Specification: JSR 208: Java Business Integration Specification (JBI) 1.0 Specification ("Specification")

Status: Public Review

Release: March 4th, 2005

Copyright 2005 Sun Microsystems, Inc.

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

All rights reserved.

NOTICE

The Specification is protected by copyright and the information described therein may be protected by one or more U.S. patents, foreign patents, or pending applications. Except as provided under the following license, no part of the Specification may be reproduced in any form by any means without the prior written authorization of Sun Microsystems, Inc. ("Sun") and its licensors, if any. Any use of the Specification and the information described therein will be governed by the terms and conditions of this Agreement.

Subject to the terms and conditions of this license, Sun hereby grants you a fully-paid, non-exclusive, non-transferable, limited license (without the right to sublicense) under Sun's intellectual property rights to review the Specification only for the purposes of evaluation. This license includes the right to discuss the Specification (including the right to provide limited excerpts of text to the extent relevant to the point(s) under discussion) with other licensees (under this or a substantially similar version of this Agreement) of the Specification. Other than this limited license, you acquire no right, title or interest in or to the Specification or any other Sun intellectual property, and the Specification may only be used in accordance with the license terms set forth herein. This license will expire on the earlier of: (i) two (2) years from the date of Release listed above; (ii) the date on which the final version of the Specification is publicly released; or (iii) the date on which the Java Specification Request (JSR) to which the Specification corresponds is withdrawn. In addition, this license will terminate immediately without notice from Sun if you fail to comply with any provision of this license. Upon termination, you must cease use of or destroy the Specification.

TRADEMARKS

No right, title, or interest in or to any trademarks, service marks, or trade names of Sun, Sun's licensors, Specification Lead or the Specification Lead's licensors is granted hereunder. Sun, Sun Microsystems, the Sun logo, Java, J2SE, J2EE, J2ME, Java Compatible, the Java Compatible Logo, and the Java Coffee Cup logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED "AS IS" AND IS EXPERIMENTAL AND MAY CONTAIN DEFECTS OR DEFICIENCIES WHICH CANNOT OR WILL NOT BE CORRECTED BY SUN. SUN MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE OR THAT ANY PRACTICE OR IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. This document does not represent any commitment to release or implement any portion of the Specification in any product.

THE SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION THEREIN; THESE CHANGES WILL BE INCORPORATED INTO NEW VERSIONS OF THE SPECIFICATION, IF ANY. SUN MAY MAKE IMPROVEMENTS AND/ OR CHANGES TO THE PRODUCT(S) AND/ OR THE PROGRAM(S) DESCRIBED IN THE SPECIFICATION AT ANY TIME. Any use of such changes in the Specification will be governed by the then-current license for the applicable version of the Specification.

LIMITATION OF LIABILITY

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

You will hold Sun (and its licensors) harmless from any claims based on your use of the Specification for any purposes other than the limited right of evaluation as described above, and from 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

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

REPORT

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

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.

Neither party may assign or otherwise transfer any of its rights or obligations under this Agreement, without the prior written consent of the other party, except that Sun may assign this Agreement to an affiliated company.

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.

## Contents

1 Preface ................................................................. vii
   Changes Since the EDR ........................................... vii

2 Introduction ......................................................... 1
   Summary ............................................................... 1
   Scope ....................................................................... 2
   Target Audience ..................................................... 2
   Organization .......................................................... 2
   Conventions ........................................................... 3
   Expert Group .......................................................... 3
   Status of this Document .......................................... 4

3 Overview ............................................................. 6
   Definitions .............................................................. 8
   Rationale & Assumptions ........................................ 8
   Goals ....................................................................... 9
   Roles ...................................................................... 9

4 Architecture of the JBI Environment ......................... 11
   WSDL-based Messaging Model ................................. 11
   Normalized Message .............................................. 11
   High-Level Architecture ........................................ 12
   Normalized Message Exchange ............................... 13
   Management .......................................................... 14
   Component Framework ......................................... 14
   Normalized Message Router ................................. 15
   Service Engines .................................................... 15
   Binding Components ............................................ 15
   Examples ............................................................. 15

5 Normalized Message Router .................................... 19
   Key Concepts ......................................................... 19
   Elements of a Message Exchange ........................... 24
   Normalized Message .............................................. 25
   Quality of Service ................................................. 26
   NMR Design ......................................................... 27
   NMR API ............................................................. 42

6 Management ......................................................... 55
   Overview ............................................................. 55
   Key Concepts ......................................................... 56
   Packaging ............................................................. 58
   Installation Service ............................................... 61
   Deployment Service .............................................. 64
   Component Life Cycle ............................................ 65
   Deployment Life Cycle .......................................... 66

7 Component Framework ........................................... 67
   Component-Supplied Management Functions ............ 68
   JBI-Supplied Environment Features ......................... 71

February 28, 2005 Java Business Integration 1.0 Public Draft v
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared Libraries and Class Loading</td>
<td>72</td>
</tr>
<tr>
<td>Error Indication</td>
<td>75</td>
</tr>
<tr>
<td>8 javax.jbi</td>
<td>77</td>
</tr>
<tr>
<td>JBIException</td>
<td>78</td>
</tr>
<tr>
<td>9 javax.jbi.component</td>
<td>81</td>
</tr>
<tr>
<td>Bootstrap</td>
<td>82</td>
</tr>
<tr>
<td>Component</td>
<td>84</td>
</tr>
<tr>
<td>ComponentContext</td>
<td>85</td>
</tr>
<tr>
<td>ComponentContext.EnvironmentType</td>
<td>89</td>
</tr>
<tr>
<td>ComponentLifeCycle</td>
<td>90</td>
</tr>
<tr>
<td>InstallationContext</td>
<td>92</td>
</tr>
<tr>
<td>ServiceUnitManager</td>
<td>94</td>
</tr>
<tr>
<td>10 javax.jbi.management</td>
<td>97</td>
</tr>
<tr>
<td>AdminServiceMBean</td>
<td>98</td>
</tr>
<tr>
<td>ComponentLifeCycleMBean</td>
<td>100</td>
</tr>
<tr>
<td>DeploymentException</td>
<td>101</td>
</tr>
<tr>
<td>DeploymentServiceMBean</td>
<td>103</td>
</tr>
<tr>
<td>InstallationServiceMBean</td>
<td>106</td>
</tr>
<tr>
<td>InstallerMBean</td>
<td>108</td>
</tr>
<tr>
<td>LifeCycleMBean</td>
<td>110</td>
</tr>
<tr>
<td>ManagementMessageBuilder</td>
<td>111</td>
</tr>
<tr>
<td>ManagementMessageFactory</td>
<td>113</td>
</tr>
<tr>
<td>MBeanNames</td>
<td>114</td>
</tr>
<tr>
<td>11 javax.jbi.messaging</td>
<td>117</td>
</tr>
<tr>
<td>DeliveryChannel</td>
<td>118</td>
</tr>
<tr>
<td>ExchangeStatus</td>
<td>122</td>
</tr>
<tr>
<td>Fault</td>
<td>124</td>
</tr>
<tr>
<td>InOnly</td>
<td>125</td>
</tr>
<tr>
<td>InOptionalOut</td>
<td>127</td>
</tr>
<tr>
<td>InOut</td>
<td>129</td>
</tr>
<tr>
<td>MessageExchange</td>
<td>131</td>
</tr>
<tr>
<td>MessagingException</td>
<td>136</td>
</tr>
<tr>
<td>NormalizedMessage</td>
<td>138</td>
</tr>
<tr>
<td>RobustInOnly</td>
<td>141</td>
</tr>
<tr>
<td>12 javax.jbi.servicedesc</td>
<td>143</td>
</tr>
<tr>
<td>Descriptor</td>
<td>144</td>
</tr>
<tr>
<td>EndpointReference</td>
<td>145</td>
</tr>
<tr>
<td>13 Almanac</td>
<td>149</td>
</tr>
<tr>
<td>14 Future Directions</td>
<td>157</td>
</tr>
<tr>
<td>15 References</td>
<td>159</td>
</tr>
<tr>
<td>16 Index</td>
<td>161</td>
</tr>
<tr>
<td>Colophon</td>
<td>167</td>
</tr>
</tbody>
</table>

**February 28, 2005**

Java Business Integration 1.0 Public Draft
Preface

This is the Public Review Draft of the Java Business Integration (JBI) 1.0 specification.

This draft is made available to the community for the purpose of giving insight into the work in progress of the JBI 1.0 Expert Group and to gather constructive feedback on the directions we are taking, as well as the specific Java interfaces have are defining.

Public Review Draft feedback to the Expert Group can be sent to jsr-208-comments@jcp.org.

1.1 Changes Since the EDR

The following list is a summary of changes made to this specification since the Early Draft Review, issued in September, 2004.

- The Normalized Message Service has been renamed Normalized Message Router (NMR). The messaging model has been reworked to allow bindings and engines to work as service providers, consumers, or both. There are no longer separate DeliveryChannel types for bindings and engines.
- The management chapter has been rewritten. It previously used a use case-driven set of illustrations of system management. It now uses functional definitions of the JBI-provided management pieces, and how they interact with management tools as well as the JBI components being managed.
- The three separate framework chapter have been consolidated into one, reflecting the convergence of bindings and engines in the NMR.
- The the separate binding and engine packages have been combined into one javax.jbi.component package.
- A new "component" interface has been added, and the component-provided interfaces refactored to use it.
- All JavaDocs for all JBI-defined APIs and SPIs are included in this specification.
Preface
Introduction

2.1 Summary

Enterprise application integration (EAI) and business-to-business integration (B2B) solutions have traditionally required the use of non-standard technologies to create functional systems. This has required end users to either “lock in” to a single vendor of such technologies, or create their own. Each approach has disadvantages. No single vendor can cover the vast functional space of EAI and B2B (consider the thousands of applications and protocols to be supported). This leaves users of integration technologies in the uncomfortable position of selecting less than ideal solutions to their integration problems, and paying dearly for them.

Java™ Business Integration (JBI) seeks to address this problem by creating a standards-based architecture for integration solutions. This infrastructure allows third-party components to be “plugged in” to a standard infrastructure, and allows those components to interoperate in a predictable, reliable fashion despite being produced by separate vendors. It is anticipated that this ability to interoperate will create a multivendor “ecosystem” which will give rise to large pool of integration-related technologies that can be sourced by end users. In addition, this ecosystem will foster new innovations in integration technologies, since it will permit innovators to concentrate on a particular technology or problem area, without having to worry about providing all the other pieces needed to build a complete integration solution.

Every integration problem is unique; an appropriate combination of JBI-compliant components will provide a solution that is sized appropriately to the problem at hand. By avoiding lock-in to a particular vendor of integration technologies, the user is free to choose components that provide the particular functions that he or she needs, and be assured that a functional integration solution can be assembled from those pieces.

In the past, attempts to compose third-party components into systems that have the attributes required of enterprise systems have not been very successful. JBI addresses this by adopting a service-oriented architecture (SOA), which maximizes the decoupling between components, and creates well-defined interoperation semantics founded on standards-based messaging. The SOA approach creates many other benefits that are applicable to enterprise integration solutions.
2.2 Scope

The membership of the JSR 208 Expert group is largely composed of business-oriented engineers and designers and this specification reflects their biases, priorities, requirements and experiences. In general, this specification is scoped to the needs and requirements of business solution providers.

This specification is considered to be complimentary to the J2EE™ Platform specification. JBI does not require the J2EE™ platform though there is a clear dependency on the J2SE™ platform. It is worth noting that the J2EE™ platform may choose to reference this specification in the future.

2.3 Target Audience

The primary audience for this specification is system level software engineers who will develop the following:

- implementations of the JBI infrastructure itself,
- the "plug in" components that provide
  - communications protocol support,
  - business logic,
  - transformation engines,
  - intelligent message routing,
  - etc.

This specification is not appropriate for application developers or business analysts. It is hoped that this specification will facilitate the development of standard service composition tools and services targeted specifically at this group.

2.4 Organization

This document is organized such that the initial sections provide some context and rationale for the JSR 208 initiative within the framework of the Java Community Process. Subsequent sections provide use-cases and relevant examples as a clarification of the System Programming Interfaces (SPI) specified in subsequent sections of the document.
2.5 Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC 2119].

Java code and XML fragments are formatted as shown in Listing 1

```java
package com.foo.example;
class Runner {
    public static void main(String[] args) {
        System.out.println("Hello, Universe!");
    }
}
```

Listing 1 Example code fragment

This specification uses a number of XML namespace prefixes throughout; they are listed in Table 1

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Namespace</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jbi</td>
<td><a href="http://java.sun.com/xml/ns/jbi">http://java.sun.com/xml/ns/jbi</a></td>
<td>Jbi descriptors for installation and deployment packages</td>
</tr>
<tr>
<td>wsd1</td>
<td><a href="http://www.w3.org/2004/08/wsd1">http://www.w3.org/2004/08/wsd1</a></td>
<td>Web services description language 2.0 last-call draft</td>
</tr>
</tbody>
</table>

Table 1 XML namespace prefixes

2.6 Expert Group

Without the contributions, input, and general guidance of the JSR 208 Expert Group this Early Draft of the specification would not have been possible. Thanks are extended to the following organizations for the support of their experts (past and present) and contributions to this JSR:

- Apache Software Foundation – James Strachan
- Sid Askary
- Borland Software Corp. – Ramanathan Ravikumar
- Cap Gemini – Steve Jones
- Collaxa – Edwin Khodabakchian
- DPWN SOP Group – Andreas Mattes, Ramtin Mesbahipour
- Fujitsu – Keith Swenson
- Intalio, Inc. – Assaf Arkin
- IONA – Peter Cousins
- IOPSIS Software Inc. – Rajesh Pradhan
- JBoss – Tom Baeyens
- Jie Liu
- Nokia Corp. – Sudhanshu Pant
- Novell, Inc. – Greg Ritzinger
2.7 Status of this Document

This specification is the Public Review Draft of JBI 1.0. The content herein is subject to change without notice and is considered as work-in-progress by the JSR 208 Expert Group members.
Overview

JBI defines an architecture that allows the construction of integration systems from plug-in components, that interoperate through the method of mediated message exchange. The message exchange model is based on the web services description language 2.0 [WSDL 2.0].

Figure 1 illustrates, at a high level, the JBI concept of plug-in components. JBI provides specific interfaces for use by a plug-in, while the plug-in provides specific interfaces for use by JBI. Components do not interact with
each other directly. Instead, as shown in Figure 2, JBI functions as an intermediary to route messages from component to component. This separation of the participants in an exchange is key to decoupling service providers from consumers, which is highly desirable in service-oriented architectures generally, and in integration solutions in particular. This preserves flexibility, since consumer and provider are minimally intertwined, and provides a key point for message processing and monitoring in JBI. Note also that this processing is inherently asynchronous; provider and consumer never share a thread. This also helps keep components decoupled.

In this WSDL-based, service-oriented model, JBI plug-in components are responsible for providing and consuming services. By providing a service, a component is making available a function or functions that can be consumed by other components (or even itself). Such functions are modelled as WSDL 2.0 operations, which involve the exchange of one or more messages. A set of four WSDL-defined, basic message exchange patterns (MEPs) crisply defines the sequence of messages allowed during execution of an operation. This shared understanding, between consumer and provider components, of the message exchange pattern is the foundation of interoperability of such components in JBI.

The services provided by components (if any) are described, by the component, to JBI using WSDL 2.0. This provides an abstract, technology-neutral model of services using XML-based message exchanges. WSDL also provides a mechanism for declaring additional service metadata of interest to service consumers and JBI itself. Components can query JBI the for the WSDL describing available services.

As shown in Figure 1, JBI components plug into what is termed the JBI framework. It is expected that such components will be made available to end users from third parties, particularly where common or standardized functions are needed in typical integration problems.

The components are divided into two distinct types:

- **Service Engine (SE)**. SEs provide business logic and transformation services to other components, as well as consume such services. SEs can integrate Java-based applications (and other resources), or applications with available Java APIs.

- **Binding Component (BC)**. BCs provide connectivity to services external to a JBI installation. This can involve communications protocols, or services provided by Enterprise Information Systems (EIS resources). BCs can integrate applications (and other resources) that use remoting technology that isn’t available directly in Java.

Service engines and binding components can both function as service providers, consumers, or both. (The distinction between them is purely pragmatic, but is based on sound architectural principles. The separation of business (and processing) logic from communications logic reduces implementation complexity, and increases flexibility.)

In addition to a messaging system designed to promote component interoperability, JBI defines a management structure, based on Java Management eXtensions (JMX). JBI provides standard mechanisms for:

- Installing components.
- Managing a component’s life cycle (stop/start etc.)
- Deploying service artifacts to components.

The last item requires some explanation. JBI components often function as a type of container, to which artifacts can be deployed to add new service consumer or provider logic. For example, a service engine that provides XSLT-based transformation services would have XSLT style sheets deployed to it, in order to add new transformation operations. This process of adding such component-specific artifacts to an installed component is called deployment, to distinguish it from component installation.
3.1 Definitions

JBI does not define a traditional application programming model. Instead, it embraces a service-oriented approach to structuring enterprise functions, where JBI plug-in components function as service providers and consumers. The APIs and SPIs defined in this specification are for the use of developers of service engines and binding components that “plug in” to the JBI infrastructure, as described above.

The term Service Engine (or SE) is used to refer to plug-in components that provide business logic, transformational services, etc. Service engines can take a wide variety of forms, depending on needs. In particular, some SEs can function as “containers,” presenting a unique programming model to users. For example:

- An XSLT XML transformation engine can support multiple transformation types. In this case, the programming interface is essentially the XSLT language itself.
- A WS-BPEL 2.0 business process execution engine. In this case, the programming interface is WS-BPEL itself.
- An EJB container. In this case, the API is Java that conforms to the appropriate EJB specification.

Binding components provide the ability to use communication protocols to both access remotely provided services, and allow remote service consumers to access services provided within the JBI environment. Protocols can be as varied as the integration problems being solved require, but typical examples include

- SOAP over HTTP [http://www.ws-i.org/Profiles/BasicProfile-1.1.html],
- JMS/MOM [http://jcp.org/aboutJava/communityprocess/final/jsr914/index.html] and

A Binding Component may choose to implement one or more communication protocols thus offering connectivity services to SEs and thereby enabling SEs to expose their services to remote consumers as well as enabling the consumption of remote services.

There are several distinct roles played by the various users of JBI:

- **Integration Architect.** This user designs the overall approach to solving integration problems. This includes selecting JBI components to be used to provide connectivity and business logic.

- **Integration Technologist.** This user devises the particular services needed to solve an integration problem, and configures a JBI system to integrate those services.

- **System Administrator.** This user installs, configures, monitors, and tunes the JBI system so that it provides the designed integration services.

- **JBI Component Developer.** JBI components can be created by users, or by third parties. In either case, the developer of JBI plug-ins must provide Java components that conform to particular contracts that are defined in this specification.

These roles are elaborated upon in “Roles” on page 9

3.2 Rationale & Assumptions

The rationale for this specification is predicated on the following key assumptions, empirical data, anecdotal evidence, and business requirements,

- **J2EE™ Platform suppliers increasingly view Web Services as a central rather than a peripheral aspect of their product offering**
• The evolution of the Web Services standards is driving Service Integration to occur at a pace and at a level requiring the inclusion of entirely new classes of “developers” trained not so much in procedural languages but in emerging declarative vocabularies.

• The need for supporting Service Composition vocabularies that depend heavily on WSDL [WSDL 2.0] presents an opportunity to reach convergence on what is meant by an abstract business message, and more specifically on what is meant by WSDL-defined messaging.

• By establishing a common abstract view of a normalized (business) message it becomes possible to decouple a component-specific application programming model (used to design and develop business logic) from the underlying communications infrastructure needed to support such logic.

3.3 Goals
The following are the primary goals of this specification:

• The establishment of a standard SPI for developers of service engines and binding components, known collectively as JBI components.

• The development of an abstract protocol-neutral Message Exchange and Normalized Message (NM)

• Provision of a standard mechanism for Message Exchanges to flow between JBI components

• Establishment of a standard for the packaging of JBI components, as well as for deployment of service artifacts to such components

• Definition of administrative and management hooks such that a diverse set of standard tools targeted at specific problem domains can emerge over time

• Given the complexity and diversity of SE and BC implementations, it must be possible for different vendors to deliver SEs, BCs or both. Such components must be able to interoperate reliably, and a system constructed of such components (atop a JBI infrastructure) must be manageable.

3.4 Roles
A functionally rich integration product is required to deliver a wide range of critical components over and above what a standard JBI environment requires. By design, JBI is silent about many necessary elements of a compelling overall solution. For example, a SE should be viewed as a container hosting business logic that is “programmed” using vocabularies ranging from annotated Java™ to potentially XSLT, neither of which is defined by JBI. JBI therefore assumes a number of different actors playing complimentary roles with the common underlying objective of delivering an overall business solution.

3.4.1 Engine Developers
A JBI compatible SE implementation is required to implement the Normalized Message Router (NMR) contract. In addition SE developers must implement the component life cycle and management interfaces, and, if needed, a deployment interface. If the SE functions as a container, SE developers may provide whatever tooling they feel appropriate for easing development of SE-specific artifacts.

3.4.2 Binding Developers
A JBI compatible BC implementation is required to implement the same contracts as an SE. Bindings are typically distinct from engines, due to their use of protocols to for remote access to, and provision of, services.
3.4.3 JBI Environment Providers
The suppliers of a JBI compatible environment must support the normative interfaces exposed in this specification. JBI compatible implementations may choose to use the J2EE™ 1.4 or newer platform but are not required to do so. If J2EE is not supported, then J2SE 1.4 or newer MUST be supported.

A JBI 1.0 compatible environment MUST support at least one implementation of a WS-I Basic Profile 1.1 [http://www.ws-i.org/Profiles/BasicProfile-1.1.html] compatible Binding Component.

A JBI compatible environment may choose to deliver a service engine implementation but are not required to do so.

3.4.4 J2EE™ Platform Providers
J2EE™ Platform providers may choose to deliver a complete JBI environment including Service Engines, Binding Components and application-level tooling. J2EE™ Platform providers are currently not required to support JBI.

3.4.5 JBI Application Developers
A JBI application developer is essentially modeling, designing, developing and deploying business components using vocabularies and tools that are defined by specific SE and BC implementations. The entire JBI environment is therefore one-step removed from a JBI application developer.

Many such developers will be developing XML artifacts used to customize engines and bindings. This is not the traditional model of a developer as used in the J2EE and J2SE domains, where the emphasis is on development of Java code.
CHAPTER 4

Architecture of the JBI Environment

JBI provides an environment in which plug-in components reside. The environment provides a set of services to facilitate execution of provided services, interaction between components, as well as management of the overall system created by a JBI installation and all installed components.

JBI provides for interoperation between plug-in components by means of message-based service invocation, described using a standard service description language. This provides a consistent model of services that the components can provide and consume.

JBI provides a set of services to facilitate management of the JBI environment, including the installed components. This includes component installation and life cycle management services.

4.1 WSDL-based Messaging Model

JBI models services produced and consumed by plug-in components using Web Services Description Language (version 2.0).

WSDL provides a declarative model of message-based services on two levels:

- **Abstract service model.** A service is defined using an abstract messaging model, without reference to a particular protocol or wire encoding.
- **Concrete (bound) model.** An abstract service that is bound to a particular protocol and communications end point.

JBI uses the abstract service model as the main basis of component interactions.

4.2 Normalized Message

A normalized message consists of two parts: the abstract XML message as described above, and message metadata (referred to as message context data). The message context data allows for the association of extra information with a particular message as it is processed by both plug-in components and system components. The metadata can affect message processing as the message moves through the JBI environment.
4.3 High-Level Architecture

A top-level view of the JBI architecture is shown in the following figure. The JBI environment exists within a single JVM. External to the JBI environment are service consumers and providers, representing the external entities that are to be integrated by JBI. These external entities can use a wide variety of technologies to communicate with Binding Components in the JBI environment. Service Engines are essentially standard containers for hosting WSDL-defined service providers and service consumers that are internal to the JBI environment.

Figure 3 above shows the major architectural components of JBI. This collection of components is called the JBI Environment. Within each JBI environment, there is a collection of services which will provide operational support. Key to this set of services is the Normalized Message router (NMR) which provides the mediated message exchange infrastructure, allowing components to interoperate. In addition, JBI defines a framework which provides the pluggable infrastructure for adding Service Engines (SEs), and protocol Binding Components (BCs). These are depicted within the yellow, C-shaped framework polygon.

The right-hand side of the JBI environment depicts the management features of JBI. JBI defines a standard set of JMX-based controls to allow external administrative tools (shown on the far right) to perform a variety of system administrative tasks, as well as administer the plug-in components themselves.

The core message exchange concept implements WSDL messaging, as discussed above. Service requests are generated by consumer components, routed by the NMR, and delivered to a provider component. For example, the BPEL SE may generate a request, which happens to be provided by the external service provider connected to the WS-I BC. The NMR will route the request to the WS-I binding. The SE in this case is a service consumer, and the BC a provider.

All provided services are exposed as WSDL-described services (end points, specifically). Services provided by SEs are described as end points in the same fashion as services provided by BCs (actually by external service providers). This provides a consistent model of service provision, without regard to location.

Service consumers identify needed services by WSDL service name, not by end point address. This decouples the consumer from the provider, and allows the NMR to select the appropriate provider.

Aside from the component framework and the NMR, the remainder of the JBI environment provides the infrastructure for life cycle management, environmental inspection, administration, and reconfiguration. These provide a predictable environment for reliable operations.
4.4 Normalized Message Exchange

The JBI environment's primary function is to route normalized message exchanges from one plug-in component to another. Messages are delivered in a normalized form.

Binding components must convert “bound” messages (messages in protocol- and transport-specific formats) to normalized form, as shown below. BCs and SEs interact with the NMR via a DeliveryChannel, which provides a bidirectional delivery contract for message reception and delivery.

An external service consumer sends a service request, bound to a particular protocol and transport, to the binding component. The binding component converts the request to normalized form, as described in a WSDL model that describes the service exposed by the end point exposed by the BC to the consumer. The BC then creates a Message Exchange (ME), which is a container for messages involved in various simple message exchange patterns. It sets ME metadata describing the desired service and operation to be performed. Finally, in step 2, the BC sends the ME, via its delivery channel, to the NMR for delivery to a service provider.

The NMR selects a suitable service provider, and routes the ME appropriately. The provider must accept (pull) the ME from the delivery channel.

Reversing the direction, as shown in Figure 5, changes very little to this processing picture, except for what the binding does. This is depicted below.

A consumer SE/BC creates a normalized message, putting it into a new MessageExchange requesting a service to be performed by a service referenced by its WSDL service name. The name is a logical name; the SE does...
Architecture of the JBI Environment

not select which component is used to perform the service. The ME is sent and accepted in exactly the same fashion as the previous example. After accepting the ME, the binding denormalizes the message into protocol and transport format, and sends the message across the wire to the external service provider.

4.5 Management

The JBI environment, including bindings and engines, is administered through JMX. Several management bean (MBean) types are defined by this specification to create a consistent management environment across the JBI-provided components and the plug-ins.

