JBoss ESB 4.2 Milestone Release 1

Programmers Guide

JBESB-PG-3/23/07

Legal Notices

The information contained in this documentation is subject to change without notice.

JBoss Inc. makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. JBoss Inc. shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Java™ and J2EE is a U.S. trademark of Sun Microsystems, Inc. Microsoft® and Windows NT® are registered trademarks of Microsoft Corporation. Oracle® is a registered U.S. trademark and Oracle9™, Oracle9 Server™ Oracle9 Enterprise Edition™ are trademarks of Oracle Corporation. Unix is used here as a generic term covering all versions of the UNIX® operating system. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

Copyright

JBoss, Home of Professional Open Source Copyright 2006, JBoss Inc., and individual contributors as indicated by the @authors tag. All rights reserved.

See the copyright.txt in the distribution for a full listing of individual contributors. This copyrighted material is made available to anyone wishing to use, modify, copy, or redistribute it subject to the terms and conditions of the GNU General Public License, v. 2.0. This program is distributed in the hope that it will be useful, but WITHOUT A WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

See the GNU General Public License for more details. You should have received a copy of the GNU General Public License, v. 2.0 along with this distribution; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.

Software Version

JBoss ESB 4.2 Milestone Release 1

Restricted Rights Legend

Use, duplication, or disclosure is subject to restrictions as set forth in contract subdivision (c)(1)(ii) of the Rights in Technical Data and Computer Software clause 52.227-FAR14.

© Copyright 2007 JBoss Inc.

Contents

Table of Contents

Contents..iv

About This Guide..5

What This Guide Contains..............................5
Audience..5
Prerequisites...5
Organization...5
Documentation Conventions..........................5
Additional Documentation.............................6
Contacting Us..6

Service Oriented Architecture............................8

Overview..8
Why SOA?...10
Basics of SOA..11
Advantages of SOA......................................12
Interoperability...12
Efficiency 12
Standardization...13

The Enterprise Service Bus..............................14

Overview..14
Architectural requirements...........................16
Registries and repositories............................17
Versioning of Services..................................17
Incorporating legacy services.......................18

When to use JBossESB......................................19

Introduction..19

JBossESB..23

Rosetta...23
The core of JBossESB in a nutshell.............24
JBossESB components.................................25
Configuration...25
The Message Store.......................................26
ESB-aware and ESB-unaware users............28
Endpoint References....................................29
Mapping of EPR to Service..........................31
Gateways to the ESB....................................33
The Message...34
The Message Header....................................37
The Message payload...................................38
The MessageFactory....................................39
Message Formats..40
MessageType.JAVA_SERIALIZED…40
MessageType.JBOSS_XML................40
Data Transformation....................................41
Listener, Courier and Action Classes............41

Process Engine Support.....................................46

jBPM...46

Configuration...47

Overview..47
Providers..48
Services...49
Transport Specific Type Implementations….54
Transitioning From The Old Configuration Model..56
Frequently Asked Questions (FAQs)............57

Glossary...58

Index...62

About This Guide

What This Guide Contains

The Programmers Guide contains descriptions on the principles behind Service Oriented Architecture and Enterprise Service Bus, as well as how they relate to JBossESB. This guide also contains information on how to use JBoss ESB 4.2 Milestone Release 1.

Audience

This guide is most relevant to engineers who are responsible for using JBoss ESB 4.2 Milestone Release 1 installations and want to know how it relates to SOA and ESB principles.

Prerequisites

None.

Organization

This guide contains the following chapters:

- **Chapter 1, What is SOA?:** JBossESB is a SOA infrastructure. This chapter gives an overview of SOA and the benefits it can provide.

- **Chapter 2, The Enterprise Service Bus:** an overview of what constitutes an ESB and how JBossESB may differ from traditional ESB definitions.

- **Chapter 3, JBossESB core:** a description of the core components within JBossESB and how they are intended to be used.

- **Chapter 4, Configuration:** a description of the configuration options within JBossESB.

Documentation Conventions

The following conventions are used in this guide:

Convention	Description
Italic	In paragraph text, italic identifies the titles of documents that are being referenced. When used in conjunction with the Code text described below, italics identify a variable that should be replaced by the user with an actual value.
Bold	Emphasizes items of particular importance.
`Code`	Text that represents programming code.
Function | Function	A path to a function or dialog box within an interface. For example, “Select File | Open.” indicates that you should select the Open function from the File menu.
( ) and |	Parentheses enclose optional items in command syntax. The vertical bar separates syntax items in a list of choices. For example, any of the following three items can be entered in this syntax: `persistPolicy (Never | OnTimer | OnUpdate | NoMoreOftenThan)`
Note: **Caution**:	A note highlights important supplemental information. A caution highlights procedures or information that is necessary to avoid damage to equipment, damage to software, loss of data, or invalid test results.

Table 1 Formatting Conventions

Additional Documentation

In addition to this guide, the following guides are available in the JBoss ESB 4.2 Milestone Release 1 documentation set:

1. **JBoss ESB 4.2 Milestone Release 1** *Trailblazer Guide*: Provides guidance for using the trailblazer example.
2. **JBoss ESB 4.2 Milestone Release 1** *Getting Started Guide*: Provides a quick start reference to configuring and using the ESB.
3. **JBoss ESB 4.2 Milestone Release 1** *Administration Guide*: How to manage JBossESB.
4. **JBoss ESB 4.2 Milestone Release 1** *Release Notes*: Information on the differences between this release and previous releases.
5. **JBoss ESB 4.2 Milestone Release 1** *Services Guides*: Various documents related to the services available with the ESB.

Contacting Us

Questions or comments about JBoss ESB 4.2 Milestone Release 1 should be directed to our support team.

Chapter 1

Service Oriented Architecture

Overview

JBossESB is a Service Oriented Architecture (SOA) infrastructure. SOA represents a popular architectural paradigm[1] for applications, with Web Services as probably the most visible way of achieving an SOA[2]. Web Services implement capabilities that are available to other applications (or even other Web Services) via industry standard network and application interfaces and protocols. SOA advocates an approach in which a software component provides its functionality as a service that can be leveraged by other software components. Components (or services) represent reusable software building blocks.

SOA allows the integration of existing systems, applications and users into a flexible architecture that can easily accommodate changing needs. Integrated design, reuse of existing IT investments and above all, industry standards are the elements needed to create a robust SOA.

As enterprises slowly emerge from the mad rush of cost reduction into a more stable period of cost management, many of them find themselves in unfamiliar territory. Prior to the economic slow down, most firms understood the options they had for IT investment. Many embarked on major package implementations (e.g., Siebel, Peoplesoft and so on), while others built on the legacy systems they have trusted for years. Either way, most firms recognized the return promised and made the investment. Today, the appetite for such large investment is gone.

