Java™ Business Integration (JBI) Specification, Version 1.0

Technical comments to: jsr-208-comments (at) jcp.org
Business comments to: jbi-business (at) sun.com

JSR208
Java Community Process
http://www.jcp.org

Ron Ten-Hove & Peter Walker,
Sun Microsystems Inc.

Version 0.4
Early Review Draft
Specification: JSR 208: Java Business Integration Specification (JBI) 1.0 Specification ("Specification")
Status: Early Draft Release
Release: September 30th, 2004

Copyright 2004 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released; or (iii) the date on which the Java Specification Request (JSR) to which the Specification corresponds is withdrawn.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.

July 2004
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.

3 July 2004
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.
Table of Contents

1 Preface ........................................................................................................... 8
2 Introduction ..................................................................................................... 8
   2.1 Summary .................................................................................................. 8
   2.2 Scope ....................................................................................................... 8
   2.3 Target audience ....................................................................................... 9
   2.4 Organization ........................................................................................... 9
   2.5 Acknowledgements for this draft ............................................................. 9
   2.6 Status of this document ......................................................................... 10
3 Overview ........................................................................................................ 11
   3.1 Definitions ............................................................................................. 12
   3.2 Rationale & Assumptions ....................................................................... 12
   3.3 Goals ..................................................................................................... 14
4 Roles .............................................................................................................. 15
   4.1 SE developers ....................................................................................... 15
   4.2 BC developers ....................................................................................... 15
   4.3 JBI framework providers ....................................................................... 15
   4.4 J2EE™ Platform providers ................................................................... 15
   4.5 JBI application developer ...................................................................... 16
5 Architecture of the JBI Environment ............................................................... 17
   5.1 WSDL-based Messaging ........................................................................ 17
   5.2 Normalized Message ............................................................................. 17
   5.3 High-level Architecture ......................................................................... 18
   5.4 Normalized Message Delivery ............................................................... 20
   5.5 Management .......................................................................................... 21
      5.5.1 Component Installation ................................................................. 21
      5.5.2 Life Cycle Management ................................................................. 21
      5.5.3 Assembly Deployment ................................................................... 22
   5.6 BC and SE Frameworks ......................................................................... 22
   5.7 Normalized Message Service ................................................................ 22
   5.8 Service Engines .................................................................................... 23
   5.9 Binding Components ............................................................................. 23
   5.10 Examples ............................................................................................... 23
      5.10.1 One-way Message Adapter ........................................................... 24
6 Normalized Message Service ......................................................................... 26
   6.1 Key Concepts ........................................................................................ 26
      6.1.1 Run-time Registration .................................................................... 26
      6.1.2 Delivery Channel ........................................................................... 26
      6.1.3 Service Invocation .......................................................................... 26
      6.1.4 Message Exchange ......................................................................... 27
      6.1.5 Exchange Summary ....................................................................... 29
   6.2 Normalized Message .............................................................................. 29
      6.2.1 Normalization Defined .................................................................... 29
      6.2.2 Structure of a Normalized Message ............................................... 30
      6.2.3 Processing Methods ....................................................................... 30
      6.2.4 Binary Inclusion ............................................................................. 30
6.3 Quality of Service .......................................................... 32
  6.3.1 Reliability ................................................................. 32
  6.3.2 Transactional ............................................................ 32
  6.3.3 Persistence ............................................................... 32
  6.3.4 Recoverability .......................................................... 33
  6.3.5 Privacy/Confidentiality .............................................. 33
6.4 NMS Design ................................................................... 33
  6.4.1 Registration .............................................................. 33
  6.4.2 Exchanging Messages .............................................. 38
  6.4.3 Reliability in Message Exchange .............................. 43
  6.4.4 Transactional Message Exchange ............................ 46
6.5 NMS API ....................................................................... 48
  6.5.1 Message API ............................................................ 48
  6.5.2 Service API ............................................................. 51
  6.5.3 Usage Examples ...................................................... 60
7 Administration .................................................................... 67
  7.1 BC and SE Services Provided to JBI ............................ 67
7.2 Binding or Engine Installation .......................................... 68
  7.2.1 Load Jar ................................................................. 68
  7.2.2 Initialize Installer .................................................... 70
  7.2.3 Run Installer .......................................................... 71
  7.2.4 Post installation (and JBI system restart) .................... 72
7.3 Customized Engine or Binding Installation ......................... 73
7.4 Binding or Engine Scripted Installation ............................ 74
7.5 Shared Name Space Installation ....................................... 75
7.6 Operation ......................................................................... 75
7.7 Operational States .......................................................... 76
  7.7.1 Operations That Change State ................................... 77
  7.7.2 Extension MBean Example: Pausing/Resuming (Sub-states) .......................................................... 78
  7.7.3 JBI System ............................................................... 78
7.8 Service Component (SC) Deployment ............................ 80
7.9 Querying Installed Bindings, Engines, Name Spaces ............ 81
7.10 Querying Service Component (SC) Deployments .............. 82
7.11 Service Component Undeployment .................................. 83
7.12 Binding, Engine, and Shared Namespace Uninstallation .... 86
7.13 Packaging ..................................................................... 87
  7.13.1 jbi.xml ................................................................. 88
  7.13.2 Installation Packaging ........................................... 90
  7.13.3 Deployment Packaging .......................................... 92
7.14 JNDI Naming Hierarchy ................................................ 93
7.15 Java Management Extensions ....................................... 94
  7.15.1 System MBeans ..................................................... 94
  7.15.2 Required MBeans ................................................ 95
7.16 Scripting Language Support ........................................... 95
8 Common Framework .......................................................... 97
  8.1 Environment Context .................................................... 97
SUN Microsystems Inc.

8.2  JBIException......................................................................................................................... 97
9   Service Engine Framework................................................................................................... 98
9.1  Installation (Bootstrap).......................................................................................................... 98
9.1.1 Installation Context............................................................................................................ 98
9.2  Life Cycle............................................................................................................................. 99
10  Binding Component Framework............................................................................................. 101
10.1 Installation (Bootstrap)......................................................................................................... 101
10.1.1 Installation Context.......................................................................................................... 101
10.2 Life Cycle............................................................................................................................ 102
11  Shared Name Spaces and Class Loading ............................................................................... 104
11.1 JBI Class Loader Hierarchy............................................................................................... 104
11.2 Installation of Shared Name Spaces .................................................................................. 105
11.3 BC and SE Viewpoint.......................................................................................................... 105
12  Security.................................................................................................................................. 107
13  References............................................................................................................................ 107
14  Change History....................................................................................................................... 107
15  Open Issues........................................................................................................................... 107
1 Preface

This is an Early 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 and the open issues indicated in this draft. Several specific areas where we would welcome Early Draft review input can be found in **bold** ‘Notes to the reader’. This draft addresses major areas of the JBI 1.0 architecture, however it is not complete.

Early Draft review feedback to the Expert Group can be sent to jsr-208-comments@sun.com.

2 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, allowing third-party components to be assembled by the end user. These components provide the many varied integration functions needed; JBI provides the mechanisms to allow such components to interoperate predictably and reliably. 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 interoperability semantics founded on standards-based messaging. The SOA approach creates many other benefits that are applicable to enterprise integrations solutions.

2.2 Scope
The term Web Services is increasingly overloaded and it has come to mean many things to different constituencies. The members of the JSR208 Expert group are 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 complementary 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

This specification is intended to be used by Java developers skilled in developing communications and business protocols in a Web Services environment. The primary audience for this specification is system level software engineers with experience in developing containers, execution engines, process engines, and specialized business messaging routers and associated protocols.

This specification is not appropriate for application developers or business analyst. 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 JSR208 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 Acknowledgements for this draft

Without the contributions, input, and general guidance of the JSR208 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:

- Sid Askary
- Borland Software Corp. – Ramanathan Ravikumar
- Cap Gemini – Steve Jones
- Collaxa – Edwin Khodabakhchian
- DPWN SOP Group – Andreas Mattes, Ramtin Mesbahipour
- Intalio, Inc. – Assaf Arkin
- IOPSIS Software Inc. – Rajesh Pradhan
- Jie Liu
- Nokia Corp. – Sudhanshu Pant

July 2004
2.6 Status of this document

This specification is an Early Release Draft. The content herein is subject to change without notice and is considered as work-in-progress by the JSR 208 Expert Group members.
3 Overview

JBI is focused on using messaging in the business integration problem domain. Integration is distinct from distributed, n-tier application construction. It involves allowing separate applications to interoperate despite the fact that the applications in question were not designed to do so. Modern integration approaches promote loose coupling between applications, and scalability by avoiding the so-called “N by M” problem, where N applications must interoperate with M others. Naïve approaches to integration can produce N x M separate connections between the applications, creating a severe bottleneck to scaling such a system past a very few applications. Finally, modern integration utilizes XML-based message exchanges between applications. This has many advantages, but requires the ability to convert such messages to and from other XML vocabularies, as well as non-XML formats as needed by certain applications.

Traditional approaches to integration involve vendor-specific approaches, technologies, application adapters, and tools. This restricts the integration user to the set of capabilities that the vendor's integration solution provides; it is impractical to mix the integration technologies, adapters, and tools from separate vendors.

JBI eliminates this vendor-specific approach to creating integration solutions by creating a standards-based framework into which integration components from multiple vendors can be plugged in. These components provide specific types of services, which can be combined in the JBI framework to provide a wide variety of integration solutions.

The services provided are described using Web Services Description Language (WSDL). This provides an XML message-based, service-oriented model of services that is technology-neutral.

JBI components plug into 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 transformational services to other components, as well as consume such services.
- **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).

The services offered by components are described using WSDL 2.0. This provides an abstract, technology-neutral model of services using XML-based message exchanges. This has several desirable benefits:
Decoupling. Services providers are decoupled from service consumers. This allows service providers to be changed without requiring simultaneous changes to all service consumers. It also allows service providers to be designed independently of consumers.

Portability. Services are described in a fashion that is independent of their implementation. (This includes services accessed via bindings, which may not be Java-based.)

Interoperability. XML message exchanges provide a reasonable basis for interoperability between components that were not designed to interoperate explicitly.

Service-oriented architecture. This provides a flexible, non-proprietary organizing principle for architecting integration solutions.

3.1 Definitions

The term Service Engine (SE) is used to refer to the portion of the JBI environment (or framework) that is responsible for exposing an application programming model to developers. JBI does not define an application programming model but rather defines a set of SPIs that enable the development of standard Web Service “containers” that can seamlessly utilize multiple communication infrastructure ranging from HTTP/SOAP [ref.] to JMS/MOM [ref.] and including specialized AS1/AS2 EDI [ref.] communications stacks. A Binding Component may choose to implement one or more communications protocols thus offering connectivity services to SEs and thereby enabling SEs to expose their services to local and remote consumers as well as enabling the consumption of remote and local Web 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.

