resource adapter module. It also
       includes
        information specific to the implementation of the
       resource
        adapter library as specified through the element
        resourceadapter.

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

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

   <xsd:complexType name="activationspecType">
      <xsd:annotation>
         <xsd:documentation>

         The activationspecType specifies an activation
         specification.  The information includes
         Java class name of an activation specification and a
         set of
         required configuration property names.

         </xsd:documentation>

         <xsd:sequence>
         <xsd:element name="activationspec-class"
         type="j2ee:fully-qualified-classType">
         <xsd:annotation>
           <xsd:documentation>
           <![CDATA[
          The element activationspec-class specifies the fully
           qualified Java class name of the activation
           specification class. This class must implement the
           javax.resource.spi.ActivationSpec interface. The
           implementation of this class is required to be a
         JavaBean.
           Example:
           <activationspec-
          class>com.wombat.ActivationSpecImpl
```

CODE EXAMPLE A-2   Connector Architecture 1.5 Resource Adapter XSD

```
                </activationspec-class>

               ]]>
               </xsd:documentation>
             </xsd:annotation>
         </xsd:element>
         <xsd:element name="required-config-property"
         type="j2ee:required-config-propertyType"
         minOccurs="0"
         maxOccurs="unbounded"/>
     </xsd:sequence>
     <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

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

  <xsd:complexType name="adminobjectType">
     <xsd:annotation>
        <xsd:documentation>

        The adminobjectType specifies information about an
        administered object.  Administered objects are specific
       to a
        messaging style or message provider.  This contains
        information on the Java type of the interface implemented
       by
        an administered object, its Java class name and its
        configuration properties.

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

     <xsd:sequence>
        <xsd:element name="adminobject-interface"
        type="j2ee:fully-qualified-classType">
           <xsd:annotation>
             <xsd:documentation>
             <![CDATA[

             The element adminobject-interface specifies the
              name of the Java type of the
             interface implemented by an administered object.

             Example:
             <adminobject-interface>javax.jms.Destination
             </adminobject-interface>
```

CODE EXAMPLE A-2 Connector Architecture 1.5 Resource Adapter XSD

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

         </xsd:element>
         <xsd:element name="adminobject-class"
         type="j2ee:fully-qualified-classType">
             <xsd:annotation>
               <xsd:documentation>
                <![CDATA[

                The element adminobject-class specifies the fully
                qualified Java class name of an administered object.

                Example:
                <adminobject-class>com.wombat.DestinationImpl
                </adminobject-class>

                ]]>
               </xsd:documentation>
             </xsd:annotation>
         </xsd:element>

         <xsd:element name="config-property"
         type="j2ee:config-propertyType"
         minOccurs="0"
         maxOccurs="unbounded"/>
       </xsd:sequence>
       <xsd:attribute name="id" type="xsd:ID"/>
     </xsd:complexType>

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

     <xsd:complexType name="authentication-mechanismType">
         <xsd:annotation>
           <xsd:documentation>

             The authentication-mechanismType specifies an
            authentication
             mechanism supported by the resource adapter. Note that
            this
             support is for the resource adapter and not for the
             underlying EIS instance. The optional description
            specifies
             any resource adapter specific requirement for the
            support of
             security contract and authentication mechanism.
```

CODE EXAMPLE A-2   Connector Architecture 1.5 Resource Adapter XSD

```
 Note that BasicPassword mechanism type should support
the
 javax.resource.spi.security.PasswordCredential
interface.
 The Kerbv5 mechanism type should support the
 org.ietf.jgss.GSSCredential interface or the deprecated
 javax.resource.spi.security.GenericCredential
interface.

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

<xsd:sequence>
<xsd:element name="description"
type="j2ee:descriptionType"
minOccurs="0"
maxOccurs="unbounded"/>
<xsd:element name="authentication-mechanism-type"
type="j2ee:xsdStringType">
   <xsd:annotation>
     <xsd:documentation>
     <![CDATA[

     The element authentication-mechanism-type specifies
     type of an authentication mechanism.

     The example values are:

     <authentication-mechanism-type>BasicPassword
     </authentication-mechanism-type>

     <authentication-mechanism-type>Kerbv5
     </authentication-mechanism-type>

     Any additional security mechanisms are outside the
     scope of the Connector architecture specification.

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

</xsd:element>
<xsd:element name="credential-interface"
type="j2ee:credential-interfaceType"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:ID"/>
```

CODE EXAMPLE A-2   Connector Architecture 1.5 Resource Adapter XSD

```
    </xsd:complexType>

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

   <xsd:complexType name="config-property-nameType">
      <xsd:annotation>
         <xsd:documentation>
         <![CDATA[

         The config-property-nameType contains the name of a
         configuration property.

         The connector architecture defines a set of well-
         defined
         properties all of type java.lang.String. These are as
         follows.

         ServerName
         PortNumber
         UserName
         Password
         ConnectionURL

          A resource adapter provider can extend this property
         set to
          include properties specific to the resource adapter
         and its
          underlying EIS.

          Possible values include
         ServerName
         PortNumber
         UserName
         Password
         ConnectionURL

          Example: <config-property-name>ServerName</config-
         property-name>

          ]]>
      </xsd:documentation>
    </xsd:annotation>
    <xsd:simpleContent>
       <xsd:restriction base="j2ee:xsdStringType"/>
    </xsd:simpleContent>
  </xsd:complexType>
```

CODE EXAMPLE A-2   Connector Architecture 1.5 Resource Adapter XSD

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

 <xsd:complexType name="config-property-typeType">
    <xsd:annotation>
       <xsd:documentation>
       <![CDATA[

       The config-property-typeType contains the fully
       qualified Java type of a configuration property.

       The following are the legal values:
       java.lang.Boolean, java.lang.String, java.lang.Integer,
       java.lang.Double, java.lang.Byte, java.lang.Short,
       java.lang.Long, java.lang.Float, java.lang.Character

       Used in: config-property

       Example:
       <config-property-type>java.lang.String</config-property-
       type>

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

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

 <xsd:complexType name="config-propertyType">
     <xsd:annotation>
         <xsd:documentation>
```

CODE EXAMPLE A-2   Connector Architecture 1.5 Resource Adapter XSD

```
      The config-propertyType contains a declaration of a
    single
     configuration property that may be used for providing
     configuration information.

     The declaration consists of an optional description,
    name,
     type and an optional value of the configuration property.
    If
     the resource adapter provider does not specify a value
    than
     the deployer is responsible for providing a valid value
    for
     a configuration property.

     Any bounds or well-defined values of properties should be
     described in the description element.

    </xsd:documentation>
</xsd:annotation>
<xsd:sequence>
    <xsd:element name="description"
    type="j2ee:descriptionType"
    minOccurs="0"
    maxOccurs="unbounded"/>
    <xsd:element name="config-property-name"
    type="j2ee:config-proper <config-property-
  value>WombatServer</config-property-value>ty-nameType"/>
    <xsd:element name="config-property-type"
    type="j2ee:config-property-typeType"/>
    <xsd:element name="config-property-value"
    type="j2ee:xsdStringType"
    minOccurs="0">
    <xsd:annotation>
       <xsd:documentation>
       <![CDATA[

       The element config-property-value contains the value
       of a configuration entry. Note, it is possible for a
       resource adapter deployer to override this
       configuration information during deployment.

       Example:
       <config-property-value>WombatServer</config-
       property-value>

       ]]>
```

CODE EXAMPLE A-2   Connector Architecture 1.5 Resource Adapter XSD

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

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

 <xsd:complexType name="connection-definitionType">
    <xsd:annotation>
        <xsd:documentation>

        The connection-definitionType defines a set of connection
        interfaces and classes pertaining to a particular
       connection
        type. This also includes configurable properties for
        ManagedConnectionFactory instances that may be produced out
        of this set.

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

    <xsd:sequence>
        <xsd:element name="managedconnectionfactory-class"
        type="j2ee:fully-qualified-classType">
            <xsd:annotation>
                <xsd:documentation>
                <![CDATA[

               The element managedconnectionfactory-class specifies
               the  name of the Java class that
               implements the
               javax.resource.spi.ManagedConnectionFactory
              interface.
               This Java class is provided as part of resource
               adapter's implementation of connector architecture
               specified contracts. The implementation of this
               class is required to be a JavaBean.

               Example:
                <managedconnectionfactory-class>
                com.wombat.ManagedConnectionFactoryImpl
                </managedconnectionfactory-class>

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

CODE EXAMPLE A-2   Connector Architecture 1.5 Resource Adapter XSD

```
        </xsd:annotation>
      </xsd:element>
      <xsd:element name="config-property"
      type="j2ee:config-propertyType"
      minOccurs="0"
      maxOccurs="unbounded"/>
      <xsd:element name="connectionfactory-interface"
      type="j2ee:fully-qualified-classType">
          <xsd:annotation>
             <xsd:documentation>
             <![CDATA[

             The element connectionfactory-interface specifies
             the  name of the ConnectionFactory
             interface supported by the resource adapter.

             Example:
             <connectionfactory-
            interface>com.wombat.ConnectionFactory
             </connectionfactory-interface>

            OR

             <connectionfactory-
            interface>javax.resource.cci.ConnectionFactory
             </connectionfactory-interface>

             ]]>
          </xsd:documentation>
       </xsd:annotation>
      </xsd:element>

      <xsd:element name="connectionfactory-impl-class"
      type="j2ee:fully-qualified-classType">
      <xsd:annotation>
        <xsd:documentation>
        <![CDATA[
        The element connectionfactory-impl-class specifies
        the  name of the ConnectionFactory
        class that implements resource adapter
        specific ConnectionFactory interface.

        Example:

        <connectionfactory-impl-
       class>com.wombat.ConnectionFactoryImpl
        </connectionfactory-impl-class>
```

CODE EXAMPLE A-2 Connector Architecture 1.5 Resource Adapter XSD

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

</xsd:element>

<xsd:element name="connection-interface"
type="j2ee:fully-qualified-classType">
   <xsd:annotation>
      <xsd:documentation>
      <![CDATA[

      The connection-interface element specifies the fully
      qualified name of the Connection interface supported
      by the resource adapter.

      Example:

      <connection-interface>javax.resource.cci.Connection
      </connection-interface>

      ]]>
      </xsd:documentation>
   </xsd:annotation>
</xsd:element>

<xsd:element name="connection-impl-class"
type="j2ee:fully-qualified-classType">
   <xsd:annotation>
      <xsd:documentation>
      <![CDATA[

      The connection-impl-classType specifies the fully
      qualified name of the Connection class that
      implements resource adapter specific Connection
      interface.  It is used by the connection-impl-class
      elements.

      Example:

      <connection-impl-class>com.wombat.ConnectionImpl
      </connection-impl-class>

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

CODE EXAMPLE A-2   Connector Architecture 1.5 Resource Adapter XSD

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

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

 <xsd:complexType name="connectorType">
     <xsd:annotation>
       <xsd:documentation>

       The connectorType defines a resource adapter.

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

     <xsd:sequence>
     <xsd:group ref="j2ee:descriptionGroup"/>
     <xsd:element name="vendor-name"
     type="j2ee:xsdStringType">
         <xsd:annotation>
             <xsd:documentation>

             The element vendor-name specifies the name of
             resource adapter provider vendor.

             </xsd:documentation>
         </xsd:annotation>
     </xsd:element>
     <xsd:element name="eis-type"
     type="j2ee:xsdStringType">
         <xsd:annotation>
             <xsd:documentation>

             The element eis-type contains information about the
             type of the EIS. For example, the type of an EIS can
             be product name of EIS independent of any version
             info.

             This helps in identifying EIS instances that can be
             used with this resource adapter.

             </xsd:documentation>
         </xsd:annotation>
     </xsd:element>
     <xsd:element name="resourceadapter-version"
     type="j2ee:xsdStringType">
```

CODE EXAMPLE A-2   Connector Architecture 1.5 Resource Adapter XSD

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

           The element resourceadapter-version specifies a string-
          based version
           of the resource adapter from the resource adapter
          provider.

            </xsd:documentation>
        </xsd:annotation>
    </xsd:element>
    <xsd:element name="license"
    type="j2ee:licenseType" minOccurs="0"/>
    <xsd:element name="resourceadapter"
    type="j2ee:resourceadapterType"/>
</xsd:sequence>
<xsd:attribute name="version"
type="j2ee:dewey-versionType"
fixed="1.5"
use="required">
    <xsd:annotation>
        <xsd:documentation>

        The version specifies the version of the
        connector architecture specification that is
        supported by this resource adapter. This information
        enables deployer to configure the resource adapter to
        support deployment and runtime requirements of the
        corresponding connector architecture specification.

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

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

<xsd:complexType name="credential-interfaceType">
    <xsd:annotation>
        <xsd:documentation>

        The credential-interfaceType specifies the
        interface that the resource adapter implementation
        supports for the representation of the
        credentials. This element(s) that use this type,
        i.e. credential-interface,  should be used by
```

CODE EXAMPLE A-2   Connector Architecture 1.5 Resource Adapter XSD

```
            application server to find out the Credential
            interface it should use as part of the security
            contract.

            The possible values are:

            javax.resource.spi.security.PasswordCredential
            org.ietf.jgss.GSSCredential
            javax.resource.spi.security.GenericCredential

        </xsd:documentation>
     </xsd:annotation>
     <xsd:simpleContent>
         <xsd:restriction base="j2ee:fully-qualified-classType">
           <xsd:enumeration value=
          "javax.resource.spi.security.PasswordCredential"/>
           <xsd:enumeration value="org.ietf.jgss.GSSCredential"/>
           <xsd:enumeration value=
          "javax.resource.spi.security.GenericCredential"/>
         </xsd:restriction>
     </xsd:simpleContent>
   </xsd:complexType>

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

   <xsd:complexType name="inbound-resourceadapterType">

      <xsd:annotation>
         <xsd:documentation>
         The inbound-resourceadapterType specifies information
         about an inbound resource adapter. This contains
        information
         specific to the implementation of the resource adapter
         library as specified through the messageadapter element.

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

         <xsd:sequence>
         <xsd:element name="messageadapter"
         type="j2ee:messageadapterType"
         minOccurs="0">
            <xsd:unique name="messagelistener-type-uniqueness">
              <xsd:annotation>
                <xsd:documentation>

               The messagelistener-type element content must be
```

CODE EXAMPLE A-2   Connector Architecture 1.5 Resource Adapter XSD

```
                      unique in the messageadapter. Several
                     messagelisteners
                       can not use the same messagelistener-type.

                       </xsd:documentation>
                   </xsd:annotation>
                   <xsd:selector xpath="j2ee:messagelistener"/>
                   <xsd:field    xpath="j2ee:messagelistener-type"/>
              </xsd:unique>
          </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

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


  <xsd:complexType name="licenseType">
      <xsd:annotation>
          <xsd:documentation>

          The licenseType specifies licensing requirements for the
          resource adapter module. This type specifies whether a
          license is required to deploy and use this resource adapter,
          and an optional description of the licensing terms
          (examples: duration of license, number of connection
          restrictions). It is used by the license element.

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

      <xsd:sequence>
          <xsd:element name="description"
          type="j2ee:descriptionType"
          minOccurs="0"
          maxOccurs="unbounded"/>
          <xsd:element name="license-required"
          type="j2ee:true-falseType">
              <xsd:annotation>
                  <xsd:documentation>

                  The element license-required specifies whether a
                  license is required to deploy and use the
                  resource adapter. This element must be one of
                  the following, "true" or "false".