However, enterprises still need to make forward progress and keep ahead of the competition. SOA (and typically Web Services as a concrete implementation of those principles) make this possible. The result is dramatic improvements in collaboration between users, applications and technology components, generating significant value for any business creating competitive advantage.

Imagine a company that has existing software from a variety of different vendors, e.g., SAP, PeopleSoft. Some of these software packages may be useful to conduct business with other companies (customers, suppliers, etc.) and therefore what the company would like to do is to take those existing systems and make them available to other companies, by exposing them as services. A service here is some software component with a stable, published interface that can be invoked by clients (other software components). So, requesting and executing services involves software components owned by one company talking to components owned by another company, i.e., *business-to-business (B2B) transactions*.

---

[1] The principles behind SOA have been around for many years, but Web Services have popularised it.

[2] It is possible to build non-SOA applications using Web Services.

Conventional distributed system infrastructures (middleware) are not sufficient for these cross-organizational exchanges. For instance

- You would need agreement between the parties involved on the middleware platform
- There is an implicit (and sometimes explicit) lack of trust between the parties involved.
- Business data is confidential and should only to be seen by the intended recipient.
- Many assumptions of conventional middleware are invalid in cross-organizational interactions. Transactions, for instance, last longer - possibly for hours or days so conventional transaction protocols such as two phase commit are not applicable.

So, in B2B exchanges the lack of standardization across middleware platforms makes point-to-point solutions costly to realize in practice. The Internet alleviated some of these problems by providing standard interaction protocols (HTTP) and data formats (XML) but by themselves these standards are not enough to support application integration. They don't define interface definition languages, name and directory services, transaction protocols, etc,. It is the gap between what the Web provides and what application integration requires that Web services are trying to fill.

However, whilst the challenge and ultimate goal of SOA is inter-company interactions, services do not need to be accessed through the Internet. They can be made available to clients residing on a local LAN. Indeed, at this current moment in time, many Web services are being used in this context - intra-company integration rather than inter-company exchanges.

An example of how Web services can connect applications both intra-company and inter-company can be understood by considering a stand-alone inventory system. If you don't connect it to anything else, it's not as valuable as it could be. The system can track inventory, but not much more. Inventory information may have to be entered separately in the accounting and customer relationship management systems. The inventory system may be unable to automatically place orders to suppliers. The benefits of such an inventory system are diminished by high overhead costs.

However, if you connect your inventory system to your accounting system with XML, it gets more interesting. Now, whenever you buy or sell something, the implications for your inventory and your cash flow can be tracked in one step. If you go further, and connect your warehouse management system, customer ordering system, supplier ordering systems, and your shipping company with XML, suddenly that inventory management system is worth a lot. You can do end-to-end management of your business while dealing with each transaction only once, instead of once for every system it affects. A lot less work and a lot less opportunity for errors. These connections can be made easily using Web services.

Businesses are waking up to the benefits of SOA. These include:

- opening the door to new business opportunities by making it easy to connect with partners;

- saving time and money by cutting software development time and consuming a service created by others;
- increasing revenue streams by easily making your own services available.

Why SOA?

The problem space can be categorized by past IT investments in the area of eProcurement, eSourcing, Supply Chain Management, Customer Relationship Management (CRM) and Internet computing in general. All of these investments were made in a silo. Along with the incremental growth in these systems to meet short-term (tactical) requirements, the decisions made in this space hurt the long-term viability of the applications and infrastructure.

The three key drivers for implementing an SOA approach are:

1) Cost Reduction: Achieved by the ways services talk to each other. The direct cost effect is delivered through enhanced operations productivity, effective sourcing options, and a significantly enhanced ability to shift ongoing costs to a variable model.

2) Delivering IT solutions faster and smarter: A standards based approach will allow organizations to connect and share information and business processes much faster and easier than before. IT delivery productivity is markedly improved through simplification of the developer's role by providing standard frameworks and interfaces. Delivery timescales have been drastically reduced by easing the integration load of individual functionality, and applying accelerated delivery techniques within the environment.

3) Maximizing return on investment: Web Services opens the way for new business opportunities by enabling new business models. Web Services present the ability to measure value and discrete return much differently than traditional functional-benefit methods. Typical Total Cost of Ownership (TCO) models do not take into account the lifetime value generated by historical investment. This cost centric view destroys many opportunities to exploit these past investments and most enterprises end up building redundancy into their architecture, not out of necessity, but of perceived need. These same organizations focus the value proposition of their IT investment on a portfolio of applications, balanced by the overhead of infrastructure. An approach based on Web Services takes into account the lifetime contribution of legacy IT investment and promotes an evolution of these investments rather than a planned replacement.

SOA/Web Services fundamentally changes the way enterprise software is developed and deployed. SOA has evolved where new applications will not be developed using monolithic approaches, but instead become a virtualized on-demand execution model that breaks the current economic and technological bottleneck caused by traditional approaches.

Software as a service has become pervasive as a model for forward looking enterprises to streamline operations, lower cost of ownership and provides competitive differentiation in the marketplace. Web Services offers a viable opportunity for enterprises to drive significant costs out of software acquisitions,

react to rapidly changing market conditions and conduct transactions with business partners at will. Loosely coupled, standards-based architectures are one approach to distributed computing that will allow software resources available on the network to be leveraged. Applications that separate business processes, presentation rules, business rules and data access into separate loosely coupled layers will not only assist in the construction of better software but also make it more adaptable to future change.

SOA will allow for combining existing functions with new development efforts, allowing the creation of composite applications. Leveraging what works lowers the risks in software development projects. By reusing existing functions, it leads to faster deliverables and better delivery quality.

Loose coupling helps preserve the future by allowing parts to change at their own pace without the risks linked to costly migrations using monolithic approaches. SOA allows business users to focus on business problems at hand without worrying about technical constraints. For the individuals who develop solutions, SOA helps in the following manner:

- Business analysts focus on higher order responsibilities in the development lifecycle while increasing their own knowledge of the business domain.
- Separating functionality into component-based services that can be tackled by multiple teams enables parallel development.
- Quality assurance and unit testing become more efficient; errors can be detected earlier in the development lifecycle
- Development teams can deviate from initial requirements without incurring additional risk
- Components within architecture can aid in becoming reusable assets in order to avoid reinventing the wheel
- Functional decomposition of services and their underlying components with respect to the business process helps preserve the flexibility, future maintainability and eases integration efforts
- Security rules are implemented at the service level and can solve many security considerations within the enterprise

Basics of SOA

Traditional distributed computing environments have been tightly coupled in that they do not deal with a changing environment well. For instance, if an application is interacting with another application, how do they handle data types or data encoding if data types in one system change? How are incompatible data-types handled?