Major functions supported by the management interfaces are:

- Installation of engines and bindings (components)
- Life cycle management of plug-ins (start/stop controls)
- Deployment of component artifacts to engines and bindings that support dynamic additions to their internal execution environments. For instance, the SE/BC is an EJB container, and the artifacts are new EJBs to be installed in the container.
- Monitoring and control.

4.5.1 Component Installation

Engines and bindings must be installed via management interfaces. The verb “install,” as used in this specification, refers to installation of the binaries and related artifacts to provide the basic functions of the component. This is distinct from deployment, which is the act of adding application-specific artifacts to the component to customize its behavior in application-specific ways. For example, an XSLT transformation engine is installed, while specific transformation style sheets may be deployed to the transformation engine to supply needed transformation services.

4.5.2 Life Cycle Management

Once an engine or binding is installed, it can be started and stopped using MBean interfaces defined in this specification. This type of control is referred to as life cycle management.

4.5.3 Service Unit Deployment

As discussed above, deployment refers to the addition of component-specific artifacts to the installed engine or binding. These artifacts are referred to as Service Units. This specification says nothing about the contents of service units; they are completely opaque to JBI.

Service units are grouped into an aggregate deployment file called a Service Assembly. This includes a deployment descriptor that indicates where each SU is to be deployed.

4.6 Component Framework

The JBI component framework provides the pluggable interface which allows Bindings and Engines to interact with the JBI environment. The framework provides the interface to all of the JBI operational services.

The plug-ins interface to JBI via two mechanisms: SPIs (service provider interfaces) and APIs (application program interfaces). SPIs are interfaces implemented by the binding or engine; APIs are interfaces exposed to
bindings or engines by the framework. The contracts between framework and component are detailed in the “Component Framework” and “Normalized Message Router” sections of this specification. This details the obligations of both framework and JBI component to achieve particular functional goals within the JBI environment.

4.7 Normalized Message Router

The normalized message exchange delivery outlined above depends on the Normalized Message Router (NMR) to route message exchanges between service consumers and providers. The NMR supports performing such message delivery with varying qualities of service, depending on application needs and the nature of the messages being delivered.

The qualities of service supplied by the NMR, in collaboration with the bindings and engines as needed, are:

- **Best effort.** Messages may be dropped, or delivered more than once.
- **At least once.** Messages may not be dropped, but duplicates may be delivered.
- **Once and only once.** Messages are guaranteed to be delivered only once.

These are detailed in the “Normalized Message Router” chapter of this specification.

4.8 Service Engines

SEs are the business logic drivers of the JBI system. Engines can orchestrate service consumption and provision, in the course of, for example, executing long-lived business processes. Other engines can provide simple services, such as data transformation. Yet other engines may provide sophisticated routing or EDI services such as message collation / de-collation facilities.

SEs can create new services by aggregating other services. This is a key compositional pattern for languages such as WS-BPEL, that construct complex processes from services.

SEs can serve as service providers, service consumers, or both.

4.9 Binding Components

BCs are used to send and receive messages via particular protocols and transports. They serve to isolate the JBI environment from the particular protocol by providing normalization and denormalization from and to the protocol-specific format, allowing the JBI environment to deal only with normalized messages. (Note that protocol-specific metadata can be attached to a normalized message, or the message exchange, in their metadata, allowing protocol-specific information to be conveyed to a SE or BC in a fashion that is opaque to other JBI environment components.)

4.10 Examples

The following examples will help illustrate the use of the pieces of the JBI environment and plug-in components to perform typical integration tasks.
4.10.1 One-way Message Adapter

In this scenario, a JBI environment supplies a simple message adaptation service for a one-way message. A message is sent to the JBI environment, transformed, then sent to a destination outside of the environment. This is a simple form of point-to-point, A2A integration, and is illustrated below.

Client1 is a service consumer that wishes to make a one-way request of a service offered by Service1. Unfortunately, Client1 and Service1 don't have a common message format or messaging protocol, thus necessitating the use of integration middleware to adapt both the message format and the messaging protocol suitably. The middleware in this case is a JBI environment, shown in the large, rounded rectangle in the middle of the message sequence chart. The individual objects shown are:

- BC1. A binding component that “speaks” Client1’s protocol, which happens to be WS-I BP 1.1 compliant SOAP.
- SE1. A service engine used to provide light-weight sequencing of services. This can be configured to perform the desired message adaptation and forwarding.
- SE2. A service engine used to transform messages using XSLT 1.0.
- BC2. A binding component that “speaks” Service1's protocol, which is (in this example) AS2 over SMTP.

The message exchanges in the message sequence chart, above, are described in detail below.

- Client1 to BC1. The client is configured to send its request of Service1, the payload of which we term REQ1, to BC1. As far as Client1 is concerned, BC1 is the endpoint for accessing Service1, using the client's own messaging protocol.
- BC1 normalizes and forwards the inbound request for Service1, using the NMR to route the message. The JBI instance is configured to send requests for Service1 to SE1. (SE1 is a light-weight engine that can sequence the conversion and forwarding of messages.)
- SE1 selects the type of conversion to be performed, and sends a request to the conversion service to have REQ1 converted to what we will label REQ1A. The NMR will route this message to the SE2, which provides the desired service. SE2 will perform the transformation and synchronously return the result SE1.
- SE1 completes the sequencing the conversion-and-forward process by sending the result of the transformation, REQ1A to Service1. (The NMR will route this to BC2).
• BC2 denormalizes the message, and sends it (one-way) to Service1.

Note that the double-headed arrows in the sequence chart indicate messages routed by the NMR. This is an important feature: the components involved are loosely coupled. Each component indicates to the NMR the desired message destination only by service name; the NMR determines which component this corresponds to.
CHAPTER 5

Normalized Message Router

The normalized message router, or NMR, receives message exchanges from JBI components (engines or bindings), and routes them to the appropriate component for processing. This mediated message-exchange processing model decouples service consumers from providers, and allows the NMR to perform additional processing during the lifetime of the message exchange.

Note that this chapter makes use of WSDL 2.0 terms, rather than the older WSDL 1.1 ones.

5.1 Key Concepts

This section introduces key concepts in the NMR architecture. The goal of the NMR is to allow components acting as service producers and consumers to interoperate with one another in a predictable fashion, using WSDL-based service descriptions (supplemented by other metadata) as the sole source of coupling between them. This is key to allowing mix-and-match assembly of integration technologies.

[WSDL 2.0] provides the basic model, and description mechanism, for interaction of JBI components. WSDL provides an abstract service model, based on operations which are defined as XML message exchanges. This abstract model can be extended to provide binding-specific information: protocol-specific information declaring how the abstract model is mapped to an actual communications protocol and endpoint [WSDL 2.0 bindings].

JBI makes extensive use of WSDL’s abstract message model, supplemented with metadata where needed. The NMR can be thought of as an abstract WSDL-defined messaging system infrastructure, where bindings and engines serve to provide and consume WSDL-defined services.

5.1.1 Service Consumers and Providers

JBI engines and bindings function as service consumers, or providers, or both. A service provider makes a WSDL-described service (a collection of operations that are accomplished by message exchanges) available through an endpoint. A consumer makes use of services by initiating message exchange that invoke particular operations.

Consumers and providers are decoupled, in that they normally share only the (abstract) service definition, and not endpoint information. This isolates the consumer from the particulars of the service provider's implementation (protocol etc.).

5.1.2 Delivery Channel

A DeliveryChannel represents a bidirectional communication pipe used by bindings and engines to communicate with the NMR. The DeliveryChannel forms the API contract between service consumers/
providers and the NMR. A service consumer uses its delivery channel to initiate service invocations, while a
service provider uses its delivery channel to receive such invocations. A component that functions as both
consumer and provider uses the same delivery channel for both roles.

Take, for example, a binding that receives requests from a service consumer that is external to the JBI system.
Upon receiving such an inbound service invocation request, the BC must act as a service consumer while
interacting with the NMR via its delivery channel. (This reflects the binding component's function as a form of
consumer proxy in this use case.)

Each BC and SE is provided with a single delivery channel. Therefore implementations of
DeliveryChannel MUST support concurrent use of a given channel from multiple threads.

5.1.3 Run-time Registration
Run-time registration is the process by which a service provider registers the actual services it provides with the
NMR. Registration is divided into two steps:

• Declaring a service endpoint to the NMR.
• Providing metadata describing the nature of an endpoint definition.

Declaration is the process by which a binding or engine registers an endpoint name with the NMR. Any
endpoint name declared within the NMR must be supported by metadata describing the details of the
registration. Such details are provided separately, using a SPI that is separate from the delivery channel.

5.1.4 Service Invocation and Message Exchange Patterns
Service invocation refers to an instance of an end-to-end interaction between a service consumer and a service
provider. While it is impossible to define a complete set of service invocation patterns within JBI, the following
list represents the most common interactions:

• **One-Way**: consumer issues a request to provider with no error path.
• **One-Way with Fault**: consumer issues a request to provider. Provider may respond with a fault if it fails to
  process request.
• **Request-Response**: consumer issues a request to provider, with expectation of response. Provider may
  respond with a fault if it fails to process request.
• **Request Optional-Response**: consumer issues a request to provider, which may result in a response.
  Consumer and provider both have the option of generating a fault in response to a message received during
  the interaction.

The consumer and provider roles described above may be performed by bindings or engines, in any
combination of the two. When a binding acts as a service consumer, an external consumer is implied. Similarly,
when the binding acts as a provider, and external service is implied. The use of a service engine in either role
implies a local actor for that role.

**WSDL 2.0 Predefined Extensions** [WSDL 2.0 Extensions] defines a message exchange pattern (MEP) as “the
sequence and cardinality of abstract messages listed in an operation.” JBI uses this concept to define
interactions between two nodes: the consumer node, and the provider node. The pattern is defined in terms of
message type (normal or fault), and message direction.

MEPs always reflect the provider's point of view. For example, in a request-response interaction, the MEP
would be in-out, which reflects the flow of messages as seen by the provider. From the consumer's perspective,
the direction of message flow is reversed, but the MEP used by the NMR in its interactions with the consumer
will always reflect the providers perspective. This is a conventional practice when dealing with WSDL MEPs.
The in-out pattern is illustrated below, in figure Figure 7.

![Diagram of in-out message exchange pattern]

**Figure 7 In-out Message Exchange Patterns**

The first message (a request) is sent from the consumer to the provider (thus it has the "in" direction, from the
point of view of the provider). The second message (a response) is sent from the provider to the consumer (thus
is has the "out" direction). These exchanges are performed in the sequence given, with single messages. From
the provider’s perspective the exchange is a "in" message, following by an "out" message, thus leading to the
name of the pattern.

### 5.1.5 Message Exchange

A Message Exchange (ME) serves as a "container" for normalized messages that are part of a service
invocation. In encapsulates not only the normalized messages that are represented by the in and out messages of
the message exchange pattern realized by the ME, but holds metadata and state information associated with the
on-going exchange. A Message Exchange represents the JBI-local portion of a service invocation. The following table shows the relationship between service invocations and message exchange patterns.

<table>
<thead>
<tr>
<th>Service Invocation</th>
<th>Message Exchange Pattern (Provider View)</th>
</tr>
</thead>
<tbody>
<tr>
<td>One-Way</td>
<td>In-Only</td>
</tr>
<tr>
<td>Reliable One-Way</td>
<td>Robust In-Only</td>
</tr>
<tr>
<td>Request-Response</td>
<td>In-Out</td>
</tr>
<tr>
<td>Request Optional-Response</td>
<td>In Optional-Out</td>
</tr>
</tbody>
</table>

Table 2 Service Invocation to MEP Mapping

The activity diagram in Figure 8 depicts a one-way service invocation between two local Service Engines, including a message exchange instance:

Figure 8 One-way Service Invocation Between two Service Engines

SE1, acting as a service consumer, invokes the desired service and operation by creating and initializing an in-only message exchange, containing the request message. SE1 then sends the message exchange instance to the NMR. This is indicated by the object flow between the SE1 and NMR swim lanes.

The NMR determines which service provider should provide the requested service and operation, and routes the message exchange instance for delivery to the provider. This is shown by the object flow between the NMR and SE1 swim lanes.

The following depicts a reliable one-way invocation between service engines hosted in separate JBI
environments. Note that there is no provision for “federating” separate JBI instances; the two JBI instances shown in this example see each other as separate consumers and providers, accessed by the communications protocol provided by BC1 and BC2. There is no special relationship between the two JBI instances that affects how service invocation is performed between them.

SE1 constructs and initializes a robust-in message exchange instance, and sends it to the NMR. (In the process of creating the ME instance, the NMR has assigned it the unique identifier ‘X’.) The NMR determines which service provider should be used, and sends the message exchange to BC1 as a robust-in patterned instance. BC1 packages the request according to the requirements of the protocol that it implements, and sends the request to the service provider, which, in this example, happens to be another JBI instance, JBI2, and is accessed via an endpoint exposed by BC2.

When BC2 of JBI2 receives the request, it constructs a robust-in-patterned message exchange instance (“Y”), and sends to the NMR. The NMR of JBI2 determines which service provider should be used, and sends the message exchange instance (“Y”) to SE2.

SE 2 then receives instance “Y”. After processing the request, SE2 may elect to return a fault message, indicating an application-level error. In this case, SE2 can use instance “Y” to return a fault message, which will follow the reverse of the path the request followed, eventually arriving at SE1 of JBI1, in message exchange.

Figure 9 SE Invokes Service Provided by Remote JBI Instance: Robust In with Fault
instance “X”. (A positive “done” response in such a MEP would follow the same path, but is not shown here for sake of clarity.)

The following depicts a request-response invocation between a JBI-hosted Service Engine and a remote service provider.

![Diagram of request-response invocation](image)

**Figure 10 SE Invokes Remote Service**

The service engine SE creates and initializes an in-out-patterned message exchange instance. SE sends this to the NMR, which determines which service provider should process this invocation. In this case binding BC is selected. When BC receives the message exchange (an in-out pattern), it denormalizes the request, and sends it using its communications protocol to the external provider. The provider processes the request, returning either the normal response or an application-level fault, as appropriate. The BC normalizes the response, and adds it to the message exchange. The message exchange is then sent to the NMR, which routes it back to the consumer, SE.

### 5.2 Elements of a Message Exchange

- **Pattern** – every message exchange is described by a pattern, which describes the direction, sequence, cardinality, and names of messages/faults which participate in the exchange.
- **Initiator** – component which creates the message exchange. Normally service consumers act as initiators, using message exchange patterns that begin with an 'in' message. All standard JBI-defined patterns are consumer initiated.
- **Servicer** – component which services the message exchange. Normally service providers act as servicers, acting to service exchanges whose patterns begin with an 'in' message. Less commonly, service consumers may also act as servicers of provider-initiated exchanges (those patterns starting with an 'out' message).
- **Address** – a service reference, endpoint reference, and operation name for the logical address which the NMR uses to route message exchanges.
- **Message** – a message exchange carries one or more messages.
• **Fault** – a message exchange may carry at most one fault.
• **Status** – describes the status of the message exchange: error, done, or active.
• **Error** – Exception object used to describe the nature/source of an error status.
• **Properties** – initiators and services of a message exchange may associate arbitrary properties with a message exchange. The NMR may choose to reserve certain property names to declare QoS, security, transaction, or other operational meta-data.

Message exchange instances are short-lived, and will not survive a system shutdown or failure. Higher level error recovery logic, provided by consumer and provider components, must handle such error cases. Durable message exchange instances, which would allow JBI to supply such recoverability, are being considered for JBI 2.0.

### 5.2.1 Exchange Summary

The following message exchange patterns are defined by WSDL 2.0. The definition includes a fault rule, which specifies which message in the exchange is created (or replaced) to convey a fault from one participant to the other.

<table>
<thead>
<tr>
<th>Pattern Name</th>
<th>Message Sequence</th>
<th>Fault Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>In-Only</td>
<td>in</td>
<td>none</td>
</tr>
<tr>
<td>Robust In-Only</td>
<td>in, on</td>
<td>in</td>
</tr>
<tr>
<td>In-Out</td>
<td>in, out</td>
<td>replaces out</td>
</tr>
<tr>
<td>In Optional-Out</td>
<td>in, out</td>
<td>on in or out</td>
</tr>
</tbody>
</table>

Table 3 WSDL 2.0 Message Exchange Patterns

### 5.3 Normalized Message

Normalized messages are the messages of a JBI message exchange. This section defines normalized messages, and discusses the processing model JBI presents for such messages.

#### 5.3.1 Normalization Defined

Message normalization is the process of storing protocol and business context in a generic, transportable fashion. All messages handled by the Normalized Message Router are in normalized form. Normalization is a two-way street:

- **Normalizing** a message consists of mapping context-specific data to a context-neutral abstraction. Essentially, a normalizer must 'translate' context information that needs to be interpreted by the NMR into a 'generic' form that the NMR understands. Any other information (e.g. payloads, additional irrelevant context) can be added to a message, and carried by the NMR, but these data are completely opaque to the NMR. Note that normalization does not imply canonicalization of the payload data.

- **Denormalizing** a message refers to the process of receiving a message from the NMR and translating it back into a context-specific representation. Denormalization represents the inverse of a normalization operation.

#### 5.3.2 Structure of a Normalized Message

There are two distinct parts of a Normalized Message: context and content.
• **Message context** refers to a set of message "properties", which may be used to associate metadata with a message.

• **Message content** is essentially a generic source abstraction which contains all of the message data. In this context, the term "contain" is defined as inclusion within the abstract data model describing the message.

### 5.3.3 Processing Methods

The Normalized Message abstraction does not prescribe a specific processing style for the message content. A consumer of message content is free to choose the processing method based on context-specific requirements. For example, a highly performant binding component may wish to stream the content using a pull parser or a raw byte stream. A transformation service engine, on the other hand, may require that the entire message be in memory to facilitate arbitrary context queries, in which case an XML DOM tree representation may be appropriate.

### 5.3.4 Binary Inclusion

Support for binary inclusion must exist in some form in the Normalized Message abstraction. Ideally, the source object used to represent the content should be flexible enough to accommodate binary content. In the absence of such support, the NMR API may have to outline a set of best practices for storing unparsed (e.g. binary) content as message properties. The latter solution would be an absolute last resort.

### 5.4 Quality of Service

#### 5.4.1 Reliability

JBI is a solution in the enterprise computing arena, thus it requires a high-level of reliability. From the NMR point of view, this means that the message delivery design needs to address features that allow for at least a minimum level message exchange reliability. The NMR API's don't need to enforce 100% reliability, they just need to make that level of service available when needed. Allowing a range of reliability enables a range of solutions for BC and SE implementers.

#### 5.4.2 Transactional

Transactions have wide applicability in JBI-based applications. Using JBI to compose services will often require the sharing of a transaction context between JBI engines/bindings. Such transactions can facilitate reliable operation of composed services. The NMR facilitates the composition of business processing and communication, including sharing of transaction contexts.

The NMR MAY include transactional pieces itself.

The unit of work in the NMR is a Message Exchange. This is also the unit of work when a transaction is being used. The NMR is considered to be transaction aware. If the NMR is told that the Message Exchange is being controlled by a transaction, the NMR will internally use the transaction, if required, and will pass the transaction context along with the Message Exchange.

The primary reason for the inclusion of transactions is to assist in the primary responsibility of reliable messaging. Without transactions the One-Way communications primitive could not be used in a reliable fashion. The extension of transactions to other communications primitives is for completeness.

It should also be noted that the preceding assumes single JVM/JBI transactions. This is not meant to preclude the use of distributed transactions, but the standards for such transactions are not yet mature enough to be incorporated by JBI directly. Future versions of JBI may address distributed transactions directly.
5.4.3 Persistence
Persistence is the ability to persist the state of a Message Exchange at defined point during the Message Exchange lifetime. Persistence is a step below the Transactional. Persistence doesn't allow the coordination of multiple components. It does allow the Message Exchange state to be saved, typically to bound the work required during restart recovery (i.e. a transformation engine may wish to save the results of the XSLT transformation so that it this won't have to be redone given the relative expense of the transformation versus persisting the result.)
The general support for persistence is tied in with support for recoverability, which is discussed in the next section.

5.4.4 Recoverability
Recoverability is the ability of an active Message Exchange to be recovered during restart recovery to some consistent point in it's lifetime. This would normally involve either persisting a Message Exchange or using finer grain transactions to save the state of the Message Exchange.

5.4.5 Secrecy
Secrecy is the protection of information from being disclosed to outside parties. This is a basic requirement in the financial industry, and often required in many other industries. In the NMR this comes about if the NMR uses external storage in the implementation of transactions or persistence. The basic approach to solving this problem is to encrypt any user data stored on external storage. This is an implementation problem, and not one directly addressed by JBI.

5.5 NMR Design
This section provides detailed design information on the Normalized Message Router. The contracts between a JBI component (binding or engine) and the NMR are detailed.

5.5.1 Registration
As stated previously, registration is the process by which service providers declare to the NMR the particular services they provide. Each declaration must be accompanied by a corresponding metadata definition, which describes the declaration. JBI requires that WSDL 2.0 be used for all such declarations. The NMR presumes (but does not enforce) that all such declarations are compliant with the WSDL 2.0 specification.
5.5.1.1 Service Provider Registration

A domain model for service provider registration is pictured below. Each endpoint registration by a provider must be accompanied by metadata describing the service, endpoint, and binding details for the midpoint. (A
service engine registers a special JBI-defined binding type for service engine-provided services being so registered.)

A logical sequence diagram depicting a registration scenario follows. This picture is limited purely to registration-related activity, and does not address the relation between endpoint registration and message exchanges.

- BC creates an implementation of NMR Resolver interface to provide metadata describing endpoint registrations.
- BC registers its Resolver implementation once with the NMR.
- BC registers an endpoint name.
- NMR initializes the Endpoint reference.
- SE queries the NMR for endpoint by name (or some other criteria).
- SE uses reference to query NMR for metadata describing the Endpoint.
- NMR invokes the BCs Resolver to fetch the appropriate meta-data.
- SE parses the Descriptor returned by the NMR.
5.5.2 Exchanging Messages
This section provides an in-depth discussion of the eight standard MEPs supported in JBI. It is important to note that the NMR provides for an extensible model of message exchange. JBI vendors that wish to extend the standard catalog of eight MEPs are free to do so.

Each pattern ends with a status exchange; this is necessary to deterministically complete the exchange. This is especially useful when optional messages (or faults) are part of the exchange pattern, but is necessary in all MEPs to support reliable message exchange.

Provider-initiated exchanges are included in these patterns. While these patterns are less commonly used, they are necessary in significant use cases.

5.5.2.1 Diagram Key
The MEP diagrams here use the symbols summarized in the following figure. The diagrams themselves present a chronological view of the exchanges, with time progressing from the top of each diagram to the bottom.

5.5.2.2 Out-Only Message Exchange
This pattern is used for a one-way, provider-initiated exchange.

Notes:
- Provider initiates with message.
- Consumer responds with status to complete exchange.
5.5.2.3 In-Only Message Exchange
This pattern is used for one-way, consumer-initiated exchanges.

![Figure 15 In-only Message Exchange Pattern](image)

Notes:
- Consumer initiates with message.
- Provider responds with status to complete exchange.

5.5.2.4 Robust Out-Only Message Exchange
This pattern is used for a reliable one-way, provider-initiated exchange.

![Figure 16 Robust Out-only Message Exchange Pattern](image)

Notes:
- Provider initiates with `message`.
- Consumer may respond with `status` or `fault`.
- If consumer response is `status`, the exchange is complete.
- If consumer response is `fault`, provider responds with `status` to complete exchange.
5.5.2.5 Robust In-Only Message Exchange
This pattern is used for reliable, one-way consumer-initiated message exchanges.

![Figure 17 Robust In-only Message Exchange Pattern](image)

Notes:
- Consumer initiates with **message**.
- Provider may respond with **status** or **fault**.
- If provider response is **status**, the exchange is complete.
- If provider response is **fault**, consumer responds with **status** to complete exchange.

5.5.2.6 Out-In Message Exchange
This pattern is used for two-way, provider-initiated exchanges.

![Figure 18 Out-in Message Exchange Pattern](image)

Notes:
• Provider initiates with **message**.
• Consumer responds with **message** or **fault**.
• Provider responds with **status**.

### 5.5.2.7 In-Out Message Exchange

This pattern is used for two-way, consumer-initiated exchanges.

![Figure 19 In-out Message Exchange Pattern](image)

Notes:
• Consumer initiates with **message**.
• Provider responds with **message** or **fault**.
• Consumer responds with **status**.
5.5.2.8 Out Optional-In Message Exchange

This pattern is used for two-way, provider-initiated exchanges where the consumer's response is optional. This is an interesting pattern, for it illustrates that without the final status exchange, the provider cannot determine when the pattern is completed.

Figure 20 Out Optional-in Message Exchange Pattern

Notes:
- Provider initiates with message.
- Consumer may respond with message, fault, or status.
- If consumer responds with status, the exchange is complete.
- If consumer responds with fault, provider must respond with status.
- If consumer responds with message, provider may respond with fault or status.
- If provider responds with fault, consumer must respond with status to complete exchange.
5.5.2.9 In Optional-Out Message Exchange

This pattern is used for a two-way, consumer-initiated exchange where the provider's response is optional.

![Diagram of In Optional-out Message Exchange Pattern]

Notes:

- Consumer initiates with message.
- Provider may respond with message, fault, or status.
- If provider responds with status, the exchange is complete.
- If provider responds with fault, consumer must respond with status.
- If provider responds with message, consumer may respond with fault or status.
- If consumer responds with fault, provider must respond with status to complete exchange.

5.5.3 Reliability in Message Exchange

The following sections show how reliable message delivery can be accomplished using the NMR API's for each of the communication patterns.

Assumptions:

- Messages are idempotent
- If the initiator doesn't receive an end-to-end indication of completion (success/failure) of a communications primitive, then the initiator will retry the communications primitive.
- If the servicer must retain some persistent result of the operation, the servicer must persist the result before responding.
• Message Exchanges have a unique identifier.
• Any Message Exchange identifier saved during normal processing should be deleted at restart of the JBI component.

This is mostly equivalent to an at-least-once message delivery mechanism.

5.5.3.1 One-way with Fault

Reliable communications protocol handlers often provide mechanisms for assuring delivery of an inbound message to the application. Failure to successfully deliver the message will result in retries. Such protocols are often realized as persistent message queues, as depicted in the message sequence chart below.

Stateful service engines use a persistent store to retain state over system shutdowns. To assure reliable message transfer from the reliable messaging queue to the SE’s persistent state store, the message exchange should not be ended (with status exchange) until the message (or the results of processing the message) are persisted by the SE.

Notes:
1. Code in or attached to BC queues a message for delivery to the BC. It expects that this information is persisted. Once on the queue it should never be removed until step 8 has been reached. This guarantees that the message will at least be sent once.
2. The BC picks a message to attempt delivery.
3. BC normalizes the message, and sends it to NMR.
4. SE accepts message from NMR
5. SE saves message. The SE could also check to see if the message has been seen before if follow-on logic can’t tolerate duplicate messages.
6. SE sends status to the NMR. SE can reject the message (e.g., structural, content or age problem) or accept it, in either case the outcome is returned to the initiator.

Figure 22 Reliable One-way MEP with Fault Message Sequence Chart
7. BC accepts status from the NMR.
8. BC removes message from the delivery queue. After this point in time no attempt to redeliver the message will be made.