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 [ED: ref. TBD] present an opportunity to reach convergence on what is meant by an abstract business message, and more specifically on what is meant by WSDL 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 (SE)
- The development of an abstract protocol-neutral Normalized Message (NM)
- The definition of a standard framework for pluggable Binding Components (BC)
- Provision of a standard mechanism for Normalized Messages to flow between Binding Components and Process Engines
- Establishment of a standard for the packaging and deployment of SE and BCs
- 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.
4 Roles

A functionally rich Web Services 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™ [ref.], 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.

4.1 SE developers

A JBI compatible SE implementation is required to implement the Normalized Message Service (NMS) contract. In addition SE developers must implement the engine lifecycle and management interfaces. SE developers may provide whatever tooling they feel appropriate for easing development at the Business Process level.

4.2 BC developers

A JBI compatible BC implementation is required to implement the Normalized Message Service (NMS) contract. In addition SE developers must implement the binding lifecycle and management interfaces.

4.3 JBI framework 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™ Platform but are not required to do so.

A JBI 1.0 compatible environment must support at least one implementation of a WS-I Basic Profile 1.1 [ref.] compatible Binding Component.

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

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.
4.5 JBI application developer

A JBI application developer is essentially modeling, designing, developing and deploying business components using vocabularies and tools that are defined by specific SE 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 used in the J2EE and J2SE domains, where the emphasis on development of Java code.
5 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.

5.1 WSDL-based Messaging

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 endpoint.

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

5.2 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.
5.3 High-level Architecture

A top-level view of the JBI architecture is depicted in the following illustration.

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. Note that Service Engines are essentially standard containers for hosting WSDL-based service providers and service consumers.

The diagram 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 Service (NMS) which provides the message and control interaction infrastructure. JBI
defines Frameworks which provide the pluggable infrastructure for adding Service Engines (SEs), and protocol Binding Components (BCs).

The core message exchange concept implements WSDL messaging, as discussed above. Inbound service requests are received by BCs, routed by the NMS, and delivered to a SE. Outbound service requests are generated by SEs, routed by the NMS, and delivered to a BC.

All inbound service capabilities are exposed as inbound WSDL-described services and each BC is defined by one or more inbound WSDL endpoints. The NMS provides the routing service, mapping from inbound endpoint to inbound service provider. All outbound messages from the SEs define the outbound destination, as an abstract service name. The NMS routes the outbound message to the appropriate BC.

Aside from the BC/SE frameworks and the NMS, 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.
5.4 Normalized Message Delivery

The JBI environment's primary function is to route normalized messages from one plug-in component to another (BC to SE, or *vice versa*). 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.

A 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 endpoint shown. (The normalized message is sent to the NMS by the BC for further processing.)

SEs similarly send normalized messages to the NMS for delivery. (The source of the message depends on the type of business logic the engine executes.)

A SE creates a normalized message, requesting a service to be performed by a service referenced by its WSDL service name. The name is a logical name; the SE does not select which binding is used to access the service. The binding denormalizes the message into protocol and transport format, and sends the message across the wire to the service provider.
In both the inbound and outbound message cases, the NMS chooses where to route the normalized message. The details of how such delivery decisions are made are given in the NMS section of this specification.

5.5 Management

[Note to reader: This is under consideration by the Expert Group. Should JBI attempt to standardize selected management functions of the JBI environment and the plug-in components it hosts? If so, which functions are appropriate?]

- Life cycle (start, stop) of components and the JBI environment as a whole.
- Installation (and un-installation) of plug-in components (and shared name spaces)?
- Deployment (and un-deployment) of service-related artifacts to components?
- Monitoring of state and performance of components?

Is JMX the appropriate technology to support JBI management? Concerning JMX remoting: what is the best way to handle localization issues (where the client locale <> JBI locale)? Or is this an unrealistic scenario?

What follows immediately below and in chapter 7 contains a description that illustrates some possibilities based on assumed answers to the question posed above.]

The JBI environment, including bindings and engines, is administered through JMX. Several 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 plug-in engines and bindings
- Life cycle management of plug-ins (start/stop controls)
- Deployment of application assembly to plug-ins
- Monitoring and control

5.5.1 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.

5.5.2 Life Cycle Management

SUN Microsystems Inc.
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 life cycle management.

[Note to reader: Other life cycle capabilities, for example for individual endpoints, may be specified in a later release of the specification. Comments are sought on the requirements in this area.]

5.5.3 Assembly Deployment

As discussed above, deployment refers to the addition of application-specific pieces to an installed engine or binding. Each piece of a larger “application” deployed to an individual binding or engine is termed a Service Assembly (SA).

A grouping of one or more SAs is termed a Service Component (SC). This includes a deployment descriptor that indicates where each SA is to be deployed.

5.6 BC and SE Frameworks

The BC and SE frameworks provide the pluggable interface which allows Bindings and Engines to interact with the JBI environment. These frameworks provide 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 engine (or binding) are detailed in the frameworks and the normalized message service sections of this specification. This details the obligations of both framework and engine (or binding) to achieve particular functional goals within the JBI environment.

5.7 Normalized Message Service

The normalized message delivery outlined above depends on the Normalized Message Service (NMS) to route messages from BCs to SEs, and from SEs to BCs. The NMS 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 NMS, 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 NMS chapter of this specification.

Messages are sent by engines and bindings through bi-directional delivery channels. Messages can be received by bindings and engines using a “Pull” model only.

[Note to reader: The EG is considering the role of policy-based message routing in JBI. This could take several forms, but in general requires that bindings and engines provide
meta-data describing capabilities and requirements that could be reconciled by a policy mechanism. We would like public feedback on a) the utility of such a concept, and b) particulars of what sorts of capabilities and requirements ought to be addressed by JBI.]

5.8 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 processes from services.

SEs describe the services they provide using WSDL, which is registered with the SE framework. The SE provides a set of uniquely named services (unique within the JBI environment).

5.9 Binding Components

BCs are used to send and receive messages via specialized 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 in its message context data, allowing protocol-specific information to be conveyed to a SE or BC in a fashion that is opaque to other JBI environment components.)

5.10 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.
5.10.1 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.

Client\textsubscript{1} is a service consumer that wishes to make a one-way request of a service offered by Service\textsubscript{1}. Unfortunately, Client\textsubscript{1} and Service\textsubscript{1} 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:

- BC\textsubscript{1}. A binding component that “speaks” Client\textsubscript{1}'s protocol, which happens to be WS-I BP 1.1 compliant SOAP.
- SE\textsubscript{1}. A service engine used to provide light-weight sequencing of services. This can be configured to perform the desired message adaptation and forwarding.
- BC\textsubscript{R}. The JBI reflector binding, sometimes called the loopback binding.
- SE\textsubscript{2}. 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 NMS 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 NMS will route this message to the reflector binding, BCᴿ, since the request will be performed by an internal service.

• The reflector binding will send (via NMS) the transformation service request to SE₂.¹ SE₂ will perform the transformation and synchronously return the result to the BCᴿ and then to SE₁.

• SE₁ completes the sequencing the conversion-and-forward process by sending the result of the transformation, REQ₁A to Service₁. (The NMS 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 NMS. This is an important feature: the components involved are loosely coupled. Each component indicates to the NMS the desired message destination only by service name; the NMS determines which component this corresponds to.

¹Note that steps 3 and 4 could be optimized somewhat by a suitably clever NMS, to eliminate the need for an actual reflector binding. This discussion does not attempt to describe such product-specific optimizations.
6 Normalized Message Service

The normalized message service, or NMS, provides the message routing service for JBI. Messages are routed between BCs and SEs.

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

6.1 Key Concepts

This section introduces key concepts in the NMS architecture independent of design, interface, or implementation details.

6.1.1 Run-time Registration

Runtime registration is the process by which a BC or SE registers itself with the NMS. Registration can be broken down into two sub-categories:

- Declaring an endpoint or service address to the NMS.
- Providing meta-data describing the nature or an endpoint or service definition.

Declaration is the process by which a binding or engine registers an endpoint or service name with the NMS. BCs may only register endpoints, while SEs may only register services. Any name declared within the NMS (service or endpoint) must be supported by meta-data describing the details of the registration.

6.1.2 Delivery Channel

A delivery channel represents a bi-directional communication pipe used by bindings and engines to communicate with the NMS. BCs and SEs have specific channel extensions which form their SPI contract with the NMS. A BC uses its delivery channel to initiate inbound invocations and service outbound invocations. An SE uses its delivery channel to initiate outbound invocations and service inbound invocations.

Each BC and SE is provided with a single delivery channel. Therefore, it is assumed that API and implementation decisions would support concurrent use of a given channel from multiple threads. This model must scale vertically (volume of message exchanges) and horizontally (number of concurrent interacting threads).

[Note to reader: This concept is under consideration by the Expert Group. Should the Delivery Channel concept exist in the JBI model, or should components use a direct NMS API for creating, sending, and receiving message exchanges? If Delivery Channels should exist, should multiple instances per component be allowed? How should such instances be distinguished?]

6.1.3 Service Invocation

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 invocations within JBI, the following list represents the most common interactions in use today:
• 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 described above may be co-located in a JBI environment, deployed in separate JBI environments, or outside of a JBI environment.

6.1.4 Message Exchange

A Message Exchange (ME) represents the JBI-local portion of a service invocation. The following table shows the relationship between service invocations and message exchanges.

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

The following depicts a one-way service invocation between two local Service Engines. Note the use of a reflector binding to recast the outbound request into a corresponding inbound request.

The following depicts a reliable one-way invocation between Service Engines hosted in separate JBI environments.

The following depicts a request-response invocation between a JBI-hosted Service Engine and a remote party.
Elements of a Message Exchange

- **Initiator** – component which creates the message exchange. BCs initiate inbound exchanges, SEs initiate outbound exchanges.

- **Servicer** – component which services the message exchange. BCs service outbound exchanges, SEs service inbound exchanges.

- **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.

- **Address** – a service reference, endpoint reference, and operation name for the logical address which the NMS 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 NMS may choose to reserve certain property names to declare QoS, security, transaction, or other operational metadata.

[Note to reader: Message Exchange instances are considered to be short-lived, which will not survive a system shutdown or failure. Higher-level error recovery logic, provided by BCs and/or SEs, will handle such error cases. We would appreciate your comments on this restriction to short-lived MEs.]

### 6.1.5 Exchange Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Initiator</th>
<th>Servicer</th>
<th>Messages</th>
<th>Fault Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out-Only</td>
<td>SE</td>
<td>BC</td>
<td>out</td>
<td>N/A</td>
</tr>
<tr>
<td>In-Only</td>
<td>BC</td>
<td>SE</td>
<td>in</td>
<td>N/A</td>
</tr>
<tr>
<td>Robust Out-Only</td>
<td>SE</td>
<td>BC</td>
<td>out</td>
<td>on out</td>
</tr>
<tr>
<td>Robust In-Only</td>
<td>BC</td>
<td>SE</td>
<td>in</td>
<td>on in</td>
</tr>
<tr>
<td>Out-In</td>
<td>SE</td>
<td>BC</td>
<td>out, in</td>
<td>replaces in</td>
</tr>
<tr>
<td>In-Out</td>
<td>BC</td>
<td>SE</td>
<td>in, out</td>
<td>replaces out</td>
</tr>
<tr>
<td>Out Optional-In</td>
<td>SE</td>
<td>BC</td>
<td>out, in</td>
<td>on out or in</td>
</tr>
<tr>
<td>In Optional-Out</td>
<td>BC</td>
<td>SE</td>
<td>in, out</td>
<td>on in or out</td>
</tr>
</tbody>
</table>

### 6.2 Normalized Message

#### 6.2.1 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 Service are in normalized form. Normalization is a two-way street:

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

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

### 6.2.2 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 meta-data 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.