The service-oriented architecture (SOA) consists of three roles: requester, provider, and broker.

- *Service Provider*: A service provider allows access to services, creates a description of a service and publishes it to the service broker.

- *Service Requestor*: A service requester is responsible for discovering a service by searching through the service descriptions given by the service broker. A requester is also responsible for binding to services provided by the service provider.

- *Service Broker*: A service broker hosts a registry of service descriptions. It is responsible for linking a requestor to a service provider.

Advantages of SOA

SOA provide several significant benefits for distributed enterprise systems. Some of the most notable benefits include: interoperability, efficiency, and standardization. We will briefly explore each of these in this section.

Interoperability

Interoperability is the ability of software on different systems to communicate by sharing data and functionality. SOA/Web Services are as much about interoperability as they are about the Web and Internet scale computing. Most companies will have numerous business partners throughout the life of the company. Instead of writing a new addition to your applications every time you gain a new partner, you can write one interface using Web service technologies like SOAP. So now your partners can dynamically find the services they need using UDDI and bind to them using SOAP. You can also extend the interoperability of your systems by implementing Web services within your corporate intranet. With the addition of Web services to your intranet systems and to your extranet, you can reduce the cost integration, increase communication and increase your customer base.

It is also important to note that the industry has even established the Web Services Interoperability Organization.

"The Web Services Interoperability Organization is an open industry effort chartered to promote Web Services interoperability across platforms, applications, and programming languages. The organization brings together a diverse community of Web services leaders to respond to customer needs by providing guidance, recommended practices, and supporting resources for developing interoperable Web services." (www.ws-i.org)

The WS-I will actually determine whether a Web service conforms to WS-I standards as well as industry standards. In order to establish integrity and acceptance, companies will seek to build their Web services in compliance with the WS-I standards.

Efficiency

SOA will enable you to reuse your existing applications. Instead of creating totally new applications, you can create them using various combinations of services exposed by your existing applications. Developers can be more efficient because they can focus on learning industry standard technology. They will not have to spend a lot of time learning every new technology that arises. For a manager this means a reduction in the cost of buying new software and having to hire new developers with new skill sets. This approach will allow developers to meet changing business requirements and reduce the length of development cycles for projects. Overall, SOA provides for an increase in efficiency by allowing

applications to be reused, decreasing the learning curve for developers and speeding up the total development process.

Standardization

For something to be a true standard, it must be accepted and used by the majority of the industry. One vendor or small group of vendors must not control the evolution of the technology or specification. Most if not all of the industry leaders are involved in the development of Web service specifications. Almost all businesses use the Internet and World Wide Web in one form or another. The underlying protocol for the WWW is of course HTTP. The foundation of Web services is built upon HTTP and XML. Although SOA does not mandate a particular implementation framework, interoperability is important and SOAP is one of the few protocols that all good SOA implementations can agree on.

Chapter 2

The Enterprise Service Bus

Overview

The ESB is seen as the next generation of EAI – better and without the vendor-lockin characteristics of old. As such, many of the capabilities of a good ESB mirror those of existing EAI offerings. Traditional EAI stacks consist of: Business Process Monitoring, Integrated Development Environment, Human Workflow User Interface, Business Process Management, Connectors, Transaction Manager, Security, Application Container, Messaging Service, Metadata Repository, Naming and Directory Service, Distributed Computing Architecture.

As with EAI systems, ESB is *not* about business logic – that is left to higher levels. It is about infrastructure logic. Although there are many different definitions of what constitutes an ESB, what everyone agrees on now is that an ESB is part of an SOA infrastructure. However, SOA is not simply a technology or a product: it's a style of design, with many aspects (such as architectural, methodological and organisational) unrelated to the actual technology. But obviously at some point it becomes necessary to map the abstract SOA to a concrete implementation and that's where the ESB comes in to play.

By considering ESB in terms of an SOA infrastructure, then we have the flexibility to abstract away from given implementation choices, such as JMS, SOAP etc. Then we define the capabilities that we want from our SOA infrastructure, which become the capabilities for the ESB. However, because of their heritage, ESBs typically come with a few assumptions that are not inherent to SOA:

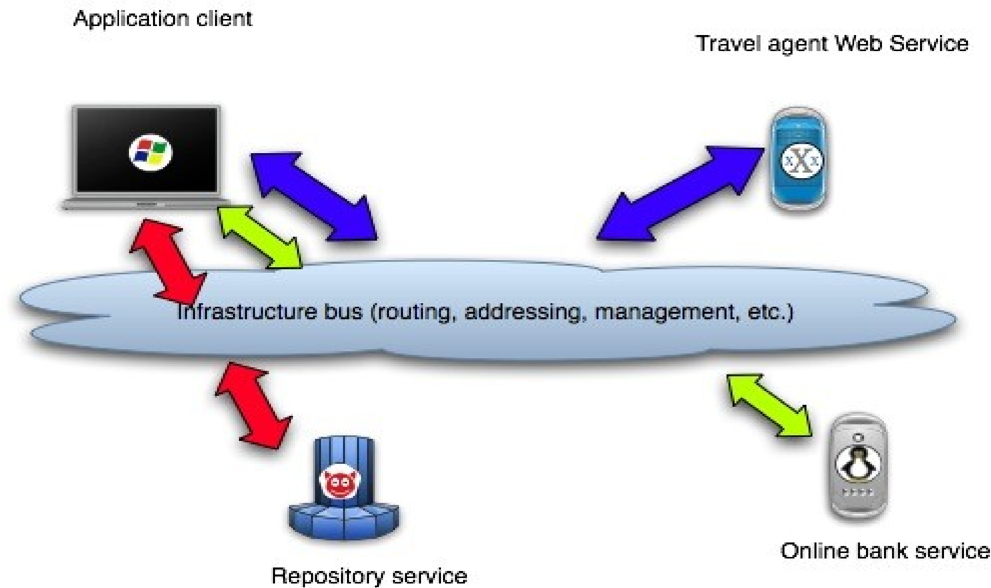
- Java specific.
- Run-time message mediator.
- Message translation.
- Security model translation.

Loose coupling *does not* require a mediator to route messages, although that is dominant ESB architecture. This is also a requirement within the JBI specification. The ESB model should not restrict the SOA model, but should be seen as a concrete representation of SOA. As a result, if there is a conflict between the way SOA would approach something and the way in which it may be done in a traditional ESB, the SOA approach will win within JBossESB.