5.5.3.2 Request-Response

Notes:

1. Code in or attached to BC queues a message for delivery to the BC. It expects that this information is persisted. Once on the queue it should never be removed until step 8 has been reached. The guarantees that the message will at least be sent once.
2. The BC picks a message to attempt delivery.
3. BC sends the message to the NMR.
4. SE accepts the message from the NMR
5. SE saves the message and its response to the message. (The SE could also check to see if the message has been seen before if follow-on logic can't tolerate duplicate messages.)
6. SE sends the response to NMR. SE can reject the message (e.g., structural, content or age problem) or accept it, in either case the outcome is returned to the initiator.
7. BC accepts response from NMR.
8. BC saves response
9. BC sends status of response to NMR
10. SE get status from the NMR
11. SE can cleanup any leftovers since it has positive acknowledgment of acceptance.
12. BC performs any cleanup.

5.5.3.3 Request Optional-Response
The Request Optional-Response communication primitive is just a dynamic selection of either Robust One-Way
or Request-Response. The selection is detected when (in the examples) the SE returns a status (Robust One-
Way) or SE returns a response message (Request-Response).

5.5.4 Transactional Message Exchange
The following sections show how reliable message delivery can be augmented with transactions to increase the
reliability and reduce work that a caller of the NMR needs to perform.

5.5.4.1 One-Way
An example sequence of operations for an exchange initiator is shown below.

![Figure 24 One-way Transactional Message Exchange]

Notes:
1. Initiator (an SE or a BC) starts a transaction. It may use its own transactional resources.
2. Initiator creates and initializes a Message Exchange instance, and sets the transaction on the Message
   Exchange properties.
3. Initiator sends the ME to NMR.
4. NMR enlists its own internal resource.
5. NMR saves the ME.
6. NMR delists its own internal resource
7. Initiator commits the transaction. The initiator could do some more work with its own resources before the
commit and after the send().

An example message sequence diagram for the servicer side of this scenario is shown below:

Figure 25

Notes: (This is the servicer side.)

1. Servicer calls accept() to get a ME.
2. NMR reads the ME transaction property, then enlists its resource manager in the transaction.
3. NMR gets the ME data persisted in the previous (initiator) example.
4. NMR delists its resource manager.
5. Initiator calls setStatus() to say that its finished with the ME. The initiator can enlist its own resources and perform operations against these.
6. Initiator commits the transaction.
5.5.4.2 One-Way with Fault

Notes:
1. Initiator starts a transaction and enlists it local resource.
2. Initiator can work with the local resource and then send the ME.
3. Servicer calls except, the transaction has been switched to the servicer's thread.
4. Servicer does whatever it wants with the message. It can enlist and delist local resource it needed.
5. Servicer sends the status back.
6. Initiator accepts the status. The transaction context has been switched back to the initiator's accept() thread.
7. Initiator can do other work with its local transaction and the commits.

Figure 26 Transacted One-way Exchange with Fault
5.5.4.3 Request-Response

The Request Response communications primitive is the one case that can't be made reliable in a practical fashion. The next section on transactional message exchange will provide a solution for this case.

1. Initiator creates a transaction.
2. Initiator enlists its local resources in the transaction and sends the message.
3. Servicer accepts the ME and the transaction context moves to the servicer's accept() thread.
4. Servicer processes the request, enlisting its local resources into the transaction if it needs.
5. Servicer delists any local resources and sends the response.
6. Initiator accepts the response and the transaction context is switched to the initiator thread.
7. Initiator can use local resources and send the status to the servicer.
8. Servicer accepts the status. Transaction context is not available, probably not much to do here.
9. Servicer can do anything it likes but can't effect the transaction.
10. Initiator can use local resources and then delists and commits the transaction.

5.5.4.4 Request Optional-Response

Just like the reliable version of Request Optional-Response, this is like a dynamic selection of Robust One-Way or Request-Response.
5.6 NMR API

The NMR exposes a standard API to SEs and BCs.

5.6.1 Message API

The Message API defines the content and behavior of normalized messages within JBI. Key functions of this API include:

- Storing/retrieving content using a standard data model (XML)
- Ability to set/get context information on a per-message basis (properties)

Figure 28 Message API Class Diagram
5.6.2 Service API

This API allows SEs to interact with the NMR.

```
<<interface>>
DeliveryChannel
+accept()
+accept(timeout : long)
+availableEndpoints()
+activateEndpoint()
+deactivateEndpoint()
+close()
+getResolver()
+setResolver()
+send()
+sendSync(timeout : long)
+createExchange([uri : URI])
+createExchange(serviceName : QName, operationName : QName)
+createInOnlyExchange()
+createOptionalInExchange()
+createOutOnlyExchange()
+createOutExchange()
+createRobustOutOnlyExchange()
+createRobustOutExchange()
```

Figure 29 Service API Classes

5.6.2.1 DeliveryChannel

The DeliveryChannel interface serves as the primary means for a JBI component to interact with the NMR. It allows the component to

- receive and send messages
- create message exchanges
- query service metadata
- register its own metadata resolver implementation.

5.6.2.1.1 Creating Message Exchanges

The DeliveryChannel provides three distinct ways to create new message exchange instances. These support three different needs:

- \texttt{createExchange(serviceName, operationName)}. Creation from service name and operation name. This permits the component to quickly put together an ME instance, without the need of inspecting MEPs in service metadata.

- \texttt{createExchange(URI)}. Creation from MEP URI. This permits extensions to the supported MEPs.

- \texttt{createInOnlyExchange()}, etc. Creation using MEP-specific convenience methods. This is convenient for creating standard MEs in situations where the MEP required is known at coding-time.

5.6.2.1.2 Sending Messages

The \texttt{DeliveryChannel} offers two basic means of sending message exchanges:
• `send(exchange)` - Asynchronous message transmission: this method doesn't block. The exchange will be delivered concurrent to the thread of execution that called `send()`.

• `sendSynch(exchange)` - Synchronous message transmission: this method blocks the current thread of execution until the exchange is returned.

The latter method has a variant that adds a time-out parameter. This allows JBI components to use simple synchronous message exchanges without needing to manage “stuck” threads if an exchange isn't returned as expected. To support this time-out feature, the `sendSynch()` method returns a boolean value to indicate successful return of the exchange instance.

Note that in the case of asynchronous exchanges, there is no guarantee that the message exchange received (by the receiver's call to `accept()`, discussed in the next section) will be the same object that was initially sent. Rather, the only guarantee is that the contents of the exchange instance, including the message exchange ID, will be the same.

### 5.6.2.1.3 Receiving Messages

The `DeliveryChannel` offers two ways of receiving message exchanges:

• `accept()` - This method blocks until a MessageExchange instance is received from the NMR.

• `accept(timeoutMS)` - This method works similarly to `accept()`, except that it is blocked for the given time-out interval, the method will return `null`, indicating that nothing was received.

Message reception always follows the “pull” model: the JBI component must pull the message exchange instance from the `DeliveryChannel`, using its own thread of execution. Components may also use multiple such threads, allowing concurrent message reception by the component. The JBI implementation can satisfy multiple `accept()` requests from a single component in arbitrary order.

### 5.6.2.1.4 Querying Metadata

The `DeliveryChannel` also allows JBI components to query the metadata of service providers in the JBI system. This allows for dynamic service discovery and service provider selection.

• `availableEndpoints()` allows a component to discover all of the endpoints providing a particular service type.

• `getEndpointDescriptor()` allows a component to get the metadata descriptor handle for a particular endpoint.

Once an `Descriptor` for an endpoint is found, the service metadata (that is, the WSDL description of the endpoint) can be retrieved using its `getDescription()` method, which returns a DOM representation of the endpoint's description.

### 5.6.2.1.5 Endpoint Activation and Deactivation

A JBI component that acts as a service provider must indicate to the NMR which endpoints it is currently providing (a component can supply multiple endpoints). Such a component can dynamically (and arbitrarily) activate and deactivate endpoints, using the following `DeliveryChannel` methods:

• `activateEndpoint(serviceName, endpointName)` - This method is used by a component to let the NMR know that the named endpoint is ready to process message exchanges. This MUST NOT be called until the component is fully ready to process message exchanges destined for the named endpoint, and the component's `Resolver` implementation is ready to supply service data about the endpoint.

• `deactivateEndpoint(endpoint)` - This method is used by a component to let the NMR know that the named endpoint is being withdrawn from service. The NMR MUST NOT allow initiators to invoke the endpoint's services after the endpoint is deactivated. Ongoing message exchanges will be permitted to continue to completion.
5.6.2.1.6 Closing Down a DeliveryChannel

The component can deactivate all endpoints for the services it provides in a single DeliveryChannel call:

- **close()**. This method causes all activate endpoints registered with the channel (that is, all services provided by the component) to be shut down, in the same fashion as if deactivateEndpoint() was called for each active endpoint.

Once a DeliveryChannel is so closed, it cannot be re-opened. Instead, a new instance of DeliveryChannel is needed to allow a component to resume interacting with the NMR.

5.6.2.2 Resolver SPI

With the Resolver interface, the component can supply service metadata concerning services it provides. All components that act as service providers MUST register a Resolver implementation prior to activating any such services.

The Resolver interface defines a single method:

- **Descriptor resolveReference(EndpointReference)**. The component's implementation must return a Descriptor (a wrapper for DOM-based WSDL) for the named endpoint, or null if the endpoint is not active.

The document provided MUST be a WSDL 2.0-compliant document, with the following additional properties:

- The document must not use the <wsdl:import> or <wsdl:include> elements (the document must be stand-alone).

- Service Engine-provided endpoints MUST use the binding type defined in the section titled “JBI Service Engine WSDL Binding” on page 54.

The document must provide the complete definition for the named service, and the named endpoint, including the service's interface and operations (including, of course, the message exchange patterns used).
5.6.2.3 Message Exchange Pattern APIs

The message exchange pattern APIs for all built-in MEPs, are depicted in the following figure:

All message exchange instances are created by calls to suitable factory methods provided by `DeliveryChannel`. `MessageExchange` instances are owned by either the initiator, servicer, or NMR. Creating or accepting a message exchange instance confers ownership on the creating (or accepting) component. Upon calling `send()` for a particular ME instance, the component surrenders ownership of the ME instance immediately. `sendSynch()` causes ownership to be surrendered during the course of the call; when the call ends (normally) the component resumes ownership of the ME instance.

Ownership of an ME instances gives the owner the right to read from and write to the instance. When a component (or the NMR) doesn't have ownership of an ME instance, it MUST NOT read from or write to the instance. If a component that is not the current owner of the instance does attempt to call any method on the instance an appropriate MessagingException MUST be thrown.

5.6.2.3.1 MessageExchange

The super-interface, `MessageExchange`, also provides common methods for in- and out-* message exchange pattern APIs. `MessageExchange` instances provide the context for a sequence of normalized message interchanges as well as status interchange. The instance can be thought of as a stateful container for normalized messages.

Each exchange instance has the following features which are accessed by the `MessageExchange` API:

- **Status. ExchangeStatus** provides a type-safe enumeration of the possible values of an ME's status during its life cycle. A new ME instance is in **ACTIVE**, and remains so until the end of its life cycle, when
it transitions to either a DONE or ERROR state. The status can be queried and set using getStatus() and setStatus(), respectively.

- **Source of an ERROR status.** This feature allows the instance to retain the root cause (Java exception) of a processing failure. See getError() and setError().

- **Exchange ID.** Each instance of a MessageExchange has a unique identifier string, assigned by the JBI implementation. This identifier can be used by JBI components to associate asynchronous responses with originating requests.

- **Exchange Pattern.** Each instance of a MessageExchange has a pattern identifier. The standard Message Exchange Patterns are identified by their WSDL 2.0-assigned URIs.

- **Service name.** The qualified-name of the service being invoked by this exchange.

- **Operation name.** The qualified-name of the operation being invoked by this exchange.

- **Endpoint name.** The optional name of the endpoint of the service provider.

- **Normalized messages.** The instance serves as a factory for NormalizedMessages associated with the exchange instance. Methods are provided to create, set, and query such messages. See createMessage(), setMessage(), and getMessage().

- **Fault messages.** The instance serves as a factory for Fault messages associated with the exchange instance. Methods are provided to create, set, and query such messages. See createFault(), setFault(), and getFault().

- **Transaction Context.** If the transaction context is non-null, the message exchange is transacted. See getTransactionContext() and setTransactionContext().

- **Properties.** Non-JBI message exchange features can be stored as properties: name-value pairs, where the value is (to JBI) an opaque java.lang.Object.

The MessageExchange API provides sufficient functions to support all standard MEPs. However, “in” and “out” messages are not distinguished at the API level, making understanding of ongoing exchanges at a Java code level difficult. To address this issue, a set of MEP-specific APIs are provided, and are detailed in the four sections below.

### 5.6.2.3.2 InOnly Message Exchange

This sub-interface provides an API for the “http://www.w3.org/2004/08/wsd1/in-only” MEP. This provides the following MEP-specific methods:

- **setInMessage(msg).** This can be called only once;

- **getInMessage().** This can be called multiple times; if the set method (or setMessage()) has not been called, it will return null.

### 5.6.2.3.3 InOptionalOut Message Exchange

This sub-interface provides an API for the “http://www.w3.org/2004/08/wsd1/in-opt-out” MEP. This provides the following MEP-specific methods:

- **setInMessage(msg).** This can be called only once, by the service consumer.

- **getInMessage().** This can be called multiple times; if the set method (or setMessage()) has not been called, it will return null.

- **setOutMessage(msg).** This can be called only once, by the service provider. Setting the “out” message is optional in this pattern.

- **getOutMessage().** This can be called multiple times; if the set method (or setMessage()) has not been called, it will return null.
5.6.2.3.4 InOut Message Exchange
This sub-interface provides an API for the “http://www.w3.org/2004/08/wsd1/in-out” MEP. This provides the following MEP-specific methods:

• `setInMessage(msg)`. This can be called only once, by the service consumer.

• `getInMessage()`. This can be called multiple times; if the set method (or `setMessage()` has not been called, it will return null.

• `setOutMessage(msg)`. This can be called only once, by the service provider.

• `getOutMessage()`. This can be called multiple times; if the set method (or `setMessage()`) has not been called, it will return null.

5.6.2.3.5 RobustInOnly Message Exchange
This sub-interface provides an API for the “http://www.w3.org/2004/08/wsd1/robust-in-only” MEP. This provides the following MEP-specific methods:

• `setInMessage(msg)`. This can be called only once, by the service consumer.

• `getInMessage()`. This can be called multiple times; if the set method (or `setMessage()` has not been called, it will return null.

5.6.3 Usage Examples
This section provides detailed examples on common BC, SE, and NMR interactions. The areas of run-time registration and message exchange are covered in detail.

5.6.3.1 Service Consumer
This example takes place within a single JBI environment. A service engine, ConsumerEngine, is the consumer of an external web service (i.e. outbound service invocation).

Notes:

• Service name : {http://abc.com/services} : service1
• Endpoint name : soap1
• Direction : Outbound
• Operation(s) : updateStatus (one way)
• For simplicity, BC and SE are single-threaded and only participate in one exchange during the course of this example.
• Assume the following constants are declared:
  • QName SERVICE1 = new QName("http://abc.com/services", "service1");
  • QName OP_UPDATE_STATUS = new QName("http://abc.com/services", "updateStatus");
  • String ENDPOINT1 = “soap1”;
5.6.3.1.1 SE Initialization

class ConsumerEngine implements ComponentLifeCycle,
{
    DeliveryChannel   channel;
    public void init(ComponentContext context)
    throws JBIException
    {
        // Obtain reference to delivery channel
        channel = context.getDeliveryChannel();
        // If this engine was also a service provider, we would register
        // our Resolver implementation at this point.
        }
        // ...
    }
}

Listing 2 Service Engine Initialization

5.6.3.1.2 BC Initialization

class SoapBinding implements ComponentLifeCycle
{
    DeliveryChannel   channel;
    EndpointReference endpointRef;
    public void init(ComponentContext context)
    throws JBIException
    {
        // Obtain reference to delivery channel
        channel = context.getDeliveryChannel();
        // Create a Resolver impl for metadata and attach to channel
        Resolver myMetaData = new MyResolver();
        channel.setResolver(myMetaData);

        // Register endpoint names
        endpointRef = channel.activateEndpoint(SERVICE1, ENDPOINT1);
        }
        // ...
    }
}

Listing 3 Binding Component Initialization
5.6.3.1.3 Message Exchange

Engine View

```java
void someEngineMethod()
{
    InOnly inOnly;
    NormalizedMessage message;
    // As initiator, we get to create the message exchange
    inOnly = channel.createInOnlyExchange();
    // Create new message for this exchange.
    message = inOnly.createMessage();
    // Populate message with data (omitted)
    // Set message as in reference in exchange
    inOnly.setInMessage(message);
    // Set operation and service details
    // (We could avoid this by using channel.createExchange(SERVICE1, 
    // OP_UPDATE_STATUS); above)
    inOnly.setOperation(OP_UPDATE_STATUS);
    inOnly.setService(SERVICE1);
    // We will allow the NMR to select the endpoint automatically.
    // Alternatively, we could query the available endpoints implementing SERVICE1,
    // choose from that list, and use inOnly.setEndpoint() to route the ME ourselves.
    // Initiate the message exchange
    channel.send(inOnly);
}
```

Listing 4 Message Exchange: Service Engine View

Binding View

```java
void someBindingMethod()
{
    MessageExchange exchange = channel.accept();
    if (exchange instanceof InOnly)
        process((InOnly) exchange);
    else // ...
}

void process(InOnly inOnly)
{
    NormalizedMessage inMsg;
    // fetch "in" message from exchange
    inMsg = inOnly.getInMessage();
    // process data
    try
    {
        // commit message to wire (omitted)
        // set status appropriately
        inOnly.setStatus(ExchangeStatus.DONE);
    }
    catch (Exception ex)
    {
        inOnly.setError(ex);
        inOnly.setStatus(ExchangeStatus.ERROR);
    }
    channel.send(outOnly);
}
```

Listing 5 Message Exchange: Binding View
5.6.3.2 Service Provider (Servicer)
This example takes place within a single JBI environment. A service engine, ProviderEngine, provides a service to external consumers (i.e. inbound service invocation) through a SOAP binding component.

Notes:
• Service name : {http://xyz.com/services} : service2
• Endpoint name : engine2
• Direction : Inbound
• Operation(s) : getStockQuote (request -response)
• For simplicity, BC and SE are single-threaded and only participate in one exchange during the course of this example.
• Assume the following constants are declared:
  • QName SERVICE2 = new QName("http://xyz.com/services", "service2");
  • String ENDPOINT2 = “engine2”;

5.6.3.2.1 BC Initialization

class SoapBinding implements ComponentLifeCycle
{
    DeliveryChannel    channel;
    public void init(ComponentContext context)
    throws JBIException
    {
        // Obtain reference to delivery channel
        channel = context.getDeliveryChannel();
        // Create listener for inbound SOAP messages
        // (omitted)
    }
}

Listing 6 Example SOAP Binding Component Initialization

5.6.3.2.2 SE Initialization

class ConsumerEngine implements ComponentLifeCycle
{
    DeliveryChannel    channel;
    EndpointReference   endpointRef;
    public void init(ComponentContext context)
    throws JBIException
    {
        // Obtain reference to delivery channel
        channel = context.getDeliveryChannel();
        // Create a Resolver impl for meta-data and attach to channel
        Resolver myMetaData = new MyResolver();
        channel.setResolver(myMetaData);
        // Register service name
        endpointRef = channel.activateEndpoint(SERVICE2, ENDPOINT2);
    }
}

Listing 7 Example Service Engine Initialization

5.6.3.2.3 Message Delivery
Binding View

```java
void someBindingMethod()
{
    InOut inOut;
    NormalizedMessage inMsg;

    // Receive message over native protocol (omitted)
    // Create InOut exchange
    inOut = channel.createInOutExchange();

    // Create new message
    inMsg = inOut.createMessage();

    // Normalize native protocol data (omitted)
    // Set message as in reference in exchange
    inOut.setInMessage(inMsg);

    // Set operation and service details
    inOut.setOperation(new QName(SERVICE2, "getStockQuote"));
    inOut.setService(SERVICE2);

    // Initiate the message exchange and wait for response
    InOut exchange = (InOut) channel.sendSync(inOut);
    process((InOut) exchange);
}

void process(InOut inOut)
{
    NormalizedMessage outMsg = inOut.getOutMessage();
    if (outMsg == null)
    {
        // check status, potential error
    }

    // commit response message to native wire protocol (omitted)
}
```

Listing 8 Example Binding Using Synchronous Message Exchange for Service Invocation
Engine View

```java
void someEngineMethod()
{
    MessageExchange exchange = channel.accept();
    if (exchange instanceof InOut)
        process((InOut) exchange);
    else // ...
}

void process(InOut inOut)
{
    NormalizedMessage inMsg; NormalizedMessage outMsg;
    // fetch in message
    inMsg = inOut.getMessage();
    // process data
    try
    {
        // perform appropriate processing to inMsg (omitted)
        outMsg = inOut.createMessage();
        // populate message content (omitted);
        // attach message to exchange
        inOut.setOutMessage(outMsg);
    }
    catch (Exception ex)
    {
        inOut.setError(ex);
        outOnly.setStatus(ExchangeStatus.ERROR);
    }
    channel.send(inOut);
}
```

Listing 9 Example Engine Service Provider

### 5.6.4 Service Provider Metadata

Each service provider must provide a Resolver implementation. This resolver is called by the NMR when it wishes to retrieve metadata about services provided by the component. Such services must have been previously registered (by name) with the NMR.

Service descriptions are supplied as WSDL 2.0 documents. JBI supplements standard WSDL 2.0 by adding a JBI-specified binding type, which MUST be used by service engines when declaring the endpoint for a service provided by that engine. This binding type is defined in the next subsection.

#### 5.6.4.1 JBI Service Engine WSDL Binding

A JBI 1.0 service engine binding is identified by assigning the value

```
“http://jcp.org/jbi/1.0/binding/service+engine”
```

to the type property of a WSDL 2.0 binding component, as defined in [WSDL 2.0].

This binding type serves to identify service engines (i.e., local service providers within the JBI environment). It does not define a mapping of WSDL operations or messages; all abstract WSDL definitions are used without modification.
As mentioned in the architecture overview section, the JBI environment is administered using JMX. This includes the JBI components (the plug-in bindings and engines), which must provide specified management interfaces. In addition, the JBI implementation must provide specified MBeans to facilitate management of the JBI environment-provided infrastructure as well as the JBI components.

### 6.1 Overview

The major administrative tasks that are supported by this specification are:

- Installation of JBI components and shared libraries.
- Deployment of artifacts (opaque to JBI) to installed components.
- Starting and stopping components (binding and engines), as well as JBI implementation components (if applicable).

All of these use cases assume the presence of an administrative tool, which serves to initiate activities using JMX. JBI implementations provide particular management beans (MBeans) that expose management functions to the administrative tool. The MBeans are:

- **InstallationServiceMBean**. The administrative tool uses this management bean to install and uninstall components and shared libraries.
- **InstallerMBean**. JBI implementations provide one of these per component installation started with the InstallationServiceMBean. This is used to manage the installation process, including component-supplied installation configuration via the component’s own MBean.
- **DeploymentServiceMBean**. The administrative tool uses this bean to deploy and undeploy artifacts to components.
- **ComponentLifeCycleMBean**. The administrative tool uses this bean to stop and start components. The component may supply additional life cycle controls, using a component-supplied extension MBean.

In addition, components MAY expose their own MBeans, to provide component-specific management functions beyond the life cycle and installation extensions mentioned above. For example, a component can expose an MBean to manage internal logging levels.

Installation and deployment involve having the administrative tool deliver standardized packages to the appropriate MBeans, listed above. JBI specifies how such packages are structured, including standard XML descriptors of the packages.
6.2 Key Concepts

6.2.1 Component Installation
JBI components (engines and bindings) are installed using standard management mechanisms, defined in this chapter. Standard packaging for such components is also defined here. The objective is to ensure the portability of components, including installation artifacts. This allows component vendors to produce a single package for distribution, rather than multiple variants based on which JBI implementation is targeted. This also benefits users by simplifying component acquisition and management logistics.

6.2.2 Shared-library Installation
Java libraries can be shared between multiple components. These are installed using standard management mechanisms.

6.2.3 Deployment
Many components function as containers, providing (or consuming) services based on artifacts contained by the component. For example, an XSLT engine contains multiple style sheets, each providing a specific transformation service.

The act of introducing new artifacts to a container component is called deployment, chiefly to avoid confusion with component or shared-library installation.

6.2.3.1 Unit Deployment
A single deployment artifact, destined for a single component, is termed a Service Unit, or SU. The contents of a SU are opaque to JBI, but are transparent to the component to which it is being deployed, as well as the design-time tooling that produced the artifacts contained by the SU.

6.2.3.2 Composite Deployment
Often multiple deployments are required to create a new service or consumer application within a JBI environment. To support this directly, JBI provides a composite deployment capability, where deployments meant for different components can be grouped into a Service Assembly, or SA. Such an assembly includes a deployment descriptor, detailing to which component each Service Unit contained in the SA is to be deployed.

6.2.4 Deployment Life Cycle
Each deployment (Service Unit) has a simple life cycle model, provided by JBI-specified interfaces. This model can be extended by the component hosting the deployment to add additional states. This is illustrated in the figure below. This is similar to the component life cycle.

![Base Deployment Life Cycle State Diagram](image)

Figure 31 Base Deployment Life Cycle State Diagram
6.2.5 Component Life Cycle

Each component has a simple life cycle model, managed by JBI-specified management interfaces. This model can be extended by the component to add additional states. The basic life cycle model is shown in the state diagram below.

![Base Component Life Cycle State Diagram](image)

The extended life cycle of a component is the base life cycle, as depicted above, plus the states introduced by component installation. The base component life cycle can be extended by a component by use of an extension MBean. For example, a component may add a "paused" state, which is entered by use of the controls offered by an extension MBean.

6.2.6 Class Loading

A JBI environment MUST provide class loaders for threads used to call component-supplied methods. Such a class loader provides access to classes available in a declared class path, as described in the Java Language Specification [JLS] (java.lang.ClassLoader). There are two types of class loaders used during the extended life cycle of the component:

- **Bootstrap.** During the installation phase of a component’s extended life cycle, the class loader provided for component-supplied installation methods provides access to the bootstrap class-path.

- **Execution.** After installation, the component enters the execution phase of its extended life cycle. The class loader supplied by the JBI implementation for threads calling a component in the execution phase must supply access to classes in the component’s declared class path, plus any shared libraries used by the component. Component class-loading is detailed in chapter 7, "Component Framework."

6.2.7 Use of Java Management Extensions (JMX)

Java Management Extensions (JMX) provides the basis of the interaction between an administrative device (command-line script engine, Ant script, browser UI, or graphical UI) and the JBI environment, including JBI components. JBI implementations MUST support JMX 1.2.1 or newer.

To support JBI-specified administrative functions, JBI components must implement specified JBI (Java) interfaces, while the JBI implementation itself supplies specified JMX MBeans that make use of the component-implemented interfaces. For example, an administrative tool uses an implementation-supplied MBean to perform deployment, which in turn will use a component-supplied ServiceUnitManager interface to deploy a service unit to that component.