                  </xsd:documentation>
```

CODE EXAMPLE A-2   Connector Architecture 1.5 Resource Adapter XSD

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

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

    <xsd:complexType name="messageadapterType">
      <xsd:annotation>
        <xsd:documentation>

       The messageadapterType specifies information about the
        messaging capabilities of the resource adapter. This
        contains information specific to the implementation of the
        resource adapter library as specified through the
        messagelistener element.

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

      <xsd:sequence>
       <xsd:element name="messagelistener"
       type="j2ee:messagelistenerType"
       maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:ID"/>
   </xsd:complexType>

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

   <xsd:complexType name="messagelistenerType">
      <xsd:annotation>
         <xsd:documentation>

         The messagelistenerType specifies information about a
         specific message listener supported by the messaging
         resource adapter. It contains information on the Java
        type
         of the message listener interface and an activation
        specification.

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

      <xsd:sequence>
         <xsd:element name="messagelistener-type"
```

CODE EXAMPLE A-2   Connector Architecture 1.5 Resource Adapter XSD

```
                type="j2ee:fully-qualified-classType">
              <xsd:annotation>
                 <xsd:documentation>
                 <![CDATA[

                 The element messagelistener-type specifies the
              fully
                 qualified name of the Java type of a message
                 listener interface.

                 Example:

                 <messagelistener-type>javax.jms.MessageListener
                 </messagelistener-type>

                 ]]>
                 </xsd:documentation>
              </xsd:annotation>
          </xsd:element>
          <xsd:element name="activationspec"
          type="j2ee:activationspecType"/>
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:ID"/>
   </xsd:complexType>

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


   <xsd:complexType name="outbound-resourceadapterType">
      <xsd:annotation>
          <xsd:documentation>

          The outbound-resourceadapterType specifies information
          about
           an outbound resource adapter. The information includes
          fully
           qualified names of classes/interfaces required as part
          of
           the connector architecture specified contracts for
           connection management, level of transaction support
           provided, one or more authentication mechanisms
          supported
           and additional required security permissions.

           If there is no authentication-mechanism specified as
          part of
```

```
               resource adapter element then the resource adapter does
             not
              support any standard security authentication mechanisms
             as
              part of security contract. The application server
             ignores
              the security part of the system contracts in this case.

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

           <xsd:sequence>
              <xsd:element name="connection-definition"
              type="j2ee:connection-definitionType"
              maxOccurs="unbounded"/>
              <xsd:element name="transaction-support"
              type="j2ee:transaction-supportType"/>
              <xsd:element name="authentication-mechanism"
              type="j2ee:authentication-mechanismType"
              minOccurs="0"
              maxOccurs="unbounded"/>
              <xsd:element name="reauthentication-support"
              type="j2ee:true-falseType">
                 <xsd:annotation>
                    <xsd:documentation>

                    The element reauthentication-support specifies
                    whether the resource adapter implementation
                   supports
                    re-authentication of existing Managed-
                   Connection
                    instance. Note that this information is for the
                    resource adapter implementation and not for the
                    underlying EIS instance. This element must have
                    either a "true" or "false" value.

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

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

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

         <xsd:complexType name="required-config-propertyType">
```

CODE EXAMPLE A-2   Connector Architecture 1.5 Resource Adapter XSD

```
        <xsd:annotation>
           <xsd:documentation>
           <![CDATA[

           The required-config-propertyType contains a
           declaration of a single configuration property used
           for specifying a required configuration property
           name. It is used by required-config-property
            elements.
           Example:

           <required-config-property>Destination</required-
          config-property>

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

        <xsd:sequence>
           <xsd:element name="description"
           type="j2ee:descriptionType"
           minOccurs="0"
           maxOccurs="unbounded"/>
           <xsd:element name="config-property-name"
           type="j2ee:config-property-nameType"/>
           </xsd:sequence>
           <xsd:attribute name="id" type="xsd:ID"/>
       </xsd:complexType>

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

       <xsd:complexType name="resourceadapterType">
          <xsd:annotation>
             <xsd:documentation>

             The resourceadapterType specifies information
           about the
            resource adapter. The information includes
            resource adapter Java class name, configuration
           properties,
            information specific to the implementation of the
           resource
            adapter library as specified through the
            outbound-resourceadapter and inbound-
           resourceadapter
            elements, and an optional set of administered
           objects.
```

CODE EXAMPLE A-2   Connector Architecture 1.5 Resource Adapter XSD

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

        <xsd:sequence>
            <xsd:element name="resourceadapter-class"
            type="j2ee:fully-qualified-classType"
            minOccurs="0">
                <xsd:annotation>
                  <xsd:documentation>

                 The element resourceadapter-class specifies
                the
                  name of a Java class that implements
                  the javax.resource.spi.ResourceAdapter
                  interface. This Java class is provided as
                part of
                  resource adapter's implementation of
                connector
                  architecture specified contracts. The
                implementation
                  of this class is required to be a JavaBean.

                  </xsd:documentation>
                  </xsd:annotation>
            </xsd:element>
            <xsd:element name="config-property"
            type="j2ee:config-propertyType"
            minOccurs="0"
            maxOccurs="unbounded"/>
            <xsd:element name="outbound-resourceadapter"
            type="j2ee:outbound-resourceadapterType"
            minOccurs="0">
                <xsd:unique name="connectionfactory-
                interface-uniqueness">
                  <xsd:annotation>
                    <xsd:documentation>

                    The connectionfactory-interface element
                   content
                    must be unique in the outbound-
                   resourceadapter.
                    Multiple connection-definitions can not
                   use the
                    same connectionfactory-type.

                    </xsd:documentation>
```

```
                </xsd:annotation>
                 <xsd:selector xpath="j2ee:connection-
                definition"/>
                 <xsd:field     xpath=
                "j2ee:connectionfactory-interface"/>
              </xsd:unique>
            </xsd:element>
          <xsd:element name="inbound-resourceadapter"
          type="j2ee:inbound-resourceadapterType"
          minOccurs="0"/>
          <xsd:element name="adminobject"
          type="j2ee:adminobjectType"
          minOccurs="0"
          maxOccurs="unbounded"/>
          <xsd:element name="security-permission"
          type="j2ee:security-permissionType"
          minOccurs="0"
          maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:ID"/>
     </xsd:complexType>

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

     <xsd:complexType name="security-permissionType">
        <xsd:annotation>
           <xsd:documentation>

           The security-permissionType specifies a
          security
           permission that is required by the resource
          adapter code.

           The security permission listed in the
          deployment descriptor
           are ones that are different from those required
          by the
           default permission set as specified in the
          connector
           specification. The optional description can
          mention specific
           reason that resource adapter requires a given
          security
           permission.

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

CODE EXAMPLE A-2   Connector Architecture 1.5 Resource Adapter XSD

```
             <xsd:sequence>
                <xsd:element name="description"
                type="j2ee:descriptionType"
                minOccurs="0"
                maxOccurs="unbounded"/>
                <xsd:element name="security-permission-spec"
                type="j2ee:xsdStringType">
                <xsd:annotation>
                  <xsd:documentation>
                  The element security-permission-spec
                 specifies a security
                  permission based on the Security policy file
                  syntax. Refer to the following URL for Sun's
                  implementation of the security permission
                  specification:

                 http://java.sun.com/products/jdk/1.4/docs/gu
                 ide/security/PolicyFiles.html#FileSyntax

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

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

       <xsd:complexType name="transaction-supportType">
         <xsd:annotation>
           <xsd:documentation>

           The transaction-supportType specifies the level of
           transaction support provided by the resource
          adapter. It is
           used by transaction-support elements.

           The value must be one of the following:

           NoTransaction
           LocalTransaction
           XATransaction

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

```
                  <xsd:restriction base="j2ee:string">
                    <xsd:enumeration value="NoTransaction"/>
                    <xsd:enumeration value="LocalTransaction"/>
                    <xsd:enumeration value="XATransaction"/>
                  </xsd:restriction>
              </xsd:simpleContent>
          </xsd:complexType>

      </xsd:schema>
```

A.3   Java EE Connector Architecture 1.6 Resource Adapter XML XSD

CODE EXAMPLE A-3   Connector Architecture 1.6 Resource Adapter XSD

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://java.sun.com/xml/ns/javaee"
        xmlns:javaee="http://java.sun.com/xml/ns/javaee"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        elementFormDefault="qualified"
        attributeFormDefault="unqualified"
        version="1.6">
  <xsd:annotation>
    <xsd:documentation>
      $Id$
    </xsd:documentation>
  </xsd:annotation>

  <xsd:annotation>
    <xsd:documentation>
      <![CDATA[

    This is the XML Schema for the Connector 1.6 deployment
    descriptor.  The deployment descriptor must be named
    "META-INF/ra.xml" in the connector's rar file.  All Connector
    deployment descriptors must indicate the connector resource
    adapter schema by using the Java EE namespace:

    http://java.sun.com/xml/ns/javaee

    and by indicating the version of the schema by
```

CODE EXAMPLE A-3   Connector Architecture 1.6 Resource Adapter XSD

```
    using the version element as shown below:

        <connector xmlns="http://java.sun.com/xml/ns/javaee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/connector_1_6.xsd"
          version="1.6">
          ...
        </connector>

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

    http://java.sun.com/xml/ns/javaee/connector_1_6.xsd

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

  <xsd:annotation>
    <xsd:documentation>

      The following conventions apply to all Java EE
      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="javaee_6.xsd"/>


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


  <xsd:element name="connector" type="javaee:connectorType">
    <xsd:annotation>
      <xsd:documentation>
```

CODE EXAMPLE A-3   Connector Architecture 1.6 Resource Adapter XSD

```
     The connector element is the root element of the deployment
     descriptor for the resource adapter. This element includes
     general information - vendor name, resource adapter version,
     icon - about the resource adapter module. It also includes
     information specific to the implementation of the resource
     adapter library as specified through the element
     resourceadapter.

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

   </xsd:element>

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

   <xsd:complexType name="activationspecType">
     <xsd:annotation>
       <xsd:documentation>

     The activationspecType specifies an activation
     specification.  The information includes
     Java class name of an activation specification and a set of
     required configuration property names.

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

     <xsd:sequence>
       <xsd:element name="activationspec-class"
        type="javaee:fully-qualified-classType">
     <xsd:annotation>
       <xsd:documentation>
         <![CDATA[

           The element activationspec-class specifies the fully
           qualified Java class name of the activation
           specification class. This class must implement the
           javax.resource.spi.ActivationSpec interface. The
           implementation of this class is required to be a
           JavaBean.

           Example:
       <activationspec-class>com.wombat.ActivationSpecImpl
       </activationspec-class>

           ]]>
```

CODE EXAMPLE A-3　Connector Architecture 1.6 Resource Adapter XSD

```
                </xsd:documentation>
        </xsd:annotation>
          </xsd:element>
          <xsd:element name="required-config-property"
           type="javaee:required-config-propertyType"
           minOccurs="0"
           maxOccurs="unbounded"/>
          <xsd:element name="config-property"
           type="javaee:config-propertyType"
           minOccurs="0"
           maxOccurs="unbounded">
   <xsd:annotation>
   <xsd:documentation>
       The required-config-property element is deprecated since
       Connector 1.6 specification. The resource adapter
       implementation is recommended to use the @NotNull
       Bean Validation annotation or its XML validation
       descriptor equivalent to indicate that a configuration
       property is required to be specified by the deployer.
       See the Connectors specification for more information.
   </xsd:documentation>
   </xsd:annotation>

          </xsd:element>

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

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

      <xsd:complexType name="adminobjectType">
        <xsd:annotation>
          <xsd:documentation>

        The adminobjectType specifies information about an
        administered object.  Administered objects are specific to a
        messaging style or message provider.  This contains
        information on the Java type of the interface implemented by
        an administered object, its Java class name and its
        configuration properties.

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

        <xsd:sequence>
          <xsd:element name="adminobject-interface"
```

CODE EXAMPLE A-3   Connector Architecture 1.6 Resource Adapter XSD

```
      type="javaee:fully-qualified-classType">
    <xsd:annotation>
      <xsd:documentation>
        <![CDATA[

          The element adminobject-interface specifies the
           name of the Java type of the
          interface implemented by an administered object.

          Example:
    <adminobject-interface>javax.jms.Destination
    </adminobject-interface>

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

      </xsd:element>
      <xsd:element name="adminobject-class"
       type="javaee:fully-qualified-classType">
    <xsd:annotation>
      <xsd:documentation>
        <![CDATA[

          The element adminobject-class specifies the fully
          qualified Java class name of an administered object.

          Example:
    <adminobject-class>com.wombat.DestinationImpl
    </adminobject-class>

          ]]>
      </xsd:documentation>
    </xsd:annotation>
      </xsd:element>

      <xsd:element name="config-property"
       type="javaee:config-propertyType"
       minOccurs="0"
       maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

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

  <xsd:complexType name="authentication-mechanismType">
```

CODE EXAMPLE A-3   Connector Architecture 1.6 Resource Adapter XSD

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

The authentication-mechanismType specifies an authentication
mechanism supported by the resource adapter. Note that this
support is for the resource adapter and not for the
underlying EIS instance. The optional description specifies
any resource adapter specific requirement for the support of
security contract and authentication mechanism.

Note that BasicPassword mechanism type should support the
javax.resource.spi.security.PasswordCredential interface.
The Kerbv5 mechanism type should support the
org.ietf.jgss.GSSCredential interface or the deprecated
javax.resource.spi.security.GenericCredential interface.

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

<xsd:sequence>
  <xsd:element name="description"
   type="javaee:descriptionType"
   minOccurs="0"
   maxOccurs="unbounded"/>
  <xsd:element name="authentication-mechanism-type"
   type="javaee:xsdStringType">
<xsd:annotation>
  <xsd:documentation>
    <![CDATA[

      The element authentication-mechanism-type specifies
      type of an authentication mechanism.

      The example values are:

      <authentication-mechanism-type>BasicPassword
      </authentication-mechanism-type>

      <authentication-mechanism-type>Kerbv5
      </authentication-mechanism-type>

      Any additional security mechanisms are outside the
      scope of the Connector architecture specification.

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

CODE EXAMPLE A-3   Connector Architecture 1.6 Resource Adapter XSD

```
      </xsd:element>
      <xsd:element name="credential-interface"
       type="javaee:credential-interfaceType"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

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

  <xsd:complexType name="config-property-nameType">
    <xsd:annotation>
      <xsd:documentation>
    <![CDATA[

      The config-property-nameType contains the name of a
      configuration property.

      The connector architecture defines a set of well-defined
      properties all of type java.lang.String. These are as
      follows.

            ServerName
            PortNumber
            UserName
            Password
            ConnectionURL

      A resource adapter provider can extend this property set to
      include properties specific to the resource adapter and its
      underlying EIS.

      Possible values include
      ServerName
      PortNumber
      UserName
      Password
      ConnectionURL

      Example: <config-property-name>ServerName</config-property-
name>

      ]]>
      </xsd:documentation>
    </xsd:annotation>
    <xsd:simpleContent>
      <xsd:restriction base="javaee:xsdStringType"/>
```

CODE EXAMPLE A-3   Connector Architecture 1.6 Resource Adapter XSD

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

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

  <xsd:complexType name="config-property-typeType">
    <xsd:annotation>
      <xsd:documentation>
    <![CDATA[

      The config-property-typeType contains the fully
      qualified Java type of a configuration property.

      The following are the legal values:
         java.lang.Boolean, java.lang.String, java.lang.Integer,
         java.lang.Double, java.lang.Byte, java.lang.Short,
         java.lang.Long, java.lang.Float, java.lang.Character

      Used in: config-property

      Example:
      <config-property-type>java.lang.String</config-property-
type>

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

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

  <xsd:complexType name="config-propertyType">
    <xsd:annotation>
      <xsd:documentation>
```

CODE EXAMPLE A-3   Connector Architecture 1.6 Resource Adapter XSD

```
    The config-propertyType contains a declaration of a single
    configuration property that may be used for providing
    configuration information.

    The declaration consists of an optional description, name,
    type and an optional value of the configuration property. If
    the resource adapter provider does not specify a value than
    the deployer is responsible for providing a valid value for
    a configuration property.

    Any bounds or well-defined values of properties should be
    described in the description element.

      </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
      <xsd:element name="description"
       type="javaee:descriptionType"
       minOccurs="0"
       maxOccurs="unbounded"/>
      <xsd:element name="config-property-name"
       type="javaee:config-property-nameType"/>
      <xsd:element name="config-property-type"
       type="javaee:config-property-typeType"/>
      <xsd:element name="config-property-value"
       type="javaee:xsdStringType"
       minOccurs="0">
    <xsd:annotation>
      <xsd:documentation>
        <![CDATA[

          The element config-property-value contains the value
          of a configuration entry. Note, it is possible for a
          resource adapter deployer to override this
          configuration information during deployment.