Therefore, in JBossESB mediation (e.g., content based routing) is a deployment choice and not a mandatory requirement. Obviously for compliance with certain specifications it may be configured by default, but if developers don't need that

compliance point, they should be able to remove it (generally or on a per service basis).

The abstract view of the ESB/SOA infrastructure is shown below in Figure 1:



At its core, a good SOA should have a good *messaging infrastructure* (MI), and JMS is a fairly good example of a standards-compliant MI. But it obviously will not be the only implementation supported. Other capabilities that an ESB provides include:

- Process orchestration, typically via WS-BPEL.
- Protocol translation.
- Adapters.
- Change management (hot deployment, versioning, lifecycle management).
- Quality of service (transactions, failover).
- Qualify of protection (message encryption, security).
- Management.

Access control lists (ACLs) are important and complimentary to security protocols, such as WS-Security/WS-Trust, and often overlooked by existing implementations. JBossESB will support ACLs are part of the security capabilities.

Many of these capabilities can be obtained by plugging in other services or layering existing functionality on the ESB. We should see the ESB as the fabric for building, deploying and managing event-driven SOA applications and systems. There are many different ways in which these capabilities can be realized, and the JBossESB does not mandate one implementation over another. Therefore, all capabilities will

be accessed as services which will give plug-and-play configuration and extensibility options.

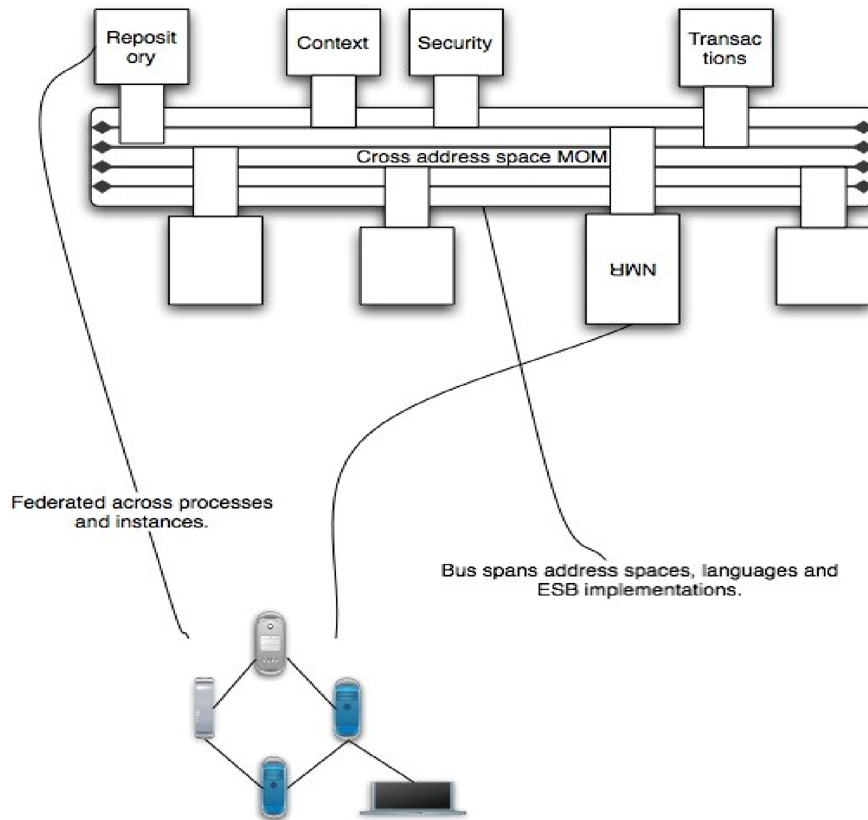


Figure 2: ESB components and multi-bus support.

Architectural requirements

In a distributed environment services can communicate with each other using a variety of message passing protocols. With the aid of client and server stub code, RPC semantics can be used to maintain the abstraction of local procedure calls across address space boundaries. Client stub code is a local proxy for the remote object, which is controlled by the corresponding server stub code. It is the responsibility of the client stub to marshal information which identifies the remote method and its parameters, transmit this information across the network to the object, receive the reply message, and un-marshal the reply to return to the invoker.

However, SOA does not imply a specific carrier protocol and neither does it imply RPC semantics (in fact, loose coupling of services forces developers into an

asynchronous message passing pattern[3]). Therefore, multiple protocols should be supported simultaneously. In most cases, clients will know the communication protocol to use when interacting with a service; however, in some situations this may not be the case, and the communication stack may need to be assembled dynamically (via a hand-shake protocol, where the client stub may have to be dynamically constructed[4]).

At the core of JBossESB is a *messaging infrastructure* (MI), but this MI is abstract, in that it does not force us into just JMS or SOAP styles. For example, a pure-play Web Services deployment within the ESB *can* be supported. As such, JBossESB assumes a single MI abstraction, but the capabilities may be provided by multiple different implementations. This is further support for the notion of having multiple buses within the ESB (each bus may be controlled by a separate MI implementation).

The service description and service contract are extremely important in the context of SOA and therefore ESB. In general, the developers create the contracts and the ESB maps it to whatever technology is being used to implement the SOA, e.g., WSDL. JBossESB allows this mapping to technology to be configurable and dynamic, i.e., it supports multiple SOA implementation technologies.

Registries and repositories

There are actually two different aspects to the service bus: first, turning legacy systems and services into services that work within the SOA infrastructure; secondly, there is taking the services and adding policy and mediation control between those services. Integral to this is the notion of SOA Repositories: a repository is a persistent representation of an SOA Registry, which is needed to publish, discover and consume services. JBossESB will support a range of registry implementations, with UDDI as one of the first.

Versioning of Services

Using the ESB/SOA actually consists of two phases: the initial creation phase and the maintenance phase, which may have different requirements from the creation phase. Services evolve over time and it is often difficult or impossible to find a quiescent period in which to replace a service. As such, in any enterprise deployment there is likely going to be multiple versions of services being used by clients at the same time. Some of the version mismatch may be hidden by suitable routing and on-the-fly message modifications. JBossESB will address the challenge of versioning of services, something that other implementations tend to ignore. Services will be identifiable via major and minor version numbers, with pattern matching capabilities provided by a pluggable rules engine, e.g., a default rule would be that all minor versions are compatible within the scope of the same major version number, but that can be overridden with a specific rule by the service provider or system administrator.