Components MAY supply their own MBeans to perform additional management tasks that fall outside the bounds of this specification. The mechanism for this is detailed in “Component Installation” on page 62.
6.3 Packaging

JBI defines standard packaging for both installation of components and deployment of artifacts to those components that function as "containers", as discussed above in Figure 6.2.3. In either case, a descriptor must be supplied to provide installation or deployment information for the JBI implementation. Because of the similarity between these two descriptor types, a single schema is employed for both.

6.3.1 Installation and Deployment Descriptor

The schema used for both installation and deployment descriptor creation is shown below, using RelaxNG compact notation for clarity.

```xml
<default namespace this = "http://java.sun.com/xml/ns/jbi"
start =
  element jbi {
    attribute version { xsd:decimal },
    ( component | service-assembly | shared-library )
  }
component =
  element component {
    attribute type { "service-engine" | "binding-component" },
    identification,
    element component-class-name { attribute description { text }?, text },
    element component-class-path { class-path },
    element bootstrap-class-name { text },
    element bootstrap-class-path { class-path },
    element shared-library { text }*,
    element* -this:* { text }*
  }
shared-library =
  element shared-library {
    identification,
    element shared-library-class-path { class-path }
  }
service-assembly =
  element service-assembly {
    identification,
    service-unit+
  }
service-unit =
  element service-unit {
    identification,
    element target {
      element artifacts-zip { text },
      element component-name { text }
    }
  }
identification =
  element identification {
    element name { text },
    element description { text },
    element* -this:* { text }*
  }
class-path =
  (element path-element { text })+
```

Listing 10 jbi.xml Schema (Relax NG)

The following subsections detail the use of the productions contained in the "jbi" schema.

6.3.1.1 Use of jbi

The top-level production is "jbi". This serves as the starting point for all deployment and installment descriptors generated by this schema.
• **Attribute version** is a decimal value indicating the version of descriptor in use. For this version of JBI, this value MUST be 1.0.

• The **type element** indicates what type of descriptor this is. It must be one of the following elements:
  - **component**. This indicates that an installation package is described by the descriptor.
  - **service-assembly**. This indicates that a deployment package is described by the descriptor.
  - **shared-library**. This indicates that a shared-library installation package is described by the descriptor.

The use of these different descriptor types is detailed below.

### 6.3.1.2 Use of identification
The identification production is used to provide identifying information for components, shared libraries, service units, and service assemblies (which are collectively referred to as *items* here). The identification production contains the following items:

• **name**. The name of the item. This is a human-friendly identifier for the item. Note that each item type imposes other restrictions on name (e.g., uniqueness).

• **description**. A longer (one sentence) description of the item.

• **extension elements** (optional). Additional, vendor-specific identification information. These are not used by JBI, but it is anticipated that tools used to create and manipulate the items identified will require additional identifying information.

### 6.3.1.3 Use of class-path
The class-path production is used to declare a Java class-path for components and shared libraries. Each path element given is an absolute path that is rooted as follows:

- Component **bootstrap-class-path** and **component-class-path**: rooted in the component’s installation package installation root. See chapter "Component Framework" for details.

- Shared library **shared-library-class-path**: rooted in the shared-library’s installation package installation root. See chapter "Component Framework" for details.

### 6.3.1.4 Use of shared-library
The shared-library production is used to describe a shared-library installation package. It has the following elements:

• **identification**. This is used to provide identifying information for the shared-library. The name provided must be unique among all installed shared-libraries in a JBI environment.

• **shared-library-class-path**. This declares the class-path elements (rooted in the shared-library installation package) that are given to the components that use the shared library.

### 6.3.1.5 Use of service-assembly
The service-assembly production is used to describe a service assembly deployment package. This deployment descriptor describes the contained service units, and to which component they are to be deployed.

• **identification**. The name given is a unique identifier (among all service assemblies deployed to the JBI environment) for the composite deployment.

• **service-units**. These elements describe the individual deployments to be made to components. See below for details.

### 6.3.1.6 Use of service-unit
The service-unit production is used to describe a service component within a service-assembly deployment package.
• **identification.** This provides a unique name (unique among service-units deployed to the target component) for the service-unit.

• **target.**
  - **artifacts-zip.** The name of the artifact archive file within the deployment package that will be deployed
  - **component-name.** The name of the component that will have `artifacts-zip` deployed to it.

The deployment process is described in detail in “Deployment Service” on page 64.

**6.3.1.7 Use of component**
The component production is used to describe a component installation package.

• **type** specifies whether the component is a service engine or a binding component. This MAY be used by JBI implementations to distinguish between the two component types.

• **bootstrap-class-path** is the class-path used during installation processing (bootstrapping) of the component. See “Installation Service” on page 61 for details on component installation processing.

• **bootstrap-class-name** is the fully-qualified class name for the component’s Bootstrap implementation. This is used during component installation, and is detailed in “Component Installation” on page 62.

• **component-class-path** is the class-path used after the component is installed. This class-path, along with shared-libraries, constitutes the full class-path available to the component through its default classloader context as provided by JBI. See chapter “Component Framework” for details.

• **component-class-name** is the fully-qualified class name for the component’s Component implementation. This is used during component start-up, and is detailed in “Component Life Cycle” on page 65.

• **Installation extension data.** Optional component-specific installation data can be supplied at the end of the component element. Such extensions MUST be in a non-JBI XML name space. The component can access these data using its `InstallationContext.getInstallationDescriptorExtension()` method.

**6.3.2 Installation Packaging**
Installation packages contain everything that is needed to install a JBI component into a JBI system, or a shared-library for use by such components. This includes an installation descriptor, which provides information for the JBI installation service to process the contents of the installation package appropriately. The actual installation process is described in “Installation Service” on page 61.

The installation package itself is a ZIP archive file, which has contents that are opaque to JBI except for one installation descriptor that MUST be named as follows:

```
/META-INF/jbi.xml
```

The contents of the installation descriptor file MUST conform to either the component installation descriptor schema or the shared-library installation descriptor schema, as described above.

**6.3.3 Deployment Packaging**
Deployment packages contain opaque (to JBI) deployment artifacts, and a deployment descriptor, which provides information for the JBI deployment service to process the contents of the deployment package appropriately. The deployment process is described in “Deployment Service” on page 64.

The deployment package itself consists of a deployment descriptor, and one or more deployment archives, all contained within a ZIP archive file. The file/directory structure of this archive is as follows:

• **/META-INF/jbi.xml.** This contains the deployment descriptor, which MUST conform to the deployment
descriptor schema described above.

- /{artifacts-file-name.zip}. This is the name of one of the deployment archives, as given in one of the deployment descriptor’s target elements.

- /{artifacts-file-name-N.zip}. As many artifact archives as are needed provide all such archives mentioned in the deployment descriptor jbi.xml.

An example deployment descriptor and deployment package directory are given below. The descriptor makes reference to two separate files in the deployment package. Those two files, plus the descriptor itself, are combined into a single ZIP archive, as shown.

```xml
<?xml version="1.0" encoding="utf-8"?>
<jbi version="1.0" xmlns="http://java.sun.com/xml/ns/jbi">
    <service-assembly>
        <identification>
            <name>example-deployment-1</name>
            <description>An example deployment of two service units</description>
        </identification>
        <service-unit>
            <identification>
                <name>SU-1</name>
                <description>service unit 1</description>
            </identification>
            <target>
                <artifacts-zip>su1-artifacts.zip</artifacts-zip>
                <component-name>bpel-engine-1</component-name>
            </target>
        </service-unit>
        <service-unit>
            <identification>
                <name>SU-2</name>
                <description>service unit 2</description>
            </identification>
            <target>
                <artifacts-zip>su2-artifacts.zip</artifacts-zip>
                <component-name>xslt-engine-1</component-name>
            </target>
        </service-unit>
    </service-assembly>
</jbi>
```

Listing 11 Example Deployment Descriptor jbi.xml

```
/META-INF/jbi.xml
/su1-artifacts.zip
/su2-artifacts.zip
```

Listing 12 Example Deployment Package Directory

6.4 Installation Service

The installation service allows the administrative tool to install and uninstall components and shared-libraries to and from the JBI environment.

Installation of components allows the component to supply an optional extension MBean to provided extended configuration capabilities. Installation of shared-libraries does not allow for extended configuration, and thus follows a simpler installation process. These two types of installation are detailed below.

Both types of installation begin with the JBI InstallationServiceMBean, a JMX management bean that JBI implementations MUST provide. Once installed, individual shared libraries and components are referred to
Management

by name. The term "name" in this context always refers to the unique library or component name, supplied in
the installation descriptor’s <identification><name> element value.

6.4.1 Shared Library Installation

Shared library installation packages can be installed and uninstalled, as follows, using the
InstallationServiceMBean:

- **Install.** The administrative tool installs a shared library by invoking the `installSharedLibrary()` method on the installation service MBean, supply a URI indicating where the installation package may be
  found. The installation package MUST have a shared library installation descriptor.

- **Uninstall.** The administrative tool uninstalls a shared library by invoking the
  `uninstallSharedLibrary()` method on the installation service MBean, supplying a shared library
  name to indicate which shared-library to remove. The name must match the installation descriptor-supplied
  library name provided during shared library installation.

6.4.2 Component Installation

Component installation is somewhat more complex than shared-library installation. The admininstrative tool
must create or recall a separate `InstallerMBean` management bean, using the
InstallationServiceMBean. An `InstallerMBean` is used to manage the installation of an
individual component.

The installer MBean is managed from the InstallationServiceMBean as follows:

- **Create installer.** Invoke the `loadNewInstaller()` method, supplying an installation package URL.
  This returns a JMX ObjectName for the newly created installer.

- **Recall installer.** Invoke the `loadInstaller()` method, supplying the component’s name. This MUST
  return the same ObjectName as returned when the component was installed.

- **Remove installer.** Invoke the `unloadInstaller()` method, supplying the component’s name.
  Optionally, this method can be used to remove the component itself.

The `InstallerMBean` is used to manage the installation of the component itself:

- **Install the component.** Using the `install()` method, the administrative tool causes the component to
  be installed as detailed in “Installation Process” on page 62. The method returns the JMX ObjectName of
  the LifeCycleMBean for the newly installed component.

- **Configure the installer.** Prior to installing the component, the installer can optionally be configured, using
  the InstallerConfigurationMBean. The JMX ObjectName for this MBean can be queried by the
  administrative tool using the `getInstallerConfigurationMBean()` method.

- **Check if already installed.** The administrative tool can check if the component associated with the
  InstallerMBean is already installed by using the `isInstalled()` method.

- **Uninstall the component.** The administrative tool can uninstall the component associated with the
  InstallerMBean by invoking the `uninstall()` method.

6.4.2.1 Installation Process

The installation process involves the following two interactions with an administrative tool:

- Loading of the installer using the `InstallerServiceMBean`. This results in the creation of an
  `InstallerMBean` for the component.
• Running the installer, using the InstallerMBean created above.

Optionally, the tool may interact with an installation extension MBean, installed during the above process. These steps are elaborated upon in the following subsections.

6.4.2.1.1 Loading the Installer
When the InstallerServiceMBean.loadNewInstaller() method is called, the JBI implementation must perform the following steps. A failure of any one of the steps will cause the entire installation to be aborted, leaving the JBI environment in the same state it was in before the installation was attempted.

1. **Prepare installation context.** When an installer MBean is created, the installation package ZIP file MUST be expanded into an install root. This root is used throughout the lifetime of the component, and MUST be located in a file system. Class-path elements (from the jbi.xml installation descriptor) refer to items in the install root. The JBI implementation must create a javax.jbi.component.InstallationContext object, initialized to provide the following data:
   - **Installation root**, as a String containing the root’s full directory pathname.
   - **Bootstrap class name**, as given in the installation descriptor.
   - **Bootstrap class path**, as given in the installation descriptor.
   - **Component name**, as given in the installation descriptor.
   - **Installation Descriptor**. A DOM version of the installation descriptor document, allowing access to non-JBI installation data if necessary.

2. **Prepare Bootstrap class loader.** The JBI implementation must create a classloader that provides access to the bootstrap class path declared in the installation package descriptor. This class loader is described in detail in chapter 7, "Component Framework."

3. **Initialize Installer.** Using a thread with the above class loader set as the context class loader, the JBI implementation creates an instance of the bootstrap class, as declared in the installation descriptor. It initializes the bootstrapper, by invoking its init() method, passing it the installation context prepared in step 1. At this point the bootstrap code provided by the component may optionally register an installation extension MBean, using the JMX server provided by the component context.

4. **Create Installer MBean.** After the bootstrap init() call, the JBI implementation MUST create and register an InstallerMBean for the component. Note that the InstallerMBean will need references to the bootstrap loader object, class loader, and component context to complete the installation.

6.4.2.1.2 Customized Installation
The administrative tool, after loading the installer as detailed in section 6.4.2.1.1, receives a JMX object name for the InstallerMBean created for the component. The administrative tool must use the InstallerMBean's getInstallerConfigurationMBean() method to query for the existence of the optional InstallerConfigurationMBean. If this does exist, the customization of the installation must be done, using the InstallerConfigurationMBean BEFORE the installer is run by the administration tool.

**QUESTION -- what about the context class loader for invocations of the installer configuration MBean.**

6.4.2.1.3 Run Installer
The administrative tool, after loading the installer as detailed in section 6.4.2.1.1, receives a JMX object name for the InstallerMBean created for the component. The installation is completed by the administration tool invoking the component’s InstallerMBean install() method. The JBI implementation must perform the following processing steps to complete the installation.

1. Invoke the bootstrap object’s onInstall() method, passing it the component context for the component.
The context class loader for the thread performing this invocation must be set to the bootstrap class loader, as previously described.

2. Optionally, at this point the bootstrap can deregister any previously registered MBeans.

3. Return the JMX object name returned by the bootstrap’s `onInstall()` to the invoker of `install()`.

A successful invocation of `install()` returns the JMX object name of the component’s `ComponentLifeCycleMBean`, which is used to control the running state of the component. See “Component Life Cycle” on page 65.

### 6.5 Deployment Service

The deployment service allows the administrative tool to deploy and undeploy service assemblies to and from the JBI environment.

A deployment consists of a service assembly, as described in the section on “Deployment Packaging” on page 60. The administrative tool requests deployment of an assembly by invoking the `DeploymentServiceMBean deploy()` method, passing it a URL for the resource containing the service assembly package. The package must conform to the requirements detailed in the section on “Deployment Packaging”, else the deployment will fail (by throwing an exception) without attempting to deploy any of the service units contained within the assembly.

When a valid deployment package is received by the "deploy" method, the JBI implementation MUST provide the following processing steps:

1. Validate the deployment package, ensuring that the deployment descriptor conforms to section 6.3.3. Also, ensure that all service-unit artifacts are present in the package, and that all named target components are installed in the JBI system.

2. For each `<service-unit>` in the deployment descriptor:
   1. Unzip the service-unit artifacts archive, and place the result in file storage. (If the service-unit is deployed to more than one component in the same deployment descriptor, the service unit must be copied such that any changes to the deployment contents made by one component to the deployment do not affect the other).
   2. Deploy the named service-unit to the named component. This is detailed in section “Service Unit Deployment Processing” on page 64.

If any iteration (or iterations) of step 2 fails to successfully deploy the named service-unit to the named component, the implementation MUST continue to deploy any remaining service units in the assembly. If all service units of the assembly fail to deploy, a failure exception must be thrown. If at least one service unit deployment succeeds, then any failures must be reported using the status return `String` of the `DeploymentServiceMBeans invoke()` method.

If the service-unit deployment is a duplicate, deployment is skipped, without being considered a deployment failure. Thus, redeployment of an assembly should succeed, but not disturb the existing deployments.

### 6.5.1 Service Unit Deployment Processing

Individual service units are deployed as detailed in this section. As noted above, the service unit artifacts are persisted by the JBI implementation. For each service unit, the JBI implementation MUST perform the following actions for non-duplicate service-unit deployments.

1. The `ServiceUnitManager` interface for the target component is located. Note that this is supplied by
the component, not the JBI implementation.

2. The implementation deploys the service-unit artifacts to the component, using the above interface’s onDeploy method, supplying it the service-unit name, as declared in the deployment descriptor, and the file path name to the root of the persisted artifacts.

At this point, the service unit deployment is ready for its normal life cycle control, as detailed in section 6.7.

6.5.2 Service Unit Undeployment Processing
To Be Supplied.

6.6 Component Life Cycle
Each installed component has a minimal life cycle, as defined here. In addition, this life-cycle can be extended by the component, using extension MBeans. This is illustrated below. The minimal life cycle is modelled by

![Figure 33 Example Component Life Cycle State Diagram](image)

Figure 33 Example Component Life Cycle State Diagram

two JBI MBeans: the InstallerMBean, which controls whether or not the component is installed in the JBI system, and the ComponentLifeCycleMBean, which controls the components more conventional start/stop life cycle. The component can extend the conventional life cycle by providing a life cycle extension MBean, which provides additional life cycle states. Figure 33 illustrates this by adding a single "paused" state to the standard (LifeCycleMBean-supplied) life cycle states. This extra state is accessed through use of the "pause" control supplied by the extension MBean shown.
6.7 Deployment Life Cycle

Each service unit deployment has a life cycle, allowing it to be stopped, started, etc. A state diagram depicting this is shown in Figure 34. The exact interpretation of this life cycle model is component-dependent, but in general components should interpret the Running state of a deployment as starting the provision and consumption of services related to the deployment itself, while the Stopped state means that provision and consumption of services have ceased. The life cycle state of a deployment is referred to as its running state.

Groups of deployments are started in two phases, to assure orderly start-up of the JBI system. On start or restart of any group of deployments, a JBI implementation MUST call init() for all deployments in the group before calling start() on any that are to be started.

MUST retain the running state of all deployments, such that the system can be restarted from a shutdown or crash, and all deployments will be restored to their previous running state.
Component Framework

JBI components (service engines and binding components) both provide interfaces for the use of JBI, and use interfaces provided by JBI. The major messaging and management contracts between JBI and a component have already been discussed in previous chapters. The remaining contracts, known collectively as the component framework, are detailed in this chapter.

JBI implementations provide an environment for components, in addition to the environment provided by the standard Java environment the implementation runs under. The JBI framework provides the following for the use of components:

- **Component installation context.** This is supplied to the component’s bootstrap installation method.
- **Component execution context.** This is supplied to the component during component start. This provides access to JBI-provided resources during normal execution of the component.
- **Class loading.** JBI ensures that the context class loader for threads used to call component-supplied methods is set appropriately at both bootstrap- and execution-time.
- **Error indication.** Exception classes provide indication of JBI-specific errors to the component, and allow the component to distinguish various error types.

In addition, components implement particular interfaces, to provide the following functions to JBI:

- **Bootstrapper.** This provides extended installation and installation configuration functions for the component, beyond those JBI’s management functions already provide.
- **Component.** This interface, implemented by the component, provides JBI will methods to get the component’s life cycle interface, provide service metadata, and get the component’s service unit manager.
- **Component life cycle.** This provides life cycle control of the component. This is utilized by the ComponentLifeCycleMBean, as described in the "Management" chapter.
- **Service unit manager.** This provides deployment and life cycle control for service units deployed to the component. This is an optional interface; components that do not support deployment need not supply a service unit manager.

These component-supplied services are related to management activities; to fully understand how these services are used please see the "Management" chapter.
7.1 Component-Supplied Management Functions

7.1.1 Bootstrapper

Installation of components is handled largely by the JBI management system (see the "Management" chapter for details), which will copy the installation image to the file system of the JBI instance. The process of installing (and the reverse, uninstalling) a component can be customized by the component. This allows further installation processing beyond the simple copying of binary images that the management system provides. For example, creation of database tables may be performed by a component’s bootstrap.

The component MUST provide, as part of its installation package, the fully-qualified name of a class that implements the Bootstrap interface. (The installation package is specified in the "Management" chapter.)

7.1.2 Component Interface

Each component MUST supply an implementation of the Component interface. The implementation class name is supplied to the management system using the installation descriptor. This descriptor is described in the "Management" chapter.

The component interface functions as a single point for the JBI implementation to query the other component-supplied interfaces described in this chapter, as well as metadata for component-supplied services. The component interface provides access to:

- The component’s life cycle implementation, as described below.
- The component’s service unit manager. This is provided by components that support deployment; those that do not must return null. The service unit manager must be a singleton object.

The JBI implementation must obtain an instance of the component’s implementation of this interface AFTER the component is installed, in order to further manage the component. This is accomplished by creating an instance of the javax.jbi.component.Component as follows:

- the class name MUST be the component-class-name as declared in the component’s installation descriptor (as described in the "Management" chapter), and
- it must be created using the component class loader (see “JBI Execution Class-Loader Context Hierarchy” on page 73 for details).

The implementation must create only one instance of the Component implementation.

7.1.3 Component Life Cycle

Each component MUST supply an implementation of the ComponentLifeCycle interface. This is accessed by JBI through the Component interface.
After installation, the component is ready for conventional life cycle control. A component's basic life cycle states, and transitions between states, are depicted below. Note that the "Paused" state illustrates an extended life cycle state that is an extension of the basic JBI life cycle states.

When first installed (as described in the "Management" chapter), the component is placed in the “Installed” state. Before using the component-supplied services beyond the bootstrap, the implementation must obtain and initialize the component’s life cycle implementation. This is accomplished by using the component’s implementation of `javax.jbi.component.Component`, as discussed above in “Component Interface”.

Using the component’s `Component` implementation, the JBI implementation obtains the component’s `ComponentLifeCycle` object.

When the `LifeCycleMBean.start()` method of the component is invoked (see the management chapter), and the SE is in the “Installed” state, the JBI implementation MUST call the component’s life cycle init() method, passing to it a valid component context, followed by a call to the life cycle start() method, as illustrated...
below. Subsequent calls to start(), from the “Stopped” state, MUST NOT cause the JBI framework to call the init() method. Subsequent starts without an intervening stop and shut down behave as shown below.

The JBI implementation MUST retain the Component implementation object created during the initial start (or whenever it is first needed). The retained object is used to access the component’s life cycle implementation.

### 7.1.4 Service Unit Management

Each component can OPTIONALLY supply a service unit manager, in the form of an implementation of the ServiceUnitManager interface. The absence of such a manager object implies that the component does not expect deployments.

The JBI implementation will call the ServiceUnitManager methods during service deployment, as described in the "Management" chapter.
7.2 JBI-Supplied Environment Features

7.2.1 Component Execution Context

The ComponentContext interface serves to allow components to query JBI-supplied aspects of the execution environment the component is running in. This context is given to the component life cycle interface using the init() method. The life cycle interface is described in detail in the "Management" chapter.

The execution context makes the following items available to the engine or binding:

- **Component Name**. The unique name assigned to the component at installation time.
- **Installation Root**. The installation root directory of the component.
- **Workspace Root**. The root directory of a "scratchpad" directory for use by the component. This is the root of a file space that is reserved for use by the component, which it can use for any purpose.
- **MBean Naming Service**. A service that provides methods to generate JMX MBean Object Names for any MBeans provided by the component.
- **MBean Server**. The JMX MBean server used to register all MBeans in the JBI system.
- **Naming Context**. A JNDI naming context for the exclusive use of the component.
- **Normalized Message Router**. This allows the engine or binding to interact with the messaging router, by use of a DeliveryChannel. See the "Normalized Message Router" chapter for details.

7.2.2 Installation (Bootstrap)

The overall installation (bootstrap) process is described in the management chapter. As part of this process, the JBI installation supplies the component’s bootstrapper with an InstallationContext, which provides the component’s bootstrap with environmental information to support installation and uninstallation activities.

7.2.2.1 Installation Context

To allow the service engine bootstrap access to the key parts of the JBI environment that may be needed during installation, the EngineBootstrap methods are passed a reference to a javax.jbi.engine.EngineInstallationContext interface. This context provides access to the following:

- **Component name**. A unique name for the component, as provided in the installation descriptor.
- **Component class name**. The name of the class that implements the component SPI, javax.jbi.component.Component, for this component. This can be read and written by the component’s bootstrap. It is initially set to the component-class-name value supplied in the component’s installation descriptor.
- **Class path elements**. A list of directories or jar files that will comprise the class path of the component when it is executed. The class that implements the Component SPI must be accessible through this class path. This can be read and written by the component bootstrap. This is initially set to the component-class-path values supplied in the component’s installation descriptor.
- **Component context**. A JBI-global component context. It gives access to the items similar to the component context given to a Component's life cycle init() method, but is global (shared by all installation context instances).
- **Installation root directory**. The full path name of the directory into which this component is being installed.
- **Component installation descriptor**. This provides access to the full installation descriptor jbi.xml. This allows the bootstrap to access extended installation information.
Unless otherwise noted, these properties are read-only.
The bootstrap MAY alter the class name and class path element list items in the installation, using the provided
setter methods. The bootstrap MUST ensure that the class name is the name of the class that implements the
javax.jbi.component.Component interface.

7.3 Shared Libraries and Class Loading

JBI implementations MUST provide correct class loaders for threads used to call component-supplied methods,
at both installation- and execution-time. This section defines how such class loaders must behave, in order to
assure a consistent class loading context across all JBI implementations. This is essential for component
portability.

There are two separate phases in the lifetime of a component that require two different class loaders:

- **Installation time.** All methods of the BootStrap interface, as implemented by the component, require a
  bootstrap class-loading context.

- **Execution time.** All methods of the Component, ComponentLifeCycle, and
  ServiceUnitManager interfaces, as implemented by a component, require an execution class-loading
  context.

Both types of class loading contexts provide the following:

- Platform classes
- JBI classes
- Class-path accessible classes:
  - For the bootstrap execution class-loading context, this means the bootstrap-class-path declared in the
    component’s installation descriptor.
  - For the execution class-loading context, this means the class-path supplied by the
    ComponentContext for the component, as described in “Component Execution Context” on page 71

The execution class-loading context adds one more item to be supported: shared libraries, which are described
in the next section.

7.3.1 Shared Libraries

JBI components are loosely-coupled components that, at a minimum, interact only through the JBI
MessageExchange and NormalizedMessage interfaces. In some important use cases, this interaction
needs to be extended through NormalizedMessage properties, where the components have a shared
domain-specific set of classes used to read and write the message properties. For example, an EDI binding
component may wish to share a set of specialized class objects that represent an EDI document with a
specialized service engine that consumes such EDI documents. Given that the binding and engine are installed
separately, there is a need for the two components to share a common library that implements the shared classes,
and must also share the class loader for the library. This requires that the JBI environment provide support to
provide for such shared libraries and class loaders in the execution class-loading context.

To facilitate sharing of class loaders between components in a controllable, predictable fashion that will not
adversely affect other system components, a specific class loading scheme for JBI environments is required,
based on a class loader hierarchy.
7.3.2 JBI Execution Class-Loader Context Hierarchy

The execution class-loader context MUST be as depicted in the figure below. This depicts how individual class loaders are chained together in a JBI implementation to create a standard execution class-loader context.