          Example:
          <config-property-value>WombatServer</config-property-
value>

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

```
    <xsd:element name="config-property-ignore" type="javaee:true-
falseType"
    minOccurs="0" maxOccurs="1">
    <xsd:annotation>
       <xsd:documentation>
       <![CDATA[

         The element config-property-ignore is used to specify
         whether the configuration tools must ignore considering
the
         configuration property during auto-discovery of
        Configuration properties. See the Connector specification
for
         more details. If unspecified, the container must not
ignore
         the configuration property during auto-discovery.
         This element must be one of the following, "true" or
"false".

         ]]>
      </xsd:documentation>
    </xsd:annotation>
    </xsd:element>
    <xsd:element name="config-property-supports-dynamic-updates"
type "javaee:true-falseType"
    minOccurs="0" maxOccurs="1">
    <xsd:annotation>
       <xsd:documentation>
       <![CDATA[
         The element config-property-supports-dynamic-updates is
used to specify
         whether the configuration property allows its value to
be updated, by
         application server's configuration tools, during the
lifetime of
        the JavaBean instance. See the Connector specification for
         more details. If unspecified, the container must not
dynamically
         reconfigure the property.
         This element must be one of the following, "true" or
"false".

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

CODE EXAMPLE A-3   Connector Architecture 1.6 Resource Adapter XSD

```
      <xsd:element name="config-property-confidential" type=
"javaee:true-falseType"
    minOccurs="0" maxOccurs="1">
    <xsd:annotation>
       <xsd:documentation>
       <![CDATA[
         The element config-property-confidential is used to
specify
         whether the configuration property is confidential and
         recommends application server's configuration tools to
use special
         visual aids for editing them. See the Connector
specification for
       more details. If unspecified, the container must not treat
the
         property as confidential.

         This element must be one of the following, "true" or
"false".
         ]]>
     </xsd:documentation>
    </xsd:annotation>
    </xsd:element>


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

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

  <xsd:complexType name="connection-definitionType">
    <xsd:annotation>
      <xsd:documentation>

    The connection-definitionType defines a set of connection
    interfaces and classes pertaining to a particular connection
    type. This also includes configurable properties for
    ManagedConnectionFactory instances that may be produced out
    of this set.

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

    <xsd:sequence>
      <xsd:element name="managedconnectionfactory-class"
       type="javaee:fully-qualified-classType">
```

CODE EXAMPLE A-3   Connector Architecture 1.6 Resource Adapter XSD

```
     <xsd:annotation>
       <xsd:documentation>
         <![CDATA[

          The element managedconnectionfactory-class specifies
          the  name of the Java class that
          implements the
          javax.resource.spi.ManagedConnectionFactory interface.
          This Java class is provided as part of resource
          adapter's implementation of connector architecture
          specified contracts. The implementation of this
          class is required to be a JavaBean.

          Example:
          <managedconnectionfactory-class>
        com.wombat.ManagedConnectionFactoryImpl
          </managedconnectionfactory-class>

          ]]>
       </xsd:documentation>
    </xsd:annotation>
      </xsd:element>
      <xsd:element name="config-property"
       type="javaee:config-propertyType"
       minOccurs="0"
       maxOccurs="unbounded"/>
      <xsd:element name="connectionfactory-interface"
       type="javaee:fully-qualified-classType">
    <xsd:annotation>
      <xsd:documentation>
        <![CDATA[

          The element connectionfactory-interface specifies
          the  name of the ConnectionFactory
          interface supported by the resource adapter.

          Example:
          <connectionfactory-
interface>com.wombat.ConnectionFactory
          </connectionfactory-interface>

          OR

          <connectionfactory-
interface>javax.resource.cci.ConnectionFactory
          </connectionfactory-interface>
```

CODE EXAMPLE A-3   Connector Architecture 1.6 Resource Adapter XSD

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

      <xsd:element name="connectionfactory-impl-class"
       type="javaee:fully-qualified-classType">
    <xsd:annotation>
      <xsd:documentation>
        <![CDATA[

          The element connectionfactory-impl-class specifies
          the  name of the ConnectionFactory
          class that implements resource adapter
          specific ConnectionFactory interface.

          Example:

          <connectionfactory-impl-
class>com.wombat.ConnectionFactoryImpl
          </connectionfactory-impl-class>

          ]]>
      </xsd:documentation>
    </xsd:annotation>
      </xsd:element>
      <xsd:element name="connection-interface"
       type="javaee:fully-qualified-classType">
    <xsd:annotation>
      <xsd:documentation>
        <![CDATA[

          The connection-interface element specifies the fully
          qualified name of the Connection interface supported
          by the resource adapter.

          Example:

      <connection-interface>javax.resource.cci.Connection
      </connection-interface>

          ]]>
      </xsd:documentation>
    </xsd:annotation>
      </xsd:element>
      <xsd:element name="connection-impl-class"
       type="javaee:fully-qualified-classType">
```

CODE EXAMPLE A-3   Connector Architecture 1.6 Resource Adapter XSD

```
      <xsd:annotation>
        <xsd:documentation>
          <![CDATA[

            The connection-impl-classType specifies the fully
            qualified name of the Connection class that
            implements resource adapter specific Connection
            interface.  It is used by the connection-impl-class
            elements.

            Example:

        <connection-impl-class>com.wombat.ConnectionImpl
        </connection-impl-class>

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

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

    <xsd:complexType name="connectorType">
      <xsd:annotation>
        <xsd:documentation>

      The connectorType defines a resource adapter.

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

      <xsd:sequence>
        <xsd:element name="module-name" type="javaee:string"
      minOccurs="0">
          <xsd:annotation>
            <xsd:documentation>

              The element module-name specifies the name of the
              resource adapter.

              If there is no module-name specified, the module-name
              is determined as defined in Section EE.8.1.1 and
              EE.8.1.2
              of the Java Platform, Enterprise Edition (Java EE)
```

CODE EXAMPLE A-3   Connector Architecture 1.6 Resource Adapter XSD

```
            Specification, version 6.
          </xsd:documentation>
        </xsd:annotation>
      </xsd:element>
      <xsd:group ref="javaee:descriptionGroup"/>
      <xsd:element name="vendor-name"
       type="javaee:xsdStringType" minOccurs="0">
    <xsd:annotation>
      <xsd:documentation>


        The element vendor-name specifies the name of
        resource adapter provider vendor.

      If there is no vendor-name specified, the application server
      server must consider the default "" (empty string) as
      the name of the resource adapter provider vendor.

      </xsd:documentation>
    </xsd:annotation>
      </xsd:element>
      <xsd:element name="eis-type"
       type="javaee:xsdStringType" minOccurs="0">
    <xsd:annotation>
      <xsd:documentation>


        The element eis-type contains information about the
        type of the EIS. For example, the type of an EIS can
        be product name of EIS independent of any version
        info.

        This helps in identifying EIS instances that can be
        used with this resource adapter.

      If there is no eis-type specified, the application
      server must consider the default "" (empty string) as
      the type of the EIS.

      </xsd:documentation>
    </xsd:annotation>
      </xsd:element>
      <xsd:element name="resourceadapter-version"
       type="javaee:xsdStringType" minOccurs="0">
    <xsd:annotation>
      <xsd:documentation>

        The element resourceadapter-version specifies a string-
based version
```

CODE EXAMPLE A-3   Connector Architecture 1.6 Resource Adapter XSD

```
          of the resource adapter from the resource adapter
          provider.

          If there is no resourceadapter-version specified, the
        application
          server must consider the default "" (empty string) as
          the version of the resource adapter.

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

      </xsd:element>
      <xsd:element name="license"
       type="javaee:licenseType"
       minOccurs="0"/>
      <xsd:element name="resourceadapter"
       type="javaee:resourceadapterType"/>

      <xsd:element name="required-work-context"
       type="javaee:fully-qualified-classType"
       minOccurs="0"
       maxOccurs="unbounded">
     <xsd:annotation>
      <xsd:documentation>

        The element required-work-context specifies the
         class name that implements WorkContext interface,
       that a resource adapter requires the application server to
support.

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

       </xsd:element>

    </xsd:sequence>
    <xsd:attribute name="version"
       type="javaee:dewey-versionType"
       fixed="1.6"
       use="required">
      <xsd:annotation>
    <xsd:documentation>

     The version indicates the version of the schema to be used by
the
```

CODE EXAMPLE A-3   Connector Architecture 1.6 Resource Adapter XSD

```
      deployment tool. This attribute does not have a default, and
the
      resource adapter developer/deployer is required to specify it.
      This attribute allows the deployment tool to choose which
      schema to validate the descriptor against.


    </xsd:documentation>
      </xsd:annotation>
   </xsd:attribute>
   <xsd:attribute name="metadata-complete" type="xsd:boolean">
   <xsd:annotation>
      <xsd:documentation>
        The metadata-complete attribute defines whether the
        deployment
        descriptor for the resource adapter module is complete,
        or whether
        the class files available to the module and packaged
        with the resource
        adapter should be examined for annotations that specify
        deployment
        information.

        If metadata-complete is set to "true", the deployment
        tool of the
        application server must ignore any annotations that
        specify
        deployment information, which might be present in the
        class files of
        the application.If metadata-complete is not specified
        or is set to
        "false", the deployment tool must examine the class
        files of the
        application for annotations, as specified by this
        specification. If
        the deployment descriptor is not included or is included
        but not
        marked metadata-complete, the deployment tool will
        process
        annotations

        Application servers must assume that metadata-complete
        is true for
        resource adapter modules with deployment descriptor
        version
        lower than 1.6.
      </xsd:documentation>
```

CODE EXAMPLE A-3　　Connector Architecture 1.6 Resource Adapter XSD

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

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

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

  <xsd:complexType name="credential-interfaceType">
    <xsd:annotation>
      <xsd:documentation>

    The credential-interfaceType specifies the
    interface that the resource adapter implementation
    supports for the representation of the
    credentials. This element(s) that use this type,
    i.e. credential-interface,  should be used by
    application server to find out the Credential
    interface it should use as part of the security
    contract.

    The possible values are:

    javax.resource.spi.security.PasswordCredential
    org.ietf.jgss.GSSCredential
    javax.resource.spi.security.GenericCredential

      </xsd:documentation>
    </xsd:annotation>
    <xsd:simpleContent>
      <xsd:restriction base="javaee:fully-qualified-classType">
    <xsd:enumeration
        value="javax.resource.spi.security.PasswordCredential"/>
    <xsd:enumeration
        value="org.ietf.jgss.GSSCredential"/>
    <xsd:enumeration
        value="javax.resource.spi.security.GenericCredential"/>
      </xsd:restriction>
    </xsd:simpleContent>
  </xsd:complexType>

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

  <xsd:complexType name="inbound-resourceadapterType">
    <xsd:annotation>
      <xsd:documentation>
```

CODE EXAMPLE A-3   Connector Architecture 1.6 Resource Adapter XSD

```
      The inbound-resourceadapterType specifies information
      about an inbound resource adapter. This contains information
      specific to the implementation of the resource adapter
      library as specified through the messageadapter element.

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

      <xsd:sequence>
        <xsd:element name="messageadapter"
         type="javaee:messageadapterType"
         minOccurs="0">
      <xsd:unique name="messagelistener-type-uniqueness">
        <xsd:annotation>
          <xsd:documentation>

            The messagelistener-type element content must be
            unique in the messageadapter. Several messagelisteners
            can not use the same messagelistener-type.

          </xsd:documentation>
        </xsd:annotation>
        <xsd:selector xpath="javaee:messagelistener"/>
        <xsd:field    xpath="javaee:messagelistener-type"/>
      </xsd:unique>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:ID"/>
    </xsd:complexType>

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

    <xsd:complexType name="licenseType">
      <xsd:annotation>
        <xsd:documentation>

      The licenseType specifies licensing requirements for the
      resource adapter module. This type specifies whether a
      license is required to deploy and use this resource adapter,
      and an optional description of the licensing terms
      (examples: duration of license, number of connection
      restrictions). It is used by the license element.

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

      <xsd:sequence>
```

CODE EXAMPLE A-3   Connector Architecture 1.6 Resource Adapter XSD

```
        <xsd:element name="description"
         type="javaee:descriptionType"
         minOccurs="0"
         maxOccurs="unbounded"/>
        <xsd:element name="license-required"
         type="javaee:true-falseType">
    <xsd:annotation>
      <xsd:documentation>

        The element license-required specifies whether a
        license is required to deploy and use the
        resource adapter. This element must be one of
        the following, "true" or "false".

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

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

   <xsd:complexType name="messageadapterType">
     <xsd:annotation>
       <xsd:documentation>

    The messageadapterType specifies information about the
    messaging capabilities of the resource adapter. This
    contains information specific to the implementation of the
    resource adapter library as specified through the
    messagelistener element.

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

     <xsd:sequence>
       <xsd:element name="messagelistener"
        type="javaee:messagelistenerType"
        maxOccurs="unbounded"/>
     </xsd:sequence>
     <xsd:attribute name="id" type="xsd:ID"/>
   </xsd:complexType>

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

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

CODE EXAMPLE A-3   Connector Architecture 1.6 Resource Adapter XSD

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

    The messagelistenerType specifies information about a
    specific message listener supported by the messaging
    resource adapter. It contains information on the Java type
    of the message listener interface and an activation
    specification.

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

    <xsd:sequence>
      <xsd:element name="messagelistener-type"
       type="javaee:fully-qualified-classType">
    <xsd:annotation>
      <xsd:documentation>
        <![CDATA[

          The element messagelistener-type specifies the fully
          qualified name of the Java type of a message
          listener interface.

          Example:

    <messagelistener-type>javax.jms.MessageListener
    </messagelistener-type>

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

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

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

  <xsd:complexType name="outbound-resourceadapterType">
    <xsd:annotation>
      <xsd:documentation>

    The outbound-resourceadapterType specifies information about
    an outbound resource adapter. The information includes fully
```

CODE EXAMPLE A-3   Connector Architecture 1.6 Resource Adapter XSD

```
    qualified names of classes/interfaces required as part of
    the connector architecture specified contracts for
    connection management, level of transaction support
    provided, one or more authentication mechanisms supported
    and additional required security permissions.

     If any of the outbound resource adapter elements (transaction-
    support,
     authentication-mechanism, reauthentication-support) is
    specified through
     this element or metadata annotations, and no  connection-
    definition is
     specified as part of this element or through annotations, the
     application server must consider this an error and fail
    deployment.

    If there is no authentication-mechanism specified as part of
    this or metadata annotations, then the resource adapter does
not
    support any standard security authentication mechanisms as
    part of security contract. The application server ignores
    the security part of the system contracts in this case.

     If there is no transaction-support specified as part of
     this element, then the application server must
     consider that the resource adapter does not support either
     the resource manager local or JTA transactions and must
    consider
     the transaction support as NoTransaction. Note that resource
    adapters may
      specify the level of transaction support to be used at runtime
      for a
      ManagedConnectionFactory through the TransactionSupport
      interface.

     If there is no reauthentication-support specified as part of
     this element or metadata annotation, then the application
    server must
     consider that the resource adapter does not support re-
    authentication of
     ManagedConnections.

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

     <xsd:sequence>
       <xsd:element name="connection-definition"
```

CODE EXAMPLE A-3 Connector Architecture 1.6 Resource Adapter XSD

```
          type="javaee:connection-definitionType"
          maxOccurs="unbounded" minOccurs="0"/>
        <xsd:element name="transaction-support"
         type="javaee:transaction-supportType" minOccurs="0"/>
        <xsd:element name="authentication-mechanism"
         type="javaee:authentication-mechanismType"
         minOccurs="0"
         maxOccurs="unbounded"/>
        <xsd:element name="reauthentication-support"
         type="javaee:true-falseType" minOccurs="0"/>
    <xsd:annotation>
      <xsd:documentation>

        The element reauthentication-support specifies
        whether the resource adapter implementation supports
        re-authentication of existing Managed- Connection
        instance. Note that this information is for the
        resource adapter implementation and not for the
        underlying EIS instance. This element must have
        either a "true" or "false" value.