[Note to reader: The definition and use of meta-data is under consideration by the Expert Group. Early Draft review input is most welcome.]

### 6.2.3 Processing Methods

The Normalized Message abstraction must not prescribe a specific processing style for the message content. A consumer of message content should be 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 user-facing engine, on the other hand, may wish to process the content using a DOM Document.

### 6.2.4 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 NMS 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.
[Note to reader: The relationship with work being done in the JAX-RPC 2.0 Expert Group is under review. This will be more fully specified in a future release of the specification.]
6.3 Quality of Service

6.3.1 Reliability

JBI is a solution in the enterprise computing arena, thus it requires a high-level of reliability. From the NMS point of view, this means that the message delivery design needs to address features that allow for at least a minimum level message reliability. The NMS 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.

[Note to reader: The requirements on the NMS in this regard are under consideration by the Expert Group. Early Draft review input is most welcome.]

6.3.2 Transactional

Transactions are often used in the construction of applications in the JBI target space. Many of the constituent pieces already use transactions, thus allowing these pieces to compose their functionality in the context of a transaction should allow for reliable operation of the composed pieces. The NMS facilitates the composition of business processing and communication. The NMS may or may not include transactional pieces itself.

The unit of work in the NMS is a Message Exchange. This is also the unit of work when a transaction is being used. The NMS is considered to be transaction aware. If the NMS is told that the Message Exchange is being controlled by a transaction, the NMS 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.

[Note to reader: The relationship between JBI and JTA within J2EE is under consideration by the Expert Group.]

6.3.3 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.

[Note to reader: This concept is under consideration by the Expert Group. Please also see below.]

6.3.4 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.

[Note to reader: This concept is under consideration by the Expert Group. Do you see a need for persistent, recoverable message exchanges? Is application-level error recovery (upon message exchange failure) sufficient, or should persistence and recoverability be required as well? Perhaps this should be addressed after JBI 1.0?]

6.3.5 Privacy/Confidentiality

Confidentiality 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 NMS this comes about if the NMS 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.

6.4 NMS Design

This section provides detailed design information on the Normalized Message Service. The contracts between the bindings/engines and the NMS are detailed.

6.4.1 Registration

As stated previously, registration is the process by which SEs declare services and BCs declare endpoints with the NMS. Each declaration must be accompanied by a corresponding meta-data definition, which describes the declaration. JBI has decided to standardize around WSDL 2.0, so the NMS presumes (but does not enforce) that all meta-data is compliant with WSDL 2.0.
6.4.1.1 BC Registration
A domain model for BC endpoint registration is pictured below. Each endpoint registration by a BC must be accompanied by meta-data describing the service, endpoint, and binding details for the endpoint.

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 Endpoint Registration Message Sequence Chart

- BC creates an implementation of NMS Resolver interface to provide meta-data describing endpoint registrations.
- BC registers its Resolver implementation once with the NMS.
- BC registers an endpoint name.
- NMS initializes the Endpoint reference.
- SE queries the NMS for endpoint by name (or some other criteria).
- SE uses reference to query NMS for meta-data describing the Endpoint.
- NMS invokes the BCs Resolver to fetch the appropriate meta-data.
- SE parses the Descriptor returned by the NMS.
6.4.1.2 SE Registration

A domain model for SE service registration is pictured below. Each service registration by a BC must be accompanied by meta-data describing the service, interface, and type (message) details for the service.

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 service registration and message exchanges.

- SE creates an implementation of NMS Resolver interface to provide meta-data describing service registrations.
- SE registers its Resolver implementation once with the NMS.
- SE registers a service name.
- NMS initializes the Service reference.
- BC queries the NMS for service by name (or some other criteria).
- BC uses reference to query NMS for meta-data describing the Service.
- NMS invokes the SEs Resolver to fetch the appropriate meta-data.
- BC parses the Descriptor returned by the NMS.
SE Service Registration Message Sequence Chart
6.4.2 Exchanging Messages

This section provides an in-depth discussion of the eight standard MEPs supported in JBI. It is important to note that the NMS provides for an extensible model of message exchange. JBI vendors that wish to extend the standard catalogue of eight MEPs are free to do so.

Diagram Key

- message
- fault
- status

6.4.2.1 Out-Only Message Exchange

Notes:
- SE initiates with message.
- BC responds with status to complete exchange.

6.4.2.2 In-Only Message Exchange
Notes:

- BC initiates with message.
- SE responds with status to complete exchange.

### 6.4.2.3 Robust Out-Only Message Exchange

Notes:

- SE initiates with message.
- BC may respond with status or fault.
- If BC response is status, the exchange is complete.
- If BC response is fault, SE responds with status to complete exchange.
6.4.2.4 Robust In-Only Message Exchange

**Notes:**
- BC initiates with **message**.
- SE may respond with **status** or **fault**.
- If SE response is **status**, the exchange is complete.
- If SE response is **fault**, BC responds with **status** to complete exchange.

6.4.2.5 Out-In Message Exchange

**Notes:**
- SE initiates with **message**.
- BC responds with **message** or **fault**.
- SE responds with **status**.
6.4.2.6 In-Out Message Exchange

Notes:
- BC initiates with message.
- SE responds with message or fault.
- BC responds with status.

6.4.2.7 Out Optional-In Message Exchange

Notes:
- SE initiates with message.
- BC may respond with message, fault, or status.
- If BC responds with status, the exchange is complete.
- If BC responds with fault, SE must respond with status.
- If BC responds with **message**, SE may respond with **fault** or **status**.
- If SE responds with **fault**, BC must respond with **status** to complete exchange.

### 6.4.2.8 In Optional-Out Message Exchange

**Notes:**
- BC initiates with **message**.
- SE may respond with **message**, **fault**, or **status**.
- If SE responds with **status**, the exchange is complete.
- If SE responds with **fault**, BC must respond with **status**.
- If SE responds with **message**, BC may respond with **fault** or **status**.
- If BC responds with **fault**, SE must respond with **status** to complete exchange.
6.4.3 Reliability in Message Exchange

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

Assumptions:

- Messages are idempotent
- If the initiator doesn't receive and end-end indication of completion (success/failure) then the communication primitive the initiator will retry the communication 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.

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

6.4.3.1 One-way with Fault

![Diagram of One-way with Fault]

Notes:

- Code in or attached to BC queues a message for delivery. 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.
- The BC picks a message to attempt delivery.
- BC send message to NMS.
- SE accepts message from NMS.
- 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.
- SE sends status to NMS. SE can reject the message (I.E. Structural, content or age problem) or accept it, in either case the outcome is returned to the initiator.
• BC accepts status from NMS.
• BC removes message from delivery queue. After this point in time no attempt to redeliver the message will be made.
6.4.3.2 Request-Response

Notes:

- Code in or attached to BC queues a message for delivery. 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.
- The BC picks a message to attempt delivery.
- BC send message to NMS.
- SE accepts message from NMS
- SE saves the response. The SE could also check to see if the message has been seen before if follow-on logic can't tolerate duplicate messages.
- SE sends response to NMS. SE can reject the message (I.E. Structural, content or age problem) or accept it, in either case the outcome is returned to the initiator.
- BC accepts response from NMS.
- BC saves response
- BC sends status of response to NMS
- SE get status from the NMS
- SE can cleanup any leftovers since it has positive acknowledgment of acceptance.
- BC performs any cleanup.

6.4.3.3 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).

6.4.4 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 NMS needs to perform.

6.4.4.1 One-Way

Notes: (This is the Out-Only side.)
- BC starts a transaction. It may use its own transactional resources.
- BC creates a Message Exchange and set the transaction on the Message Exchange.
- BC send() ME to NMS.
- NMS enlists its own internal resource.
- NMS saves the ME.
- NMS delists its own internal resource
- BC commits the transaction. BC could do some more work with its own resources before the commit and after the send().
Notes: (This is the In-Only side.)

- SE call accept to get a ME.
- NMS starts a transaction and enlists its resource manager.
- NMS get the ME.
- NMS delist the resource manager.
- BC call setStatus to say that its finished with the MEP. BC can enlist BC resources and perform operations against these.
- BC commits the transaction.

6.4.4.2 One-Way with Fault

Notes:

- BC starts a transaction and enlists it local resource.
- BC can work with the local resource and then send the ME.
- SE calls except, the transaction has been switched to the SE thread.
- SE does whatever it wants with the message. It can enlist and delist local resource it needed.
- SE sends the status back.
- BC accepts the status. The transaction context has been switched back to the BC thread.
- BC can do other work with its local transaction and the commits.
6.4.4.3 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.

- BC creates a transaction.
- BC enlists and local resources and sends the message.
- SE accept the ME and the transaction context moves to the SE thread.
- SE processes the request, enlist local resources if it needs.
- SE delists any local resources and sends the response.
- BC accepts the response and the transaction context is switched to the BC thread.
- BC can use local resources and send the status to the SE.
- SE accepts the status. Transaction context is not available, probably not much to do here.
- SE can do anything it likes but can't effect the transaction.
- BC can use local resources and then delists and commits the transaction.

6.4.4.4 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.

6.5 NMS API

The NMS exposes a standard API to SEs and BCs.

6.5.1 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)

### 6.5.1.1 NormalizedMessage

**public interface** NormalizedMessage

Represents a JBI Normalized Message.

#### Method Summary

<table>
<thead>
<tr>
<th>Class/Type</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>javax.xml.transform.Source</td>
<td>getContent()</td>
<td>Retrieve the content of the message.</td>
</tr>
<tr>
<td>Object</td>
<td>getProperty(String name)</td>
<td>Retrieve a message property.</td>
</tr>
<tr>
<td>java.util.Iterator</td>
<td>getPropertyNames()</td>
<td>Lists all properties by name for this message.</td>
</tr>
<tr>
<td>void</td>
<td>setContent(Source content)</td>
<td>Set the content of the message.</td>
</tr>
<tr>
<td>void</td>
<td>setProperty(String name, Object value)</td>
<td>Set a message property.</td>
</tr>
</tbody>
</table>
6.5.1.2 Fault
public interface Fault
extends NormalizedMessage
Models WSDL fault messages.

6.5.1.3 MessagingException
public class MessagingException
extends javax.jbi.JBIException
Generic exception used to report messaging related errors in the Normalized Message Service.