Each component has a separate JBI-provided component class loader, that is always the class-loader context for threads used to invoke component-supplied methods. Component-created threads should retain this class-loader context if they will participate in interactions with JBI. The component class loader MUST give access to the classes in the component’s class path, as specified by the `component-class-path` declaration in the component installation descriptor. The component class loader MUST search the component class path in the order given in the installation descriptor.

The component class loader is chained to a delegating class loader. As with the component class loader, there is a separate instance of the delegating class loader for each component installed in the JBI environment. The delegating class loader serves to delegate class loading operations to shared class loaders (zero or more, as dictated by configuration). The component’s delegating class loader MUST delegate only to those shared library class loaders that correspond to the shared libraries named in `shared-library` elements of the component’s installation descriptor. The delegating class loader MUST search the shared class loaders in the order given in the installation descriptor.

Shared class loaders are installed separately from components, with the intention that they be used by sets of components to allow sharing of Java objects based on non-standard classes and interfaces. This creates a unique name space for the objects created from classes loaded from the shared class loader. (Objects created from
classes loaded by separate class loaders, even if from identical .class resources, will not be considered to be of
the same type by the Java run-time because the class types came from separate name spaces.)

Each shared class loader is a child of the JBI life cycle class loader. This is used to load the JBI classes.
Optionally, the JBI life cycle class loader can be chained to other class loaders, such as the J2EE platform
shared class loader shown in Figure 38. This is appropriate for environments such as application servers, which
provide other classes beyond the virtual machine's built-in class loader (the so-called JVM bootstrap loader).

7.3.2.1 Installation of Shared Libraries
Shared libraries (shared class loaders) are installed (and uninstalled) using the Management
InstallationServiceMBean. See the "Management" chapter for details.
A shared library must be installed before a component can be installed that makes use of the shared library,
otherwise the installation will fail. See the "Management" chapter, sections “Shared Library Installation” and
"Component Installation" for details.

7.3.2.2 Component Viewpoint
There is no explicit API exposed by JBI that bindings and engines can use to examine or manipulate the JBI
class loading scheme. The JBI environment guarantees that that the context class loader (returned by the
java.lang.Thread.getContextClassLoader() method, will be the component class loader, as defined above, unless
the component chooses to set the context class loader. In this event, the component MUST set the JBI-provided
context class loader as the parent of the component-provided class loader for all threads that interact with JBI.
7.3.3 Bootstrap Class-loader Context

The bootstrap class-loader context MUST be as depicted in the figure below. This depicts how individual class loaders are linked together in a JBI implementation to create a standard bootstrap class-loader context.

Each component has a separate JBI-provided bootstrap class loader, that is always the class-loader context for threads used to invoke bootstrap-related, component-supplied methods. The bootstrap class loader MUST give access to the classes in the component’s bootstrap class path, as specified by the bootstrap-class-path declaration in the component installation descriptor. The bootstrap class loader MUST search the bootstrap class path in the order given in the installation descriptor.

The bootstrap class-loader is a child of the JBI class loader. This is used to load the JBI classes.

Optionally, the JBI class loader can be chained to other class loaders, such as the J2EE platform shared class loader shown in Figure 39. This is appropriate for environments such as application servers, which provide other classes beyond the virtual machine's built-in class loader (the so-called JVM bootstrap loader).

Figure 39 Bootstrap class-loader context hierarchy

7.4 Error Indication

Errors are indicated by use of Java Exceptions. All JBI exceptions are based on javax.jbi.JBIException, to allow component and administrative code to distinguish JBI-related error conditions. The JBI exception types are:

- DeploymentException. This is thrown by the DeploymentServiceBMBean to indicate, to an administrative tool, that a deployment has failed. See "Management" for details.
- MessagingException. This is thrown by the NMR, indicating a variety of error conditions when creating, manipulating, sending and receiving message exchanges and normalized messages. See "Normalized Message Routing" for details.
7.4.1 JBIException

All exception types defined by this specification are derived from this base exception class. This allows component code to easily catch all possible JBI-related exceptions when needed.
CHAPTER 8

Package
javax.jbi

Class Summary

Exceptions

<table>
<thead>
<tr>
<th>JBIException</th>
<th>JBIException is the top-level exception thrown by all JBI system components.</th>
</tr>
</thead>
</table>
javax.jbi

JBIException

Declaration
public class JBIException extends java.lang.Exception

java.lang.Object
    +--java.lang.Throwable
        +--java.lang.Exception
            +--javax.jbi.JBIException

All Implemented Interfaces: java.io.Serializable

Direct Known Subclasses: javax.jbi.management.DeploymentException, javax.jbi.messaging.MessagingException

Description
JBIException is the top-level exception thrown by all JBI system components.

Member Summary

Constructors

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Method Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>JBIException(java.lang.String message)</td>
<td></td>
</tr>
<tr>
<td>JBIException(java.lang.String message, java.lang.Throwable cause)</td>
<td></td>
</tr>
<tr>
<td>JBIException(java.lang.Throwable cause)</td>
<td></td>
</tr>
</tbody>
</table>

Inherited Member Summary

Methods inherited from class Object

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), wait(long, int), wait(long, int), wait(long, int)</td>
<td></td>
</tr>
</tbody>
</table>

Methods inherited from class Throwable

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fillInStackTrace(), getCause(), getLocalizedMessage(), getMessage(), getStackTrace(), initCause(Throwable), printStackTrace(PrintWriter), printStackTrace(PrintWriter), printStackTrace(PrintWriter), setStackTrace(StackTraceElement[]), toString()</td>
<td></td>
</tr>
</tbody>
</table>
Constructors

JBIException(String)

```java
public JBIException(java.lang.String message)

Creates a new instance of JBIException with an exception message.

Parameters:
  message - String describing this exception.
```

JBIException(String, Throwable)

```java
public JBIException(java.lang.String message, java.lang.Throwable cause)

Creates a new instance of JBIException with the specified message and cause.

Parameters:
  message - String describing this exception.
  cause - Throwable which represents an underlying problem (or null).
```

JBIException(Throwable)

```java
public JBIException(java.lang.Throwable cause)

Creates a new instance of JBIException with the specified cause.

Parameters:
  cause - Throwable which represents an underlying problem (or null).
```
JBIException
JBIException(Throw)
## Class Summary

### Interfaces

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bootstrap</td>
<td>Implemented by a JBI Component to provide any special processing required at install/uninstall time.</td>
</tr>
<tr>
<td>Component</td>
<td>This interface, implemented by component implementations, allows the JBI implementation to query the component for particular information.</td>
</tr>
<tr>
<td>ComponentContext</td>
<td>This provides access to data needed by all JBI components running in the JBI environment.</td>
</tr>
<tr>
<td>ComponentLifeCycle</td>
<td>Implemented by a JBI component to provide initialization, start, stop, and shut down processing.</td>
</tr>
<tr>
<td>InstallationContext</td>
<td>This context contains information necessary for a JBI component to perform its installation/uninstallation processing.</td>
</tr>
<tr>
<td>ServiceUnitManager</td>
<td>This interface defines component-supplied methods for managing service unit deployments.</td>
</tr>
</tbody>
</table>

### Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ComponentContext.EnvironmentType</td>
<td>Enumeration class for types of JBI execution environments.</td>
</tr>
</tbody>
</table>
bootstrap

javax.jbi.component

Bootstrap

Declaration

public interface Bootstrap

Description

Implemented by a JBI Component to provide any special processing required at install/uninstall time. Things such as creation/deletion of directories, files, and database tables can be done by the onInstall() and onUninstall() methods, respectively. Also allows the component to terminate the installation or uninstallation in the event of an error.

Member Summary

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>getExtensionMBeanName()</td>
</tr>
<tr>
<td>init(InstallationContext installContext)</td>
</tr>
</tbody>
</table>

Methods

getExtensionMBeanName()

public javax.management.ObjectName getExtensionMBeanName()

Obtains the optional installer configuration MBean ObjectName. If none is provided by this component, this method must return null.

Returns: ObjectName which represents the MBean registered by the init(InstallationContext) method. If none was registered, returns null.

init(InstallationContext)

public void init(javax.jbi.component.InstallationContext installContext)
throws JBIException

Initializes the installation environment for a component. This method is expected to save any information from the installation context that may be needed by other methods.

If the component needs to register an optional installer configuration MBean, it MUST do so during the call to this method.

Parameters:
installContext - the context containing information from the install command and from the component installation ZIP file.
Throws:
   javax.jbi.JBIException - when there is an error requiring that the installation be terminated.

onInstall()

   public void onInstall()
       throws JBIException

   Called at the beginning of installation of a component to perform any special installation tasks required by the component.

   Throws:
   javax.jbi.JBIException - when there is an error requiring that the installation be terminated.

onUninstall()

   public void onUninstall()
       throws JBIException

   Called at the beginning of uninstallation of a component to perform any special uninstallation tasks required by the component.

   Throws:
   javax.jbi.JBIException - when there is an error requiring that the uninstallation be terminated.
Component javax.jbi.component

getchLifeCycle() javajbi.component

Component

Declaration
public interface Component

Description
This interface, implemented by component implementations, allows the JBI implementation to query the component for particular information.

<table>
<thead>
<tr>
<th>Member Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods</td>
</tr>
<tr>
<td>ComponentLifeCycle getLifeCycle()</td>
</tr>
<tr>
<td>ServiceUnitManager getServiceUnitManager()</td>
</tr>
<tr>
<td>javax.jbi.servicedesc. Descriptor resolveReference(javax.jbi.servicedesc. EndpointReference ref)</td>
</tr>
</tbody>
</table>

Methods

getLifeCycle()

```java
public javax.jbi.component.ComponentLifeCycle getLifeCycle()
```

Get the life cycle control interface for this component.

Returns: The life cycle control interface for this component.

getServiceUnitManager()

```java
public javax.jbi.component.ServiceUnitManager getServiceUnitManager()
```

Get the ServiceUnitManager for this component. If this component does not support deployments, it must return null.

Returns: The ServiceUnitManager for this component, null if none.

resolveReference(EndpointReference)

```java
public javax.jbi.servicedesc. Descriptor resolveReference(javax.jbi.servicedesc. EndpointReference ref)
```

Resolves descriptor details for the specified reference, which is for a service provided by this component.

Parameters:
    ref - entity reference

Returns: description for specified reference.
javax.jbi.component

ComponentContext

Declaration

public interface ComponentContext

Description

This provides access to data needed by all JBI components running in the JBI environment.

<table>
<thead>
<tr>
<th>Nested Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>static class</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>void activateEndpoint(javax.xml.namespace.QName serviceName, java.lang.String endpointName)</td>
</tr>
<tr>
<td>java.lang.String getComponentName()</td>
</tr>
<tr>
<td>getDeliveryChannel()</td>
</tr>
<tr>
<td>java.lang.String getInstallRoot()</td>
</tr>
<tr>
<td>getManagementMessageFactory()</td>
</tr>
<tr>
<td>javax.management.MBeanServer getMBeanServer()</td>
</tr>
<tr>
<td>javax.naming.InitialContext getNamingContext()</td>
</tr>
<tr>
<td>getTransactionManager()</td>
</tr>
<tr>
<td>java.lang.String getWorkspaceRoot()</td>
</tr>
</tbody>
</table>
Methods

activateEndpoint(QName, String)

```java
public javax.jbi.servicedesc.EndpointReference activateEndpoint(javax.xml.namespace.QName serviceName,
java.lang.String endpointName)
throws JBIException
```

Registers the named endpoint with the NMR.

**Parameters:**
- `serviceName` - qualified name of the service the endpoint exposes
- `endpointName` - the name of the endpoint to be registered

**Returns:** a reference to the registered endpoint

**Threws:**
- `javax.jbi.JBIException` - if the endpoint cannot be activated

availableEndpoints(QName)

```java
public javax.jbi.servicedesc.EndpointReference[] availableEndpoints(javax.xml.namespace.QName serviceName)
throws JBIException
```

Queries the NMR for endpoints registered against the specified service.

**Parameters:**
- `serviceName` - qualified name of service

**Returns:** EndpointReference[] list of available endpoints for the specified service; potentially zero-length.

**Threws:**
- `javax.jbi.JBIException` - invalid service reference

deactivateEndpoint(EndpointReference)

```java
public void deactivateEndpoint(javax.jbi.servicedesc.EndpointReference endpoint)
throws JBIException
```

Deregisters the given endpoint with the NMR.

**Parameters:**
- `endpoint` - reference to the endpoint to be deactivated

**Threws:**
- `javax.jbi.JBIException` - if the endpoint cannot be deactivated

getComponentName()

```java
public java.lang.String getComponentName()
```

Get the unique component name of this component.

**Returns:** the component name.
getDeliveryChannel()

```java
public javax.jbi.messaging.DeliveryChannel getDeliveryChannel()
    throws MessagingException

Get a channel for this component to use to communicate with the Normalized Message Router.

Returns: the delivery channel for this component.

Throws:
    javax.jbi.messaging.MessagingException - if the channel cannot be activated.
```

getAddressEndpointDescriptor(EndpointReference)

```java
public javax.jbi.servicedesc.Descriptor getEndpointDescriptor(
    javax.jbi.servicedesc.EndpointReference endpoint)
    throws JBIException

Retrieve metadata for the specified endpoint.

Parameters:
    endpoint - endpoint reference

Returns: metadata describing endpoint, or null if metadata is unavailable.

Throws:
    javax.jbi.JBIException - invalid endpoint reference
```

getAddressEnvironmentType()

```java
public javax.jbi.component.ComponentContext.EnvironmentType
    getEnvironmentType()

Get the type of JBI execution environment this JBI system supports: J2SE, or J2EE.

Returns: the type of execution environment this implementation supports.
```

getAddressInstallRoot()

```java
public java.lang.String getInstallRoot()

Get the installation root directory path for this component.

This method MUST return the file path formatted for the underlying platform.

Returns: The installation root directory path, in platform-specific form.
```

getAddressManagementMessageFactory()

```java
public javax.jbi.management.ManagementMessageFactory
    getManagementMessageFactory()

Get the management message factory which enables JBI components to construct status and exception messages.

```

getAddressMBeanNames()

```java
public javax.jbi.management.MBeanNames
    getMBeanNames()

Get a handle to the MBeanNames service for use in creating MBean names.

Returns: Handle to the MBeanNames service.
ComponentContext javax.jbi.component
getMBeanServer()

getMBeanServer() 1

   public javax.management.MBeanServer getMBeanServer() 2

   Get the JMX MBean server used to register all MBeans in the JBI framework. 3
   
   Returns: The MBean server handle. 4

getNamingContext() 5

   public javax.naming.InitialContext getNamingContext() 6

   Get the JNDI naming context for this implementation. This context is a standard JNDI InitialContext but its 7
   content will vary based on the environment in which the JBI implementation is running. 8
   
   Returns: The JNDI naming context. 9

getTransactionManager() 10

   public javax.transaction.TransactionManager getTransactionManager() 11

   Get the TransactionManager for this implementation. The instance returned is an implementation of the 12
   standard JTS interface. If none is available, returns null. 13
   
   Returns: A TransactionManager instance. 14

getWorkspaceRoot() 15

   public java.lang.String getWorkspaceRoot() 16

   Get the root directory path for this component’s private workspace. 17
   
   This method MUST return the file path formatted for the underlying platform. 18
   
   Returns: The private workspace root path, in platform-specific form. 19
javax.jbi.component

ComponentContext.EnvironmentType

Declaration
public static final class ComponentContext.EnvironmentType

declared in java.lang.Object

+---javax.jbi.component.ComponentContext.EnvironmentType

Enclosing Class: ComponentContext

Description
Enumeration class for types of JBI execution environments.

Member Summary

<table>
<thead>
<tr>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>static J2EE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ComponentContext.EnvironmentType</th>
</tr>
</thead>
<tbody>
<tr>
<td>static J2SE</td>
</tr>
</tbody>
</table>

Inherited Member Summary

Methods inherited from class java.lang.Object
clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), toString(), wait(long, int), wait(long, int), wait(long, int)

<table>
<thead>
<tr>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>J2EE</td>
</tr>
<tr>
<td>public static final javax.jbi.component.ComponentContext.EnvironmentType J2EE</td>
</tr>
</tbody>
</table>

| J2SE |
| public static final javax.jbi.component.ComponentContext.EnvironmentType J2SE |
Declaration
public interface ComponentLifeCycle

Description
Implemented by a JBI component to provide initialization, start, stop, and shut down processing. These methods comprise the life cycle contract between the JBI framework and the component. The life cycle of a component begins with a call to the init() method on an instance of the component’s implementation of this interface, and ends with the first call to the shutDown() method on that instance. Between these two calls, there can be any number of stop() and start() calls.

Member Summary

Methods
javax.management.ObjectName getExtensionMBeanName()
java.lang.Void init(ComponentContext context)
java.lang.Void shutDown()
java.lang.Void start()
java.lang.Void stop()

getExtensionMBeanName()

public javax.management.ObjectName getExtensionMBeanName()

Get the JMX ObjectName for any additional MBean for this component. If there is none, return null.

Returns: The JMX object name of the additional MBean or null if there is no additional MBean.

init(ComponentContext)

public void init(javax.jbi.component.ComponentContext context)
throws JBIException

Initialize the component. This performs initialization required by the component but does not make it ready to process messages. This method is called once for each life cycle of the component.

If the component needs to register an additional MBean to extend its life cycle, or provide other component management tasks, it should be registered during this call.

Parameters:
context - the component’s context, providing access to component data provided by the JBI environment
javax.jbi.component

ComponentLifeCycle

shutDown()

public void shutDown()
    throws JBIException

Shut down the component. This performs cleanup before the component is terminated. Once this method
has been called, init(ComponentContext) must be called before the component can be started
again with a call to start().

Throws:
    javax.jbi.JBIException - if the component is unable to shut down

start()

public void start()
    throws JBIException

Start the component. This makes the component ready to process messages. This method is called after
init(ComponentContext), and when the component is being restarted after a previous call to
shutDown(). If stop() was called previously but shutDown() was not, start() can be
called again without another call to init(ComponentContext).

Throws:
    javax.jbi.JBIException - if the component is unable to start

stop()

public void stop()
    throws JBIException

Stop the component. This makes the component stop accepting messages for processing. After a call to this
method, start() can be called again without first calling init(ComponentContext).

Throws:
    javax.jbi.JBIException - if the component is unable to stop
InstallationContext

glassPathElements()

javax.jbi.component
InstallationContext

Declaration
public interface InstallationContext

Description
This context contains information necessary for a JBI component to perform its installation/uninstallation processing. This is provided to the init() method of the component Bootstrap interface.

Member Summary

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getClassPathElements()</td>
<td>Get a list of elements that comprise the class path for this component. Each element represents either a directory (containing class files) or a jar file. All elements are reachable from the install root. Returns: a list of String objects, each of which contains a class path elements. The list may not be empty.</td>
</tr>
<tr>
<td>getComponentClassName()</td>
<td>Get the name of the class that implements the Component interface for this component. Returns: the Component implementation class name</td>
</tr>
<tr>
<td>getComponentName()</td>
<td>Get the unique name assigned to this component. Returns: the unique component name</td>
</tr>
</tbody>
</table>

Methods

glassPathElements()

public java.util.List getClassPathElements()

Get a list of elements that comprise the class path for this component. Each element represents either a directory (containing class files) or a jar file. All elements are reachable from the install root.

Returns: a list of String objects, each of which contains a class path elements. The list may not be empty.

glassPathElements

public java.lang.String getComponentClassName()

Get the name of the class that implements the Component interface for this component.

Returns: the Component implementation class name

glassPathElements

public java.lang.String getComponentName()

Get the unique name assigned to this component.

Returns: the unique component name
getContext()

public javax.jbi.component.ComponentContext getContext()

Get the JBI context for this component.

Returns: the JBI context for this component.

getInstallationDescriptorExtension()

public org.w3c.dom.DocumentFragment getInstallationDescriptorExtension()

Return a DOM document fragment representing the installation descriptor (jbi.xml) extension data for the
component, if any.

The Installation Descriptor Extension data is located in the <identification

Returns: a DOM document containing the installation descriptor (jbi.xml) content, or null if non
is
present in the descriptor.

getInstallRoot()

public java.lang.String getInstallRoot()

Get the installation root directory full path name for this component.

Returns: the installation root directory name.

setClassPathElements(List)

public void setClassPathElements(java.util.List classPathElements)

Set the list of elements that comprise the class path for this component. Each element represents either a
directory (containing class files) or a jar file. Elements are reached from the install root.

Parameters:

classPathElements - a list of String objects, each of which contains a class path elements. The
list may not be empty.
javas.jbi.component
ServiceUnitManager

Declaration
public interface ServiceUnitManager

Description
This interface defines component-supplied methods for managing service unit deployments. It is implemented by the component. The JBI implementation queries the component for the implementation of this object using the ComponentLifeCycleMBean.

Member Summary

Methods
java.lang.String deploy(java.lang.String serviceUnitName, java.lang.String serviceUnitRootPath)
void init(java.lang.String serviceUnitName, java.lang.String serviceUnitRootPath)
void shutDown(java.lang.String serviceUnitName)
void start(java.lang.String serviceUnitName)
void stop(java.lang.String serviceUnitName)
java.lang.String undeploy(java.lang.String serviceUnitName, java.lang.String serviceUnitRootPath)

Methods

deploy(String, String)

public java.lang.String deploy(java.lang.String serviceUnitName, java.lang.String serviceUnitRootPath)
throws DeploymentException

Deploy a Service Unit to the component. This is called by the JBI implementation in order to deploy the given artifact to the implementing component.

Parameters:
- serviceUnitName - name of the service unit being deployed
- serviceUnitRootPath - path of the service unit artifact root

Returns: deployment status message

Throws:
- javax.jbi.management.DeploymentException - if deployment operation is unsuccessful.

init(String, String)

public void init(java.lang.String serviceUnitName, java.lang.String serviceUnitRootPath)
throws DeploymentException
Initialize the given deployed service unit. This is the first phase of a two-phase start, where the component must prepare to receive service requests related to the deployment (if any).

The serviceUnitRootPath parameter is provided to facilitate restart of the component. This allows simply components to rely entirely on JBI’s ability to persist deployment information, avoiding the need to build persistance into the component.

Parameters:

- serviceUnitName - name of the service unit being initialized
- serviceUnitRootPath - path of the service unit artifact root

Throws:

- javax.jbi.management.DeploymentException - if the service unit is not deployed, or if it is in an incorrect state.

shutDown(String)

```java
public void shutDown(java.lang.String serviceUnitName)
    throws DeploymentException
```

Shut down the deployed service unit. This causes the component to return to the state it was in after deploy(String, String) and before init(String, String).

Parameters:

- serviceUnitName - name of the service unit being shut down

Throws:

- javax.jbi.management.DeploymentException - if the service unit is not deployed, or if it is in an incorrect state.

start(String)

```java
public void start(java.lang.String serviceUnitName)
    throws DeploymentException
```

Start the deployed service unit. This is the second phase of a two-phase start, where the component can now initiate service requests related to the deployment.

Parameters:

- serviceUnitName - name of the service unit being started

Throws:

- javax.jbi.management.DeploymentException - if the service unit is not deployed, or if it is in an incorrect state.

stop(String)

```java
public void stop(java.lang.String serviceUnitName)
    throws DeploymentException
```

Stop the deployed service unit. This causes the component to cease generating service requests related to the given service unit. This returns the service unit to a state equivalent to after init(String, String) was called.

Parameters:

- serviceUnitName - name of the service unit being stopped
Throws:

   javax.jbi.management.DeploymentException - if the service unit is not deployed, or if it is in an incorrect state.

undeploy(String, String)

   public java.lang.String undeploy(java.lang.String serviceUnitName,
   java.lang.String serviceUnitRootPath)
   throws DeploymentException

    Undeploy a Service Unit from the component. The service unit must be shutdown to undeploy it.

    Parameters:
    serviceUnitName - name of the service unit being deployed
    serviceUnitRootPath - the root file directly path for the contents of the deployment.

    Returns:  deployment status message

    Throws:
    javax.jbi.management.DeploymentException - if undeployment operation is unsuccessful.
# Chapter 10

## Package

`javax.jbi.management`

## Class Summary

<table>
<thead>
<tr>
<th>Interfaces</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdminServiceMBean</td>
<td>AdminServiceMBean defines the interface that must be implemented by the AdminService in a JBI Framework.</td>
</tr>
<tr>
<td>DeploymentServiceMBean</td>
<td>DeploymentServiceMBean defines the interface that must be implemented and exposed as a JMX MBean by JBI implementations.</td>
</tr>
<tr>
<td>InstallationServiceMBean</td>
<td>The InstallationService is responsible for the installation and uninstallation of components (engines and bindings) and shared libraries.</td>
</tr>
<tr>
<td>InstallerMBean</td>
<td>InstallerMBean defines standard installation and uninstallation controls for components.</td>
</tr>
<tr>
<td>LifeCycleMBean</td>
<td>LifeCycleMBean is a base interface that defines standard life cycle controls for JBI Framework services and JBI Components (bindings and engines).</td>
</tr>
<tr>
<td>ManagementMessageBuilder</td>
<td>ManagementMessageBuilder defines the interface that must be invoked by components in order to construct exception messages before throwing them back to the JBI Framework, and task status messages for logging.</td>
</tr>
<tr>
<td>ManagementMessageFactory</td>
<td>This provides access to message builder needed by all components running running in the JBI framework.</td>
</tr>
<tr>
<td>MBeanNames</td>
<td>Provide methods to create standard ObjectNames for JBI Framework engine, binding, and framework system services.</td>
</tr>
</tbody>
</table>

## Exceptions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeploymentException</td>
<td>DeploymentException is an exception thrown by the Deployment Service.</td>
</tr>
</tbody>
</table>
AdminServiceMBean

javax.jbi.management

AdminServiceMBean

Declaration

public interface AdminServiceMBean

Description

AdminServiceMBean defines the interface that must be implemented by the AdminService in a JBI Framework. The AdminService is responsible for creating other system services at startup, starting and stopping the system, authenticating administrators, configuring the routing of notifications and alerts to email addresses, setting security for vendor components, etc.

Member Summary

<table>
<thead>
<tr>
<th>Methods</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>getBindingComponents()</td>
<td></td>
</tr>
<tr>
<td>public javax.management.ObjectName[] getBindingComponents()</td>
<td></td>
</tr>
<tr>
<td>Get a list of all binding components currently installed.</td>
<td></td>
</tr>
<tr>
<td>Returns: array of JMX object names of all installed BCs.</td>
<td></td>
</tr>
<tr>
<td>GetComponentByName(java.lang.String name)</td>
<td></td>
</tr>
<tr>
<td>public javax.management.ObjectName GetComponentByName(java.lang.String name)</td>
<td></td>
</tr>
<tr>
<td>Lookup a JBI Installable Component by its unique name.</td>
<td></td>
</tr>
<tr>
<td>getEngineComponents()</td>
<td></td>
</tr>
<tr>
<td>public javax.management.ObjectName[] getEngineComponents()</td>
<td></td>
</tr>
<tr>
<td>getSystemService(java.lang.String serviceName)</td>
<td></td>
</tr>
<tr>
<td>public javax.management.ObjectName getSystemService(java.lang.String serviceName)</td>
<td></td>
</tr>
<tr>
<td>Set a JBI service.</td>
<td></td>
</tr>
<tr>
<td>getSystemInfo()</td>
<td></td>
</tr>
<tr>
<td>public java.lang.String getSystemInfo()</td>
<td></td>
</tr>
<tr>
<td>Get system information.</td>
<td></td>
</tr>
<tr>
<td>getSystemServices()</td>
<td></td>
</tr>
<tr>
<td>public java.management.ObjectName[] getSystemServices()</td>
<td></td>
</tr>
<tr>
<td>List all JBI system services.</td>
<td></td>
</tr>
<tr>
<td>isBinding(java.lang.String componentName)</td>
<td></td>
</tr>
<tr>
<td>public boolean isBinding(java.lang.String componentName)</td>
<td></td>
</tr>
<tr>
<td>Check if the given component is a binding component.</td>
<td></td>
</tr>
<tr>
<td>isEngine(java.lang.String componentName)</td>
<td></td>
</tr>
<tr>
<td>public boolean isEngine(java.lang.String componentName)</td>
<td></td>
</tr>
<tr>
<td>Check if the given component is a JBI engine.</td>
<td></td>
</tr>
</tbody>
</table>
getEngineComponents()

public javax.management.ObjectName[] getEngineComponents()

Get a list of all engines currently installed.

Returns: array of JMX object names of all installed service engines

getSystemInfo()

public java.lang.String getSystemInfo()

Return current version and other info about this JBI Framework.

Returns: info String

getSystemService(String)

public javax.management.ObjectName getSystemService(java.lang.String serviceName)

Lookup a system service by name.

Parameters:

serviceName - is the name of the system service

Returns: the JMX object name of the service or null

getSystemServices()

public javax.management.ObjectName[] getSystemServices()

Looks up all JBI Framework System Services currently installed.

Returns: array of JMX object names of system services

isBinding(String)

public boolean isBinding(java.lang.String componentName)

Check if a given JBI Component is a Binding Component.

Parameters:

componentName - the unique name of the component

Returns: true if the component is a binding

isEngine(String)

public boolean isEngine(java.lang.String componentName)

Check if a given JBI Component is a service engine.

Parameters:

componentName - the unique name of the component

Returns: true if the component is a service engine
ComponentLifeCycleMBean

javax.jbi.management

ComponentLifeCycleMBean

Declaration

public interface ComponentLifeCycleMBean extends LifeCycleMBean

All Superinterfaces: LifeCycleMBean

Description

ComponentLifeCycleMBean defines standard lifecycle controls for JBI Installable Components, and adds getters for optional deployment and extension MBeans.

Member Summary

Methods

javax.management.ObjectName getExtensionMBeanName() throws JBIException

Inherited Member Summary

Methods inherited from interface LifeCycleMBean

shutDown(), start(), stop()

Methods

getExtensionMBeanName()

public javax.management.ObjectName getExtensionMBeanName() throws JBIException

Get the JMX ObjectName for the life cycle extension MBean for this component. If there is none, return null.

Returns: ObjectName the JMX object name of the additional life cycle MBean or null if there is no additional MBean.

Throws:

javax.jbi.JBIException - if there is a failure getting component information for the component to which this life cycle applies.
javax.jbi.management

DeploymentException

Declaration
public class DeploymentException extends javax.jbi.JBIException

java.lang.Object
   |--- java.lang.Throwable
      |   |--- java.lang.Exception
         |   |--- javax.jbi.JBIException
            |--- javax.jbi.management.DeploymentException

All Implemented Interfaces: java.io.Serializable

Description
DeploymentException is an exception thrown by the Deployment Service.

Member Summary

Constructors
DeploymentException(java.lang.String message)
DeploymentException(java.lang.String message, java.lang.Throwable cause)
DeploymentException(java.lang.Throwable cause)

Inherited Member Summary

Methods inherited from class Object
clon(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(), wait(long, int), wait(long, int), wait(long, int)

Methods inherited from class Throwable
fillInStackTrace(), getCause(), getLocalizedMessage(), getMessage(), getStackTrace(), initCause(Throwable), printStackTrace(PrintWriter), printStackTrace(PrintWriter), printStackTrace(PrintWriter), setStackTrace(StackTraceElement[]), toString()
Constructors

DeploymentException(String)

```java
public DeploymentException(java.lang.String message)

Creates a new instance of DeploymentException with an exception message.