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

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

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

  <xsd:complexType name="required-config-propertyType">
    <xsd:annotation>
      <xsd:documentation>
    <![CDATA[

      The required-config-propertyType contains a declaration
      of a single configuration property used for specifying a
      required configuration property name. It is used
      by required-config-property elements.


      Usage of this type is deprecated from Connector 1.6
specification.
      Refer to required-config-property element for more
information.
```

CODE EXAMPLE A-3   Connector Architecture 1.6 Resource Adapter XSD

```
      Example:

      <required-config-property>Destination</required-config-
property>

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

    <xsd:sequence>
      <xsd:element name="description"
       type="javaee:descriptionType"
       minOccurs="0"
       maxOccurs="unbounded"/>
      <xsd:element name="config-property-name"
       type="javaee:config-property-nameType"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

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

  <xsd:complexType name="resourceadapterType">
    <xsd:annotation>
      <xsd:documentation>

    The resourceadapterType specifies information about the
    resource adapter. The information includes
    resource adapter Java class name, configuration properties,
    information specific to the implementation of the resource
    adapter library as specified through the
    outbound-resourceadapter and inbound-resourceadapter
    elements, and an optional set of administered objects.

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

    <xsd:sequence>
      <xsd:element name="resourceadapter-class"
       type="javaee:fully-qualified-classType"
       minOccurs="0">
    <xsd:annotation>
      <xsd:documentation>

        The element resourceadapter-class specifies the
         name of a Java class that implements
         the javax.resource.spi.ResourceAdapter
```

```
              interface. This Java class is provided as part of
              resource adapter's implementation of connector
              architecture specified contracts. The implementation
              of this class is required to be a JavaBean.

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

      </xsd:element>
      <xsd:element name="config-property"
       type="javaee:config-propertyType"
       minOccurs="0"
       maxOccurs="unbounded"/>
      <xsd:element name="outbound-resourceadapter"
       type="javaee:outbound-resourceadapterType"
       minOccurs="0">
    <xsd:unique name="connectionfactory-interface-uniqueness">
      <xsd:annotation>
        <xsd:documentation>

          The connectionfactory-interface element content
          must be unique in the outbound-resourceadapter.
          Multiple connection-definitions can not use the
          same connectionfactory-type.

        </xsd:documentation>
      </xsd:annotation>
      <xsd:selector xpath="javaee:connection-definition"/>
      <xsd:field    xpath="javaee:connectionfactory-interface"/>
    </xsd:unique>
      </xsd:element>
      <xsd:element name="inbound-resourceadapter"
       type="javaee:inbound-resourceadapterType"
       minOccurs="0"/>
      <xsd:element name="adminobject"
       type="javaee:adminobjectType"
       minOccurs="0"
       maxOccurs="unbounded"/>

    <xsd:unique name="adminobject-type-uniqueness">
      <xsd:annotation>
        <xsd:documentation>

          The adminobject-interface and adminobject-class element
content must be
          unique in the resourceadapterType. Several admin objects
```

CODE EXAMPLE A-3   Connector Architecture 1.6 Resource Adapter XSD

```
           can not use the same adminobject-interface and
adminobject-class.

         </xsd:documentation>
      </xsd:annotation>
      <xsd:selector xpath="javaee:adminobject"/>
      <xsd:field    xpath="javaee:adminobject-interface"/>
      <xsd:field    xpath="javaee:adminobject-class"/>
    </xsd:unique>

      <xsd:element name="security-permission"
       type="javaee:security-permissionType"
       minOccurs="0"
       maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

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

  <xsd:complexType name="security-permissionType">
    <xsd:annotation>
      <xsd:documentation>

    The security-permissionType specifies a security
    permission that is required by the resource adapter code.

    The security permission listed in the deployment descriptor
    are ones that are different from those required by the
    default permission set as specified in the connector
    specification. The optional description can mention specific
    reason that resource adapter requires a given security
    permission.

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

    <xsd:sequence>
      <xsd:element name="description"
       type="javaee:descriptionType"
       minOccurs="0"
       maxOccurs="unbounded"/>
      <xsd:element name="security-permission-spec"
       type="javaee:xsdStringType">
    <xsd:annotation>
      <xsd:documentation>
```

CODE EXAMPLE A-3   Connector Architecture 1.6 Resource Adapter XSD

```
         The element security-permission-spec specifies a security
         permission based on the Security policy file
         syntax. Refer to the following URL for Sun's
         implementation of the security permission
         specification:


http://java.sun.com/javase/6/docs/technotes/guides/security/Polic
yFiles.html
      </xsd:documentation>
    </xsd:annotation>
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"/>
  </xsd:complexType>

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

  <xsd:complexType name="transaction-supportType">
    <xsd:annotation>
      <xsd:documentation>

    The transaction-supportType specifies the level of
    transaction support provided by the resource adapter. It is
    used by transaction-support elements.

    The value must be one of the following:

        NoTransaction
        LocalTransaction
        XATransaction

      </xsd:documentation>
    </xsd:annotation>
    <xsd:simpleContent>
      <xsd:restriction base="javaee:string">
    <xsd:enumeration value="NoTransaction"/>
    <xsd:enumeration value="LocalTransaction"/>
    <xsd:enumeration value="XATransaction"/>
      </xsd:restriction>
    </xsd:simpleContent>
  </xsd:complexType>