---

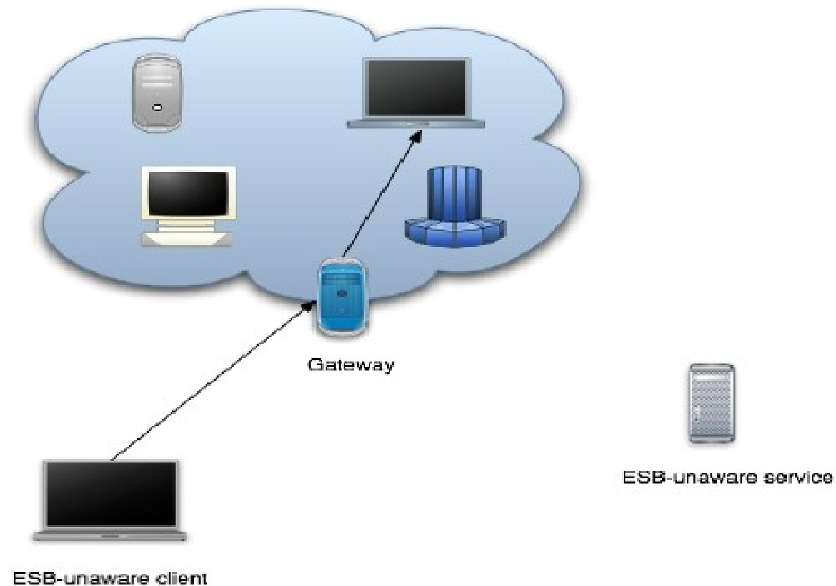
[3] Actually true asynchrony is often not necessary: synchronous one-way (void returns) RPCs can be used and often are in Web Services.

[4] Services may be available via multiple different protocols simultaneously, e.g., CORBA IIOP and JMS. A service repository (aka Name Service/Trading Service) will maintain service identities with their endpoint references and contract definitions (CORBA IDL, WSDL, etc.)

Incorporating legacy services

One of the key aspects of SOA is the ability to leverage existing infrastructural investments. Being required to cast aside software systems in order to incorporate a new technology such as an ESB, is not good practice and we would caution against using such systems since they could lead to vendor lock-in.

JBossESB will allow existing services to be incorporated within the ESB environment without modification to those services. Likewise, clients and services that are deployed within JBossESB will be able to use services that are external to the ESB in an automatic manner. This is illustrated in the figure below and explained in more detail in subsequent chapters.



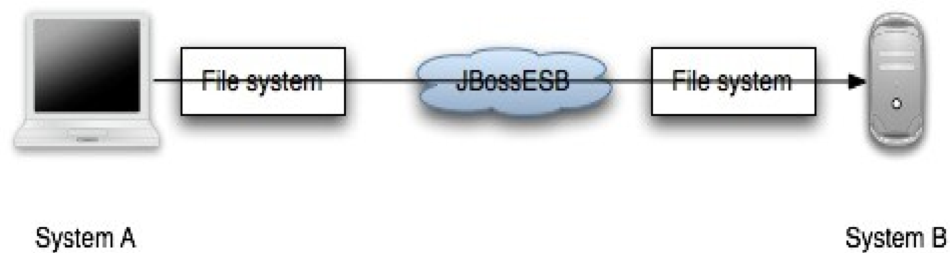
Chapter 3

When to use JBossESB

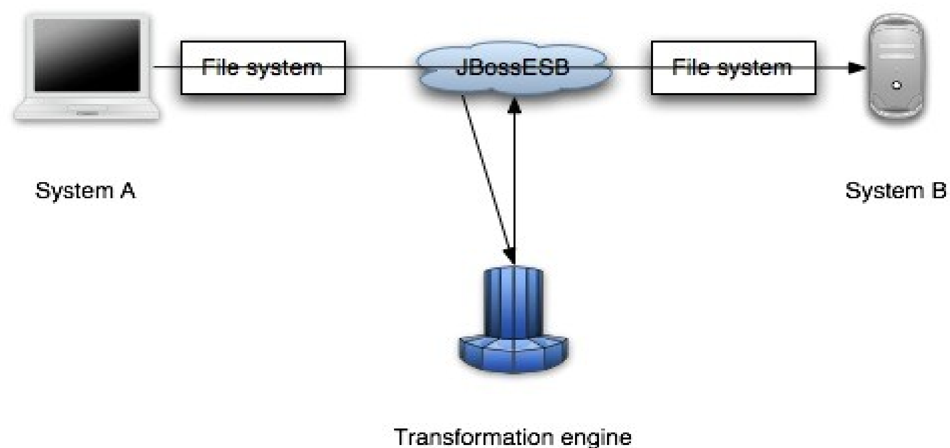
Introduction

We have already discussed when SOA principles and an ESB implementation may be useful. The table below illustrates some further, concrete examples where JBossESB would be useful. Although these examples are specific to interactions between participants using non-interoperable JMS implementations, the principles are general.

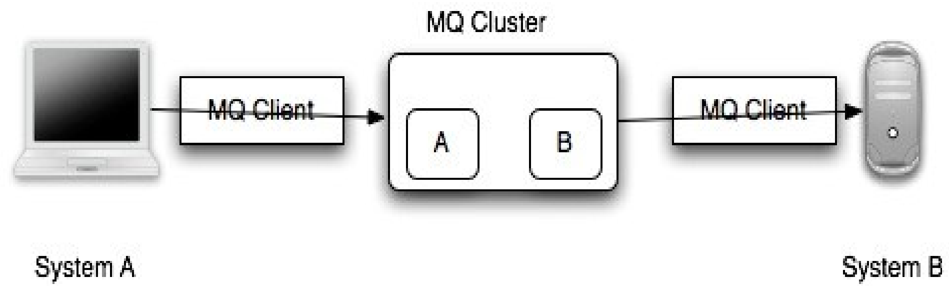
The diagram below shows simple file movement between two systems where messaging queuing is not involved.