Parameters:
  message - String describing this exception.
```

DeploymentException(String, Throwable)

```java
public DeploymentException(java.lang.String message, java.lang.Throwable cause)

Creates a new instance of DeploymentException with the specified message and cause.

Parameters:
  message - String describing this exception.
  cause - Throwable which represents an underlying problem (or null).
```

DeploymentException(Throwable)

```java
public DeploymentException(java.lang.Throwable cause)

Creates a new instance of DeploymentException with the specified cause.

Parameters:
  cause - Throwable which represents an underlying problem (or null).
```
javax.jbi.management

DeploymentServiceMBean

Declaration

public interface DeploymentServiceMBean

Description

DeploymentServiceMBean defines the interface that must be implemented and exposed as a JMX MBean by JBI implementations. This MBean allows the administrative tool to manage service assembly deployments.

Member Summary

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
</tr>
<tr>
<td>java.lang.String</td>
</tr>
<tr>
<td>java.lang.String</td>
</tr>
<tr>
<td>java.lang.String[]</td>
</tr>
<tr>
<td>java.lang.String[]</td>
</tr>
<tr>
<td>java.lang.String[]</td>
</tr>
<tr>
<td>java.lang.String[]</td>
</tr>
<tr>
<td>boolean</td>
</tr>
<tr>
<td>java.lang.String</td>
</tr>
</tbody>
</table>

Methods

canDeployToComponent(String, String)

public boolean canDeployToComponent(java.lang.String componentName, java.lang.String serviceUnitName)

Indicates if the named Service Unit can be deployed to the named component.

Parameters:

componentName - name of the component

serviceUnitName - name of the service unit being queried

Returns: true if the service unit can be deployed

deploy(String)

public java.lang.String deploy(java.lang.String serviceAssemblyZipUrl)
throws Exception

Deploys the given Service Assembly to the JBI environment
DeploymentServiceMBean

getAssemblyUnitDescriptor(String)

Parameters:
  serviceAssemblyZipUrl - String containing the location URL of the Service Assembly ZIP file

Returns: Result/Status of the current deployment

Throws:
  java.lang.Exception - if complete deployment fails

getAssemblyUnitDescriptor(String)

public java.lang.String getAssemblyUnitDescriptor(java.lang.String serviceAssemblyName)
  throws Exception

Returns the descriptor of the Service Assembly that was deployed to the JBI environment.

Parameters:
  serviceAssemblyName - name of the SA to be queried

Returns: descriptor of the Assembly Unit

Throws:
  java.lang.Exception - if a processing error occurs

componentsForDeployedAssemblyUnit(String)

public java.lang.String[] getComponentsForDeployedAssemblyUnit(java.lang.String serviceAssemblyName)
  throws Exception

Returns a list of components to which Service Units are targeted for in a Service Assembly.

Parameters:
  serviceAssemblyName - the service assembly to be queried

Returns: list of component names

Throws:
  java.lang.Exception - if a processing error occurs

gDeployedAssemblyUnitsForComponent(String)

public java.lang.String[] getDeployedAssemblyUnitsForComponent(java.lang.String componentName)
  throws Exception

Returns a list of Service Assemblies that contain Service Units for the given component.

Parameters:
  componentName - name of the component to query

Returns: list of Service Assembly names deployed to the named component

Throws:
  java.lang.Exception - if a processing error occurs

gDeployedServiceAssemblies()

public java.lang.String[] getDeployedServiceAssemblies()
  throws Exception

Returns a list of Service Assemblies deployed to the JBI environment.
javax.jbi.management

DeploymentServiceMBean

getDeployedServiceUnitList(String)

public java.lang.String[] getDeployedServiceUnitList(java.lang.String componentName) throws Exception

Returns a list of service unit names that are currently deployed to the named component.

Parameters:
- componentName - the name of the component to query

Returns: List of service unit names deployed in the named component

getDeployedServiceUnitList(String)

Throws: java.lang.Exception - if a processing error occurs

isDeployedServiceUnit(String, String)

public boolean isDeployedServiceUnit(java.lang.String componentName, java.lang.String serviceUnitName) throws Exception

Returns a boolean value indicating whether the Service Unit is currently deployed.

Parameters:
- componentName - name of the component to query
- serviceUnitName - name of the service unit being sought

Returns: true if the named service unit is currently deployed to the named component

Throws: java.lang.Exception

unDeploy(String)

public java.lang.String unDeploy(java.lang.String serviceAssemblyName) throws Exception

Undeploys the given Service Assembly from the JBI environment

Parameters:
- serviceAssemblyName - name of the Service Assembly that has to be undeployed

Returns: Result/Status of the current undeployment.

Throws: java.lang.Exception - if complete undeployment fails
InstallationServiceMBean

javax.jbi.management

InstallationServiceMBean

Declaration
public interface InstallationServiceMBean

Description
The InstallationService is responsible for the installation and uninstallation of components (engines and bindings) and shared libraries.

Member Summary

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.String installSharedLibrary(java.lang.String slZipURL)</td>
</tr>
<tr>
<td>javax.management.ObjectName loadInstaller(java.lang.String aComponentName)</td>
</tr>
<tr>
<td>javax.management.ObjectName loadNewInstaller(java.lang.String installZipURL)</td>
</tr>
<tr>
<td>boolean uninstallSharedLibrary(java.lang.String slName)</td>
</tr>
<tr>
<td>boolean unloadInstaller(java.lang.String aComponentName, boolean isToBeDeleted)</td>
</tr>
</tbody>
</table>

Methods

installSharedLibrary(String)

public java.lang.String installSharedLibrary(java.lang.String slZipURL)

Install a shared library zip archive.

The return value is the unique name for the shared-library, as found in the the value of the installation descriptor's <identification><name> element.

Parameters:
slZipURL - URL locating a zip file containing a shared library installation package

Returns: the unique name of the shared library loaded from slZipURL.

loadInstaller(String)

public javax.management.ObjectName loadInstaller(java.lang.String aComponentName)

Load the InstallerMBean for a previously installed component.

The “component name” refers to the <identification><name> element value from the component’s installation package (see loadNewInstaller(String)).

Parameters:
aComponentName - the component name identifying the installer to load.
Returns: the JMX ObjectName of the InstallerMBean loaded from an existing installation context.

loadNewInstaller(String)

public javax.management.ObjectName loadNewInstaller(java.lang.String installZipURL)

Load the installer for a new component from a component installation package.

Parameters:
installZipURL - URL locating a ZIP file containing the JBI Installation package to be installed

Returns: the JMX ObjectName of the InstallerMBean loaded from installZipURL.

uninstallSharedLibrary(String)

public boolean uninstallSharedLibrary(java.lang.String slName)

Uninstall a shared library.

Parameters:
slName - the name of the shared name space to uninstall.

Returns: true if the uninstall was successful.

unloadInstaller(String, boolean)

public boolean unloadInstaller(java.lang.String aComponentName, boolean isToBeDeleted)

Unload a JBI Installable Component installer.

Parameters:
aComponentName - the component name identifying the installer to unload.
isToBeDeleted - true if the component is to be deleted as well.

Returns: true if the operation was successful, otherwise false.
**InstallerMBean**

**javax.jbi.management**

### InstallerMBean

#### Declaration

```java
public interface InstallerMBean
```

#### Description

InstallerMBean defines standard installation and uninstallation controls for components.

---

**Member Summary**

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>getInstallerConfigurationMBean()</td>
</tr>
<tr>
<td>install()</td>
</tr>
<tr>
<td>getInstallRoot()</td>
</tr>
<tr>
<td>isInstalled()</td>
</tr>
<tr>
<td>uninstall()</td>
</tr>
</tbody>
</table>

---

**Methods**

#### getInstallerConfigurationMBean()

```java
public javax.management.ObjectName getInstallerConfigurationMBean() throws JBIException
```

Get the installer configuration MBean name for this component.

**Returns:** the MBean object name of the Installer Configuration MBean.

**Throws:**

- `javax.jbi.JBIException` - if the component is not in the LOADED state or if any error occurs during processing.

#### getInstallRoot()

```java
public java.lang.String getInstallRoot() throws JBIException
```

Get the installation root directory path for this component.

**Returns:** the full installation path of this component.

#### install()

```java
public javax.management.ObjectName install() throws JBIException
```

Install a component.
javax.jbi.management

InstallerMBean

isInstalled()

Returns: JMX ObjectName representing the LifeCycleMBean for the installed component, or null if the installation did not complete.

Throws:

javax.jbi.JBIException - if the installation fails.

isInstalled()

public boolean isInstalled()

Determine whether or not the component is installed.

Returns: true if this component is currently installed, false if not.

uninstall()

public void uninstall()

throws JBIException

Uninstall a component. This completely removes the component from the JBI system.

Throws:

javax.jbi.JBIException - if the uninstallation fails.
Declaration

public interface LifeCycleMBean

All Known Subinterfaces: ComponentLifeCycleMBean

Description

LifeCycleMBean is a base interface that defines standard life cycle controls for JBI Framework services and JBI Components (bindings and engines).

Member Summary

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>void shutDown()</td>
</tr>
<tr>
<td>void start()</td>
</tr>
<tr>
<td>void stop()</td>
</tr>
</tbody>
</table>

Methods

shutDown()

public void shutDown()
throws JBIException

Shut down the item. The releases resources, preparatory to uninstallation.

Throws:
javax.jbi.JBIException - if the item fails to shut down

start()

public void start()
throws JBIException

Start the item.

Throws:
javax.jbi.JBIException - if the item fails to start

stop()

public void stop()
throws JBIException

Stop the item. This suspends current messaging activities.

Throws:
javax.jbi.JBIException - if the item fails to stop
javax.jbi.management
ManagementMessageBuilder

Declaration
public interface ManagementMessageBuilder

Description
ManagementMessageBuilder defines the interface that must be invoked by components in order to construct exception messages before throwing them back to the JBI Framework, and task status messages for logging. Using structured messages (XML) allows JBI implementations to report such messages in a uniform manner for all components.

Methods

buildComponentExceptionMessage(HashMap)

public java.lang.String buildComponentExceptionMessage(java.util.HashMap params)
throws Exception

Construct an exception message encoded as an XML string. TODO: Schema?

Parameters:
params - a hashmap containing the message parameters

Returns: exception message encoded as an XML string.

Throws:
java.lang.Exception - if a processing error occurs

buildComponentTaskStatusMessage(String, String, String, String)

public java.lang.String buildComponentTaskStatusMessage(java.lang.String componentName, java.lang.String taskName, java.lang.String taskResult, java.lang.String taskStatusMessage)
throws Exception

Construct a task status message, encoded as an XML String. TODO: Schema?

Parameters:
componentName - the name of the component reporting task status
taskName - the name of the task being reported on
**ManagementMessageBuilder** javax.jbi.management

buildComponentTaskStatusMessage(String, String, String, String)

- **taskResult** - the result of the task
- **taskStatusMessage** - the status of the task

**Returns:** task status message, encoded as an XML string.

**Throws:**
- `java.lang.Exception` - if a processing error occurs
javax.jbi.management

ManagementMessageFactory

Declaration
public interface ManagementMessageFactory

Description
This provides access to message builder needed by all components running running in the JBI framework.

Member Summary
Methods

newBuildManagementMessage()

public javax.jbi.management.ManagementMessageBuilder newBuildManagementMessage()

Get an instance of message builder.

Returns: management message builder.
MBeanNames

gGetComponentMBeanName(String, String)

javax.jbi.management
MBeanNames

Declaration
public interface MBeanNames

Description
Provide methods to create standard ObjectNames for JBI Framework engine, binding, and framework system services.

Member Summary

||
| Methods | javax.management.ObjectName getComponentMBeanName(java.lang.String componentName, java.lang.String controlType)114 |
| Methods | javax.management.ObjectName getCustomComponentMBeanName(java.lang.String customName, java.lang.String componentName)114 |
| Methods | java.lang.String getJmxDomainName()115 |
| Methods | javax.management.ObjectName getSystemServiceMBeanName(java.lang.String name, java.lang.String type)115 |

Methods

gGetComponentMBeanName(String, String)

public javax.management.ObjectName getComponentMBeanName(java.lang.String componentName, java.lang.String controlType)

Formulate and return the MBean ObjectName of a JBI component.

Parameters:
componentName -- the name (alias) of the Binding
controlType -- the type of control (mbean type)

Returns: the JMX ObjectName of the service, or null if illegal name.

gGetCustomComponentMBeanName(String, String)

public javax.management.ObjectName getCustomComponentMBeanName(java.lang.String customName, java.lang.String componentName)

Formulate and return the MBean ObjectName of Custom Control for a JBI component.

Parameters:
customName - the name of the custom control.
componentName - the name of the component
javax.jbi.management

MBeanNames

getJmxDomainName()

public java.lang.String getJmxDomainName()

Retrieve the default JMX Domain Name for MBean’s registered in this instance of the JBI Framework.

Returns: the JMX domain name for this instance of the JBI Framework.

getSystemServiceMBeanName(String, String)

public javax.management.ObjectName getSystemServiceMBeanName(java.lang.String name, java.lang.String type)

Formulate and return the MBean ObjectName of a JBI Framework system service.

Parameters:
   name - the name of the system service.
   type - the type of the MBean.

Returns: the JMX ObjectName of the service, or null if illegal name.
MBeanNames in javax.jbi.management

getSystemServiceMBeanName(String, String)
## Package

```java
javax.jbi.messaging
```

### Class Summary

#### Interfaces
- **DeliveryChannel**
  - Bi-directional communication channel used by engines and bindings (hereafter called JBI components) to interact with the Normalized Message Router.
- **Fault**
  - Marker interface for WSDL fault messages.
- **InOnly**
  - Supports operations used to process an In Only MEP to completion.
- **InOptionalOut**
  - Supports operations used to process an In Optional Out MEP to completion.
- **InOut**
  - Supports operations used to process an In Out MEP to completion.
- **MessageExchange**
  - MessageExchange represents a container for normalized messages which are described by an exchange pattern.
- **NormalizedMessage**
  - Represents a JBI Normalized Message.
- **RobustInOnly**
  - Supports operations used to process an Robust In Only MEP to completion.

#### Classes
- **ExchangeStatus**
  - Typesafe enumeration containing status values for a message exchange.

#### Exceptions
- **MessagingException**
  - Generic exception used to report messaging related errors in the Normalized Message Router.
javax.jbi.messaging

DeliveryChannel

Declaration
public interface DeliveryChannel

Description
Bi-directional communication channel used by engines and bindings (hereafter called JBI components) to interact with the Normalized Message Router. Each JBI component has one DeliveryChannel associated with it.

Member Summary

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>MessageExchange accept()</td>
</tr>
<tr>
<td>MessageExchange accept(long timeoutMS)</td>
</tr>
<tr>
<td>void close()</td>
</tr>
<tr>
<td>MessageExchange createExchange(javax.xml.namespace.QName serviceName, javax.xml.namespace.QName operationName)</td>
</tr>
<tr>
<td>MessageExchange createExchange(java.net.URI pattern)</td>
</tr>
<tr>
<td>InOnly createInOnlyExchange()</td>
</tr>
<tr>
<td>InOptionalOut createInOptionalOutExchange()</td>
</tr>
<tr>
<td>InOut createInOutExchange()</td>
</tr>
<tr>
<td>RobustInOnly createRobustInOnlyExchange()</td>
</tr>
<tr>
<td>void send(MessageExchange exchange)</td>
</tr>
<tr>
<td>boolean sendSynch(MessageExchange exchange)</td>
</tr>
<tr>
<td>boolean sendSynch(MessageExchange exchange, long timeoutMS)</td>
</tr>
</tbody>
</table>

Methods

accept()

public javax.jbi.messaging.MessageExchange accept()

throws MessagingException

Blocking call used to service a MessageExchange instance which has been initiated by another component. This method supports concurrent invocation for multi-threaded environments.

Returns: message exchange instance

Throws:

MessagingException - failed to accept

accept(long)

public javax.jbi.messaging.MessageExchange accept(long timeoutMS)

throws MessagingException, IllegalArgumentException

February 28, 2005 Java Business Integration 1.0 Public Draft 118
Blocking call used to service a MessageExchange instance which has been initiated by another component, within the given timeout period. This method supports concurrent invocation for multi-threaded environments.

Parameters:
- timeoutMS - timeout period, in milliseconds (0 -> no timeout)

Returns: message exchange instance, or null if timed out

Throws:
- java.lang.IllegalArgumentException - if timeoutMS < 0
- MessagingException - failed to accept

public void close() throws MessagingException

Closes the delivery channel, halting all message traffic.

Throws:
- MessagingException - fatal error while closing channel

public javax.jbi.messaging.MessageExchange createExchange(javax.xml.namespace.QName serviceName, javax.xml.namespace.QName operationName) throws MessagingException

Creates a new MessageExchange instance used to initiate a service invocation.

JBI defines a set of four basic message exchange types, corresponding to the predefined in-* WSDL 2.0 Message Exchange Patterns. Message exchanges of any type are created using the DeliveryChannel.

Parameters:
- serviceName - the name of the service to be invoked
- operationName - the name of the operation to be invoked

Returns: new message exchange, initialized for invoking the given service and operation

Throws:
- MessagingException - if the given service and operation are not registered with the NMR.

public javax.jbi.messaging.MessageExchange createExchange(java.net.URI pattern) throws MessagingException

Creates a new MessageExchange instance used to initiate a service invocation. JBI defines a set of eight fundamental message exchange types which are created using binding and engine delivery channels. This base method is provided for extensibility, to satisfy the need for vendor-specific message exchange patterns. The registration and administration of these patterns is not addressed by JBI.

Parameters:
- pattern - message exchange pattern

Returns: new message exchange

Throws:
- MessagingException - specified pattern is not registered to a message exchange type
createInOnlyExchange()

public javax.jbi.messaging.InOnly createInOnlyExchange() throws MessagingException

Convenience method that creates an In-Only message exchange.

Returns: new InOnly message exchange

Throws:
    MessagingException - failed to create exchange

createInOptionalOutExchange()

public javax.jbi.messaging.InOptionalOut createInOptionalOutExchange() throws MessagingException

Convenience method that creates an In-Optional-Out message exchange.

Returns: new InOptionalOut message exchange

Throws:
    MessagingException - failed to create exchange

createInOutExchange()

public javax.jbi.messaging.InOut createInOutExchange() throws MessagingException

Convenience method that creates an In Out message exchange.

Returns: new InOut message exchange

Throws:
    MessagingException - failed to create exchange

createRobustInOnlyExchange()

public javax.jbi.messaging.RobustInOnly createRobustInOnlyExchange() throws MessagingException

Convenience method that creates a Robust-In-Only message exchange.

Returns: new RobustInOnly message exchange

Throws:
    MessagingException - failed to create exchange

send(MessageExchange)

public void send(javax.jbi.messaging.MessageExchange exchange) throws MessagingException

Routes a MessageExchange instance through the Normalized Message Router to the appropriate servicing component. This method supports concurrent invocation for multi-threaded environments.

This is used not only to send the initial message in an exchange, but also for the servicer to “return” the exchange with the appropriate response (response, fault, or ExchangeStatus). In more complex message exchange patterns, a single MessageExchange can be sent back-and-forth via send() several times, but always terminating with an ExchangeStatus being sent.
sendSynch(MessageExchange)

    public boolean sendSynch(javax.jbi.messaging.MessageExchange exchange)
    throws MessagingException

    Routes a MessageExchange instance through the Normalized Message Router to the appropriate servicing
    component, and blocks until the exchange is returned. This method supports concurrent invocation for
    multi-threaded environments.

    Parameters:
    exchange - message exchange to send

    Returns: true if the exchange has been returned (always)

    Throws: MessagingException - unable to send exchange, or no response is expected from the send() operation (i.e., the MessageExchange is being used to convey an ExchangeStatus)

sendSynch(MessageExchange, long)

    public boolean sendSynch(javax.jbi.messaging.MessageExchange exchange, long timeoutMS)
    throws MessagingException

    Routes a MessageExchange instance through the Normalized Message Router to the appropriate servicing
    component, and blocks until the exchange is returned or the specified timeout interval elapses. This method
    supports concurrent invocation for multi-threaded environments.

    Parameters:
    exchange - message exchange to send
    timeoutMS - timeout period, in milliseconds (0 -> no timeout)

    Returns: true if the exchange has been returned, or false if the method timed out while waiting

    Throws: MessagingException - unable to send exchange, or no response is expected from the send() operation (i.e., the MessageExchange is being used to convey an ExchangeStatus)
javax.jbi.messaging

ExchangeStatus

Declaration

public final class ExchangeStatus

java.lang.Object

+-javax.jbi.messaging.ExchangeStatus

Description
Typesafe enumeration containing status values for a message exchange.

Member Summary

<table>
<thead>
<tr>
<th>Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>static ExchangeStatus ACTIVE 122</td>
</tr>
<tr>
<td>static ExchangeStatus DONE 122</td>
</tr>
<tr>
<td>static ExchangeStatus ERROR 123</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean equals(ExchangeStatus status 123</td>
</tr>
<tr>
<td>java.lang.String toString() 123</td>
</tr>
<tr>
<td>static ExchangeStatus valueOf(java.lang.String status 123</td>
</tr>
</tbody>
</table>

Inherited Member Summary

Methods inherited from class java.lang.Object

clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(),
wait(long, int), wait(long, int), wait(long, int)

Fields

ACTIVE

public static final javax.jbi.messaging.ExchangeStatus ACTIVE

Indicates that an ME has not been processed to completion.

DONE

public static final javax.jbi.messaging.ExchangeStatus DONE

Indicates that an ME has been processed to completion.
ERROR

public static final javax.jbi.messaging.ExchangeStatus ERROR

Indicates that an ME has terminated abnormally within the JBI environment.

Methods

equals(ExchangeStatus)

public boolean equals(javax.jbi.messaging.ExchangeStatus status)

Equality test.

Returns: boolean result of test.

toString()

public java.lang.String toString()

Returns string value of enumerated type.

Overrides: toString in class Object

Returns: String representation of status value.

valueOf(String)

public static javax.jbi.messaging.ExchangeStatus valueOf(java.lang.String status)

Returns instance of ExchangeStatus that corresponds to given string.

Returns: ExchangeStatus corresponding to the given string value

Throws:

java.lang.IllegalArgumentException - if string can’t be translated
javax.jbi.messaging Fault

Declaration
public interface Fault extends NormalizedMessage

All Superinterfaces: NormalizedMessage

Description
Marker interface for WSDL fault messages.

Inherited Member Summary

<table>
<thead>
<tr>
<th>Methods inherited from interface NormalizedMessage</th>
</tr>
</thead>
<tbody>
<tr>
<td>addAttachment(String, DataHandler), addAttachment(String, Object), getAttachment(String), getAttachment(String, Object), getContent(), getContent(Node), getProperty(String), getProperty(String, Object), getPropertyNames(), getPropertyNames(String), getSecuritySubject(), getSecuritySubject(), getSecuritySubject(), getSecuritySubject(), listAttachments(), listAttachments(String), removeAttachment(String), removeAttachment(String, Object), setContent(Source), setContent(Source, String), setContent(Source, String, Object), setProperty(String, Object), setProperty(String, Object, String), setSecuritySubject(Subject), setSecuritySubject(Subject, String), setSecuritySubject(Subject, String, Object)</td>
</tr>
</tbody>
</table>
javax.jbi.messaging

InOnly

Declaration
public interface InOnly extends MessageExchange

All Superinterfaces: MessageExchange

Description
Supports operations used to process an In Only MEP to completion.

Member Summary

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>getInMessage()</td>
<td>in message</td>
</tr>
<tr>
<td>setInMessage(NormalizedMessage msg)</td>
<td></td>
</tr>
</tbody>
</table>

Inherited Member Summary

Methods inherited from interface MessageExchange

createFault(), createMessage(), getEndpoint(), getError(),
getExchangeId(), getFault(), getMessage(), getOperation(),
getPattern(), getProperty(), getService(), getStatus(),
getTransactionContext(), isTransacted(), setEndpoint(), setError(),
setFault(), setMessage(), setOperation(),
setProperty(), setService(), setStatus(),
setTransactionContext()

Methods

getInMessage()

public javax.jbi.messaging.NormalizedMessage getInMessage()

Retrieves the in normalized message from this exchange.

Returns: in message

setInMessage(NormalizedMessage)

public void setInMessage(javax.jbi.messaging.NormalizedMessage msg)

throws MessagingException

Sets the in normalized message for this exchange.
InOnly
javax.jbi.messaging

setInMessage(NormalizedMessage)

Parameters:
  msg - in message

Throws:
  MessagingException - unable to set in message
InOptionalOut

Declaration
public interface InOptionalOut extends MessageExchange

All Superinterfaces: MessageExchange

Description
Supports operations used to process an In Optional Out MEP to completion.

Member Summary

Methods
- NormalizedMessage getInMessage()
- NormalizedMessage getOutMessage()
- void setInMessage(NormalizedMessage msg)
- void setOutMessage(NormalizedMessage msg)

Inherited Member Summary

Methods inherited from interface MessageExchange
- createFault()
- createMessage()
- getEndpoint()
- getError()
- getExchangeId()
- getFault()
- getMessage(String)
- getOperation()
- getPattern()
- getProperty(String)
- getService()
- getStatus()
- getTransactionContext()
- isTransacted()
- setEndpoint(EndpointReference)
- setError(Exception)
- setFault(Fault)
- setMessage(NormalizedMessage, String)
- setOperation(QName)
- setProperty(String, Object)
- setService(QName)
- setStatus(ExchangeStatus)
- setTransactionContext(Transaction)

Methods

getInMessage()

public javax.jbi.messaging.NormalizedMessage getInMessage()
Retrieves the in message reference from this exchange.

Returns: in message

getOutMessage()

public javax.jbi.messaging.NormalizedMessage getOutMessage()
setInMessage(NormalizedMessage)

Retrieves the *out* message reference from this exchange.

**Returns:** out message

**setInMessage(NormalizedMessage)**