</xsd:schema>
```

Caching Manager

This chapter describes how the connector architecture supports caching.

This section serves as a brief introduction to the caching support in the connector architecture. A future version of the connector architecture will address this issue in detail.

B.1   Overview

The connector architecture provides a standard way of extending an application manager for plugging in caching managers. A caching manager may be provided by a third-party vendor or a resource adapter provider.

A caching manager manages cached state for application components while they access EISs across transactions.
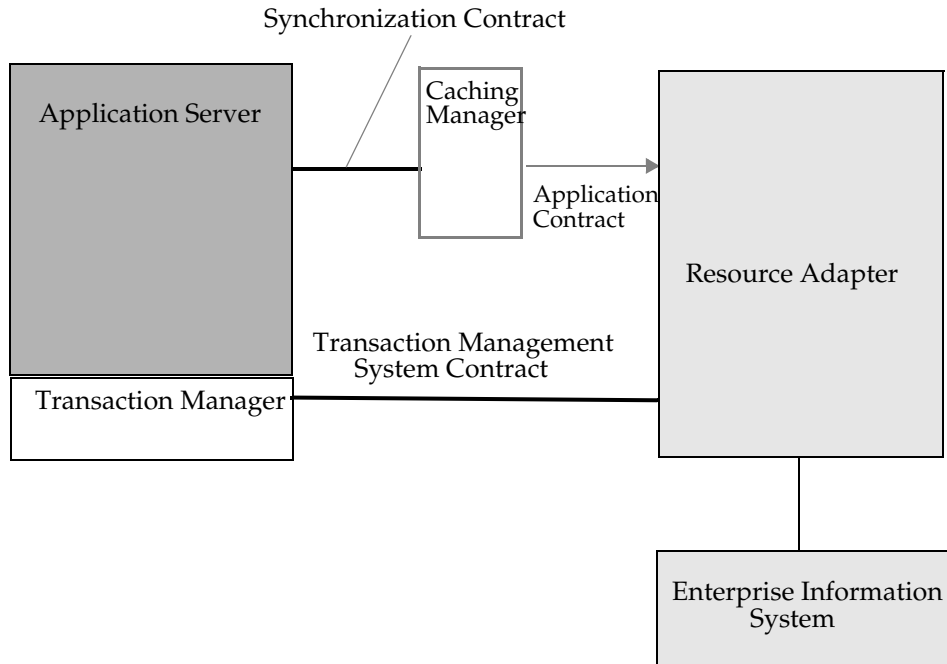
A caching manager is provided above a resource adapter. An application component may access a resource manager either through a caching manager (thereby maintaining a cached state across application requests) or directly through the resource adapter with no caching involved.

The `XAResource` based transaction management contract enables an external transaction manager to control and coordinate transactions across multiple resource managers. A caching manager (provided above the resource adapter) requires to be synchronized relative to the transaction coordination flow (defined by the JTA `XAResource` interface) on the underlying resource manager. This leads to a requirement for a synchronization contract between the application server and caching manager.

The connector architecture defines a standard synchronization contract between the application server and caching manager. The caching manager uses the synchronization notifications to manage its cached state and to flush it to the

resource adapter. The resource adapter then takes the responsibility of managing its recoverable units of work and participates in the transaction coordination protocol from the transaction manager.

FIGURE B-1 Synchronization Contract between Caching Manager and Application Server

The above diagram shows a caching manager layered above a resource adapter. The contract between caching manager and resource adapter is specific to a resource adapter.

B.2　Synchronization Contract

Note – To support a caching manager as a standard extension to the application server, additional contracts between the application server and the caching manager are required. This version of the specification introduces only the synchronization contract.

This section specifies the synchronization contract between the application server and the caching manager.

B.2.1 Interface

Each caching manager implements the `javax.transaction.Synchronization` interface. A caching manager registers its `Synchronization` instance with the application server when it is configured with the application server.

The caching manager receives synchronization notifications only for transactions managed by an external transaction manager. In the case of transactions managed internally by a resource manager, the resource adapter and caching manager define their own implementation-specific mechanisms for synchronizing caches.

The `Synchronization.beforeCompletion` method is called prior to the start of the two-phase commit transaction completion process. This call executes in the same transaction context of the caller who initiated the transaction completion. The caching manager uses this notification to flush its cached state to the resource adapter.

The `Synchronization.afterCompletion` method is called after the transaction has completed. The status of transaction completion is passed in as a parameter. The caching manager uses this notification to do any cache cleanups if a rollback has occurred.

B.2.2 Implementation

The caching manager must support the `javax.transaction.Synchronization` interface. If the caching manager implements the `Synchronization` interface and registers it with the application server, then the application server must invoke the `beforeCompletion` and `afterCompletion` notifications.

The application server is responsible for ensuring that synchronization notifications are delivered first to the application components (that have expressed interest in receiving synchronization notification through their respective application component and container-specific mechanisms) and then to the caching managers that implement the `Synchronization` interface.

Security Scenarios

This chapter describes various scenarios for EIS integration. These scenarios focus on security aspects of the connector architecture.

Note that these scenarios establish the requirements to be addressed by the connector architecture. Chapter 8, "Security Architecture" and Chapter 9, "Security Contract" specify the requirements that are supported in this version of the specification.

A Java EE application is a multi-tier, web-enabled application that accesses EISs. It consists of one or more application components—EJBs, JSPs, servlets—which are deployed on containers. These containers can be one of the following:

- Web containers that host JSP, servlets, and static HTML pages
- EJB containers that host EJB components
- Application client containers that host standalone application clients

In the following scenarios, the description of the architecture and security environments are illustrative in scope.
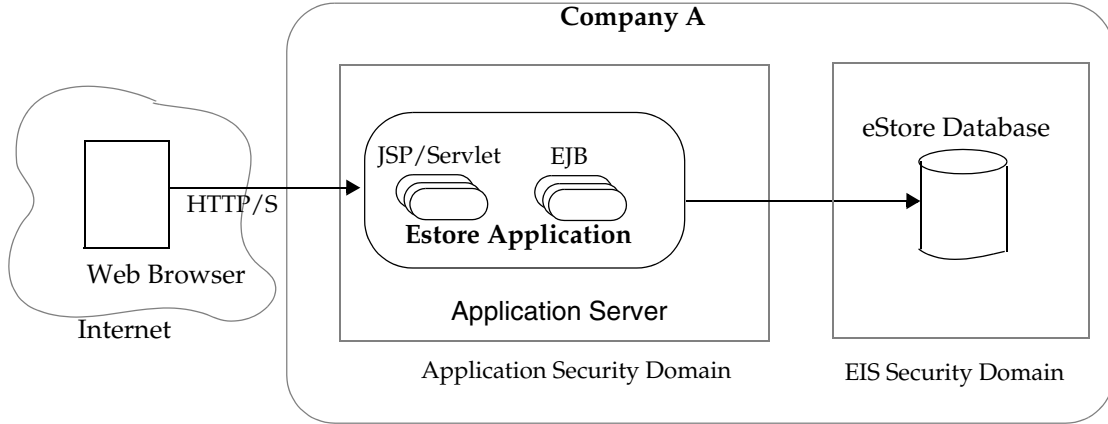
C.1 eStore Application

Company A has an eStore application based on the Java EE platform. The eStore application is composed of EJBs and JSP/servlets; together they collaborate to provide the overall functionality of the application. The application also utilizes an eStore database to store data related to product catalog, shopping carts, customer registration and profiles, transaction status and records, and order status.

The architecture of this application is illustrated in the following diagram.

FIGURE C-1   Illustrative Architecture of an eStore Application

C.1.1   Scenario

A customer, using a web browser, initiates an e-commerce transaction with the eStore application. The e-commerce transaction consists of a series of customer actions. The customer performs the following actions to place an order.

1. Browses the catalog

2. Makes a selection of products

3. Puts the selected products into a shopping cart

4. Enters her user name and password to initiate a secure transaction

5. Fills in order-related information

6. Places an order

In this scenario, the eStore application stores all persistent information about customers and their transactions in a database.

C.1.2   Security Environment

To support the above interaction scenario, the system administrator configures a unique security domain (with specific security technology and security policies) for the eStore application. A firewall protects this security domain from unauthorized Internet access.

The security domain configuration for the eStore application includes secure web access to the eStore application. Secure web access is set up based on the requirements specified in the Java EE specification. Note that the focus of this section is security related to EIS integration, not on web access security. As a result, this description ignores web access security.

The system administrator sets up a database to manage persistent data for the eStore application. In terms of security, the database system is configured with an independent security domain. This domain has its own set of user accounts, plus its own security policies and mechanisms for authentication and authorization.

The system administrator (or database administrator DBA) creates a unique database account (called `EStoreUser`) to handle database transactions; the database transactions correspond to different customer-driven interactions with the eStore application. He also sets up an additional database account (called `EStoreAdministrator`) to manage the database on behalf of the eStore administrator. This administrative account has a higher level of access privileges.

To facilitate better scaling of the eStore application, the system administrator may choose to set the load balancing of database operations across multiple databases. He may also partition persistent data and transactions across multiple database accounts, based on various performance optimization criteria. These areas are out of the scope for this document.

This scenario deals only with the simple case of a single database and a single user account to handle all database transactions.

C.1.3 Deployment

Note – This document does not address how principal delegation happens between the web and EJB containers. When an EJB instance acquires an EIS connection, a caller principal is associated with the EJB instance. This document does not address determining which caller principal is associated with the EJB instance.

During the deployment of the eStore application, the deployer sets up access control for all authenticated customer accounts—the customer accounts that are driving e-commerce transactions over the web—based on a single role `eStoreUserRole`.

The deployer configures the resource adapter with the security information that is required for the creation of database connections. This security information is the database account `EStoreUser` and its password.

The deployer sets up the resource principal for accessing the database system as illustrated in the figure below.

FIGURE C-2   Resource Principal for eStore Application Scenario
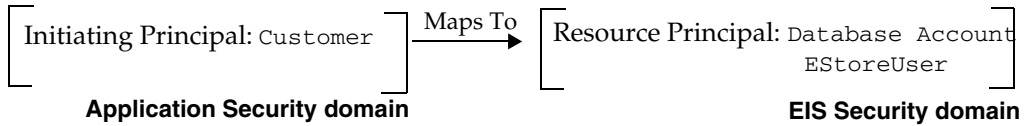
The deployment configuration ensures that all database access is always performed under the security context of the database account `EStoreUser`.

All authenticated customers (referred to as `Initiating Principal`) map to a single `EStoreUser` database account. The eStore application uses an implementation-specific mechanism to tie database transactions (performed under a single database account) to the unique identity (social security number or eStore account ID) of the initiating principal. To ensure that database access has been properly authorized, the eStore application also performs access control based on the role of the initiating principal. Because all initiating principals map to a single role, this is in effect a simple case.

This scenario describes an n-to-1 mapping. However, depending on the requirements of an application, the deployer can set the principal mapping to be different from an n-to-1 mapping. For example, the deployer can map each role to a single resource principal, where a role corresponds to an initiating principal. This results in a [*m* principals and *n* roles] to [*p* resource principals] mapping. When doing such principal mapping, the deployer has to ensure not to compromise the access rights of the mapped principals. An illustrative example is:

- User is in administrator role: Principal `EISadmin`
- User is in manager role: Principal `EISmanager`
- User is in employee role: Principal `EISemployee`

C.2   Employee Self-Service Application

Company B has developed and deployed an employee self-service (ESS) application based on the Java EE platform. This application supports a web interface to the existing Human Resources (HR) applications, which are supported by the ERP system from Vendor X. The ESS application also provides additional business processes customized to the requirements of Company B.

The application tier is composed of EJBs and JSPs that provide the customization of the business processes and support a company-standardized web interface. The ESS application enables an employee (under the roles of Manager, HR manager, and Employee) to perform various HR functions, including personal information management, payroll management, compensation management, benefits administration, travel management, and HR cost planning.

C.2.1   Architecture

The IS department of Company B has deployed its HR ESS application and ERP system in a secure environment on a single physical location. Any access to the HR application is permitted Only legal employees of the organization are permitted access to the HR application. Access is based on the employee's roles and access privileges. In addition, access to the application can only be from within the organization-wide intranet. See FIGURE C-3.

C.2.2   Security Environment

To support the various interaction scenarios related to the ESS application, the system administrator sets up an end-to-end Kerberos-based security domain for this application environment.


Note – The Security policies and mechanisms that are required to achieve this single security domain are technology dependent. Refer to the Kerberos V5 specification for more details.

The system administrator configures the security environment to support single sign-on; the user logs on only once and can then access all the services provided by the ESS application and its underlying ERP system. Single sign-on is achieved through the security mechanism and policies specific to the underlying security technology, which in this case is Kerberos.

The ERP system administrator configures all legal employees as valid user accounts in the ERP system. He also must set up various roles (Manager, HRManager, and Employee), default passwords, and access privileges. This security information is kept synchronized with the enterprise-wide directory service, which is used by Kerberos to perform the initial authentication of end-users.

FIGURE C-3 Illustrative Architecture of an Employee Self-service Application

Company B

Kerberos based Integrated Security Domain
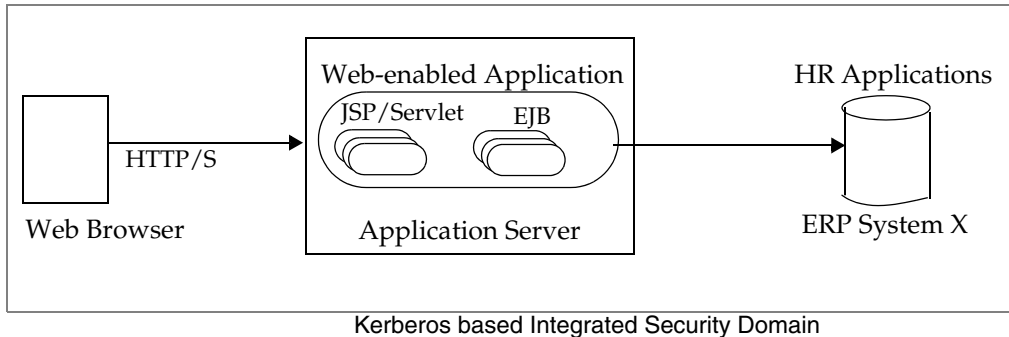
C.2.3   Deployment