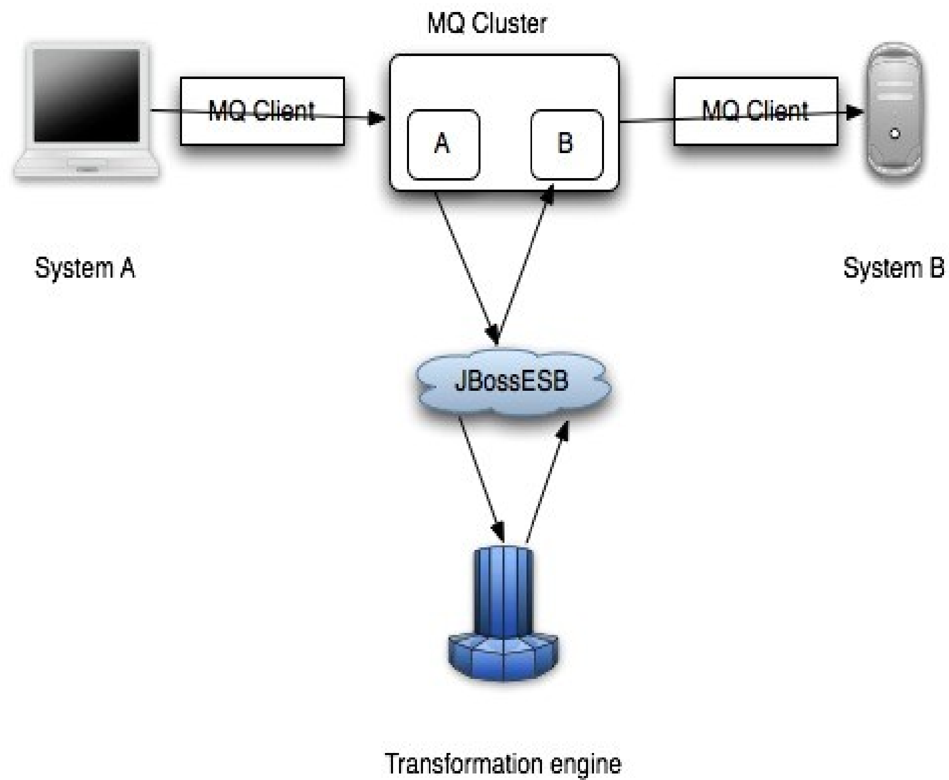
The next diagram illustrates how transformation can be injected into the same scenario using JBossESB.



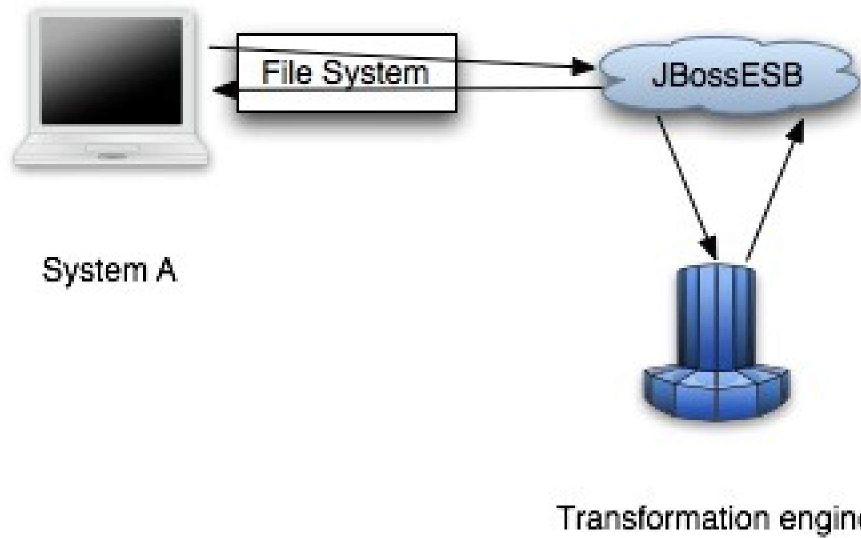
In the next series of examples, we use a queuing system (e.g., a JMS implementation).



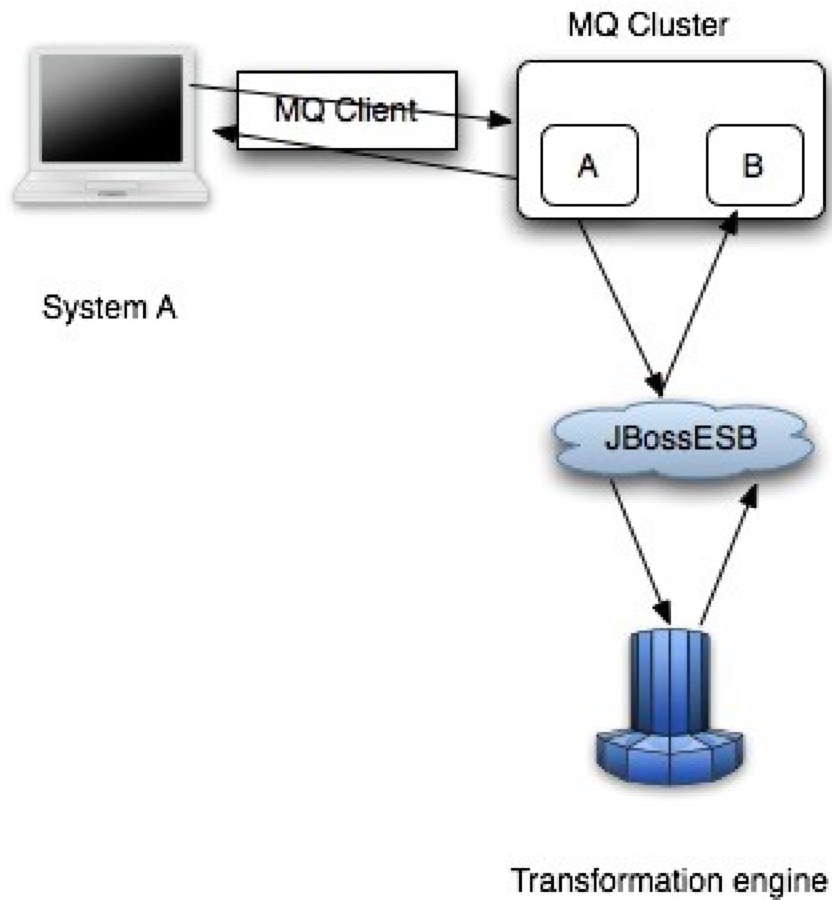
The diagram below shows transformation and queuing in the same situation.



JBossESB can be used in more than multi-party scenarios. For example, the diagram below shows basic data transformation via the ESB using the file system.



The final scenario is again a single party example using transformation and a queuing system.