```java
public void setInMessage(javax.jbi.messaging.NormalizedMessage msg)
throws MessagingException
```

Specifies the *in* message reference for this exchange.

**Parameters:**

- `msg` - in message

**Throws:**

- `MessagingException` - unable to set in message

setOutMessage(NormalizedMessage)

```java
public void setOutMessage(javax.jbi.messaging.NormalizedMessage msg)
throws MessagingException
```

Specifies the *out* message reference for this exchange.

**Parameters:**

- `msg` - out message

**Throws:**

- `MessagingException` - unable to set out message
javax.jbi.messaging

InOut

Declaration
public interface InOut extends MessageExchange

All Superinterfaces: MessageExchange

Description
Supports operations used to process an In Out MEP to completion.

Member Summary

Methods

<table>
<thead>
<tr>
<th>NormalizedMessage</th>
<th>getInMessage()</th>
</tr>
</thead>
<tbody>
<tr>
<td>NormalizedMessage</td>
<td>getOutMessage()</td>
</tr>
<tr>
<td>void</td>
<td>setInMessage(NormalizedMessage msg)</td>
</tr>
<tr>
<td>void</td>
<td>setOutMessage(NormalizedMessage msg)</td>
</tr>
</tbody>
</table>

Inherited Member Summary

Methods inherited from interface MessageExchange

<table>
<thead>
<tr>
<th>createFault()</th>
<th>createMessage()</th>
<th>getEndpoint()</th>
<th>getError()</th>
<th>getExchangeId()</th>
<th>getFault()</th>
<th>getMessage(String)</th>
<th>getOperation()</th>
<th>getPattern()</th>
<th>getProperty(String)</th>
<th>getService()</th>
<th>getStatus()</th>
</tr>
</thead>
<tbody>
<tr>
<td>createFault()</td>
<td>createMessage()</td>
<td>getEndpoint()</td>
<td>getError()</td>
<td>getExchangeId()</td>
<td>getFault()</td>
<td>getMessage(String)</td>
<td>getOperation()</td>
<td>getPattern()</td>
<td>getProperty(String)</td>
<td>getService()</td>
<td>getStatus()</td>
</tr>
<tr>
<td>getTransactionContext()</td>
<td>isTransacted()</td>
<td>setEndpoint(EndpointReference)</td>
<td>setError(Exception)</td>
<td>setFault(Fault)</td>
<td>setMessage(NormalizedMessage, String)</td>
<td>setOperation(QName)</td>
<td>setProperty(String, Object)</td>
<td>setService(QName)</td>
<td>setStatus(ExchangeStatus)</td>
<td>setTransactionContext(Transaction)</td>
<td></td>
</tr>
</tbody>
</table>

Methods

getInMessage()

public javax.jbi.messaging.NormalizedMessage getInMessage()

Retrieves the in message reference from this exchange.

Returns: in message

getOutMessage()

public javax.jbi.messaging.NormalizedMessage getOutMessage()
setInMessage(NormalizedMessage)

Retrieves the out message reference from this exchange.

Returns: out message

setInMessage(NormalizedMessage)

public void setInMessage(javax.jbi.messaging.NormalizedMessage msg)
throws MessagingException

Specifies the in message reference for this exchange.

Parameters:
    msg - in message

Throws:
    MessagingException - unable to set in message

setOutMessage(NormalizedMessage)

public void setOutMessage(javax.jbi.messaging.NormalizedMessage msg)
throws MessagingException

Specifies the out message reference for this exchange.

Parameters:
    msg - out message

Throws:
    MessagingException - unable to set in message
javax.jbi.messaging

MessageExchange

Declaration

public interface MessageExchange

All Known Subinterfaces: InOnly, InOptionalOut, InOut, RobustInOnly

Description

MessageExchange represents a container for normalized messages which are described by an exchange pattern. The exchange pattern defines the names, sequence, and cardinality of messages in an exchange.

Member Summary

<table>
<thead>
<tr>
<th>Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault</td>
<td>createFault()</td>
</tr>
<tr>
<td>NormalizedMessage</td>
<td>createMessage()</td>
</tr>
<tr>
<td>java.jbi.servicedesc.EndpointReference</td>
<td>getEndpoint()</td>
</tr>
<tr>
<td>java.lang.Exception</td>
<td>getError()</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>getExchangeId()</td>
</tr>
<tr>
<td>Fault</td>
<td>getFault()</td>
</tr>
<tr>
<td>NormalizedMessage</td>
<td>getMessage(java.lang.String name)</td>
</tr>
<tr>
<td>java.xml.namespace.QName</td>
<td>getOperation()</td>
</tr>
<tr>
<td>java.net.URI</td>
<td>getPattern()</td>
</tr>
<tr>
<td>java.lang.Object</td>
<td>getProperty(java.lang.String name)</td>
</tr>
<tr>
<td>java.xml.namespace.QName</td>
<td>getService()</td>
</tr>
<tr>
<td>ExchangeStatus</td>
<td>getStatus()</td>
</tr>
<tr>
<td>javax.transaction.Transaction</td>
<td>getTransactionContext()</td>
</tr>
<tr>
<td>boolean</td>
<td>isTransacted()</td>
</tr>
<tr>
<td>void</td>
<td>setEndpoint(java.jbi.servicedesc.EndpointReference)</td>
</tr>
<tr>
<td>void</td>
<td>setError(java.lang.Exception error)</td>
</tr>
<tr>
<td>void</td>
<td>setFault(Fault fault)</td>
</tr>
<tr>
<td>void</td>
<td>setMessage(NormalizedMessage msg, java.lang.String name)</td>
</tr>
<tr>
<td>void</td>
<td>setOperation(java.xml.namespace.QName name)</td>
</tr>
<tr>
<td>void</td>
<td>setMessage(java.lang.String name, java.lang.Object obj)</td>
</tr>
<tr>
<td>void</td>
<td>setService(java.xml.namespace.QName service)</td>
</tr>
<tr>
<td>void</td>
<td>setStatus(ExchangeStatus status)</td>
</tr>
<tr>
<td>void</td>
<td>setTransactionContext(javax.transaction.Transaction xact)</td>
</tr>
</tbody>
</table>
Methods

createFault()

```java
public javax.jbi.messaging.Fault createFault() throws MessagingException

Generic factory method for Fault objects.

Returns: a new fault message

Throws: MessagingException - failed to create fault
```

createMessage()

```java
public javax.jbi.messaging.NormalizedMessage createMessage() throws MessagingException

Creates a normalized message based on the specified message reference. The pattern governing this exchange must contain a definition for the reference name supplied.

Returns: a new normalized message

Throws: MessagingException - failed to create message
```

genericEndpoint()

```java
public javax.jbi.servicedesc.EndpointReference getEndpoint() throws Exception

Retrieves the endpoint used by this exchange.

Returns: endpoint address for this message exchange
```

getError()

```java
public java.lang.Exception getError()

Retrieves the Exception describing the exchanges error status.

Returns: exception associated with this exchange
```

getExchangeId()

```java
public java.lang.String getExchangeId()

Returns the unique identifier assigned by the NMR for this exchange.

Returns: unique id for this exchange
```

getFault()

```java
public javax.jbi.messaging.Fault getFault()

Retrieves the fault message for this exchange, if one exists. A fault/message reference is unnecessary, since an exchange can carry at most one fault, and it is always the final message in an exchange.

Returns: fault associated with the exchange, or null if not present
```
getMessage(String)

```java
public javax.jbi.messaging.NormalizedMessage getMessage(java.lang.String name)
```

Retrieves a normalized message based on the specified message reference.

**Parameters:**
- name - message reference

**Returns:** message with the specified reference name

getOperation()

```java
public javax.xml.namespace.QName getOperation()
```

Retrieves the operation used by this exchange.

**Returns:** operation name for this message exchange

getPattern()

```java
public java.net.URI getPattern()
```

Returns the URI of the pattern for this exchange.

**Returns:** pattern URI for this exchange

getProperty(String)

```java
public java.lang.Object getProperty(java.lang.String name)
```

Retrieves the specified property from the exchange.

**Parameters:**
- name - property name

**Returns:** property value

getService()

```java
public javax.xml.namespace.QName getService()
```

Retrieves the service used by this exchange.

**Returns:** service address for this message exchange

getStatus()

```java
public javax.jbi.messaging.ExchangeStatus getStatus()
```

Returns the processing status of the exchange.

**Returns:** status of the exchange

getTransactionContext()

```java
public javax.transaction.Transaction getTransactionContext()
```

Retrieves the transaction context used by this exchange.

**Returns:** transaction transaction context for this exchange
isTransacted()

    public boolean isTransacted()
    Queries the existence of a transaction context.
    Returns: boolean transactional state of the exchange

setEndpoint(EndpointReference)

    public void setEndpoint(javax.jbi.servicedesc.EndpointReference endpoint)
    Specifies the endpoint used by this exchange.
    Parameters:
        endpoint - endpoint address

setError(Exception)

    public void setError(java.lang.Exception error)
    Used to specify the source of a failure status. Invoking this method automatically adjusts the status of the
    ME to ExchangeStatus.ERROR.
    Parameters:
        error - error cause

setFault(Fault)

    public void setFault(javax.jbi.messaging.Fault fault)
    throws MessagingException
    Specifies the fault message for this exchange, if one exists. A fault/message reference is unnecessary, since
    an exchange can carry at most one fault, and it is always the final message in an exchange.
    Parameters:
        fault - fault message
    Throws:
        MessagingException - operation not permitted in the current exchange state

setMessage(NormalizedMessage, String)

    public void setMessage(javax.jbi.messaging.NormalizedMessage msg,
                            java.lang.String name)
    throws MessagingException
    Sets a normalized message with the specified message reference. The pattern governing this exchange must
    contain a definition for the reference name supplied.
    Parameters:
        msg - normalized message
        name - message reference
    Throws:
        MessagingException - operation not permitted in the current exchange state

setOperation(QName)

    public void setOperation(javax.xml.namespace.QName name)
Specifies the operation used by this exchange.

**Parameters:**
- `name` - operation name

In **setProperty(String, Object)**

```java
public void setProperty(java.lang.String name, java.lang.Object obj)
```

Specifies a property for the exchange.

**Parameters:**
- `name` - property name
- `obj` - property value

In **setService(QName)**

```java
public void setService(javax.xml.namespace.QName service)
```

Specifies the service used by this exchange.

**Parameters:**
- `service` - service address

In **setStatus(ExchangeStatus)**

```java
public void setStatus(javax.jbi.messaging.ExchangeStatus status)
```

Sets the processing status of the exchange.

**Parameters:**
- `status` - exchange status

**Throws:**
- `MessagingException` - failed to set status, possibly due to an invalid state transition.

In **setTransactionContext(Transaction)**

```java
public void setTransactionContext(javax.transaction.Transaction xact)
```

Sets the transaction context for this exchange.

**Parameters:**
- `xact` - transaction context to use with this exchange

**Throws:**
- `MessagingException` - problem with transaction context
javax.jbi.messaging

MessagingException

Declaration
public class MessagingException extends javax.jbi.JBIException

java.lang.Object
   |-- java.lang.Throwable
      |-- java.lang.Exception
         |-- javax.jbi.JBIException
             |-- javax.jbi.messaging.MessagingException

All Implemented Interfaces: java.io.Serializable

Description
Generic exception used to report messaging related errors in the Normalized Message Router.

Member Summary

Constructors
MessagingException(java.lang.String msg)
MessagingException(java.lang.String msg, java.lang.Throwable cause)
MessagingException(java.lang.Throwable cause)

Inherited Member Summary

Methods inherited from class Object
clone(), equals(Object), finalize(), getClass(), hashCode(), notify(), notifyAll(),
wait(long, int), wait(long, int), wait(long, int)

Methods inherited from class Throwable
fillInStackTrace(), getCause(), getLocalizedMessage(), getMessage(), getStackTrace(),
initCause(Throwable), printStackTrace(PrintWriter), printStackTrace(PrintWriter),
printStackTrace(PrintWriter), setStackTrace(StackTraceElement[]), toString()
Constructors

MessagingException(String)

    public MessagingException(java.lang.String msg)

    Create a new MessagingException.

    Parameters:
      msg - error detail

MessagingException(String, Throwable)

    public MessagingException(java.lang.String msg, java.lang.Throwable cause)

    Create a new MessagingException with the specified cause and error text.

    Parameters:
      msg - error detail
      cause - underlying error

MessagingException(Throwable)

    public MessagingException(java.lang.Throwable cause)

    Create a new MessagingException with the specified cause.

    Parameters:
      cause - underlying error
NormalizedMessage
javax.jbi.messaging
addAttachment(String, DataHandler)

javax.jbi.messaging
NormalizedMessage

Declaration
public interface NormalizedMessage

All Known Subinterfaces: Fault

Description
Represents a JBI Normalized Message.

Member Summary

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void addAttachment(java.lang.String id, javax.activation.DataHandler content)</td>
<td>Add an attachment to this message.</td>
</tr>
<tr>
<td>javax.activation.DataHandler getAttachment(java.lang.String id)</td>
<td></td>
</tr>
<tr>
<td>java.xml.transform.Source getContent()</td>
<td></td>
</tr>
<tr>
<td>java.lang.Object getProperty(java.lang.String name)</td>
<td></td>
</tr>
<tr>
<td>java.util.Iterator getPropertyNames()</td>
<td></td>
</tr>
<tr>
<td>java.util.Iterator listAttachments()</td>
<td></td>
</tr>
<tr>
<td>void removeAttachment(java.lang.String id)</td>
<td></td>
</tr>
<tr>
<td>void setContent(java.xml.transform.Source content)</td>
<td></td>
</tr>
<tr>
<td>void setProperty(java.lang.String name, java.lang.Object value)</td>
<td></td>
</tr>
<tr>
<td>void setSecuritySubject(java.security.auth.Subject subject)</td>
<td></td>
</tr>
</tbody>
</table>

Methods

addAttachment(String, DataHandler)

public void addAttachment(java.lang.String id, javax.activation.DataHandler content) throws MessagingException

Add an attachment to this message.

Parameters:
  id - unique identifier for the attachment
  content - attachment content

Throws:
  MessagingException - failed to add attachment
getAttachment(String)

    public javax.activation.DataHandler getAttachment(java.lang.String id)

Retrieve attachment to this message with the specified identifier.

Parameters:
    id - unique identifier for attachment

Returns: javax.activation.DataHandler representing attachment content, or null if an attachment with the specified identifier is not found

getContent()

    public javax.xml.transform.Source getContent()

Retrieve the content of this message.

Returns: message content

getProperty(String)

    public java.lang.Object getProperty(java.lang.String name)

Retrieve a property from the message.

Parameters:
    name - property name

Returns: property value, or null if the property does not exist

getPropertyNames()

    public java.util.Iterator getPropertyNames()

Retrieve a list of property names for the message.

Returns: list of property names

getSecuritySubject()

    public javax.security.auth.Subject getSecuritySubject()

Get the security subject associated with this message.

Returns: the security subject associated with this message; null if none.

listAttachments()

    public java.util.Iterator listAttachments()

Returns a list of identifiers for all attachments to this message.

Returns: iterator over String attachment identifiers

removeAttachment(String)

    public void removeAttachment(java.lang.String id)
        throws MessagingException

Removes attachment to the message with the specified unique identifier.
Parameters:

  id - attachment identifier

Throws:

  MessagingException - failed to remove attachment

setContent(Source)

public void setContent(javax.xml.transform.Source content)
  throws MessagingException

Set the content of this message.

Parameters:

  content - message content

Throws:

  MessagingException - failed to set content

setProperty(String, Object)

public void setProperty(java.lang.String name, java.lang.Object value)

Set a property on the message.

Parameters:

  name - property name
  value - property value

setSecuritySubject(Subject)

public void setSecuritySubject(javax.security.auth.Subject subject)

Set the security subject associated with the message.

Parameters:

  subject - the security subject to associate with this message
javax.jbi.messaging

RobustInOnly

Declaration

public interface RobustInOnly extends MessageExchange

All Superinterfaces: MessageExchange

Description
Supports operations used to process an Robust In Only MEP to completion.

Member Summary

Methods

NormalizedMessage getInMessage()
void setInMessage(NormalizedMessage msg)

Inherited Member Summary

Methods inherited from interface MessageExchange

createFromFault(), createMessage(), getEndpoint(), getError(),
getMessageId(), getFault(), getMessage(String), getOperation(),
getPattern(), getProperty(String), getService(), getStatus(),
getTransactionContext(), isTransacted(), setEndpoint(EndpointReference),
setError(Exception), setFault(Fault), setMessage(NormalizedMessage, String),
setOperation(QName), setProperty(String, Object), setService(QName),
setStatus(ExchangeStatus), setTransactionContext(Transaction)

Methods

getInMessage()

    public javax.jbi.messaging.NormalizedMessage getInMessage()

    Retrieves the in normalized message from this exchange.

    Returns: in message

setInMessage(NormalizedMessage)

    public void setInMessage(javax.jbi.messaging.NormalizedMessage msg)
    throws MessagingException

    Sets the in normalized message for this exchange.
setInMessage(NormalizedMessage)

Parameters:
msg - in message

Throws:
   MessagingException\_\_\_136 - unable to set in message
## Class Summary

### Interfaces

<table>
<thead>
<tr>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Descriptor</td>
<td>Basic interface for metadata information.</td>
</tr>
<tr>
<td>EndpointReference</td>
<td>Reference generated by the NMR to refer to an endpoint registration.</td>
</tr>
</tbody>
</table>
javax.jbi.servicedesc

**Descriptor**

Declaration

```java
public interface Descriptor
```

Description

Basic interface for metadata information.

### Member Summary

<table>
<thead>
<tr>
<th>Methods</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>org.w3c.dom.Document</td>
<td>getDescription()</td>
</tr>
</tbody>
</table>

### Methods

#### getDescription()

```java
public org.w3c.dom.Document getDescription()
```

Retrieves a DOM representation containing metadata which describes an entity registered with NMR.

**Returns:** DOM representation of entity metadata.
javax.jbi.servicedesc

EndpointReference

Declaration
public interface EndpointReference

Description
Reference generated by the NMR to refer to an endpoint registration.

Member Summary

<table>
<thead>
<tr>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.String getEndpointName()</td>
</tr>
<tr>
<td>javax.xml.namespace.QName getServiceName()</td>
</tr>
</tbody>
</table>

Methods

getEndpointName()

public java.lang.String getEndpointName()

Returns the endpoint name of this reference.

Returns: endpoint name

ggetServiceName()

public javax.xml.namespace.QName getServiceName()

Returns the service name of this reference.

Returns: qualified service name
<table>
<thead>
<tr>
<th>EndpointReference</th>
<th>javax.jbi.servicedesc</th>
</tr>
</thead>
<tbody>
<tr>
<td>getServiceName()</td>
<td></td>
</tr>
</tbody>
</table>
ALMANAC LEGEND
The almanac presents classes and interfaces in alphabetic order, regardless of their package. Fields, methods and constructors are in alphabetic order in a single list.

This almanac is modeled after the style introduced by Patrick Chan in his excellent book Java Developers Almanac.

1. Name of the class, interface, nested class or nested interface. Interfaces are italic.
2. Name of the package containing the class or interface.
3. Inheritance hierarchy. In this example, RealtimeThread extends Thread, which extends Object.
4. Implemented interfaces. The interface is to the right of, and on the same line as, the class that implements it. In this example, Thread implements Runnable, and RealtimeThread implements Schedulable.
5. The first column above is for the value of the @since comment, which indicates the version in which the item was introduced.
6. The second column above is for the following icons. If the “protected” symbol does not appear, the member is public. (Private and package-private modifiers also have no symbols.) One symbol from each group can appear in this column.
7. Return type of a method or declared type of a field. Blank for constructors.
8. Name of the constructor, field or method. Nested classes are listed in 1, not here.

Modifiers  Access Modifiers  Constructors and Fields
abstract ☐  final ●  static □  static final ■  protected ♦  constructor ♣  field

RealtimeThread

void addToFeasibility()

void currentRealtimeThread()

void getScheduler()

RealtimeThread()

RealtimeThread(SchedulingParameters scheduling)

void sleep(Clock clock, HighResolutionTime time)

throws InterruptedException
CHAPTER 13

Almanac

AdminServiceMBean

AdminServiceMBean

javax.management.ObjectName[] getBindingComponents()
javax.management.ObjectName getComponentByName(String name)
javax.management.ObjectName[] getEngineComponents()
String getSystemInfo()
javax.management.ObjectName getSystemService(String serviceName)
javax.management.ObjectName[] getSystemServices()
boolean isBinding(String componentName)
boolean isEngine(String componentName)

Bootstrap

Bootstrap

javax.management.ObjectName getExtensionMBeanName()
void init(InstallationContext installContext) throws javax.jbi.JBIException
void onInstall() throws javax.jbi.JBIException
void onUninstall() throws javax.jbi.JBIException

Component

Component

ComponentLifeCycle getLifeCycle()
ServiceUnitManager getServiceUnitManager()
javax.jbi.servicedesc.Descriptor resolveReference(javax.jbi.servicedesc.EndpointReference ref)
### ComponentContext

```java
javax.jbi.component
```

```java
ComponentContext
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>activateEndpoint()</td>
<td><code>activateEndpoint(javax.xml.namespace.QName serviceName, String endpointName)</code></td>
</tr>
<tr>
<td>availableEndpoints()</td>
<td><code>availableEndpoints(javax.xml.namespace.QName serviceName)</code></td>
</tr>
<tr>
<td>deactivateEndpoint()</td>
<td><code>deactivateEndpoint(javax.xml.jbi.component.EndpointReference endpoint)</code></td>
</tr>
<tr>
<td>getComponentName()</td>
<td><code>String getComponentName()</code></td>
</tr>
<tr>
<td>getDeliveryChannel()</td>
<td><code>DeliveryChannel getDeliveryChannel()</code></td>
</tr>
<tr>
<td>getEndpointDescriptor()</td>
<td><code>Descriptor getEndpointDescriptor(javax.xml.jbi.component.EndpointReference endpoint)</code></td>
</tr>
<tr>
<td>getEnvironmentType()</td>
<td><code>EnvironmentType getEnvironmentType()</code></td>
</tr>
<tr>
<td>getInstallRoot()</td>
<td><code>getInstallRoot()</code></td>
</tr>
<tr>
<td>getManagementMessageFactory()</td>
<td><code>ManagementMessageFactory getManagementMessageFactory()</code></td>
</tr>
<tr>
<td>getMBeanNames()</td>
<td><code>MBeanNames getMBeanNames()</code></td>
</tr>
<tr>
<td>getMBeanServer()</td>
<td><code>MBeanServer getMBeanServer()</code></td>
</tr>
<tr>
<td>getNamingContext()</td>
<td><code>NamingContext getNamingContext()</code></td>
</tr>
<tr>
<td>getTransactionManager()</td>
<td><code>TransactionManager getTransactionManager()</code></td>
</tr>
<tr>
<td>getWorkspaceRoot()</td>
<td><code>String getWorkspaceRoot()</code></td>
</tr>
</tbody>
</table>

### ComponentContext.EnvironmentType

```java
javax.jbi.component
```

```java
ComponentContext.EnvironmentType
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>J2EE</td>
<td><code>EnvironmentType J2EE</code></td>
</tr>
<tr>
<td>J2SE</td>
<td><code>EnvironmentType J2SE</code></td>
</tr>
</tbody>
</table>

### ComponentLifeCycle

```java
javax.jbi.component
```

```java
ComponentLifeCycle
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>extensionMBeanName()</td>
<td><code>getExtensionMBeanName()</code></td>
</tr>
<tr>
<td>init()</td>
<td><code>init(ComponentContext context)</code></td>
</tr>
<tr>
<td>shutDown()</td>
<td><code>shutdown()</code></td>
</tr>
<tr>
<td>start()</td>
<td><code>start()</code></td>
</tr>
<tr>
<td>stop()</td>
<td><code>stop()</code></td>
</tr>
</tbody>
</table>

### ComponentLifeCycleMBean

```java
javax.jbi.management
```

```java
ComponentLifeCycleMBean
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>extensionMBeanName()</td>
<td><code>getExtensionMBeanName()</code></td>
</tr>
</tbody>
</table>
```java
// DeliveryChannel
package javax.jbi.messaging;

public class DeliveryChannel extends Object {
    private MessageExchange accept() throws MessagingException {
    }
    private void close() throws MessagingException {
    }
    private void send(MessageExchange exchange) throws MessagingException {
    }
    private boolean sendSynch(MessageExchange exchange) throws MessagingException {
    }
    private boolean sendSynch(MessageExchange exchange, long timeoutMS) throws MessagingException {
    }
}