During deployment of the ESS application, the deployer sets a default delegation policy of client impersonation for EIS sign-on. In this case, the application server and ERP system detect that it is the initiating principal accessing their respective services and they perform access control based on this knowledge.

FIGURE C-4 Principal Mapping

Application Security domain   **ERP System Security domain**
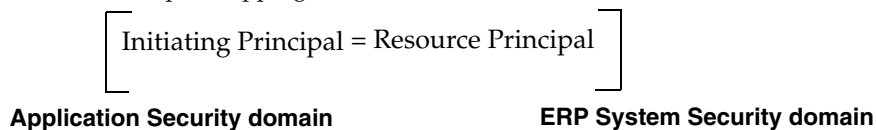
In this scenario, both the initiating principal and the resource principal refer to the same principal. This common principal is authenticated using Kerberos and its Kerberos credentials are valid in the security domains of both the application and the ERP system.

The deployer sets up access control for all authenticated employees (initiating principal) based on the configured roles—Manager, HR Manager, and Employee.

If the ERP system does not support Kerberos, then an alternate scenario is utilized. The deployer or application server administrator sets up an automatic mapping of Kerberos credentials (for the initiating principal) to valid credentials (for the same principal) in the security domain of the ERP system. Note that when the ERP system does support Kerberos, the application server performs no credentials mapping.

C.2.4   Scenario

An employee initiates an initial login to his client desktop. He enters his username and password. As part of this initial login, the employee (called initiating principal C) gets authenticated with Kerberos KDC. Refer to the details for Kerberos KDC authentication in the Kerberos v5 specification.

After a successful login, the employee starts using his desktop environment. He directs his web browser to the URL for the ESS application deployed on the application server. At this point, the initiating principal C authenticates itself to the application server and establishes a session key with the application server.

The ESS application is set up to impersonate initiating principal C when accessing the ERP system, which is running on another server. Though the application server directly connects to the ERP system, access to the ERP system is requested on behalf of the initiating principal. For this to work, principal C is required to delegate its identity and Kerberos credential to the application server and allow the application server to make requests to the ERP system on C's behalf.

C.3   Integrated Purchasing Application

Company C has an integrated purchasing application that enables an employee to use a web-based interface to perform multiple purchasing transactions. An employee can manage the entire procurement process, from creating a purchase requisition through invoice approval. The purchasing application also integrates with the enterprise's existing financial applications so that the accounting and financial aspects of the procurement business processes can be tracked.

C.3.1   Architecture

The following figure illustrates an architecture for this purchasing application. The application has been developed and deployed based on the Java EE platform and is composed of EJBs and JSPs. The EJB components provide the integration across the different applications—the logistics application from a separate vendor (this

application provides integrated purchasing and inventory management functions) and the financial accounting applications (the applications supported by the legacy system from vendor Y).

Company B is a huge decentralized enterprise; its business units and departments are geographically distributed. In this scenario, different IS departments manage ERP system X and legacy system Y. In addition, ERP system X and legacy system Y have been deployed at secured data centers in different geographic locations. Lastly, the integrated purchasing application has been deployed at a geographic location different from both ERP system X and legacy system Y.

FIGURE C-5   Illustrative Architecture of an Integrated Purchasing Application
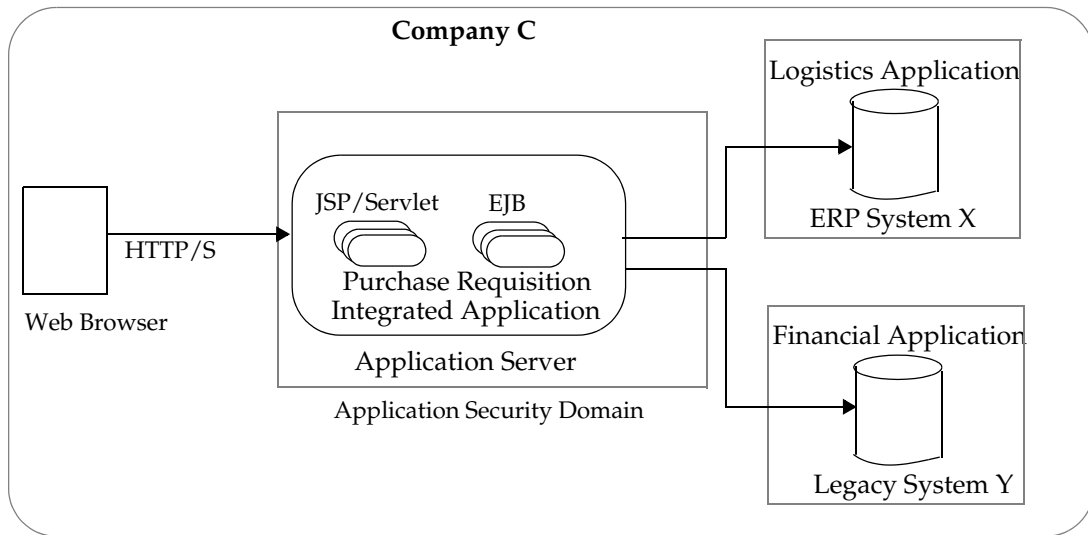
C.3.2   Security Environment

ERP system X and legacy system Y are also in different security domains; they use different security technologies and have their own specific security policies and mechanisms. The integrated purchasing application is deployed in a security domain that is different from both that of ERP system X and legacy system Y.

To support the various interaction scenarios for this integrated purchasing application, the ERP system administrator creates a unique account `LogisticsAppUser` in the ERP system. He sets up the password and specific access rights for this account. This user account is allowed access only to the logistics business processes that are used by the integrated purchasing application.

Likewise, the system administrator for the legacy system creates a unique account `FinancialAppUser`. He also sets up the password and specific access rights for this account.

The application server administrator, as part of the operational environment of the application server, configures the access to an organization-wide directory. This directory contains security information (name, password, role, and access rights) for all the employees in the organization. It is used for authentication and authorization of employees accessing the purchasing application.

Due to their physical separation in this scenario, EISs X and Y are accessed over either a secure private network or over the Internet. This requires that a secure association be established between the application server and the EISs. A secure association allows a component on the application server to communicate securely with an EIS.

C.3.3   Deployment

During the deployment of this application, the deployer configures the security information (that is, the user account `LogisticsAppUser` and its password) required to create connections to the ERP system. This configuration is done using the resource adapter for ERP system X. The deployer also configures the security information (that is, user account `FinancialAppUser` and its password) required to create connections to the legacy system Y.

The deployer configures security information in the application server to achieve the principal mapping shown in the following figure.

FIGURE C-6   Principal Mapping
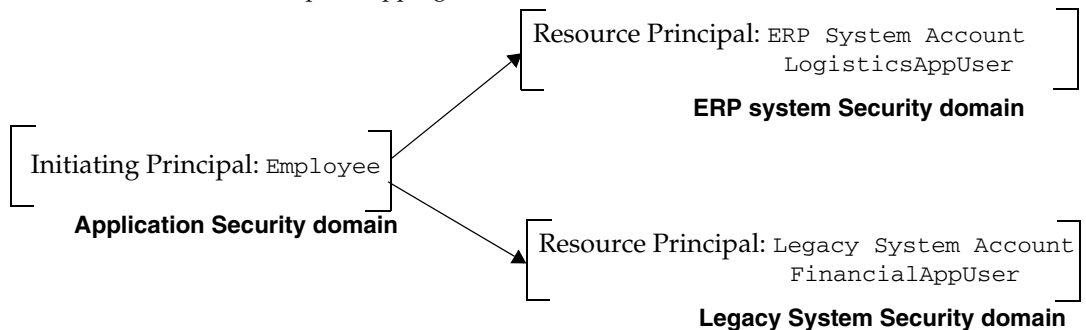
This principal mapping ensures that all connections to the ERP system are established under the security context of `LogisticsAppUser`, the resource principal for the ERP system security domain. Similarly, all connections to legacy system Y are established under the security context of the `FinancialAppUser`.

The application server does this principal mapping for all authenticated initiating principals (that is, employees accessing the integrated purchasing application) when the application connects to either the ERP system or the legacy system.

APPENDIX **D**

JAAS Based Security Architecture

This chapter extends the security architecture specified in Chapter 8, "Security Architecture" and Chapter 9, "Security Contract" to include support for JAAS-based pluggable authentication. The chapter refers to the following documents:

- White Paper on User Authentication and Authorization in Java platform: `http://java.sun.com/security/jaas/doc/jaas.html`
- JAAS 1.0 documentation

D.1 Java Authentication and Authorization Service (JAAS)

JAAS provides a standard Java framework and programming interface that enables applications to authenticate and enforce access controls upon users. JAAS is divided into two parts based on the security services that it provides:

- **Pluggable Authentication**. This part of the JAAS framework allows a system administrator to plug in the appropriate authentication services to meet the security requirements of an application environment. There is no need to modify or recompile an existing application to support new or different authentication services.
- **Authorization**. Once authentication has successfully completed, JAAS provides the ability to enforce access controls based upon the principals associated with an authenticated subject. The JAAS principal-based access controls (access controls based on who runs code) supplement the existing Java 2 code source-based access controls (access controls based on where code came from and who signed it).

D.2 Requirements

The connector security architecture uses JAAS in two ways:

- **Security Contract**. The connector security architecture uses the JAAS `Subject` class as part of the security contract between an application server and a resource adapter. Use of JAAS interfaces enables the security contract to remain independent of specific security technologies or mechanisms. The security contract has been specified in Section 9.2, "Requirements" on page 9-15.
- **JAAS Pluggable Authentication framework**. This framework lets an application server and its underlying authentication services remain independent from each other. When additional EISs and new authentication services are required (or are upgraded), they can be plugged in an application server without requiring modifications to the application server.

The connector architecture requires that the application server and the resource adapter must support the JAAS `Subject` class as part of the security contract. However, it recommends (but does not mandate) that an application server use the JAAS pluggable authentication framework.

The connector architecture does not require support for the authorization portion of the JAAS framework.
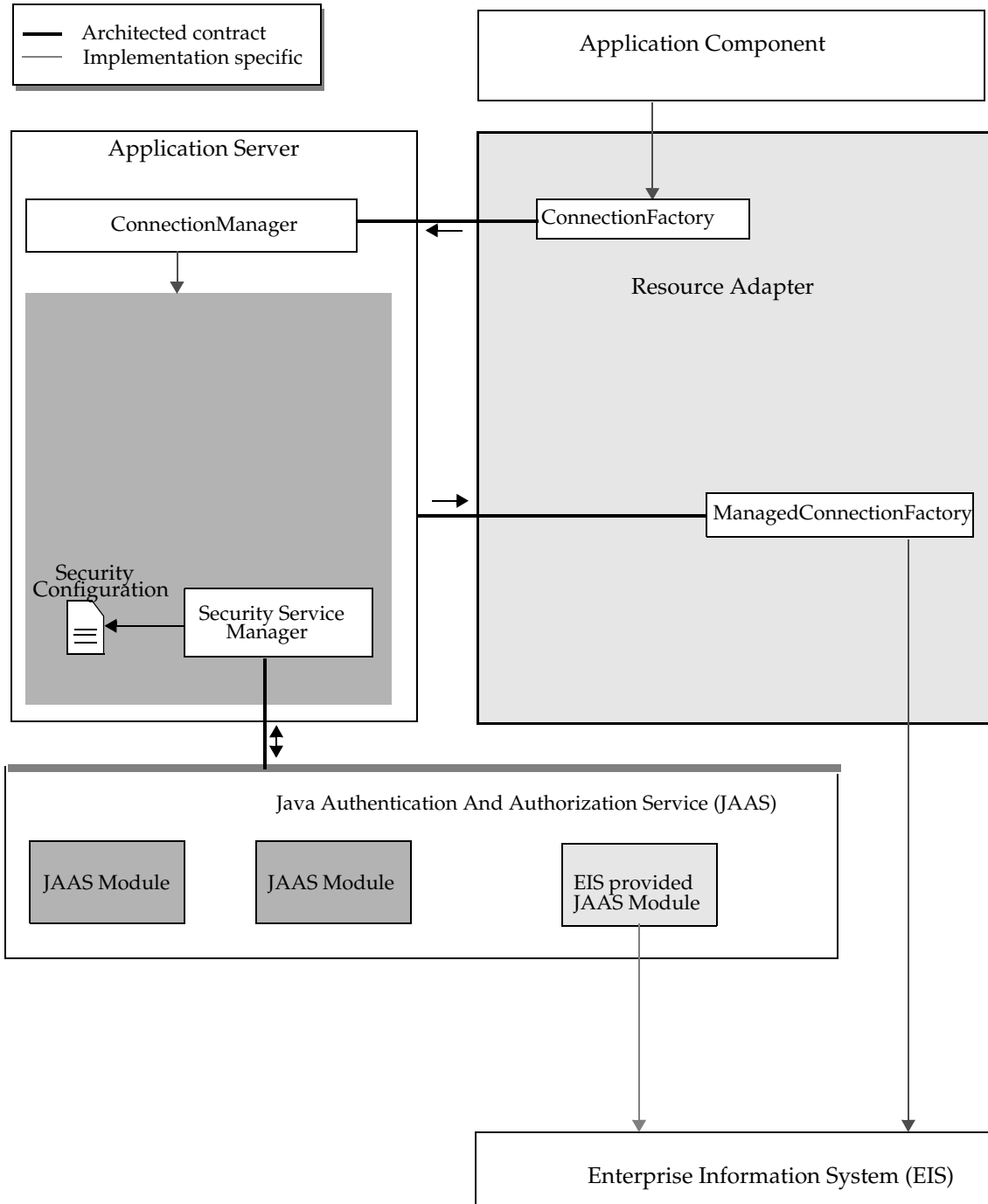
D.3 Security Architecture

The following section specifies the JAAS based security architecture. The security architecture addresses how JAAS may be used by an application server to support authentication requirements of heterogeneous EISs.

FIGURE D-1 Security Architecture.

D.3.1 JAAS Modules

The connector architecture recommends (but does not mandate) that an application server support *platform-wide* JAAS modules (also called authentication modules) for authentication mechanisms that are common across multiple EISs. The implementation of these JAAS modules is typically specific to an application server. However, these modules may be developed to be reusable across application servers.

A resource adapter provider can provide a resource adapter-specific *custom* implementation of a JAAS module. The connector architecture recommends that a resource adapter provider provide a *custom* JAAS module when the underlying EIS supports an authentication mechanism that is EIS specific and is not supported by an application server.

A *custom* `JAAS` module can be packaged together with a resource adapter and can be pluggable into an application server using the JAAS architecture.

The JAAS specification (Section 7., "Java Authentication and Authorization Service Specification, version 1.0" on page E-1) specifies requirements for developing and configuring JAAS modules.

D.3.2 Illustrative Examples: JAAS Module

The connector architecture is not intended to specify a standard architecture for JAAS modules. The following are illustrative examples of JAAS modules used typically in the JAAS-based security architecture:

D.3.2.1 Principal Mapping Module

The application server invokes the principal mapping module passing in the `Subject` instance corresponding to the caller/initiating principal. The JAAS specification specifies the interfaces/classes and mechanisms involved in the invocation of a JAAS module.

The principal mapping module maps a caller/initiating principal to a valid resource principal and returns the mapped resource principal as part of a `Subject` instance. The authentication data (example, password) for the mapped resource principal is added to the `Subject`'s credentials. The authentication data is used later to authenticate the resource principal to the underlying EIS.

A special case of the principal mapping module takes a null `Subject` as an input parameter and forms a `Subject` instance with a valid resource principal and authentication data. This is the case of default principal mapping.

The principal mapping module achieves its mapping functionality by using security information configured in the application server or an enterprise directory.

The principal mapping module does not authenticate a resource principal and is configured to perform only principal mapping. The authentication of a mapped resource principal is performed separately by an authentication mechanism-specific JAAS module.

D.3.2.2 Credential Mapping Module

The credential mapping module automatically maps credentials from one authentication domain to those in a different target authentication domain. For example, an application server can provide a module that maps the public key certificate-based credential associated with a principal to a Kerberos credential.

The credentials mapping module can use the JAAS callback mechanism to get authentication data from the application server. Note that this operation involves no user-inteface based interaction. The authentication data is used to authenticate the principal to the target authentication domain during the credentials mapping. This module can also use an enterprise directory to get security information or pre-configured mapped credentials.

D.3.2.3 Kerberos Module

This type of JAAS module supports Kerberos-based authentication for a principal. A sample Kerberos module supports:

- Getting a TGT (ticket granting ticket) to the Kerberos server in the local domain. The TGT is created by the KDC. The TGT is placed on the credentials structure for a principal.
- Delegation of authentication based on either a forwardable or proxy mechanism as specified in the Kerberos specification.

Generic Security Service API: GSS-API