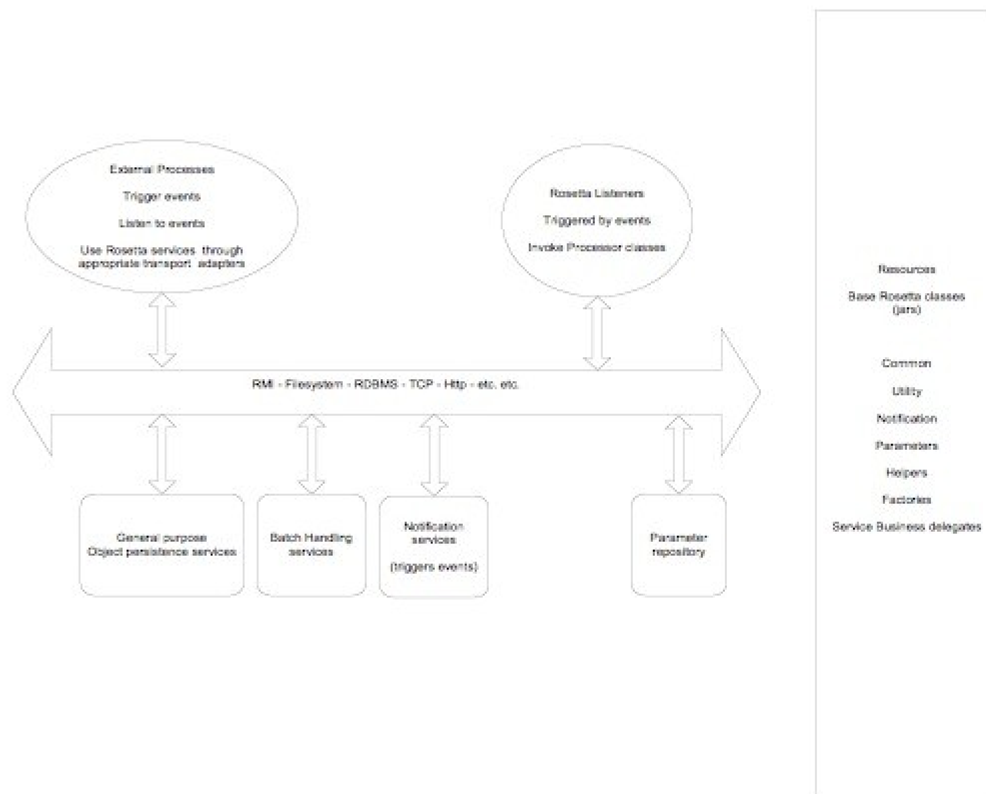
Chapter 4

JBossESB

Rosetta

The core of JBossESB is *Rosetta*[5], an ESB that has been in commercial deployment at a mission critical site for over 3 years. The architecture of Rosetta is shown below in Figure 3:

> Note: In the diagram, *processor classes* refer to the Action classes within the core that are responsible for processing on triggered events.



There are many reasons why users may want disparate applications, services and components to interoperate, e.g., leveraging legacy systems in new deployments. Furthermore, as we have seen such interactions between these entities may occur both synchronously or asynchronously. As with most ESBs, Rosetta was developed

---

[5] Rosetta borrowed its name from the stone found in 1799 by French soldiers in the Nile delta's town of Rosetta (French for Rashid) that was instrumental in Jean-François Champollion deciphering of Egyptian hieroglyphs.

to facilitate such deployments, but providing an infrastructure and set of tools that could:

- Be easily configured to work with a wide variety of transport mechanisms (e.g., email and JMS).
- Offer a general purpose object repository.
- Enable pluggable data transformation mechanisms.
- Provide a batch handling capability.
- Support logging of interactions.

To date, Rosetta has been used in mission critical deployments using Oracle Financials. The multi platform environment included an IBM mainframe running z/OS, DB2 and Oracle databases hosted in the mainframe and in smaller servers, with additional Windows and Linux servers and a myriad of third party applications that offered dissimilar entry points for interoperation. It used JMS and MQSeries for asynchronous messaging and Postgress for object storage. Interoperation with third parties outside of the corporation's IT infrastructure was made possible using IBM MQSeries, FTP servers offering entry points to pick up and deposit files to/from the outside world and attachments in e-mail messages to 'well known' e-mail accounts.

As we shall see when examining the JBossESB core, which is based on Rosetta, the challenge was to provide a set of tools and a methodology that would make it simple to isolate business logic from transport and triggering mechanisms, to log business and processing events that flowed through the framework and to allow flexible plug ins of ad hoc business logic and data transformations. Emphasis was placed on ensuring that it possible (and simple) for future users to replace/extend the standard base classes that come with the framework (and are used for the toolset), and to trigger their own 'action classes' that can be unaware of transport and triggering mechanisms.

The core of JBossESB in a nutshell

Rosetta is built on three core architectural components:

- Message Listener and Message Filtering code. Message Listeners act as "inbound" message routers that listen for messages (e.g. on a JMS Queue/Topic, or on the filesystem) and present the message to a message processing pipeline that filters the message and routes it ("outbound" router) to another message endpoint.
- Data transformation via the SmooksTransformer action processor. See the Message Transformation Guide.
- A Content Based Routing Service. See the CBR Guide.
- A Message Repository, for saving messages/events exchanged within the ESB.

These capabilities are offered through a set of business classes, adapters and processors, which will be described in detail later. Interactions between clients and

services are supported via a range of different approaches, including JMS, flat-file system and email.

A typical JBossESB deployment is shown below. We shall return to this diagram in subsequent sections.

> Note: Some of the components in the diagram (e.g., LDAP server) are configuration choices and may not be provided out-of-the-box. Furthermore, the Processor and Action distinction shown in the above diagram is merely an illustrative convenience to show the concepts involved when an incoming event (message) triggers the underlying ESB to invoke higher-level services.

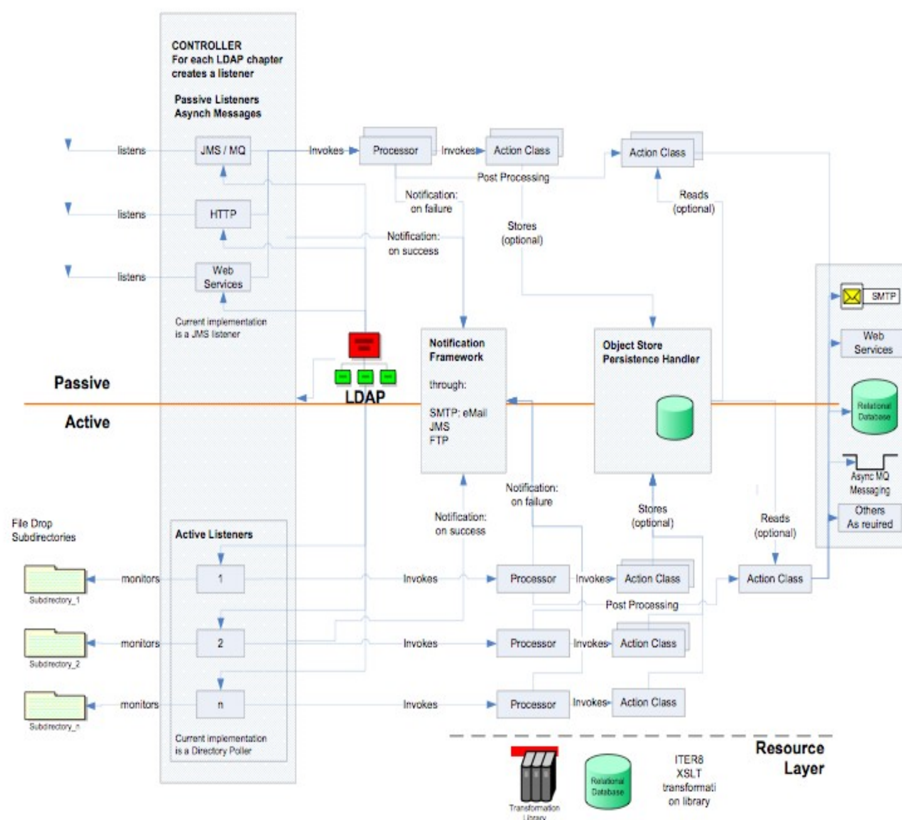


Figure 4: ESB Core components.