<table>
<thead>
<tr>
<th>Constructor Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>MessagingException(java.lang.String msg)</td>
</tr>
<tr>
<td>MessagingException(java.lang.String msg, java.lang.Throwable cause)</td>
</tr>
<tr>
<td>MessagingException(java.lang.Throwable cause)</td>
</tr>
</tbody>
</table>
6.5.2 Service API

This API allows SEs to interact with the NMS.
6.5.2.1 DeliveryChannel
public interface DeliveryChannel

Bi-directional communication channel used to interact with the Normalized Message Service.

[Note to reader: This concept is under consideration by the Expert Group. Should the Delivery Channel concept exist in the JBI model, or should components use a direct NMS API for creating, sending, and receiving message exchanges? If Delivery Channels should exist, should multiple instances per component be allowed? How should such instances be distinguished?]

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>MessageExchange</td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>MessageExchange</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Descriptor</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

6.5.2.2 BindingChannel
public interface BindingChannel
extends DeliveryChannel

Delivery channel for a JBI binding component (BC). A BindingChannel is used to initiate inbound message exchanges.

[Note to reader: This concept is under consideration by the Expert Group. Should the Delivery Channel concept exist in the JBI model, or should components use a direct NMS API for creating, sending, and receiving message exchanges? If Delivery Channels should exist, should multiple instances per component be allowed? How should such instances be distinguished?]
<table>
<thead>
<tr>
<th>Method Summary</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EndpointReference</td>
<td><code>activateInboundEndpoint(javax.xml.namespace.QName service, java.lang.String endpoint)</code></td>
<td>Used by a BC to register an inbound endpoint.</td>
</tr>
<tr>
<td>EndpointReference</td>
<td><code>activateOutboundEndpoint(javax.xml.namespace.QName service, java.lang.String endpoint)</code></td>
<td>Used by a BC to register an outbound endpoint.</td>
</tr>
<tr>
<td>InOnly</td>
<td><code>createInOnlyExchange()</code></td>
<td>Creates an InOnly message exchange.</td>
</tr>
<tr>
<td>InOptionalOut</td>
<td><code>createInOptionalOutExchange()</code></td>
<td>Creates an In Optional Out message exchange.</td>
</tr>
<tr>
<td>InOut</td>
<td><code>createInOutExchange()</code></td>
<td>Creates an In Out message exchange.</td>
</tr>
<tr>
<td>RobustInOnly</td>
<td><code>createRobustInOnlyExchange()</code></td>
<td>Creates a Robust In Only message exchange.</td>
</tr>
<tr>
<td>ServiceReference</td>
<td><code>locateService(EndpointReference ref)</code></td>
<td>Queries the NMS for a service reference related to the specified endpoint.</td>
</tr>
<tr>
<td>Descriptor</td>
<td><code>getServiceDescriptor(ServiceReference svcRef)</code></td>
<td></td>
</tr>
</tbody>
</table>
6.5.2.3 EngineChannel

public interface EngineChannel
extends DeliveryChannel

Delivery channel for a JBI service engine (SE). An EngineChannel is used to initiate outbound message exchanges.

[Note to reader: This concept is under consideration by the Expert Group. Should the Delivery Channel concept exist in the JBI model, or should components use a direct NMS API for creating, sending, and receiving message exchanges? If Delivery Channels should exist, should multiple instances per component be allowed? How should such instances be distinguished?]

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>ServiceReference activateInboundService(javax.xml.namespace.QName service)</td>
</tr>
<tr>
<td>Used by an SE to register an inbound service.</td>
</tr>
<tr>
<td>ServiceReference activateOutboundService(javax.xml.namespace.QName service)</td>
</tr>
<tr>
<td>Used by an SE to register an outbound service.</td>
</tr>
<tr>
<td>EndpointReference assignEndpoint(ServiceReference ref)</td>
</tr>
<tr>
<td>Request the NMS to assign an endpoint using implementation-specific policy.</td>
</tr>
<tr>
<td>EndpointReference[] availableEndpoints(ServiceReference ref)</td>
</tr>
<tr>
<td>Queries the NMS for endpoints registered against the specified service.</td>
</tr>
<tr>
<td>Descriptor getEndpointDescriptor(EndpointRef endpoint)</td>
</tr>
<tr>
<td>OutIn createOutInExchange()</td>
</tr>
<tr>
<td>Creates an Out In message exchange.</td>
</tr>
<tr>
<td>OutOnly createOutOnlyExchange()</td>
</tr>
<tr>
<td>Creates an OutOnly message exchange.</td>
</tr>
<tr>
<td>OutOptionalIn createOutOptionalInExchange()</td>
</tr>
<tr>
<td>Creates an Out Optional In message exchange.</td>
</tr>
<tr>
<td>RobustOutOnly createRobustOutOnlyExchange()</td>
</tr>
<tr>
<td>Creates a Robust Out Only message exchange.</td>
</tr>
</tbody>
</table>

6.5.2.4 Reference

public interface Reference

Represents a logical identifier for a component registration with the NMS. A reference may be used for routing message exchanges as well as querying meta-data about a specific reference type.

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.String getOwnerId()</td>
</tr>
</tbody>
</table>
6.5.2.5 ServiceReference
public interface ServiceReference

Reference generated by the NMS to refer to a service registration.

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>QName serviceName()</td>
</tr>
<tr>
<td>Returns the service name of this reference.</td>
</tr>
</tbody>
</table>

6.5.2.6 EndpointReference
public interface EndpointReference

Reference generated by the NMS to refer to an endpoint registration.

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.String endpointName()</td>
</tr>
<tr>
<td>Returns the endpoint name of this reference.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>QName serviceName()</td>
</tr>
<tr>
<td>Returns the service name of this reference.</td>
</tr>
</tbody>
</table>

6.5.2.7 Descriptor
public interface Descriptor

Basic interface for meta-data information.

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.w3c.dom.Document description()</td>
</tr>
<tr>
<td>Retrieves a DOM representation containing meta-data which describes an entity registered with NMS.</td>
</tr>
</tbody>
</table>

6.5.2.8 Resolver
public interface Resolver

Used by the NMS to resolve meta-data describing a specific reference (e.g. Endpoint, Service, etc.).

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Descriptor resolveReference(Reference ref)</td>
</tr>
<tr>
<td>Resolves descriptor details for the specified reference.</td>
</tr>
</tbody>
</table>
6.5.2.9 MessageExchange

public interface MessageExchange

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.

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fault</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>NormalizedMessage</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>EndpointReference</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>java.lang.Exception</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>java.lang.String</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Fault</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>NormalizedMessage</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>java.lang.String</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>java.net.URI</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>java.lang.Object</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>ServiceReference</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>ExchangeStatus</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
6.5.2.10 OutOnly
public interface OutOnly
extends MessageExchange

Supports operations used to process an Out Only MEP to completion.

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>NormalizedMessage</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

6.5.2.11 RobustOutOnly
public interface RobustOutOnly
extends MessageExchange

Supports operations used to process an Robust Out Only MEP to completion.

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>NormalizedMessage</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

6.5.2.12 OutIn
public interface OutIn
extends MessageExchange

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

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>NormalizedMessage</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>NormalizedMessage</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
6.5.2.13 OutOptionalIn

public interface OutOptionalIn
extends MessageExchange

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

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>NormalizedMessage getInMessage() Retrieves the &quot;in&quot; message reference from this exchange.</td>
</tr>
<tr>
<td>NormalizedMessage getOutMessage() Retrieves the &quot;out&quot; message reference from this exchange.</td>
</tr>
<tr>
<td>void setInMessage(NormalizedMessage msg) Specifies the &quot;in&quot; message reference for this exchange.</td>
</tr>
<tr>
<td>void setOutMessage(NormalizedMessage msg) Specifies the &quot;out&quot; message reference from this exchange.</td>
</tr>
</tbody>
</table>

6.5.2.14 InOnly

public interface InOnly
extends MessageExchange

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

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>NormalizedMessage getInMessage() Retrieves the in normalized message from this exchange.</td>
</tr>
<tr>
<td>void setInMessage(NormalizedMessage msg) Sets the in normalized message for this exchange.</td>
</tr>
</tbody>
</table>

6.5.2.15 RobustInOnly

public interface RobustInOnly
extends MessageExchange

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

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>NormalizedMessage getInMessage()</td>
</tr>
<tr>
<td>void setInMessage(NormalizedMessage msg)</td>
</tr>
</tbody>
</table>
6.5.2.16 InOut

public interface InOut
extends MessageExchange

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

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>NormalizedMessage</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>NormalizedMessage</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

6.5.2.17 InOptionalOut

public interface InOptionalOut
extends MessageExchange

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

<table>
<thead>
<tr>
<th>Method Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>NormalizedMessage</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>NormalizedMessage</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
6.5.2.18 ExchangeStatus

public class ExchangeStatus
extends java.lang.Object

Typesafe enumeration containing status values for a message exchange.

<table>
<thead>
<tr>
<th>Field Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>static ExchangeStatus ACTIVE</td>
</tr>
<tr>
<td>Indicates that an ME has not been processed to completion.</td>
</tr>
<tr>
<td>static ExchangeStatus DONE</td>
</tr>
<tr>
<td>Indicates that an ME has been processed to completion.</td>
</tr>
<tr>
<td>static ExchangeStatus ERROR</td>
</tr>
<tr>
<td>Indicates that an ME has terminated abnormally within the JBI environment.</td>
</tr>
</tbody>
</table>

6.5.3 Usage Examples

This section provides detailed examples on common BC, SE, and NMS interactions. The areas of runtime registration and message exchange are covered in detail.

6.5.3.1 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:

```java
QName SERVICE1 = new QName("http://abc.com/services", service1");
```

6.5.3.1.1 String ENDPOINT1 = "soap1"; SE Registration

```java
class ConsumerEngine {
    EngineChannel engineChannel;
    ServiceReference serviceRef;

    public void init(EngineContext engCtx)
```
throws JBIException
{
    // Obtain reference to delivery channel
    engineChannel = engCtx.getEngineChannel();

    // Create a Resolver impl for meta-data and attach to channel
    Resolver myMetaData = new MyResolver();
    channel.setResolver(myMetaData);

    // Register service name
    serviceRef = channel.activateOutboundService(SERVICE1);
}

6.5.3.1.2 BC Registration

class SoapBinding
{
    BindingChannel bindingChannel;
    EndpointReference endpointRef;

    public void init(BindingContext bindCtx)
        throws JBIException
    {
        // Obtain reference to delivery channel
        bindingChannel = bindCtx.getBindingChannel();

        // Create a Resolver impl for meta-data and attach to channel
        Resolver myMetaData = new MyResolver();
        channel.setResolver(myMetaData);

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

6.5.3.1.3 Message Exchange

Engine View

void someEngineMethod()
{
    OutOnly outOnly;
    NormalizedMessage message;
// Create OutOnly
outOnly = engineChannel.createOutOnlyExchange();

// Create new message
message = outOnly.createMessage();

// Populate message with data (omitted)

// Set message as out reference in exchange
outOnly.setOutMessage(message);

// Set operation and service details
outOnly.setOperation("updateStatus");
outOnly.setService(serviceRef);

// Assign endpoint using NMS policy
EndpointReference endpoint = engineChannel.assignEndpoint(serviceRef);
outOnly.setEndpoint(endpoint);

// Initiate the message exchange
engineChannel.send(outOnly);
}

**Binding View**

```java
void someBindingMethod()
{
    MessageExchange exchange = bindingChannel.accept();
    process(exchange);
}

void process(OutOnly outOnly)
{
    NormalizedMessage outMsg;

    // fetch out message
    outMsg = outOnly.getOutMessage();

    // process data
    try
```
6.5.3.2 Service Provider (Servicer)

This example takes place within a single JBI environment. A service engine, ProviderEngine, provides a web service to external consumers (i.e. inbound service invocation).