The GSS-API is a standard API that provides security services to caller applications in a generic fashion. These security services include authentication, authorization, principal delegation, secure association establishment, per-message confidentiality, and integrity. These services can be supported by a wide range of security mechanisms and technologies. However, an application using GSS-API accesses these services in a generic mechanism-independent fashion and achieves source-level portability.

In the context of the connector architecture, a resource adapter uses GSS-API to establish a secure association with the underlying EIS. The use of the GSS mechanism by a resource adapter is typical in the following scenarios:

- The EIS supports Kerberos as a third-party authentication service and uses GSS-API as a generic API for accessing security services.
- The resource adapter and EIS need data integrity and confidentiality services during their communication over insecure links.

The GSS-API has been implemented over a range of security mechanisms, including Kerberos V5. See Section 6., "Java Specification Request: Generic Security Service API (GSS-API), Java bindings" on page E-1 for a Java binding of GSS-API.

Note – The connector architecture does not require a resource adapter to use GSS-API.

D.4 Security Configuration

During deployment of a resource adapter, the deployer is responsible for configuring JAAS modules in the operational environment. The configuration of JAAS modules is based on the security requirements specified by a resource adapter in its deployment descriptor or through metadata annotations discussed in Chapter 18, "Metadata Annotations". Refer to Section 20.6.4, "Requirements" on page 20-24.

The element `authentication-mechanism` in the deployment descriptor specifies an authentication mechanism supported by a resource adapter. The standard types of authentication mechanisms are: `BasicPassword` and `Kerbv5`. For example, if a resource adapter specifies support for `Kerbv5` authentication mechanism, the deployer configures a Kerberos JAAS module in the operational environment.

D.4.1 JAAS Configuration

The deployer sets up the configuration of JAAS modules based on the JAAS-specified mechanism. Refer to `javax.security.auth.login.Configuration` specification for more details. The JAAS configuration includes the following information on a per resource adapter basis:

- One or more authentication modules used to authenticate a resource principal.
- The order in which authentication modules need to invoked during a stacked authentication.
- The flag value controlling authentication semantics if stacked modules are invoked.

The format for the above configuration is specific to an application server implementation.

D.5 Scenarios

The following section illustrates security scenarios for JAAS based security architecture.

D.5.1 Scenario: Resource Adapter Managed Authentication

This scenario enables the connector architecture to support EIS specific username and pasword-based authentication. It involves the following steps:

1. The application component invokes connection request method on the resource adapter without passing in any security arguments. The resource adapter passes the connection request to the application server.

2. During the deployment of the resource adapter, the application server is configured to use a principal mapping module. This principal mapping module takes a `Subject` instance with the caller principal and returns a `Subject` instance with a valid resource principal and `PasswordCredential` instance. The `PasswordCredential` has the password for authentication of the resource principal.

3. The application server calls `LoginContext.login` method. On a successful return from the principal mapping module, the application server gets a `Subject` instance that has the mapped resource principal with a valid `PasswordCredential`.

FIGURE D-2   Resource Adapter-Managed Authentication
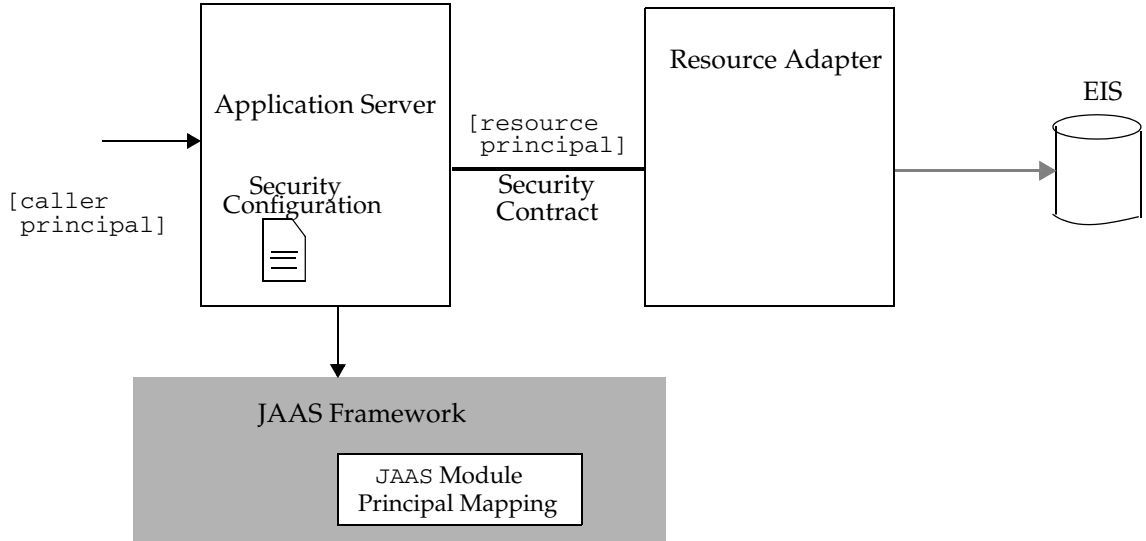
4. The application server invokes the method `ManagedConnectionFactory.create-ManagedConnection` passing in a non-null `Subject` instance. The `Subject` instance carries the resource principal and its corresponding `PasswordCredential`, which holds the user name and password.

5. The resource adapter extracts the user name and password from the `Password-Credential` instance. The resource adapter uses the getter methods (`getPrivateCredentials` method) defined on the `Subject` interface to extract the `PasswordCredential` instance.

6. The resource adapter uses username and password information (extracted from the `PasswordCredential` instance) to authenticate the resource principal to the EIS. The authentication happens during the creation of the connection through an authentication mechanism specific to the underlying EIS.

D.5.2　Scenario: Kerberos and Principal Delegation

The scenario in the following figure involves the following steps:

FIGURE D-3　Kerberos Authentication with Principal Delegation
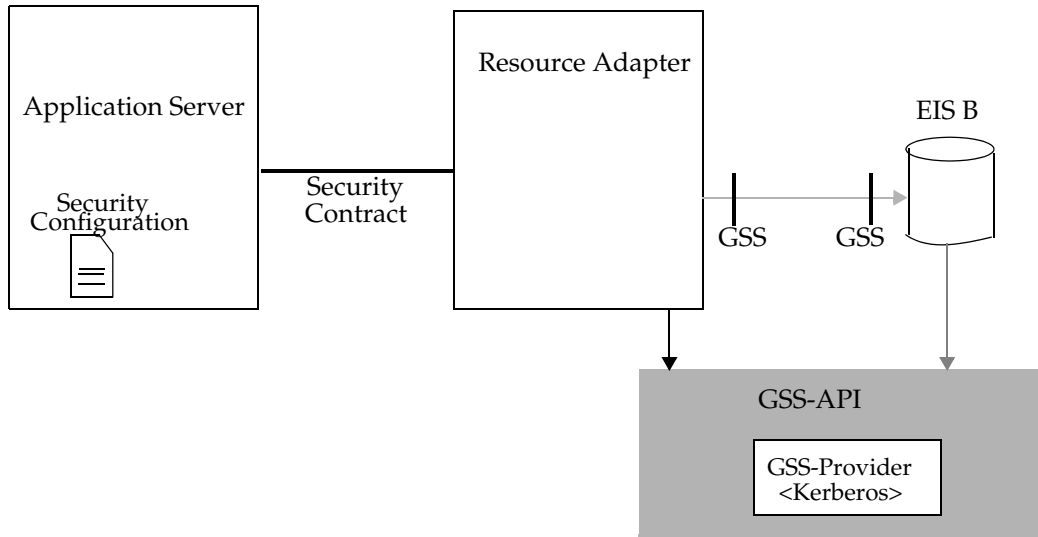
1. The initiating principal has already authenticated itself to the application server using Kerberos. The initiating principal has a service ticket for the application server and a TGT (ticket granting ticket issued by the KDC) as part of its Kerberos based credentials.

2. In this scenario, the application server is configured to impersonate the initiating principal when connecting to the EIS instance. So even though application server is directly connecting to the EIS, access to the EIS is being requested on behalf of the initiating principal. The initiating principal is required to pass its identity to the application server and allow the application server to make requests to the EIS on behalf of the initiating principal. The above is achieved through delegation of authentication.

3. The application server calls the method `ManagedConnectionFactory.createManagedConnection` by passing in a `Subject` instance with the initiating principal and its Kerberos credentials. The credentials contain a Kerberos TGT and are represented through the `GSSCredential` interface.

4. The resource adapter extracts the resource principal and its Kerberos credentials from the `Subject` instance.

5. The resource adapter creates a new physical connection to the EIS.

6. If the resource adapter and EIS support GSS-API for establishing a secure association, the resource adapter uses the Kerberos credentials based on the GSS mechanism as follows. For details, see GSS-API specification:

   - The resource adapter calls `GSS_Acquire_cred` method to acquire `cred_handle` in order to reference the credentials for establishing the shared security context.
   - The resource adapter calls the `GSS_Init_sec_context` method. The method `GSS_Init_sec_context` yields a service ticket to the requested EIS service with the corresponding session key.

Note – The mechanism and representation through which Kerberos credentials are shared across the underlying JAAS module and GSS provider is beyond the scope of the connector architecture.


   - After success, `GSS_Init_sec_context` builds a specific Kerberos-formatted message and returns it as an output token. The resource adapter sends the output token to the EIS instance.
   - The EIS service passes the received token to the `GSS_Accept_sec_context` method.
   - The resource adapter and EIS now hold the shared security context (so have established a secure association) in the form of a session key associated with the service ticket. They can now use the session key in the subsequent per-message methods: `GSS-GetMIC`, `GSS_VerifyMIC`, `GSS_Wrap`, `GSS_Unwrap`.

7. If the resource adapter and EIS fail to establish a secure association, the resource adapter cannot use the physical connection as a valid connection to the EIS instance. The resource adapter returns a security exception on the `createManagedConnection` method.

D.5.3   Scenario: GSS-API

If an EIS supports the GSS mechanism, a resource adapter may (but is not required to) use GSS-API to set up a secure association with the EIS instance. The section "Generic Security Service API: GSS-API" on page D-6 gives a brief overview of GSS-API.

**FIGURE D-4**   GSS-API use by Resource Adapter
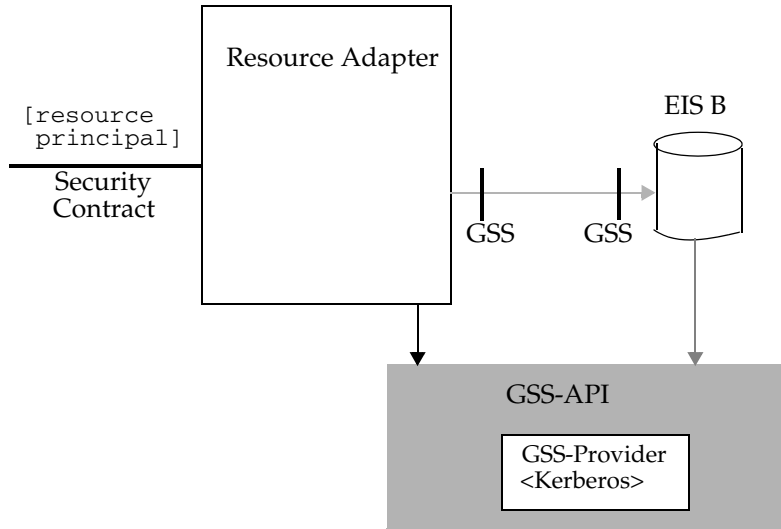
A formal specification of the use of GSS-API by a resource adapter is beyond the scope of the connector architecture. However, GSS-API has been mentioned as a possible implementation option for a resource adapter that has the GSS mechanism supported by its underlying EIS.

D.5.4 Scenario: Kerberos Authentication After Principal Mapping

The scenario depicted in the following figure involves the following steps:

**FIGURE D-5** Kerberos Authentication After Principal Mapping
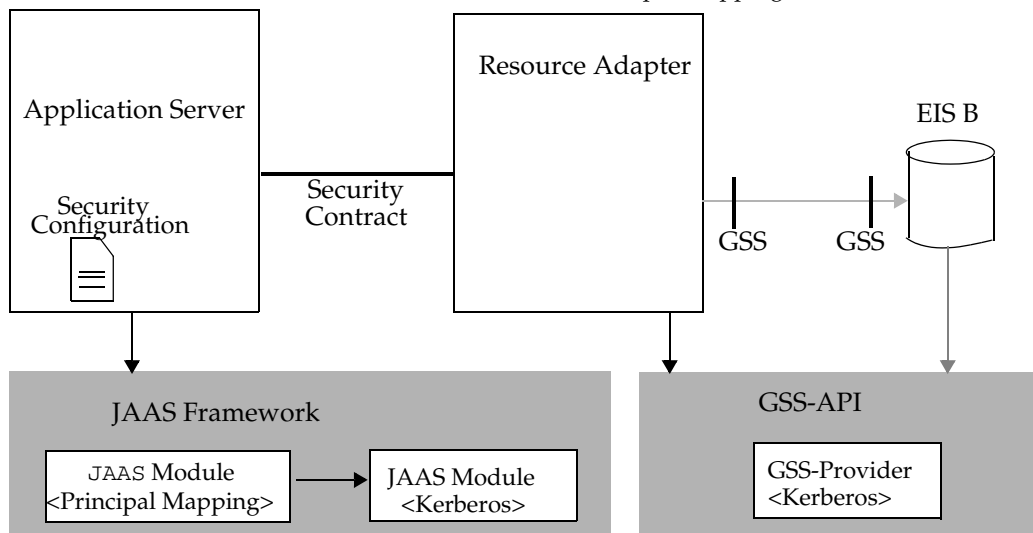
1. The application server is configured to use the principal mapping module and Kerberos module. The two authentication modules are stacked together with the principal mapping module first.

2. The application server creates a `LoginContext` instance by passing in the `Subject` instance for the caller principal and a `CallbackHandler` instance. Next, the application server calls the `login` method on the `LoginContext` instance.

3. The principal mapping module takes a `Subject` instance with caller principal and returns a `Subject` instance with a valid resource principal and Kerberos-based authentication data. The principal mapping module does not authenticate the resource principal; it does only principal mapping to find the mapped resource principal and its authentication data.

4. The Kerberos module (called after the principal mapping module) uses the resource principal and its authentication data to authenticate the resource principal. The Kerberos module leads to a valid TGT for the Kerberos domain supported by the EIS. The TGT is contained in the Kerberos credentials represented through the `GSSCredential` interface.

5. The application server calls the method `ManagedConnectionFactory.create-ManagedConnection` passing in a `Subject` instance with the resource principal and its Kerberos credentials.

6. The remaining steps are the same as in the previous scenario, Section D.5.2, "Scenario: Kerberos and Principal Delegation" on page D-10

D.5.5   Scenario: EIS-Specific Authentication

FIGURE D-6   Authentication Through EIS-Specific JAAS Module
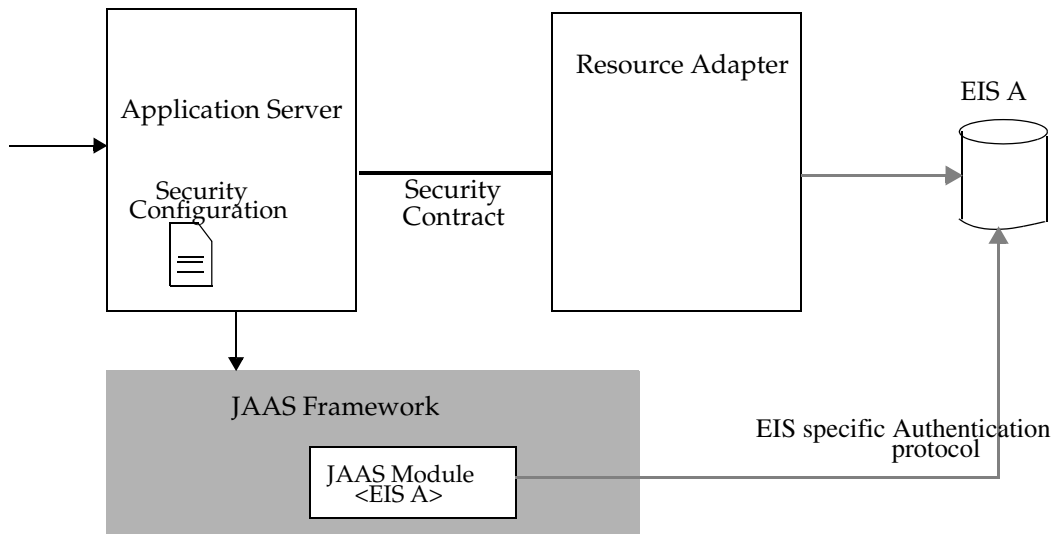
The scenario in the preceding figure involves the following steps:

1. During the configuration of a resource adapter, the application server is configured to use an EIS-specific JAAS module for authentication to the underlying EIS.

2. The configured JAAS module supports an authentication mechanism specific to the EIS. The application server is responsibility for managing the authentication data and JAAS configuration.

3. The application server gets a request from the application component to create a new physical connection to the EIS. Creating a new physical connection requires the resource principal to authenticate itself to the underlying EIS instance.

4. The application server initiates the authentication of the resource principal. It creates a `LoginContext` instance by passing in the `Subject` instance and a `CallbackHandler` instance. Next, the application server calls the `login` method on the `LoginContext` instance.

5. The JAAS module authenticates the resource principal to the underlying EIS. It uses the callback handler provided by the application server to get the authentication data.

6. The application server invokes the method `ManagedConnectionFactory`. `createManagedConnection` passing in the `Subject` instance with the authenticated resource principal and its credential.

7. The resource adapter extracts the credential (associated with the `Subject` instance) for the resource principal using the getter methods defined on the `Subject` interface. The resource adapter uses this credential to create a connection to the underlying EIS.

In this scenario, authenticating a resource principal (initiated by the application server and performed by the JAAS module) is separate from creating a connection to the EIS. The resource adapter uses the credential of the resource principal to create a connection to the EIS. This connection creation can involve further authentication.

After successfully creating a connection to the EIS, the resource adapter returns the newly created connection from the method `ManagedConnectionFactory.createManagedConnection`.

Related Documents

1. Enterprise JavaBeans (EJB) Specification, version 3.2

   `http://jcp.org/en/jsr/detail?id=345`

2. Java Transaction API (JTA) Specification, version 1.2

   `http://jcp.org/en/jsr/detail?id=907`

3. JDBC API Specification, version 4.1

   `http://jcp.org/en/jsr/detail?id=221`

4. X/Open CAE Specification -- Distributed Transaction Processing: the XA
   Specification, X/Open document

5. RFC: Generic Security Service API (GSS-API) Specification, version 2

   `http://www.ietf.org/rfc/rfc2078.txt`

6. Java Specification Request: Generic Security Service API (GSS-API), Java bindings

   `http://jcp.org/en/jsr/detail?id=72`

7. Java Authentication and Authorization Service Specification, version 1.0

   `http://www.oracle.com/technetwork/java/javase/jaas/`

8. Java Platform, Enterprise Edition (Java EE) Specification, version 7

   `http://jcp.org/en/jsr/detail?id=342`

9. JavaServer Pages (JSP) Specification, version 2.3

   `http://jcp.org/en/jsr/detail?id=245`

10. Java Servlet Specification, version 3.1

    `http://jcp.org/en/jsr/detail?id=340`

11. Java Message Service Specification, version 2.0

    `http://jcp.org/en/jsr/detail?id=343`

12. Java Platform, Standard Edition 7API Specification