// DeploymentException
package javax.jbi.management;

public class DeploymentException extends Exception {
    public DeploymentException(String message) {
    }
    public DeploymentException(String message, Throwable cause) {
    }
    public DeploymentException(Throwable cause) {
    }
}

// DeploymentServiceMBean
package javax.jbi.management;

public interface DeploymentServiceMBean {
    boolean canDeployToComponent(String componentName, String serviceUnitName) throws Exception {
    }
    String deploy(String serviceAssemblyZipUrl) throws Exception {
    }
    String getAssemblyUnitDescriptor(String serviceAssemblyName) throws Exception {
    }
    String[] getDeployedAssemblyUnitsForComponent(String componentName) throws Exception {
    }
    String[] getDeployedServiceAssemblies() throws Exception {
    }
    String[] getDeployedServiceUnitList(String componentName) throws Exception {
    }
    boolean isDeployedServiceUnit(String componentName, String serviceUnitName) throws Exception {
    }
    String undeploy(String serviceAssemblyName) throws Exception {
    }
}
```
### Descriptor
```java
javax.jbi.servicedesc
```

- **getDescription()**

### EndpointReference
```java
javax.jbi.servicedesc
```

- **getEndpointName()**
- **getServiceName()**

### ExchangeStatus
```java
javax.jbi.messaging
```

- **ACTIVE**
- **DONE**
- **ERROR**
- **valueOf(String status)**
- **toString()**
- **equals(ExchangeStatus status)**

### Fault
```java
javax.jbi.messaging
```

- **getInMessage()**
- **getOutMessage()**
- **setInMessage(NormalizedMessage msg)**
- **setInMessage(NormalizedMessage msg)**
- **setOutMessage(NormalizedMessage msg)**

### InOnly
```java
javax.jbi.messaging
```

- **getInMessage()**
- **setInMessage(NormalizedMessage msg)**

### InOptionalOut
```java
javax.jbi.messaging
```

- **getInMessage()**
- **getOutMessage()**
- **setInMessage(NormalizedMessage msg)**
- **setOutMessage(NormalizedMessage msg)**

### InOut
```java
javax.jbi.messaging
```

- **getInMessage()**
- **getOutMessage()**
**InstallationContext**

```java
package javax.jbi.component;

public interface InstallationContext {
    java.util.List getClassPathElements();
    String getComponentClassName();
    String getComponentName();
    ComponentContext getContext();
    org.w3c.dom.DocumentFragment getInstallationDescriptorExtension();
    String getInstallRoot();
    void setClassPathElements(java.util.List classPathElements);
}
```

**InstallationServiceMBean**

```java
package javax.jbi.management;

public interface InstallationServiceMBean {
    String installSharedLibrary(String slZipURL);
    java.management.ObjectName loadInstaller(String aComponentName);
    java.management.ObjectName loadNewInstaller(String installZipURL);
    boolean uninstallSharedLibrary(String slName);
    boolean unloadInstaller(String aComponentName, boolean isToBeDeleted);
}
```

**InstallerMBean**

```java
package javax.jbi.management;

public interface InstallerMBean {
    java.management.ObjectName getInstallerConfigurationMBean() throws javax.jbi.JBIException;
    String getInstallRoot();
    java.management.ObjectName install() throws javax.jbi.JBIException;
    boolean isInstalled();
    void uninstall() throws javax.jbi.JBIException;
}
```

**JBIException**

```java
package javax.jbi;

public class JBIException extends Exception {
    public JBIException(String message);
    public JBIException(String message, Throwable cause);
    public JBIException(Throwable cause);
}
```

**LifeCycleMBean**

```java
package javax.jbi.management;

public interface LifeCycleMBean {
    void shutDown() throws javax.jbi.JBIException;
    void start() throws javax.jbi.JBIException;
    void stop() throws javax.jbi.JBIException;
}
```
ManagementMessageBuilder

ManagementMessageBuilder

javax.jbi.management

String buildComponentExceptionMessage(java.util.HashMap params)
  throws Exception

String buildComponentTaskStatusMessage(String componentName,
  String taskName, String taskResult, String taskStatusMessage)
  throws Exception

ManagementMessageFactory

ManagementMessageFactory

javax.jbi.management

ManagementMessageBuilder newBuildManagementMessage()

MBeanNames

javax.jbi.management

javax.management.ObjectName getComponentMBeanName(String componentName,
  String controlType)

javax.management.ObjectName getCustomComponentMBeanName(String customName,
  String componentName)

String getJmxDomainName()

javax.management.ObjectName getSystemServiceMBeanName(String name,
  String type)

MessageExchange

javax.jbi.messaging

Fault createFault() throws MessagingException

NormalizedMessage createMessage() throws MessagingException

javax.jbi.servicedesc.EndpointReference getEndpoint()

javax.management.ObjectName getComponentMBeanName(String componentName,
  String controlType)

javax.management.ObjectName getCustomComponentMBeanName(String customName,
  String componentName)

String getJmxDomainName()

javax.management.ObjectName getSystemServiceMBeanName(String name,
  String type)

javax.jbi.servicedesc.EndpointReference getReference()

javax.management.ObjectName getComponentMBeanName(String componentName,
  String controlType)

javax.management.ObjectName getCustomComponentMBeanName(String customName,
  String componentName)

String getJmxDomainName()

javax.management.ObjectName getSystemServiceMBeanName(String name,
  String type)

javax.jbi.servicedesc.EndpointReference getReference()

javax.management.ObjectName getComponentMBeanName(String componentName,
  String controlType)

javax.management.ObjectName getCustomComponentMBeanName(String customName,
  String componentName)

String getJmxDomainName()

javax.management.ObjectName getSystemServiceMBeanName(String name,
  String type)

javax.jbi.servicedesc.EndpointReference getReference()

javax.management.ObjectName getComponentMBeanName(String componentName,
  String controlType)

javax.management.ObjectName getCustomComponentMBeanName(String customName,
  String componentName)

String getJmxDomainName()

javax.management.ObjectName getSystemServiceMBeanName(String name,
  String type)

javax.jbi.servicedesc.EndpointReference getReference()

javax.management.ObjectName getComponentMBeanName(String componentName,
  String controlType)

javax.management.ObjectName getCustomComponentMBeanName(String customName,
  String componentName)

String getJmxDomainName()

javax.management.ObjectName getSystemServiceMBeanName(String name,
  String type)

javax.jbi.servicedesc.EndpointReference getReference()

javax.management.ObjectName getComponentMBeanName(String componentName,
  String controlType)

javax.management.ObjectName getCustomComponentMBeanName(String customName,
  String componentName)

String getJmxDomainName()

javax.management.ObjectName getSystemServiceMBeanName(String name,
  String type)

javax.jbi.servicedesc.EndpointReference getReference()

javax.management.ObjectName getComponentMBeanName(String componentName,
  String controlType)

javax.management.ObjectName getCustomComponentMBeanName(String customName,
  String componentName)

String getJmxDomainName()

javax.management.ObjectName getSystemServiceMBeanName(String name,
  String type)

javax.jbi.servicedesc.EndpointReference getReference()

javax.management.ObjectName getComponentMBeanName(String componentName,
  String controlType)

javax.management.ObjectName getCustomComponentMBeanName(String customName,
  String componentName)

String getJmxDomainName()

javax.management.ObjectName getSystemServiceMBeanName(String name,
  String type)

javax.jbi.servicedesc.EndpointReference getReference()

javax.management.ObjectName getComponentMBeanName(String componentName,
  String controlType)

javax.management.ObjectName getCustomComponentMBeanName(String customName,
  String componentName)

String getJmxDomainName()

javax.management.ObjectName getSystemServiceMBeanName(String name,
  String type)

javax.jbi.servicedesc.EndpointReference getReference()

javax.management.ObjectName getComponentMBeanName(String componentName,
  String controlType)

javax.management.ObjectName getCustomComponentMBeanName(String customName,
  String componentName)

String getJmxDomainName()

javax.management.ObjectName getSystemServiceMBeanName(String name,
  String type)

javax.jbi.servicedesc.EndpointReference getReference()

javax.management.ObjectName getComponentMBeanName(String componentName,
  String controlType)

javax.management.ObjectName getCustomComponentMBeanName(String customName,
  String componentName)

String getJmxDomainName()

javax.management.ObjectName getSystemServiceMBeanName(String name,
  String type)

javax.jbi.servicedesc.EndpointReference getReference()

javax.management.ObjectName getComponentMBeanName(String componentName,
  String controlType)

javax.management.ObjectName getCustomComponentMBeanName(String customName,
  String componentName)

String getJmxDomainName()

javax.management.ObjectName getSystemServiceMBeanName(String name,
  String type)

javax.jbi.servicedesc.EndpointReference getReference()

javax.management.ObjectName getComponentMBeanName(String componentName,
  String controlType)

javax.management.ObjectName getCustomComponentMBeanName(String customName,
  String componentName)

String getJmxDomainName()

javax.management.ObjectName getSystemServiceMBeanName(String name,
  String type)

javax.jbi.servicedesc.EndpointReference getReference()

javax.management.ObjectName getComponentMBeanName(String componentName,
  String controlType)

javax.management.ObjectName getCustomComponentMBeanName(String customName,
  String componentName)

String getJmxDomainName()

javax.management.ObjectName getSystemServiceMBeanName(String name,
  String type)

javax.jbi.servicedesc.EndpointReference getReference()

javax.management.ObjectName getComponentMBeanName(String componentName,
  String controlType)

javax.management.ObjectName getCustomComponentMBeanName(String customName,
  String componentName)

String getJmxDomainName()

javax.management.ObjectName getSystemServiceMBeanName(String name,
  String type)

javax.jbi.servicedesc.EndpointReference getReference()

javax.management.ObjectName getComponentMBeanName(String componentName,
  String controlType)

javax.management.ObjectName getCustomComponentMBeanName(String customName,
  String componentName)

String getJmxDomainName()

javax.management.ObjectName getSystemServiceMBeanName(String name,
  String type)

javax.jbi.servicedesc.EndpointReference getReference()

javax.management.ObjectName getComponentMBeanName(String componentName,
  String controlType)

javax.management.ObjectName getCustomComponentMBeanName(String customName,
  String componentName)

String getJmxDomainName()

javax.management.ObjectName getSystemServiceMBeanName(String name,
  String type)

javax.jbi.servicedesc.EndpointReference getReference()

javax.management.ObjectName getComponentMBeanName(String componentName,
  String controlType)

javax.management.ObjectName getCustomComponentMBeanName(String customName,
  String componentName)

String getJmxDomainName()

javax.management.ObjectName getSystemServiceMBeanName(String name,
  String type)

javax.jbi.servicedesc.EndpointReference getReference()

javax.management.ObjectName getComponentMBeanName(String componentName,
  String controlType)

javax.management.ObjectName getCustomComponentMBeanName(String customName,
  String componentName)

String getJmxDomainName()

javax.management.ObjectName getSystemServiceMBeanName(String name,
  String type)

javax.jbi.servicedesc.EndpointReference getReference()

javax.management.ObjectName getComponentMBeanName(String componentName,
  String controlType)

javax.management.ObjectName getCustomComponentMBeanName(String customName,
  String componentName)

String getJmxDomainName()

javax.management.ObjectName getSystemServiceMBeanName(String name,
  String type)

javax.jbi.servicedesc.EndpointReference getReference()

javax.management.ObjectName getComponentMBeanName(String componentName,
  String controlType)

javax.management.ObjectName getCustomComponentMBeanName(String customName,
  String componentName)

String getJmxDomainName()

javax.management.ObjectName getSystemServiceMBeanName(String name,
  String type)

javax.jbi.servicedesc.EndpointReference getReference()

javax.management.ObjectName getComponentMBeanName(String componentName,
  String controlType)

javax.management.ObjectName getCustomComponentMBeanName(String customName,
  String componentName)

String getJmxDomainName()
### MessagingException

```java
type MessagingException
extends java.io.Serializable
```

- `Object`: Throwable
- `Exception`: java.io.Serializable
- `javax.jbi.JBIException`: MessagingException

### normalizedMessage

```java
type NormalizedMessage
```

- `Object`: throw MessagingException
- `javax.jbi.messaging`: getInMessage()
- `String`: deploy(String serviceUnitName, String serviceUnitRootPath)
- `String`: init(String serviceUnitName, String serviceUnitRootPath)
- `String`: shutdown(String serviceUnitName)

### RobustInOnly

```java
type RobustInOnly
```

- `Object`: getInMessage()

### ServiceUnitManager

```java
type ServiceUnitManager
```

- `String`: deploy(String serviceUnitName, String serviceUnitRootPath)
- `String`: init(String serviceUnitName, String serviceUnitRootPath)
- `String`: shutdown(String serviceUnitName)
void start(String serviceUnitName)
    throws javax.jbi.management.DeploymentException
void stop(String serviceUnitName)
    throws javax.jbi.management.DeploymentException
String undeploy(String serviceUnitName, String serviceUnitRootPath)
    throws javax.jbi.management.DeploymentException
Like any 1.0 version of a specification, JBI 1.0 is the start of what in the future will be a more elaborate, and complete specification. This section summarizes what directions are anticipated in future versions of JBI. This chapter is provided for informational purposes; it is non-normative.
Future Directions

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
CHAPTER 15

References

[JAX-RPC 2.0]

[JLS]

[RFC 2119]

[WSDL 2.0]

[WSDL 2.0 Bindings]

[WSDL 2.0 Extensions]

[XML 1.0]
Index

A
accept()
of javax.jbi.messaging.DeliveryChannel 118
accept(long)
of javax.jbi.messaging.DeliveryChannel 118
activateEndpoint(QName, String)
of javax.jbi.component.ComponentContext 86
ACTIVE
of javax.jbi.messaging.ExchangeStatus 122
addAttachment(String, DataHandler)
of javax.jbi.messaging.NormalizedMessage 138
AdminServiceMBean
of javax.jbi.management 98
availableEndpoints(QName)
of javax.jbi.component.ComponentContext 86

B
Bootstrap
of javax.jbi.component 82
buildComponentExceptionMessage(HashMap)
of javax.jbi.management.ManagementMessageBuilder 111
buildComponentTaskStatusMessage(String, String, String)
of javax.jbi.management.ManagementMessageBuilder 111

C
canDeployToComponent(String, String)
of javax.jbi.management.DeploymentServiceMBean 103
close()
of javax.jbi.messaging.DeliveryChannel 119
Component
of javax.jbi.component 84
ComponentContext
of javax.jbi.component 85
ComponentContext.EnvironmentType
of javax.jbi.component 89
ComponentLifeCycle
of javax.jbi.component 90
ComponentLifeCycleMBean
of javax.jbi.management 100
createExchange(QName, QName)
of javax.jbi.messaging.DeliveryChannel 119
createExchange(URI)
of javax.jbi.messaging.DeliveryChannel 119
createFault()
of javax.jbi.messaging.DeliveryChannel 119
createInOnlyExchange()
of javax.jbi.messaging.DeliveryChannel 120
createInOptionalOutExchange()
of javax.jbi.messaging.DeliveryChannel 120
createInOutExchange()
of javax.jbi.messaging.DeliveryChannel 120
createMessage()
of javax.jbi.messaging.MessageExchange 132
createRobustInOnlyExchange()
of javax.jbi.messaging.DeliveryChannel 120
deactivateEndpoint(EndpointReference)
of javax.jbi.component.ComponentContext 86
DeliveryChannel
of javax.jbi.messaging 118
deploy(String)
of javax.jbi.management.DeploymentServiceMBean 103
deploy(String, String)
of javax.jbi.component.ServiceUnitManager 94
DeploymentException
of javax.jbi.management 101
DeploymentException(String)
of javax.jbi.management.DeploymentException 102
DeploymentException(String, Throwable)
of javax.jbi.management.DeploymentException 102
DeploymentException(Throwlable)
of javax.jbi.management.DeploymentException 102
DeploymentServiceMBean
of javax.jbi.management 103
Descriptor
of javax.jbi.servicedesc 144
DONE
of javax.jbi.messaging.ExchangeStatus 122
Index

E
EndpointReference
  of javax.jbi.servicedesc 145
equals(ExchangeStatus)
  of javax.jbi.messaging.ExchangeStatus 123
ERROR
  of javax.jbi.messaging.ExchangeStatus 123
ExchangeStatus
  of javax.jbi.messaging 122

F
Fault
  of javax.jbi.messaging 124

G
getAssemblyUnitDescriptor(String)
  of javax.jbi.management.DeploymentServiceMBean 104
getAttachment(String)
  of javax.jbi.messaging.NormalizedMessage 139
getBindingComponents()
  of javax.jbi.management.AdminServiceMBean 98
classPathElements()
  of javax.jbi.component.InstallationContext 92
getComponentByObjectName(String)
  of javax.jbi.management.AdminServiceMBean 98
getComponentClassName()
  of javax.jbi.component.InstallationContext 92
getComponentMBeanName(String, String)
  of javax.jbi.management.MBeanNames 114
getComponentName()
  of javax.jbi.component.ComponentContext 86
getComponentsForDeployedAssemblyUnit(String)
  of javax.jbi.management.DeploymentServiceMBean 104
c-content()
  of javax.jbi.messaging.NormalizedMessage 139
c-Context()
  of javax.jbi.component.InstallationContext 93
customComponentMBeanName(String, String)
  of javax.jbi.management.MBeanNames 114

deliveryChannel()
  of javax.jbi.component.ComponentContext 87
deployedAssemblyUnitsForComponent(String)
  of javax.jbi.management.DeploymentServiceMBean 104
deployedServiceAssemblies()
  of javax.jbi.management.DeploymentServiceMBean 104
deployedServiceUnitList(String)
  of javax.jbi.management.DeploymentServiceMBean 105
description()
  of javax.jbi.servicedesc.Descriptor 144
d-endpoint()
  of javax.jbi.messaging.MessageExchange 132
d-endpointDescriptor(EndpointReference)
  of javax.jbi.component.ComponentContext 87
d-endpointName()
  of javax.jbi.servicedesc.EndpointReference 145
d-engineComponents()
  of javax.jbi.management.AdminServiceMBean 99
d-environmentType()
  of javax.jbi.component.ComponentContext 87
d-error()
  of javax.jbi.messaging.MessageExchange 132
d-exchangeId()
  of javax.jbi.messaging.MessageExchange 132
d-extensionMBeanName()
  of javax.jbi.messaging.MessageExchange 132
d-fault()
  of javax.jbi.messaging.MessageExchange 132
d-inMessage()
  of javax.jbi.messaging.InOnly 125
getInstallationDescriptorExtension()
  of javax.jbi.component.InstallationContext 93
g-getInstallerConfigurationMBean()
  of javax.jbi.management.InstallerMBean 108
g-installRoot()
  of javax.jbi.component.ComponentContext 87
g-robustInOnly 141
**Index**

<table>
<thead>
<tr>
<th>Method/Property</th>
<th>Class/Package</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getJmxDomainName()</code></td>
<td><code>javax.jbi.management.MBeanNames</code></td>
<td>115</td>
</tr>
<tr>
<td><code>getLifeCycle()</code></td>
<td><code>javax.jbi.component.Component</code></td>
<td>84</td>
</tr>
<tr>
<td><code>getManagementMessageFactory()</code></td>
<td><code>javax.jbi.component.ComponentContext</code></td>
<td>87</td>
</tr>
<tr>
<td><code>getMBeanNames()</code></td>
<td><code>javax.jbi.component.ComponentContext</code></td>
<td>87</td>
</tr>
<tr>
<td><code>getMBeanServer()</code></td>
<td><code>javax.jbi.component.ComponentContext</code></td>
<td>88</td>
</tr>
<tr>
<td><code>getMBeanServer()</code></td>
<td><code>javax.jbi.component.ComponentContext</code></td>
<td>88</td>
</tr>
<tr>
<td><code>getMessage(String)</code></td>
<td><code>javax.jbi.messaging.MessageExchange</code></td>
<td>133</td>
</tr>
<tr>
<td><code>getNamingContext()</code></td>
<td><code>javax.jbi.component.ComponentContext</code></td>
<td>88</td>
</tr>
<tr>
<td><code>getOperation()</code></td>
<td><code>javax.jbi.component.ComponentContext</code></td>
<td>88</td>
</tr>
<tr>
<td><code>getOutMessage()</code></td>
<td><code>javax.jbi.messaging.InOptionalOut</code></td>
<td>127</td>
</tr>
<tr>
<td><code>getPattern()</code></td>
<td><code>javax.jbi.messaging.MessageExchange</code></td>
<td>133</td>
</tr>
<tr>
<td><code>getProperty(String)</code></td>
<td><code>javax.jbi.messaging.MessageExchange</code></td>
<td>133</td>
</tr>
<tr>
<td><code>getPropertyNames()</code></td>
<td><code>javax.jbi.messaging.NormalizedMessage</code></td>
<td>139</td>
</tr>
<tr>
<td><code>getSecuritySubject()</code></td>
<td><code>javax.jbi.messaging.NormalizedMessage</code></td>
<td>139</td>
</tr>
<tr>
<td><code>getService()</code></td>
<td><code>javax.jbi.messaging.MessageExchange</code></td>
<td>133</td>
</tr>
<tr>
<td><code>getServiceName()</code></td>
<td><code>javax.jbi.servicedesc.EndpointReference</code></td>
<td>145</td>
</tr>
<tr>
<td><code>getServiceUnitManager()</code></td>
<td><code>javax.jbi.component.Component</code></td>
<td>84</td>
</tr>
<tr>
<td><code>getStatus()</code></td>
<td><code>javax.jbi.messaging.MessageExchange</code></td>
<td>133</td>
</tr>
<tr>
<td><code>getSystemInfo()</code></td>
<td><code>javax.jbi.management.AdminServiceMBean</code></td>
<td>99</td>
</tr>
<tr>
<td><code>getSystemService(String)</code></td>
<td><code>javax.jbi.management.AdminServiceMBean</code></td>
<td>99</td>
</tr>
<tr>
<td><code>getSystemServiceMBeanName(String, String)</code></td>
<td><code>javax.jbi.management.MBeanNames</code></td>
<td>115</td>
</tr>
<tr>
<td><code>getSystemServices()</code></td>
<td><code>javax.jbi.management.InstallerMBean</code></td>
<td>108</td>
</tr>
<tr>
<td><code>getTransactionContext()</code></td>
<td><code>javax.jbi.messaging.MessageExchange</code></td>
<td>133</td>
</tr>
<tr>
<td><code>getTransactionManager()</code></td>
<td><code>javax.jbi.component.ComponentContext</code></td>
<td>88</td>
</tr>
<tr>
<td><code>getWorkspaceRoot()</code></td>
<td><code>javax.jbi.component.ComponentContext</code></td>
<td>88</td>
</tr>
</tbody>
</table>

*Note: The page numbers indicate the location of the method or property in the specified class or package.*

**In**

<table>
<thead>
<tr>
<th>Method/Property</th>
<th>Class/Package</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>init(ComponentContext)</code></td>
<td><code>javax.jbi.component.ComponentLifeCycle</code></td>
<td>90</td>
</tr>
<tr>
<td><code>init(InstallationContext)</code></td>
<td><code>javax.jbi.component.Bootstrap</code></td>
<td>82</td>
</tr>
<tr>
<td><code>init(String, String)</code></td>
<td><code>javax.jbi.component.ServiceUnitManager</code></td>
<td>94</td>
</tr>
<tr>
<td><code>install()</code></td>
<td><code>javax.jbi.management.InstallerMBean</code></td>
<td>108</td>
</tr>
<tr>
<td><code>InstallationContext</code></td>
<td><code>javax.jbi.component</code></td>
<td>92</td>
</tr>
<tr>
<td><code>InstallationServiceMBean</code></td>
<td><code>javax.jbi.management</code></td>
<td>106</td>
</tr>
<tr>
<td><code>InstallerMBean</code></td>
<td><code>javax.jbi.management</code></td>
<td>108</td>
</tr>
<tr>
<td><code>installSharedLibrary(String)</code></td>
<td><code>javax.jbi.management.InstallationServiceMBean</code></td>
<td>106</td>
</tr>
<tr>
<td><code>isBinding(String)</code></td>
<td><code>javax.jbi.management.InstallationServiceMBean</code></td>
<td>106</td>
</tr>
<tr>
<td><code>isDeployedServiceUnit(String, String)</code></td>
<td><code>javax.jbi.management.DeploymentServiceMBean</code></td>
<td>105</td>
</tr>
<tr>
<td><code>isEngine(String)</code></td>
<td><code>javax.jbi.management.AdminServiceMBean</code></td>
<td>99</td>
</tr>
<tr>
<td><code>isInstalled()</code></td>
<td><code>javax.jbi.management.InstallerMBean</code></td>
<td>108</td>
</tr>
<tr>
<td><code>isTransacted()</code></td>
<td><code>javax.jbi.messaging.MessageExchange</code></td>
<td>134</td>
</tr>
</tbody>
</table>

*Note: The page numbers indicate the location of the method or property in the specified class or package.*
setInMessage(NormalizedMessage)
of javax.jbi.messaging.InOnly 125
of javax.jbi.messaging.InOptionalOut 128
of javax.jbi.messaging.InOut 130
of javax.jbi.messaging.RobustInOnly 141

setMessage(NormalizedMessage, String)
of javax.jbi.messaging.MessageExchange 134

setOperation(QName)
of javax.jbi.messaging.MessageExchange 134

setOutMessage(NormalizedMessage)
of javax.jbi.messaging.InOptionalOut 128
of javax.jbi.messaging.InOut 130

setProperty(String, Object)
of javax.jbi.messaging.MessageExchange 135
of javax.jbi.messaging.NormalizedMessage 140

setSecuritySubject(Subject)
of javax.jbi.messaging.NormalizedMessage 140

setService(QName)
of javax.jbi.messaging.MessageExchange 135

setStatus(ExchangeStatus)
of javax.jbi.messaging.MessageExchange 135

setTransactionContext(Transaction)
of javax.jbi.messaging.MessageExchange 135

shutDown()
of javax.jbi.component.ComponentLifeCycle 91
of javax.jbi.management.LifeCycleMBean 110

shutDown(String)
of javax.jbi.component.ServiceUnitManager 95

start()
of javax.jbi.component.ComponentLifeCycle 91
of javax.jbi.management.LifeCycleMBean 110

start(String)
of javax.jbi.component.ServiceUnitManager 95

stop()
of javax.jbi.component.ComponentLifeCycle 91
of javax.jbi.management.LifeCycleMBean 110

stop(String)
of javax.jbi.component.ServiceUnitManager 95

toString()
of javax.jbi.messaging.ExchangeStatus 123

undeploy(String)
of javax.jbi.management.DeploymentService-MBean 105

undeploy(String, String)
of javax.jbi.component.ServiceUnitManager 96

unbind()
of javax.jbi.management.InstallerMBean 109

unloadInstaller(String, boolean)
of javax.jbi.management.InstallationServiceMBean 107

valueOf(String)
of javax.jbi.messaging.ExchangeStatus 123
Colophon

This specification document was generated from a set of Java, FrameMaker, StarOffice, and HTML source files. They were compiled using javadoc and the MIF Doclet.