Notes:

- Service name: {http://xyz.com/services} : service2
- Endpoint name: soap2
- 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:

```java
QName SERVICE2 = new QName("http://xyz.com/services", service2");
String ENDPOINT2 = "soap2";
```

6.5.3.2.1 BC Registration

```java
class SoapBinding {
    BindingChannel bindingChannel;
    EndpointReference endpointRef;

    public void init(BindingContext bindCtx) throws JBIException {
        // Obtain reference to delivery channel
```
bindingChannel = bindCtx.getBindingChannel();

// Create a Resolver impl for meta-data and attach to channel
Resolver myMetaData = new MyResolver();
channel.setResolver(myMetaData);

// Register endpoint name
endpointRef = channel.activateInboundEndpoint(SERVICE2, ENDPOINT2);

6.5.3.2.2 SE Registration

class ConsumerEngine
{
    EngineChannel   engineChannel;
    ServiceReference serviceRef;

    public void init(EngineContext engCtx)
        throws JBIException
    {
        // Obtain reference to delivery channel
        engineChannel = engCtx.getEngineChannel();

        // Create a Resolver impl for meta-data and attach to channel
        Resolver myMetaData = new MyResolver();
        channel.setResolver(myMetaData);

        // Register service name
        serviceRef = channel.activateInboundService(SERVICE2);
    }
}

6.5.3.2.3 Message Delivery

Binding View

void someBindingMethod()
{
    InOut             inOut;
    NormalizedMessage inMsg;

    // Receive message over native protocol (omitted)

    // Create InOut exchange
inOut = bindingChannel.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("getStockQuote");
inOut.setEndpoint(endpointRef);

// Use NMS to locate associated service reference
ServiceReference service = bindingChannel.locateService(endpointRef);
inOut.setService(service);

// Initiate the message exchange
bindingChannel.send(inOut);

// Wait for response message
MessageExchange exchange = bindingChannel.accept();
process(exchange);
}

void process(InOut inOut)
{
    NormalizedMessage outMsg = inOut.getOutMessage();

    if (outMsg == null)
    {
        // check status, potential error
    }

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

Engine View

void someEngineMethod()
{  
    MessageExchange exchange = engineChannel.accept();  
    process(exchange);  
}

void process(InOut inOut)  
{  
    NormalizedMessage inMsg;  
    NormalizedMessage outMsg;  
    // fetch in message  
    inMsg = inOut.getInMessage();

    // process data  
    try  
    {  
        // invoke appropriate process, etc. (omitted)  
        // create response message  
        outMsg = inOut.createMessage();  

        // populate message content (omitted);  
        // attach message to exchange  
        inOut.setOutMessage(outMsg)  
    }  
    catch (Exception ex)  
    {  
        inOut.setError(ex);  
        outOnly.setStatus(ExchangeStatus.ERROR);  
    }  

    engineChannel.send(inOut);  
}
7 Administration

[Note to reader: This is under consideration by the Expert Group. Should JBI attempt to standardize selected management functions of the JBI environment and the plug-in components it hosts? If so, which functions are appropriate?

- Life cycle (start, stop) of components and the JBI environment as a whole.
- Installation (and un-installation) of plug-in components (and shared name spaces)?
- Deployment (and un-deployment) of service-related artifacts to components?
- Monitoring of state and performance of components?

Is JMX the appropriate technology to support JBI management? Concerning JMX remoting: what is the best way to handle localization issues (where the client locale <> JBI locale)? Or is this an unrealistic scenario?

What follows is a description that illustrates some possibilities based on assumed answers to the question posed above.]

As mentioned in the architecture overview section, the JBI environment is administered using JMX. This includes the plug-in bindings and engines, which must provide specified management bean interfaces (MBeans). In addition, JBI system components must provide specified MBeans to facilitate management of the JBI environment-provided infrastructure.

The major administrative use cases are:

- Install (BC, SE, shared name space; customized)
- Operate (start, stop, shut down)
- Deployment (deploy, undeploy)
- Uninstall (BC, SE, shared name space)

The JBI administrator may install a binding (or engine) jar file interactively, via a JMX client, or non-interactively via Apache Ant scripts. The administrator may interact with an optional Installation Extension MBean if provided by the binding (or engine) (for example: providing an “Installation Configuration” MBean to set a proxy URL needed during the installation). The binding (or engine) developer can provide an ant script to help automate the installation (that is, the administrator can edit the ant script with the proxy URL and use ant to invoke methods on the Installation Configuration MBean).

The JBI administrator may install a shared name space jar file interactively, via a JMX client, or non-interactively via Ant scripts.

The JBI administrator may start, stop, or shut down a binding (or engine) interactively, via a JMX client, or non-interactively via Ant scripts.

The JBI administrator may uninstall a binding, engine, or shared name space interactively, via a JMX client, or non-interactively via Ant scripts.

7.1 BC and SE Services Provided to JBI
Bindings and engines MUST supply the following for use by the JBI environment:

- Bootstrap class - to alter the class path and/or provide an extension MBean
- Life cycle class - to support operations and to provide optional MBeans.

Bindings and engines MAY supply the following for use by the JBI environment:

- Installation Extension MBean - provided by Bootstrap
- Deployer MBean - provided by Life cycle
- Life cycle Extension MBean - provided by Life cycle

The binding or engine developer must provide a bootstrap class and a life cycle class along with implementation classes packaged in a single jar file, containing a jbi.xml descriptor to identify the classes.

The binding or engine developer may optionally provide an installation extension MBean (by creating it in the bootstrap init method and registering it with JMX). For example, if an installation requires network access, it may need to know a proxy URL; this can be set by the administrator via an installation configuration MBean provided by the bootstrap as an extension to the installation MBean provided by the JBI framework.

The binding or engine developer may optionally provide a deployer MBean that is invoked when there are data to be deployed to the binding or engine (usually as part of larger deployment).

The binding or engine developer may optionally provide an extension MBean that can be used to provide additional controls or states for the binding or engine, or to locate related MBeans.

The following sections describe how the above services function in compliant JBI implementations. Unless otherwise noted, all behaviors and responsibilities described for JBI implementations are REQUIRED. Note that the implementation is sometimes referred in the message sequence diagrams. This does not imply that implementations must modularized in the fashion suggested by these diagrams; this is used to conveniently separate JMX-based interfaces from implementations of those interfaces. This reference implementation refers to this part of the implementation as framework/services.

### 7.2 Binding or Engine Installation

To interactively install a binding or engine via JMX client requires four steps:

- Load the jar file containing the engine or binding distribution.
- Initialize Installer (a.k.a. binding or engine bootstrap)
  - with optional Extension MBean, see the section “Customized Installation”, below.
- Run Installer
- Post Installation Initialization (also performed after JBI System restarts)

These steps are detailed in the following subsections.

#### 7.2.1 Load Jar
The administrator prepares to install a binding or engine.

1. Load Installer - administrator provides jar file for the binding or engine to be installed to the JMX client.
   a) Client forwards “load installer, jar” request to JMX server
   b) Server forwards request to management service JBI Management Installer Service MBean (JMIS)
   c) JMIS invokes management installation service to load specified jar.
   d) Installation service creates an installation context for this installation jar. This includes “exploding” the JAR such that relative class path items will reference the contents of the jar.
   e) Installation service invoke an implementation-provided service to load bootstrap from passed context
   f) Implementation service gets information from the context
   g) Implementation instantiates and initializes the binding (or engine) Bootstrap (see below).
7.2.2 Initialize Installer

The implementation prepares the installer.

g) Implementation/services instantiates an instance of the Bootstrap class, then invokes its init method, passing it the installation context.

h) Bootstrap gets information from installation context

i) Bootstrap gets information from environment context

j) (optional) Bootstrap creates Installation Extension MBean (IX)

k) (optional) Bootstrap registers IX. (note: see “Customized Installation” and “Optional Installation Extension”)

l) Implementation/services creates a JBI Installation MBean (JI).

m) Implementation registers JI MBean with JMX.
7.2.3 Run Installer

The administrator runs the installer.

2. Run Installer - administrator uses JMX client to complete the installation.
   a) Client forwards installation request to JMX server.
   b) Server forwards request to JBI Installation MBean (JI).
   c) JI forwards request to implementation/services.
   d) Implementation/services invokes the Bootstrap `onInstall` method, passing it the environment context
   e) Bootstrap updates environment context (as needed)
   f) (optional) Bootstrap de-registers any previously registered MBean.
   g) (optional) Implementation/services initializes installed binding (or engine)
### 7.2.4 Post installation (and JBI system restart)

The implementation initializes (or reinitializes) the installed binding (or engine).

- **g)** Implementation/services instantiates binding (or engine) life cycle and invokes its `init` method, passing it the environment context.
- **h)** LifeCycle gets information from the environment context.
- **i)** *(optional)* LifeCycle creates a LifeCycle Deployment MBean (LCD MB)
- **j)** *(optional)* LifeCycle registers the LCD MB with JMX
- **k)** *(optional)* LifeCycle creates a LifeCycle Extension MBean (LCX MB)
- **l)** *(optional)* LifeCycle registers the LCX MB with JMX
- **m)** Implementation/services creates the JBI LifeCycle MBean (JLC MB).
- **n)** Implementation/services registers the JLCMB with JMX.
7.3 Customized Engine or Binding Installation

To interactively install a binding or engine that provides an installation customization MBean, the administrator uses the MBean before running the installer.

1. Find Installation Extension MBean - administrator invokes \texttt{getExtensionMBean} on the JBI Installation MBean (JI) via JMX client
   a) Client forwards request to JMX server
   b) Server invokes JI
   c) JI invokes implementation/services
   d) Implementation/services invokes Bootstrap \texttt{getExtensionMBean} method and returns the Installation Extension MBean (IX).

2. Use Installation Extension MBean - administrator invokes setters on the IX via JMX client.
   e) Client forwards request to JMX server
   f) Server invokes IX
   g) IX invokes methods on Bootstrap
   h) Bootstrap may harden data in the binding (or engines) installation root, which is determined by the bootstrap by calling the environment context.
7.4 Binding or Engine Scripted Installation

A JBI implementation is REQUIRED to support scripted installation, using Apache Ant as a scripting language. This is accomplished using JMX tasks for Apache Ant. Such tasks invoke management beans via the JMX server, in much the same fashion as the interactive installation use case.