```
http://docs.oracle.com/javase/7/docs/api/
```

13. J2EE Deployment API Specification

```
http://jcp.org/en/jsr/detail?id=088
```

14. JavaBeans Specification 1.01 Final Release

```
http://www.oracle.com/technetwork/java/javase/documentation/spec-
136004.html
```

15. Java Authentication Service Provider Interface for Containers Specification, version 1.4

```
http://jcp.org/en/jsr/detail?id=196
```

16. Bean Validation Specification, version 1.1

```
http://jcp.org/en/jsr/detail?id=303
```

17. A Metadata Facility for the Java Programming Language

```
http://jcp.org/en/jsr/detail?id=175
```

18. Common annotations for the Java Platform Specification, version 1.2

```
http://jcp.org/en/jsr/detail?id=250
```

19. Contexts and Dependency Injection for the Java EE Platform 1.1

```
http://jcp.org/en/jsr/detail?id=346
```

Change History for Version 1.0

F.1 Version 0.9

- Editorial run through the document
- Added section 1.4 on relationship between JDBC and Connector architecture
- Added scenario on B2B in the chapter 4
- Added java.io.Serializable to the code specification of interfaces that are required to support Serializable interface
- Added clarifications in the chapter 5 based on the expert comments. The changes are marked by change bars
- Added equals and hashCode methods to interface ConnectionRequestInfo
- Added section 6.8 on Connection Association
- Added clarifications to the chapter 7. Did minor restructuring of the chapter based on review comments. The changes are marked by change bars
- Added clarifications to the chapter 8 based on expert comments
- Changed few details and added clarifications in the chapter 9 based on the review comments. The changes are marked by change bars
- Added more description for packaging and deployment in the chapter 10
- Clarified version dependencies in the chapter 11
- Introduced interface javax.resource.Referenceable for the standard setReference method
- Removed scenarios on Credentials Mapping and Single sign-on from Appendix C. Updated scenario C.6.2 to refer GSS-API.

F.2 Version 1.0 - Public Draft 1

- Removed definition of "Connector" from 2.1. The term Connector is now used broadly refer to the Connector architecture, while resource adapter refers to the system library.
- Added requirement for ConnectionEventListener to 6.9.2: Application Server
- Added connection handle property to the ConnectionEvent, section 5.5.7
- Introduced getResultSetInfo method in the Connection interface
- Added "Administered Object" in the section 9.6.2
- Added more details to section "Auto Commit" in 9.5.2
- Introduced separate interface for ResultSetInfo in the section 9.10.3
- Changed specification of element config-property-type in section 10.6
- Added an example to illustrate security permission specification in the section 11.3
- Added CCI related information to Projected Items, chapter 12

F.3 Version 1.0 - Public Draft 2

- Section 5.5.1: Change based on introduction of ConnectionSpec interface
- Section 5.5.1: Added clarification to ConnectionRequestInfo section
- Section 5.5.4: Added clarification to section on "Cleanup of ManagedConnection"
- Section 5.5.6: Added clarification to paragraph after the interface for ConnectionEventListener
- Section 5.9.1: Added clarification to description of the scenario
- Section 6.8: Moved earlier section "Details on Local transaction" ahead of connection sharing section and renamed it "Scenarios: Local Transaction Management". No change in any content.
- Section 6.9: Added more details on connection sharing based on the changes in EJB 2.0 and J2EE 1.3 platform specification
- Section 6.10: Added this section to clarify local transaction optimization. This is based on changes in EJB 2.0 and J2EE 1.3 platform specification
- Section 6.11: Made a new section on "Scenarios: Connection sharing". No change in content
- Section 6.12: Added clarifications and requirements in the section on "Connection Association"
- Section 6.13.2: Moved requirements on connection sharing to section 6.9
- Section 7.4.2: Code sample changed to reflect ConnectionSpec usage
- Section 9.5.1: Changed getConnection(Map) to getConnection(ConnectionSpec) and added clarifications
- Section 9.5.2: Introduced a section on ConnectionSpec

- Section 9.7.2: Added methods to ResourceAdapterMetaData interface. Added description of these methods
- Section 9.9.1: Record, MappedRecord and IndexedRecord now extend Serializable interface
- Section 9.10: Added note on JDBC semantics in relation to CCI ResultSet
- Section 9.10.3: Added note on ResultSetInfo implementation requirements
- Section 10.6: Change to auth-mechanism specification in DTD. Removed + from credential-interface
- Figure 29: Added clarifications for the diagram
- Section 8.3: Clarified security contract requirements for the application server
- Section 9.5.1: Moved method getRecordFactory from Interaction to ConnectionFactory. Note that it is not necessary to have an active connection to create generic record instances

F.4   Version 1.0 - Proposed Final Draft 2

- Reviewed requirements in terms of compliance testing. marked with change bars in the document
- Fixed documentation errors
- System Contracts:
  - Section 5.5.4: Clarified requirements for the method `matchManagedConnections` on `ManagedConnectionFactory` interface
  - Section 6.9: Made requirements for connection sharing consistent with J2EE 1.3 platform specification
  - Section 6.10: Added specification of requirements for different transaction scenarios. Added illustrative scenarios
  - Section 6.11: Removed a transaction scenario that illustrated connection sharing
  - Section 6.11: Clarified requirements for connection association
- Common Client Interface:
  - Removed `setLogWriter`, `getLogWriter`, `setTimeout`, `getTimeout` methods from `ConnectionFactory` interface
  - Added description for exceptions in the API documentation for the CCI interfaces. Note that no new exception has been introduced
  - `ConnectionFactory` implementation class required to provide a default constructor
  - Added clarifications; marked by change bars
- Deployment and Packaging:
  - Section 10.2: Clarified requirements for packaging and deployment of a resource adapter
- DTD changes based on a review of DTDs for various J2EE specifications:
  - Ordered elements alphabetically except the root element
  - `<display-name>` changed to optional in `<connector>` element

- Used common elements from other DTDs: `<description>`, `<small-icon>`, `<large-icon>`
- Used common header comments across all J2EE DTDs
- `<auth-mechanism>` changed to `<authentication-mechanism>`
- `<auth-mech-type>` changed to `<authentication-mechanism-type>`
- Added `java.lang.Character` to `<config-property-type>`:
- Changed defined values in `<authentication-mechanism-type>`: `basic-password` to `BasicPassword`, `kerbv5` to `Kerbv5`
- Changed defined values in `<transaction-support>` element: `no_transaction` to `NoTransaction`, `local_transaction` to `LocalTransaction`, `xa_transaction` to `XATransaction`

F.5   Version 1.0 - Final Release

- Clarification on reauthentication in the section 8.2.7
- Change in auto-commit in section 9.5.3. Removed set/getAutoCommit methods from the Connection interface

APPENDIX **G**

Change History for Version 1.5

G.1 Public Review Draft

- Some minor clarifications to sections 5.3.1, 5.3.4, 5.3.5
- Some minor clarifications to section 10.3
- Added chapter 11: Inbound communication
- Added chapter 13: EJB invocation
- Section 12.3
- Added method release on Endpoint interface
- Added method getXAResources on ResourceAdapter interface
- Updated section 12.5.1 to illustrate XAResource implementation
- Added section 12.5.2: Message redelivery upon crash recovery
- Section 12.5.6: Added an application server requirement to set thread context class loader
- Section 12.7: Updated the JMS use case to comply with the Connector 1.5 XSD
- Updated Section 16.2: Application server requirements
- Some minor clarifications to sections 17.4.1, 17.4.2
- Section 17.6: Converted Connector 1.5 DTD to Connector 1.5 XSD
- Added Appendix B: Activation Configuration for Message Inflow to JMS endpoints

G.2 Proposed Final Draft

- Section 7.9: Connection sharing requirements have been updated
- Section 9.2.3: `GenericCredential` interface has been deprecated
- Section 9.3: Added a resource adapter requirement to throw a `SecurityException` if the authentication information provided is erroneous

- Section 10.3.1: Clarified use of synchronized blocks within a `Work` implementation
- Section 12.3: Anew method parameter `java.lang.reflect.Method` has been added to `beforeDelivery` and `isDeliveryTransacted` methods
- Section 12.4.2: Added separate sections to describe `ActivationSpec` JavaBean and Administered objects
- Section 12.4.9: Clarified multiple endpoint activations
- Section 12.4.7: This new section describes the structure of a message listener interface
- Section 12.5.2: Clarified crash recovery processing
- Section 12.7.2: Updated the sample code to use JMS 1.1 style generic APIs
- Section 16.2: Removed application client container requirements for supporting connector contracts
- Section 17.2: Added description of RAR file contents
- Section 17.6: Updated Connector 1.5 XSD - `resourceadapter-class` element has been made optional, and `config-property` element has been added to adminobject element
- Section 18.4: Added an example to illustrate usage of `doPrivileged` blocks in resource adapter code
- Section 19.2: Added `InvalidPropertyException` and `UnavailableException` to the exception list
- Appendix B: Added descriptions to the various `ActivationSpec` JavaBean properties
- Appendix F: Updated links to related documents

G.3   Proposed Final Draft 2

- Section 6.5.2: Clarifications on `ConnectionManager` requirements.
- Section 7.9.1: This new section describes how a resource adapter may detect and report sharing violations.
- Section 7.13.1: Added description to clarify the auto-commit behavior of transactional resource adapters.
- Section 7.14: This new section describes two optional connection optimizations.
- Section 12.5.6: Clarified that the message delivery must not rely on the class loader that is associated with the `java.lang.reflect.Method` object parameter.
- Section 12.5.8: Clarified the usage of source managed transactions.
- Section 14.4.4: Clarified the requirements on transaction inflow contracts.
- Section 17.6: XSD modification: The contents of the `connectionfactory-interface` element must be unique in the `outbound-resourceadapter`.

G.4 Final Release

- Section 5.3.1: Clarification on the behavior of `start` method call
- Section 5.3.2: `ResourceAdapterAssociation` interface
- Section 5.3.4.1, 5.3.4.2: Clarifications on resource adapter shutdown
- Section 6.4.1: Fifth bullet item, clarification on access to application JNDI context
- Section 6.5.3: `ValidatingManagedConnectionFactory` interface
- Section 6.5.4: Clarifications on connection sharing and multiple connection handles
- Section 6.5.6: Clarifications on event listener mechanism
- Section 7.7.2: Clarifications on local transaction events and implicit transaction begin
- Section 7.9: `SharingViolationException`, and clarification on non-transactional resource adapters
- Section 7.9.1: Clarifications on sharing violation detection
- Section 9.1.8.2: Clarifications on security options A, B and C
- Section 10.3: Numeric string values for `WorkException` error code values
- Section 10.3.1: Clarification on access to JNDI context from `Work` objects
- Section 10.3.7: `CreateTimer` method throws `UnavailableException`
- Section 12.4.2.2: Clarifications on required configuration properties on `ActivationSpec` JavaBean, and JMS `ActivationSpec` JavaBean
- Section 12.4.2.4: Clarifications on configuring administered objects
- Section 12.3: Changes to `endpointActivation`, `beforeDelivery`, `afterDelivery` and `isDeliveryTransacted` method signatures
- Section 12.4.4: Clarifications on message delivery during endpoint activation
- Section 12.4.4: Clarifications on `endpointDeactivation` method
- Section 12.4.6: Clarifications on contract between deployment tool and application server
- Section 12.5.6: Description of `IllegalStateException` thrown by `beforeDelivery` and `afterDelivery` method calls
- Section 12.5.8: Clarifications on use of transaction inflow
- Section 15.8: Service endpoint `javax.resource.cci.MessageListener` interface
- Section 16.2: Table 16-1, updates to column corresponding to lifecycle management
- Section 16.3: JavaBean requirements
- Section 16.4: Equality constrains
- Section 17.5.3: Clarifications on `getReference` method
- All exception classes have been provided constructors that have a cause param
- Section 17.6: XSD modifications: `vendor-name` has been assigned a string type, `security-permission` has been moved one level up such that the permissions apply to the whole resource adapter code, deployment-extension element has been removed, indentations have been reformatted to follow standard XML conventions

Change History for Version 1.6

H.1   Final Draft

- Various sections (no semantic changes) to address editorial feedback

- Corrected the phrase non-"Classic Profile" to non-"Full Profile" in Section 3.5 to reflect the nomenclature followed in the platform spec.

- Removed JTA as an optional requirement in Section 3.5 as the Platform specification (v6 PFD, Section EE.9.5 -- Requirements for All Java EE Profiles) requires JTA support in all Java EE Profiles.

- Added a requirement in Section 3.5 regarding the deployment of bi-directional resource adapters in standalone connector containers that do not support Message Inflow.

- Added BeanValidation as an optional requirement for standalone connector containers in Section 3.5

- Added JASPIC baseline compatibility and Security Inflow support requirements in Section 3.5

- Clarified in Section 3.5 that the specification does not define new or mandate support for existing application components in the standalone connector environment.

- Removed the note to reviewer and removed `InteractionSpec` from list of JavaBeans that must be validated in Section 5.3.7.5

- Added a requirement for mandatory support of BeanValidation

- Added a requirement in Section 5.3.7.5 for the connector container to perform validation on the `javax.validation.groups.Default` in the JavaBean.

- Added validation failure related requirement in Section 5.3.7.5

- Added a requirement in Section 5.3.7.5 for the application server to treat all JavaBean properties as reachable and cascadable

- Clarified in Section 11.4 that if `WorkContextProvider.getWorkContexts` returns null/empty `List` the `WorkManager` must treat it as if no additional execution context information needs to be established for that `Work` instance and continue `Work` processing.

- Clarified in Section 11.4 that the resource adapter is not allowed to modify the state of a `WorkContext` after the `Work` instance that is associated with it has been submitted to the `WorkManager`

- Clarified that the resource adapter must be configured to have the necessary security permissions to add a Principal in Section 16.4.5.1

- Changed the definition of `AuthenticationMechanism` and `SecurityPermission`'s `Target` as (`@Target({})`) in Section 18.4.3 and 18.4.4

- Removed the "default {}" clause for the `messageListeners` annotation element in Section 18.7

- Updated example in Section 18.7 to add a `messageListeners` annotation element to the `Activation` annotation

- Clarified in Section 18.5 that the application server must process `ConfigProperty` annotations on JavaBeans irrespective of whether they are specified by way of annotations or DD.

- Added a Section 18.3.3 "Annotation Processing Requirements of Superclasses" to discuss how annotations on super-classes must be processed by the application server

- Added Connectors 1.5 XSD to Section A.2 and clarified that application servers must support RARs with Connectors 1.5 DD.

- Clarified in 20.2.0.1 that Java EE Platform's library support can be used by resource adapters to express library dependencies.

- Clarified in Section 20.4.1 that the `resourceadapter-class` needs to be provided only if available

---

H.2   Proposed Final Draft

- Renamed `InflowContext` to `WorkContext` in multiple sections, as the contextual information being passed for a `Work` instance is not limited to Message Inflow scenarios alone as the name originally appeared to suggest
- Added a paragraph in Section 7.1 clarifying that connection handles may not be portably be passed across application component boundaries
- Clarified Section 10.3.3 that the constant `IMMEDIATE` requires the WorkManager to timeout the action as soon as possible
- Clarified in Section 11.6 that the `WorkManager` ignore unknown hint names and added a new Section 11.6.1 for standard Hint names. Added LongRunning hint in 11.6.1.2

- Added a requirement in Section 13.5.6 on the application server to support cases where a resource adapter may not perform a message delivery between `beforeDelivery` and `afterDelivery` method pair
- Added a paragraphin Section 16.3 allowing a `WorkManager` to enable access policy enforcement prior to `Work` execution
- Corrected grammatical errors and typos in Chapter 18
- Corrected in section 18.3.2 a mis-arrangement of rules for merging DD and annotations
- Removed in section 18.4 the `specVersion` annotation element and corrected defaults for array elements
- Clarified in Section 18.5 that the values of `type` is limited to the XML schema constraints and ensured that the type of a field/method and the corresponding `ConfigProperty.type` are equal and updated Section 20.7 schema to reflect `specVersion` related changes
- Added a new Section 5.3.7.6 describing Configuration Property attributes such as dynamically reconfigurable and confidential parameters. Updated Section 18.5 and 20.7 to update annotation and deployment descriptor equivalents for these attributes
- Added a new method that accepts a timeout parameter to `MessageEndpointFactory.createMessageEndpoint` in Section 13.3
- Defined retryable equivalents of `UnavailableException` and `WorkRejectedException` and clarified their usage in Section 10.3.3 and Section 13.3
- Clarified that the `@Connector` annotated JavaBean must implement the `ResourceAdapter` interface in Section 18.4.1 and 18.4.2
- Defined the standard hint names as constants in `HintsContext` and explained their usage in Section 11.6.1
- Made the following clarifications in the Connection and Transaction Management chapters (Chapter 6 and 7) of the spec
  - Clarified, in Section 6.5.4.2 (in the section detailing the "at most one active connection handle" scenario), when an application server may modify active connection handles and clarified behaviour of this scheme in the absence of Lazy Connection Association support.
  - Clarified that connections may not be portably cached under Local Transaction Scenarios, in the absence of Lazy Connection Association optimization, detailed in Section 7.10.3
  - Clarified that the `ManagedConnection.associateConnection` method may be called to set the same `ManagedConnection` in Section 7.11.2
  - Clarified that the `DissociatableManagedConnection.dissociateConnections` may be called even when a transaction is in progress in Section 7.16.1
  - Added a new Section 6.4.3 to provide guidelines on how a connection handle should be obtained and used in managed environments
- Added a new Section 1.5 that provides guidelines on the relationship of the Connector Architecture with other integration technologies such as JBI and SCA

H.3 Public Review

- Added a requirement in Ch.20 Runtime Environment to support Java SE 6.0. This is aligned with Section EE.2.3.1 of the Java EE 6 ED
- Modified expert group member list in Section 1.7 to include Eugene and Jesper
- Modified Section 11.3 and 11.4 InflowContext and InflowContext provider to extend `Serializable`
- Added sections 10.3.10 and 10.3.11 to address distributed work processing
- Added section 5.3.7.5 capturing JSR 303 requirements
- Added JSR 303: Bean Validation to Appendix F: Related documents
- Modified Section 13.4.2.2 to clarify that JavaBean validation mechanisms are recommended to be used against the `ActivationSpec.validate` method and deprecate `required-config-property`
- Added Chapter 18: Metadata annotations
- Added JSR175 and JSR 250 references in Appendix F: Related Documents
- Added a new Section 7.14 TransactionSynchronizationRegistry to describe how a resource adapter may get access to the `TransactionSynchronizationRegistry`
- Updated 4.1.6 to indicate application assemblers may not result in a deployment descriptor when annotations are used
- Added requirement for the container to merge annotation and deployment descriptor information during configuration of JavaBeans in Sections 5.3.7.1, 5.3.7.3 and 5.3.7.4
- Updated 6.3.1, 7.13, 7.15.2, 8.5.1, 8.6.1, 8.6.2, 8.6.5, 9.1.4.2, 9.1.5.1, 9.2.1, 11.4.1, 13.4.1, 13.4.2, 13.4.2.4, 13.4.3, 13.4.6, 13.4.7, 20.6.2, 20.6.3.1 to indicate use of annotations and deployment descriptor elements for configuration.
- Clarified that there are new annotation based equivalents for various deployment-descriptor elements in older sections - Section 7.9, 13.7.2 for Resource annotation, Section 8.4 and 8.4.1 for res-auth specification, Section 13.4.1, 13.7.1.1, B.1, B.2.1.1, B.2.1.2, B.3, B.4, E.4 for endpoint annotations like MessageDriven
- Clarified about use of new metadata annotations for resource adapter entities: Section 20.5.1 - @Connector for ResourceAdapter, Section 20.5.2.1 @ConnectionDefinition for ManagedConnectionFactory, Section20.5.3 for Configuration Properties, Section 20.5.4 for Bean Validation, Section 21.3 and 21.3.1 for SecurityPermission
- Clarified that the deployment descriptor is optional. removed requirement in Section 20.2.0.1 that required the bundling of a resource adapter deployment descriptor
- Clarified in Section 20.4, 20.4.1, 20.4.2, 20.4.2.3 that the deployment information may also be specified through metadata annotations. clarified that the RA provider may provide deployment information through the deployment descriptor

H.4 Early Draft

- Added new chapters. 11. Generic Inflow Context and 16. Security Inflow
- Added the new system contracts, Generic Inflow context and Security Inflow to Section 1.2 Scope and Section 3.1 System Contracts
- Added a new Section 3.5 Standalone Connector Environment
- Clarified ValidatingManagedConnectionFactory usage in Section 6.5.3.4
- Clarified ConnectionEventListener concurrency behaviour in Section 6.5.4.1
- Clarified local transaction related requirements in section 7.10.1
- Added new section 7.13 Runtime Transaction Support Level specification
- Clarified application server requirements in light of section 7.13
- Added inactiveConnectionClosed method to LazyAssociatableConnectionManager interface in section 7.15.1
- Clarified JNDI context availability in the application server thread that invokes run and release methods on a `Work` instance in Section 10.3.2
- Clarified JMS and Generic Message Inflow contracts relationship in sections 13.4.1 and 13.7
- Added ResourceAdapterAssociation related requirements to administered objects in Section 13.4.2.3
- Corrected errors in the activation spec and resource adapter classnames in the example in Section 13.8.3.1
- Added a new Section 19.3 Class Loading Requirements
- Clarified resource adapter classloading in application server requirements in Section 19.2.0.4 in light of Section 19.3
- Clarified ManagedConnectionFactory creation/reuse in Section 19.6.3.3
- Added description of ActivationSpec's config-property element in section 13.4.2.2 and config-property deployment descriptor element description for ActivationSpec in Section 19.4.1. added a config-property element in activationSpecType in Section 19.7
- Added admin object uniqueness requirements under Administered object in Section 19.4.1 and added adminobject-type uniqueness constraint in the XML schema in Section 19.7
- Added description of required-inflow-contexts element in section 19.4.1 and addeda required-inflow-contexts element in connectorType in Section 19.7
- Corrected minor typo in LocalTransactionException section in Section 21.2.1

- Removed Security Inflow and making CCI mandatory in a future release related items in Chapter 22
- Updated J2EE Platform specification references to Java EE 6 Platform specification references in Appendix E
- Updated all appropriate J2SE references to Java SE and J2EE references to Java EE

Change History for Version 1.7

I.1 Maintenance Release 1

- General: Fixed sun.com and legacy URLs and references to point to the latest specification and technology pages, Updated Appendix E to reflect the latest technology versions in the Java EE 7 Platform

- Added application component environment namespace availability requirements during `endpointActivation` and `endpointDeactivation` in Section 13.4.4, "Application Server" on page 13-15

- Added requirements in Section 13.3, "Message Inflow Model" on page 13-4 for the `MessageEndpointFactory` to provide a name for the message endpoint deployment it represents, and to provide the message endpoint class to a resource adapter.

- Added a new section, Section 18.9, "Resource Definition Annotations" on page 18-16, to discuss requirements around connection factory and administered object resource definition annotations

- Errata: Removed the paragraph that made it optional for an @Activation annotated JavaBean to implement the `ActivationSpec` interface (See Section 18.7, "@Activation" on page 18-14)

- Removed Appendix B "Activation Configuration for Message Inflow to JMS Endpoints". Chapter 12 of the Java Message Service Specification, version 2.0 defines the standard configuration elements for JMS resource adapters

- Clarified the dependency injection behavior when JavaBeans that are managed by the application server are used as CDI Managed Beans in Section 21.5, "Dependency Injection" on page 21-8

- Updated Section 20.7, "Resource Adapter XML Schema Definition" on page 20-24 to reflect the new 1.7 resource adapter deployment descriptor and added a new Section A.3, "Java EE Connector Architecture 1.6 Resource Adapter XML XSD" on page A-37 to house the 1.6 resource adapter deployment descriptor.