To run a script that installs a binding or engine, with or without an installation customization MBean:

1. Install via Ant script - administrator runs ant script to install binding (or engine) jar file.
   - Load Installer
   - Find Extension MBean (optional)
   - Configure via Extension MBean (optional)
   - Run Installer
7.5 Shared Name Space Installation

The administrator may interactively install a shared name space (or use ant for a noninteractive installation).

1. Load Shared Name Space - administrator provides a shared name space jar to be installed, via JMX client
   a) Client forwards request to JMX server
   b) Server forwards request to JBI Management Installation Service MBean (JMIS)
   c) JMIS forwards request to the implementation's Management Installation Service
   d) Installation Service forwards request to implementation/services

7.6 Operation

The administrator can interactively operate installed bindings (and engines) via a JMX client, or non-interactively via Ant script. The operations are:

1. Start - put the binding (or engine) into the running state
2. Stop - put the binding (or engine) into the stopped state
3. Shutdown - put the binding into the installed state (enables uninstall)
7.7 Operational States

The operational states required for bindings and engines are shown in illustration.

EXAMPLE: optional
JMX managed
‘Running’ sub-state
7.7.1 Operations That Change State

To interactively, or via a script, start, stop, or shut down an installed binding or engine.

The calling sequence for start, stop, and shut down is very similar:

1. Start/Stop/Shut Down BC (or SE)

<table>
<thead>
<tr>
<th>JMX Client</th>
<th>JMX Server</th>
<th>JLC MBean</th>
<th>Impl. services</th>
<th>LifeCycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td></td>
</tr>
</tbody>
</table>

1. Start/Stop/Shut Down - the administrator picks one of the JBI Life Cycle MBean (JLC) methods and requests the operation via a JMX client.
   a) Client forwards request to JMX server
   b) Server forwards request to JLC
   c) JLC forwards request to implementation/services
   d) Implementation/services invokes the Life Cycle start, stop, or shutdown method.

After a shutdown in performed, the implementation must remove all references to the life cycle object used, allowing it to be garbage-collected.

Variations:

- **Stop followed by start** (restart)
  
  To restart an installed binding (or engine), the administrator issues a stop command followed by a start command. This uses the life cycle stop and start methods to change the state of the binding (or engine) from running to stopped back to running.

- **Start followed by shut down** (implicit stop)
  
  If the administrator requests a shut down for a started binding (or engine) the implementation must first call the life cycle stop method to transition the binding (or engine) from the running state to the stopped state. After this implicit stop is performed, the implementation must call the shutdown method to transition the binding (or engine) to the installed state.

- **Shutdown followed by start** (implicit init)
  
  If the administrator requests a start when the binding (or engine) has been previously shut down, the implementation must create a new life cycle object for the binding (or engine),
call the `init` method on it to place it in the stopped state, and then call the `start` method to put it in the running state.

### 7.7.2 Extension MBean Example: Pausing/Resuming (Sub-states)

Extension MBeans can be used to introduce additional, SE/BC-specific states to the life cycle. This example illustrates how such an extension MBean could be used.

1. Finding Extension MBean - Administrator uses JBI Life Cycle MBean (LC) via a JMX client.
   a) Client forwards `getExtensionMBean` request to JMX server
   b) Server forwards request to JLC
   c) JLC forwards request to RI implementation/services
   d) Implementation calls Life Cycle `getExtensionMBean` method.

2. Pause/resume binding (or engine) - Administrator uses Life Cycle Extension MBean (LCX) via a JMX client.
   e) Client forwards request to JMX server
   f) Server forwards request to LCX (pause, resume)
   g) LCX forwards request to pause (or resume) method on Life Cycle

### 7.7.3 JBI System

The administrator can start/stop the JBI system (in a managed environment such as an application server, JBI should run as a sub-component of an application server, and start (and
start) with the application server, not independently). The start (or restart) JBI (implementation) processing ensures that all installed bindings and engines are initialized and started. System stop JBI (implementation) processing ensures that all running bindings and engines are stopped and then shut down.

1. Start JBI System
1a1-3. Create a LifeCycle instance for each installed binding and engine and initialize each via LifeCycle.init

1b1-3. Start each installed binding and engine via LifeCycle.start

The order that the bindings and engines are initialized and started in is implementation dependent.
2. Stop JBI system
   2a1-3 Stop each binding and engine via LifeCycle.stop
   2b1-3 Shut down each binding and engine via LifeCycle.shutDown and then remove the last reference to the LifeCycle instance (making it collectable)

Note that the order that engines and bindings are stopped and shut down is implementation dependent.

**Variations:**

1. **JBI Restart** (implicit init, start)
   
   When JBI is stopping, the implementation invokes all binding and engine life cycle **stop** methods, and then all **shutdown** methods, before it, and the rest of the JBI implementation, cleans up and stops.

   When JBI is starting, the implementation instantiates all binding and engine life cycle objects, invokes **init** on each, and then invokes **start** on each.

### 7.8 Service Component (SC) Deployment

To interactively, or via a script, deploy or undeploy to or from an installed binding or engine.

Note: Deployment must be performed in the SC's jbi.xml document order.

1. Administrator uses JMX client to issue a deploy command for a specified AU jar file.
   a. The JMX client uses JMX-remote to locate/invoke the JBI Management Administration Service MBean (JMAS MBean).
   b. The JMX server invokes the JMAS MBean for each SA in the specified SC:

   **repeat c-d for each target binding or engine**
c1. The JMAS Mbean locates the target binding (or engine) LifeCycle Deployment MBean (LCD MBean).

d1. The JMAS MBean sends the SC unique ID and the SC jar to the LCD MBean.

(repeat for c2, d2, ... cN, dN)

Note: a particular binding or engine can be invoked multiple times for one SC (multi-SAs)

7.9 Querying Installed Bindings, Engines, Name Spaces

To determine which components are installed and for the engines and bindings, what operational state they are in and which optional MBeans they provide.

Use-cases:

- List all installed bindings, engines, and name spaces
- Show the status of a specific binding, engines, or name space (by unique ID)

1. Administrator runs query (list: bindings, engines, name spaces, or all) ant script target
   a) The custom ant task use JMX-remote to forward the request to the JBM Management Administration Service MBean (JMAS MBean)
   b) The JMAS MBean delegates to the implementation.
   c) The implementation queries its internal records for the list of bindings, engines, and/or name spaces then determines their states: installed, stopped, running, or is in a transient state: starting or stopping.

Notes:

- Installed state is the result of a prior shut down
- Name spaces have no LifeCycle nor any concept of “running or stopped” states only the “installed” state.
- Listing may be done in any order.
- If a list command is processed during an uninstall, the state may be shown as “installed” or the binding (engine, name space) may not appear in the list at all.
- The administrator must directly query any optional extension MBean(s) to determine sub-states, if any (e.g. pausing, paused, resuming).

### 7.10 Querying Service Component (SC) Deployments

To determine which artifacts have been deployed to which components.

**Use-cases:**

- List all deployed SCs
- List all deployed SCs for a specified binding or engine (unique ID)
- List all bindings and engines for a specified SC (unique ID)

1. Administrator runs Query (list all SCs) ant script target
   a. Custom ant task uses JMX-Remote to locate/invoke JBI Management Administrative Service MBean (JMAS MBean)
   b. The JMX Server invokes the JMAS MBean: for each registered LifeCycle Deployment MBean (LCD MBean):
      c1. The JMAS MBean locates the LCD MBean
      d1. The JMAS MBean invokes the LCD MBean method to query all SCs
          (repeat for c2, d2, ... cN, dN)

Note: Listing may be done in any order.
7.11 Service Component Undeployment

To interactively, or via a script, undeploy artifacts from a previously deployed SC and all affected bindings and engines.

Note: Undeployment must be done in reverse SC deployment order (i.e., reverse jbi.xml document order).

1. Administrator uses JMX client to issue an undeploy command for a specified SC unique ID
   a. The jmx client uses jmxremote to locate/invoke the JBI Management Administration Service MBean (JMAS MBean).
   b. The JMX server invokes the JMAS MBean for each SA in the specified SC:
      c1. The JMAS Mbean locates the target binding (or engine) LifeCycle Deployment MBean (LCD MBean).
      d1. The JMAS MBean sends the SC unique ID and the SA jar to the LCD MBean.
          (repeat for c2, d2, ... cN, dN)

Notes:
  - A particular binding or engine can be undeployed to multiple times for one AU (multi-SAs)
  - The order of SA undeployments must be the reverse of the original SA deployments (that is, in reverse jbi.xml document order).
1. Administrator runs Query (list all SCs by specified binding or engine) ant script
   a) The custom ant task uses jmxremote to locate/invoke the JBI Management Administration Service MBean (JMAS MBean)
   b) The JMX server invokes the JMAS MBean
   c) The JMAS MBean locates the specified binding or engines's LifeCycle Deployment MBean (LCD MBean)
   d) The JMAS MBean invokes the LCD MBean query method to list all SCs/SAs
       Note: Listing may be done in SC deployment order and SA packaging order.
1 administrator runs query (list all bindings and engines for a specific SC) ant script target
   1a. custom ant task uses jmxremote to locate/invoke the JBI Management Administration Service MBean (JMAS MBean)
   1b. the JMX server invokes the JMAS MBean
   for each installed binding and engines:
      c1. the JMAS MBean locates the LifeCycle Deployment MBean (LCD MBean)
      d1. the JMAS MBean invokes the LCD MBean query method to list SCs
          (repeat for c2, d2, ... cN, dN)

Note: Listing may be done in installation order.
7.12 Binding, Engine, and Shared Namespace Uninstallation

Interactive uninstall is shown below:

0. **(precondition)** The administrator has shut down the binding (or engine) $X_0$ - the BindingLifeCycle (or EngineLifeCycle) has no references and is eligible for garbage collection.

1. **“Load:”** administrator uses JMX client (via JMX-remote) to request Load of an installation jar file
   a) JMX server invokes JBI Management Installation Service MBean (JMIS MBean)
   b) JMIS MBean delegates to implementation.
   c) Implementation instantiates Installer (BindingBootstrap or EngineBootstrap), calls \texttt{init}
   d) Implementation creates a temporary JBI Installation MBean proxy (JI MBean) for the specified binding (or engine) Bootstrap.
   e) Implementation registers the JI MBean

2. **“Uninstall:”** administrator uses JMX client (via JMX-remote) to find/invoke the JI MBean uninstall method:
   f) JMX server calls the JI MBean
   g) JI MBean forwards request to implementation (for before/after processing)
h) Implementation de-registers the JBI LifeCycle MBean proxy (JLC MBean).
   o  $X_1$ - the JLC MBean has no references and is eligible for garbage collection.

i) The implementation calls BindingBootstrap.uninstall (or EngineBootstrap.uninstall) to complete the set up for the uninstallation
   o  $X_2$ - the BindingBootstrap (or EngineBootstrap) has no references and is eligible for garbage collection.

j) The implementation calls the JBI Management Installation Service to release resources that were created during/after the prior installation of the binding (or engine). e.g. remove the “exploded” jar file contents from the JBI file system. (Note: the implementation can call the management code directly without using the MBean)

k) The implementation de-registers the JI Mbean
   o  $X_3$ - the JI MBean has no references and is eligible for garbage collection.

administrator runs the “uninstall binding (or engine)” ant script/target
   1. Custom ant task uses JMX-remote to request Load of installation jar file
   2. Custom ant task uses JMX-remote to request uninstallation of binding (or engine)

7.13 Packaging

This specification defines standard packaging formats for installable and deployable items.
All JBI-defined packages are Java archive files (JAR files), which must contain a file named as follows:

/META-INF/jbi.xml

The contents of this file are described below.
During the process of installation (or deployment) of packages, they will be “exploded” (unarchived) into the file store of the JBI environment. The location of this file store (called an installation or deployment root) is available using the appropriate context object. The root is also used to resolve relative class path item specified in the jbi.xml.

7.13.1 jbi.xml

A single XML document type, referred to as jbi.xml, is used to describe the contents of packages defined in this specification. Specifically, it is used to

- Describe a SE installation package.
- Describe a BC installation package.
- Describe a shared name space installation package.
- Describe a deployment package.

Each use of jbi.xml in the above packages MUST conform to the schema for jbi.xml, shown below in schema 1, “jbi.xml Packaging Descriptor XML Schema”. The schema is given in Relax-NG 1.0 compact notation.
The use of the schema for describing each type of JBI package will be detailed in the section defining the package, in this chapter.

default namespace = "http://java.sun.com/xml/ns/jbi"

start =
  element jbi {
    attribute version { xsd:decimal },
    element type { "binding" | "engine" | "service-component" | "sharedNameSpace" },
    ( binding | engine | service-component | sharedNameSpace )
  }

binding =
  element binding {
    identification,
    element componentClassName { attribute description { text }?, text },
    element componentClassPath { classpath },
    element bootstrapClassName { text },
    element bootstrapClassPath { classpath },
    element sharedNameSpaceId { text }*
  }

engine =
  element engine {
    identification,
    element componentClassName { attribute description { text }?, text },
    element componentClassPath { classpath },
    element bootstrapClassName { text },
    element bootstrapClassPath { classpath },
    element sharedNameSpaceId { text }*
  }

service-component =
  element service-component {
    identification,
    service-assembly+
  }

sharedNameSpace =
  element sharedNameSpace {
    element sharedNameSpaceId { text },
    element sharedNameSpaceClassPath { classpath }
  }

service-assembly =
  element service-assembly {
    identification,
    element target {
      element artifacts_jar { text },
      element componentId { text }
    }
  }

identification =
  element identification {
    element alias { xsd:NCName },
    element componentId { text },
    element description { text }
  }

classpath =
  (element pathelement { text })*
7.13.2 Installation Packaging

There are three types of installation packages defined by this specification: engine, binding, and shared name space. Engine and binding packages are sufficiently similar that they are defined together, in the following subsection.

7.13.2.1 BC / SE Installation Packaging

BC and SE installation packages MUST conform to this specification. The JBI installation service shall generate an error if non-conformance is detected at installation time.

Other than the package descriptor the installation package format is not defined by this specification. During installation the contents of the installation JAR will be “exploded” by the JMIS in the “installation root” assigned to the component. This is to ensure that the contents of the JAR are available to the installation class loader, using the class path given in the package descriptor defined below.

The package descriptor, jbi.xml, MUST conform to the packaging description definition given in section “jbi.xml”, above with the additional constraints given below:

- For bindings: The <jbi> document element may only contain a <binding> child element.
- For engines: The <jbi> document element may only contain an <engine> child element.

The <binding> (or <engine>) element contains the following information:

- One <identification> element, which supplies a unique component ID string, a “friendly” alias for that ID, and some descriptive text.
- One <bootstrapClassName> element, which gives the name of the Bootstrap class to be used during the installation process.
- One <bootstrapClassPath> element, giving the class path elements needed during execution of the bootstrap class.
- One <componentClassName> element, giving the class name of the Life Cycle class for the binding (or engine).
- One <componentClassPath> element, giving the class path elements needed when then the binding (or engine) is started (exclusive of shared name spaces).
- Zero or more <sharedNameSpaceID> elements, which indicate which shared name spaces are to be included in the binding's (or engine's) class path at when it is started.

The binding and engine installation packages are used by the installer service, as detailed in the section “Binding or Engine Installation”.

The bootstrap information is used by the JBI installer.
7.13.2.1.1 Example Binding Installation Package Descriptor

```xml
<?xml version="1.0"?>
<jbi version = "1.0"
 xmlns = "http://java.sun.com/xml/ns/jbi"
 xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation = "http://java.sun.com/xml/ns/jbi http://java.sun.com/xml/ns/jbi/jbi_1_0.xsd">
  <!-- identification information about this binding -->
  <type>binding</type>
  <binding>
    <identification>
      <alias>SoapBinding</alias>
      <componentId>e63055a3-b393-4f54-soap-9294ae9c5a21</componentId>
      <description>SOAP binding to send messages to partners.</description>
    </identification>
    <componentClassName description="The lifecycle implementation class">
      soapBinding.SoapBinding
    </componentClassName>
    <componentClassPath>
      <pathelement>lib/soapbinding_rt.jar</pathelement>
    </componentClassPath>
    <bootstrapClassName>soapBinding.SoapInstaller</bootstrapClassName>
    <bootstrapClassPath>
      <pathelement>lib/soapbinding_boot.jar</pathelement>
    </bootstrapClassPath>
  </binding>
</jbi>
```

The illustration gives an example of a binding installation package descriptor (jbi.xml).

7.13.2.2 Shared Name Space Installation Packaging

Shared name space installation packages MUST conform to this specification. The JBI installation service shall generate an error if non-conformance is detected at installation time. The installation package format is shown schematically in illustration [ED: TBD].

The package descriptor jbi.xml must conform to the schema given in section “jbi.xml”, above with the additional constraints given below:

- The `<jbi>` document element may only contain a `<sharedNameSpace>` child element.

The `<sharedNameSpace>` element contains the following information:

- One `<sharedNameSpaceId>` element, that contains a unique identifier for the shared name space. (It must be unique within the JBI environment in which the shared name space is being installed).
• One `<sharedNameSpaceClassPath>` element, supplying the class path elements for the shared name space.

Installation of a shared name space installation package is detailed in section “Shared namespace Installation”.

### 7.13.3 Deployment Packaging

A deployment JAR file is structured to contain a deployment descriptor (jbi.xml), and one or more ASAs (application sub-assemblies), which are JARs that contain data meant for a particular BC or SE, and which contents are opaque to the JBI environment.

![Deployment JAR structure diagram](image-url)
### 7.14 JNDI Naming Hierarchy

Configuration information for installed components MUST be made available through JNDI, using the naming hierarchy defined in this section.

**JBIROOT**

- Components
  - Bindings
    - Binding_1
      - Classes
      - Configuration
      - Deployments
      - Installation_Root
      - UUID
    - ... Binding_N
  - Engines
    - Engine_1
      - Classes
      - Configuration
      - Deployments
      - Installation_Root
      - UUID
        - Deployment_1
          - Configuration
          - Installation_Root
          - Service_Deployment_UUID
        - Deployment_2
          - Configuration
          - Installation_Root
          - Service_Deployment_UUID
          - ... more deployments
    - ... Engine_N
      - Classes
      - Configuration
      - Deployments
      - Installation_Root
      - UUID
      - Deployments
        - Deployment_1
          - Configuration
          - Installation_Root
          - Service_Deployment_UUID
        - Deployment_2
          - Configuration
          - Installation_Root
          - Service_Deployment_UUID
          - ... more deployments

Configuration
7.15 Java Management Extensions

JBI implementations MUST support JMX 1.2.1 or newer.

JBI implementations MUST provide the following management beans (MBeans), as defined in this subsection:

- Administration Service. Top-level service for discovering engines, bindings, and system components, and deploying (and undeploying) AUs to them.
- Installation Service. This allows the administrator to install and uninstall engines, bindings and name spaces.

In addition, bindings and engines MAY provide additional MBeans to provide additional management functions necessary to the engine or binding type.

All MBeans defined in this specification must be JMX Standard MBeans, as defined in JMX 1.2.1.

7.15.1 System MBeans

System MBeans are those used to provide services needed for a variety of administrative tasks. JBI implementations MUST provide the service MBeans listed in the following subsections.

7.15.1.1 Installation Service MBean

The installation service MBean is used to allow the administrator to initiate and monitor installing and uninstalling engines and bindings. This is distinct from the installation MBean, which is used during the installation process to manage the installation of an individual engine or binding.

7.15.1.2 Administration Service MBean

The JBI administration service (JMAS) MBean is used by the administrator to discover what engines, bindings and name spaces are installed in the JBI environment. In addition, it allows the administrator to deploy (and undeploy) application units (AUs), as well as perform management tasks related to AUs.

7.15.1.3 Common MBeans

The MBeans specified in this section are common to both engines and bindings, as well as some system components.

7.15.1.4 Installer MBean

Once the installation service has successfully loaded an installer using its `loadNewInstaller` or `loadInstaller` methods, the Installation Service returns the JMX `ObjectName` of an
**InstallerMBean.** This bean is created by the JBI environment, allowing the administrator to control the actual installation of the engine or binding.

### 7.15.1.5 Deployer MBean

The deployer MBean must be implemented by engines and bindings. It is used to manage deployment of ASAs to engines and bindings. It is the responsibility of the individual engines and bindings to keep track of such deployments, persisting such information from the ASA as is needed to be able to restart the engine (or binding) after a shutdown or crash.

The deployer MBean is used to deploy ASAs, undeploy them, and query what ASAs are deployed. It is normally used by the JMAS MBean to deploy the contents of an AU, rather than used directly by the administrative user. Implementations of administrative tools (such as deployment tools) are free to make direct use of this MBean if need be.

### 7.15.1.6 Life Cycle MBean

The life cycle MBean is used to control the life cycle of a system component.

### 7.15.1.7 Component Life Cycle MBean

This component life cycle MBean extends the life cycle MBean by adding support for binding (and engine) deployment and installation extension. This MBean MUST be implemented by bindings and engines.

The deployment MBean is used to receive ASAs during invocation of the system deployment service. This is detailed in the section entitled “

### 7.15.2 Required MBeans

The following table summarizes the which MBeans are required by system components, engines, and bindings.

<table>
<thead>
<tr>
<th>Mbean</th>
<th>System Components</th>
<th>Engine</th>
<th>Logger</th>
</tr>
</thead>
<tbody>
<tr>
<td>Installation Service</td>
<td>Required</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Deployment Service</td>
<td>Required</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Life Cycle</td>
<td>Required</td>
<td>Required</td>
<td>Required</td>
</tr>
<tr>
<td>Component Life Cycle</td>
<td>n/a</td>
<td>Optional</td>
<td>Optional</td>
</tr>
<tr>
<td>Deployer</td>
<td>n/a</td>
<td>Required</td>
<td>Required</td>
</tr>
<tr>
<td>Installer</td>
<td>n/a</td>
<td>(implicit)</td>
<td>(implicit)</td>
</tr>
</tbody>
</table>

The term “implicit” for the installer MBean refers to the fact that the JBI implementation supplies such beans, rather than the engine or binding directly.

### 7.16 Scripting Language Support

JBI implementations MUST provide Apache Ant-based scripting language support, by providing JMX tasks to invoke MBeans in the JBI environment.

Implementations MUST supply Apache Ant, version 1.5 or newer, plus required definition scripts and supporting Java binary code to provide JMX task support.
Assuming an environment variable JBIROOT which points at the root of a JBI installation, the following script could be used to install and start a binding whose installation package is contained in filebinding.jar.

```
$JBIROOT/jbi/bin/ant -f $JBIROOT/jbi/bin/jbi_admin.xml -Djbi.componentJar=filebinding.jar install
$JBIROOT/jbi/bin/ant -f $JBIROOT/jbi/bin/jbi_admin.xml -Djbi.componentId="fb-b393-4f54-aaad-9294ae9c5a21" start
```

[ED: Property and target definitions To Be Supplied.]
8 Common Framework

The SE and BC frameworks expose several common interfaces, which are defined in this section of the specification to avoid duplication. SE- and BC-specific framework interfaces are specified in the chapters devoted to those particular frameworks.

The frameworks rely on interfaces exposed to engines and bindings (APIs), and interfaces exposed by engines and bindings to the framework (SPIs). This section defines the contracts between JBI and the plug-in engines and bindings using the defined APIs and SPIs.

8.1 Environment Context

The EnvironmentContext interface serves to allow bindings and engines to query architected aspects of the environment the binding or engine is running in. This context is given to the binding or engine life cycle interface using the init() method; see the chapters on the SE Framework and BC Framework for details.

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

- Component ID. The unique identifier assigned to the BC or SE at installation time.
- Component Root. The installation root directory of the BC or SE. This allows the BC or SE to
- JBI Install Root. The installation root directory of the JBI system.
- MBean Naming Service. A service that provides methods to generate JMX MBean Object Names for any MBeans provided by the BC or SE.
- 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 engine or binding.
- Normalized Message Service. This allows the engine or binding to interact with the messaging service.

8.2 JBIEexception

All exception types defined by this specification are derived from this base exception class. This allows application code to easily catch all possible JBI-related exceptions when needed.
9 Service Engine Framework

Service engines (SEs) are plugged into the JBI environment using the SE framework. This framework encompasses the API/SPI contracts between the engine and the environment, as well as a set of services provided to engines.

This framework is similar to that for binding components (BCs), but is separately defined to provide SE writers a clear set of contracts and services meant only for SEs. In addition, the asymmetric nature of WSDL endpoints (for BCs) and services (for SEs) makes the role of BCs and SEs different within the WSDL model that JBI utilizes.

9.1 Installation (Bootstrap)

Installation of service engines is handled largely by the JBI Management system (see the chapter on management in this specification), which will copy the installation image to the file system of the platform the JBI system will run on. The process of installing (and the reverse, uninstalling) a service engine can be customized by the SE. 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 the SE bootstrap.

The service engine MUST provide, as part of its installation package, a bootstrap class that implements the javax.jbi.engine.EngineBootstrap interface. (The installation package is specified in the Management chapter.)

9.1.1 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:

• Alias name. A simple name for the SE provided in the installation package.
• Class name. The name of the class that implements the life cycle SPI, javax.jbi.engine.EngineLifeCycle, for this SE. This can be read and written by the SE bootstrap.
• Class path elements. A list of directories or jar files that will comprise the class path of the SE when it is started. The class that implements the life cycle SPI must be accessible through this class path. This can be read and written by the SE bootstrap.
• Component identifier. A system-assigned, unique identifier for the SE.
• Environment context. A JBI-global environment context. It gives access to the items similar to the environment context given to a SE's life cycle init() method, but is global to the SE framework, rather than a single SE.
• Installation root directory. The full path name of the directory into which this SE is being installed.
• XML SE definition. [Editorial: needs definition.]

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.engine.EngineLifeCycle` interface.

### 9.2 Life Cycle

The service engine MUST provide, as part of its installation package, a life cycle class that implements the `javax.jbi.engine.EngineLifeCycle` interface. (The installation package is specified in the Management chapter.)

After installation, the SE is ready for conventional life cycle control. A SE's possible life cycle states, and transitions between states, are depicted below.

When first installed, the SE is placed in the “Installed” state.

When `start()` is invoked (see management chapter and diagram below), and the SE is in the “Installed” state, the SE framework MUST call the SE life cycle `init()` method, passing to it a valid environment context, followed by a call to the life cycle `start()` method. Subsequent calls to `start()`, from the “Stopped” state, MUST NOT cause the SE framework to call the `init()` method.

![Life Cycle Diagram](image-url)
Subsequent starts without an intervening stop and shutdown behave as shown below.
10 Binding Component Framework

Binding components (BCs) are plugged into the JBI environment using the BC framework. This framework encompasses the API/SPI contracts between the binding and the environment, as well as a set of services provided to bindings.

This framework is similar to that for service engines (SEs), but is separately defined to provide BC writers a clear set of contracts and services meant only for BCs. In addition, the asymmetric nature of WSDL endpoints (for BCs) and services (for SEs) makes the role of BCs and SEs different within the WSDL model that JBI utilizes.

10.1 Installation (Bootstrap)

Installation of binding components is handled largely by the JBI Management system (see the chapter on management in this specification), which will copy the installation image to the file system of the platform the JBI system will run on. The process of installing (and the reverse, uninstalling) a binding component can be customized by the BC. 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 the BC bootstrap.

The binding component MUST provide, as part of its installation package, a bootstrap class that implements the `javax.jbi.binding.BindingBootstrap` interface. (The installation package is specified in the Management chapter.)

10.1.1 Installation Context

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

- **Alias name.** A simple name for the BC provided in the installation package.
- **Class name.** The name of the class that implements the life cycle SPI, `javax.jbi.binding.BindingLifeCycle`, for this BC. This can be read and written by the BC bootstrap.
- **Class path elements.** A list of directories or jar files that will comprise the class path of the BC when it is started. The class that implements the life cycle SPI must be accessible through this class path. This can be read and written by the BC bootstrap.
- **Component identifier.** A system-assigned, unique identifier for the BC.
- **Environment context.** A JBI-global environment context. It gives access to the items similar to the environment context given to a BC's life cycle `init()` method, but is global to the BC framework, rather than a single BC.
- **Installation root directory.** The full path name of the directory into which this BC is being installed.
• XML BC definition. [TBD]

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.binding.BindingLifeCycle` interface.

10.2 Life Cycle

The binding component MUST provide, as part of its installation package, a life cycle class that implements the `javax.jbi.binding.BindingLifeCycle` interface.

After installation, the BC is ready for conventional life cycle control. A BC's possible life cycle states, and transitions between states, are depicted in illustration 1, below.

![Life Cycle Diagram]

When first installed, the BC is placed in the “Installed” state.

When `start()` is invoked (see management chapter and diagram below), and the BC is in the “Installed” state, the BC framework MUST call the BC life cycle `init()` method, passing to it a valid environment context, followed by a call to the life cycle `start()` method. Subsequent calls to `start()`, from the “Stopped” state, MUST NOT cause the BC framework to call the `init()` method.

The initial start is shown in the following sequence diagram.
Subsequent starts without an intervening stop and shutdown behave as shown below.

![Diagram showing the life cycle of components: Admin Console, Life Cycle MBean, BC Framework, BC Life-Cycle Object, with arrows indicating start events.](image-url)
11 Shared Name Spaces and Class Loading

JBI components (BCs and SEs) are loosely-coupled components that, at a minimum, interact only through the JBI NormalizedMessage. In some important use cases, this interaction needs to be extended through the JBI ContentBags, where the components have a shared domain-specific set of classes used to read and write the content bag contents. For example, an EDI BC may wish to share a set of specialized class objects that represent an EDI document with a specialized SE that consumes such EDI documents. Given that the BC and SE 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.

It is the responsibility of the JBI environment to ensure that the correct class loaders are associated with any threads used to call a BC or SE. This includes NMS message delivery and SPI invocation.

To facilitate sharing of class loaders between BCs and SEs 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.

11.1 JBI Class Loader Hierarchy

The hierarchy of class loaders required by the JBI class loading scheme is shown in the following illustration. This depicts how individual class loaders are chained together in a JBI environment.

Each component (BC or SE) has a separate JBI-provided component class loader, that is always associated with threads of execution for that component. This class loader gives the component access to classes that are private to the component's implementation. These classes are specified during component installation.

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 (BC or
SE) 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 delegating class loader allows its associated component to use only those shared class loaders which it is allowed access (by configuration). In addition, the delegating class loader enforces ordering of the use of those class loaders. This order is used to predictably search for a class; the first shared loader to not return a `ClassNotFoundException` will be used to load and return the class. The order is specified by the order in which the shared class loader names are listed in the installation package for each component.

**Shared class loaders** are installed separately from BCs and SEs, with the intention that they be used by sets of BCs and SEs 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.) Because of this unique name space associated with a class loader, shared class loaders are sometimes referred to as shared name spaces.

Each shared class loader is a child of the **JBI life cycle class loader**. This is used to load the JBI life cycle module.

Optionally, the JBI life cycle class loader can be chained to other class loaders, such as the application server shared class loader shown in illustration 1. 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 bootstrap loader).

The **JBI implementation class loader** is a child of the JBI life cycle class loader. This is used to load classes for the implementation of the JBI environment. It is separated from the component and shared class loaders to avoid any possible collisions with component classes, and to ensure that the components do not have access to the private implementation classes of the JBI environment.

### 11.2 Installation of Shared Name Spaces

Shared name spaces (shared class loaders) are installed (and uninstalled) using the Management InstallationMBean. See the chapter on Management for details.

A shared name space must be installed before a BC or SE can be installed that makes use of the shared name space, otherwise the installation will fail. See the Management chapter, section “Shared Name Space Installation” for details on installation of BCs and SEs that use shared name spaces.

### 11.3 BC and SE 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, it is strongly recommended that the component set the JBI-provided context class loader as the parent of the component-provided class loader.
12 Security

[Note to reader: The security requirements of JBI are under review by the Expert Group. Early Draft review input is most welcome.]

BC and SEs are assumed to be operating in a trusted environment. The JBI environment must rely on the underlying J2SE™ platform for its security services. The Normalized Message is required to convey any needed security credentials, tokens, public and or private keys. These security elements must be represented as part of the Normalized Message Context in an extensible manner. Such tokens may be used for, but not limited to, application specific authentication, authorization, digital signing, and data encryption. The standard serialized form of individual security tokens is outside the scope of this document. It is anticipated that future versions of this specification may make normative reference to standard serialized security formats.

13 References

[Ed: TBD]

14 Change History

[Ed: TBD]

15 Open Issues

[Ed: The list of open issues is being compiled separately and will be tracked in detail after Early Draft review]