JBossESB components

In the following sections we shall examine the core components of JBossESB.

Configuration

All components within the core receive their configuration parameters as XML. How these parameters are provided to the system is hidden by the `org.jboss.soa.esb.parameters.ParamRepositoryFactory`:

```
public abstract class ParamRepositoryFactory
{
```

```
  public static ParamRepository getInstance();
}
```

This returns implementations of the `org.jboss.soa.esb.parameters.ParamRepository` interface which allows for different implementations:

```
public interface ParamRepository
{
  public void add(String name, String value) throws
                                           ParamRepositoryException;
  public String get(String name) throws ParamRepositoryException;
  public void remove(String name) throws ParamRepositoryException;
}
```

Within this version of the JBossESB, there is only a single implementation, the `org.jboss.soa.esb.parameters.ParamFileRepository`, which expects to be able to load the parameters from a file. The implementation to use may be overridden using the `org.jboss.soa.esb.paramsRepository.class` property.

> Note: we recommend that you construct your ESB configuration file using Eclipse or some other XML editor. The JBossESB configuration information is supported by an annotated XSD which should help if using a basic editor.

The Message Store

The message store mechanism in JBossESB is designed with audit tracking purposes in mind. As with other ESB services, it is a pluggable service, which allows for you, the developer to plug in your own persistence mechanism should you have special needs. The implementation supplied with JBossESB is a database persistence mechanism. If you require say, a file persistence mechanism, then it's just a matter of you writing your own service to do this, and override the default behaviour with a configuration change.

One thing to point out with the Message Store – this is a base implementation. We will be working with the community and partners to drive the feature functionality set of the message store to support advanced audit and management requirements. This is meant to be a starting point.

First, let's discuss the Message Store interface. It is quite simple:

The interface, part of the Rosetta core, is defined as follows:

```
package org.jboss.soa.esb.services.persistence;

public interface MessageStore {
  public URI addMessage(Message message);
  public Message getMessage(URI uid) throws Exception;
}
```

It can read and write messages, returning or taking a standard URI. This URI is used as the "key" for that message in the database, for the default database implementation.

The class which implements this interface, providing the out of the box implementation, can be found in the Services tree under the package `org.jboss.internal.soa.esb.persistence.format.db`. The methods in this implementation make the required DB connections (using a pooled Database Manager DBConnectionManager), inserting the Message, and retrieving the message.

To configure your Message Store, you can change and override the default service implementation through the following settings found in the `jbossesb-properties.xml`:

```
<properties name="dbstore">
<property name="org.jboss.soa.esb.persistence.messagestore.factory"
value="org.jboss.internal.soa.esb.persistence.format.MessageStoreFac
toryImpl"/>
<property name="org.jboss.soa.esb.persistence.db.connection.url"
value="jdbc:hsqldb:hsql://localhost:9001/jbossesb"/>
<property name="org.jboss.soa.esb.persistence.db.jdbc.driver"
value="org.hsqldb.jdbcDriver"/>
<property name="org.jboss.soa.esb.persistence.db.user" value="sa"/>
<property name="org.jboss.soa.esb.persistence.db.pwd"
value=""/>
<property name="org.jboss.soa.esb.persistence.db.pool.initial.size"
value="2"/>
<property name="org.jboss.soa.esb.persistence.db.pool.min.size"
value="2"/>
<property name="org.jboss.soa.esb.persistence.db.pool.max.size"
value="5"/>
<property name="org.jboss.soa.esb.persistence.db.pool.test.table"
value="pooltest"/>
<property
name="org.jboss.soa.esb.persistence.db.pool.timeout.millis"
value="5000"/>
</properties>
```

The section in the property file called “dbstore” has all the settings required by the database implementation of the message store. The standard settings, like URL, db user, password, pool sizes can all be modified here.

The scripts for the required database schema, are again, very simple. They can be found under ESB_ROOT/install/message-store/sql/<db_type>/ create_database.sql. Only Hypersonic SQL and PostgresSQL are provided, but you should be able to create your own database specific table definition very easily.

The structure of the table is:

```
Column Name Type
uuid TEXT
type TEXT
message text
```

the uuid column is used to store a unique key for this message, in the format of a standard URI. A key for a message would look like:

```
urn:jboss:esb:message:UID: + UUID.randomUUID()
```

This logic uses the new UUID random number generator in jdk 1.5.the type will be the type of the stored message. JBossESB ships with JBOSS_XML and JAVA_SERIALIZED currently.

The “message” column will contain the actual message content.

The supplied database message store implementation works by invoking a connection manager to your configured database. Supplied with Jboss ESB is a standalone connection manager, and another for using a JNDI datasource.

To configure the database connection manager, you need to provide the connection manager implementation in the *jbossesb-properties.xml*. The properties that you would need to change are:

```
<!--  connection manager type -->
<property name="org.jboss.soa.esb.persistence.db.conn.manager"
value="org.jboss.internal.soa.esb.persistence.format.db.StandaloneCo
nnectionManager"/>
<!--  property name="org.jboss.soa.esb.persistence.db.conn.manager"
value="org.jboss.soa.esb.persistence.manager.J2eeConnectionManager"/
-->
<!-- this property is only used if using the j2ee connection manager
-->
<property name="org.jboss.soa.esb.persistence.db.datasource.name"
value="java:/JBossesbDS"/>
```

The two supplied connection managers for managing the database pool are

```
org.jboss.soa.esb.persistence.manager.J2eeConnectionManager
org.jboss.soa.esb.persistence.manager.StandaloneConnectionManager
```

The Standalone manager uses C3PO to manage the connection pooling logic, and the J2eeConnectionManager uses a datasource to manage it's connection pool. This is intended for use when deploying your ESB endpoints inside a container such as Jboss AS or Tomcat, etc. You can plug in your own connection pool manager by implementing the interface:

```
org.jboss.internal.soa.esb.persistence.manager.ConnectionManager
```

Once you have implemented this interface, you update the properties file with your new class, and the connection manager factory will now use your implementation.

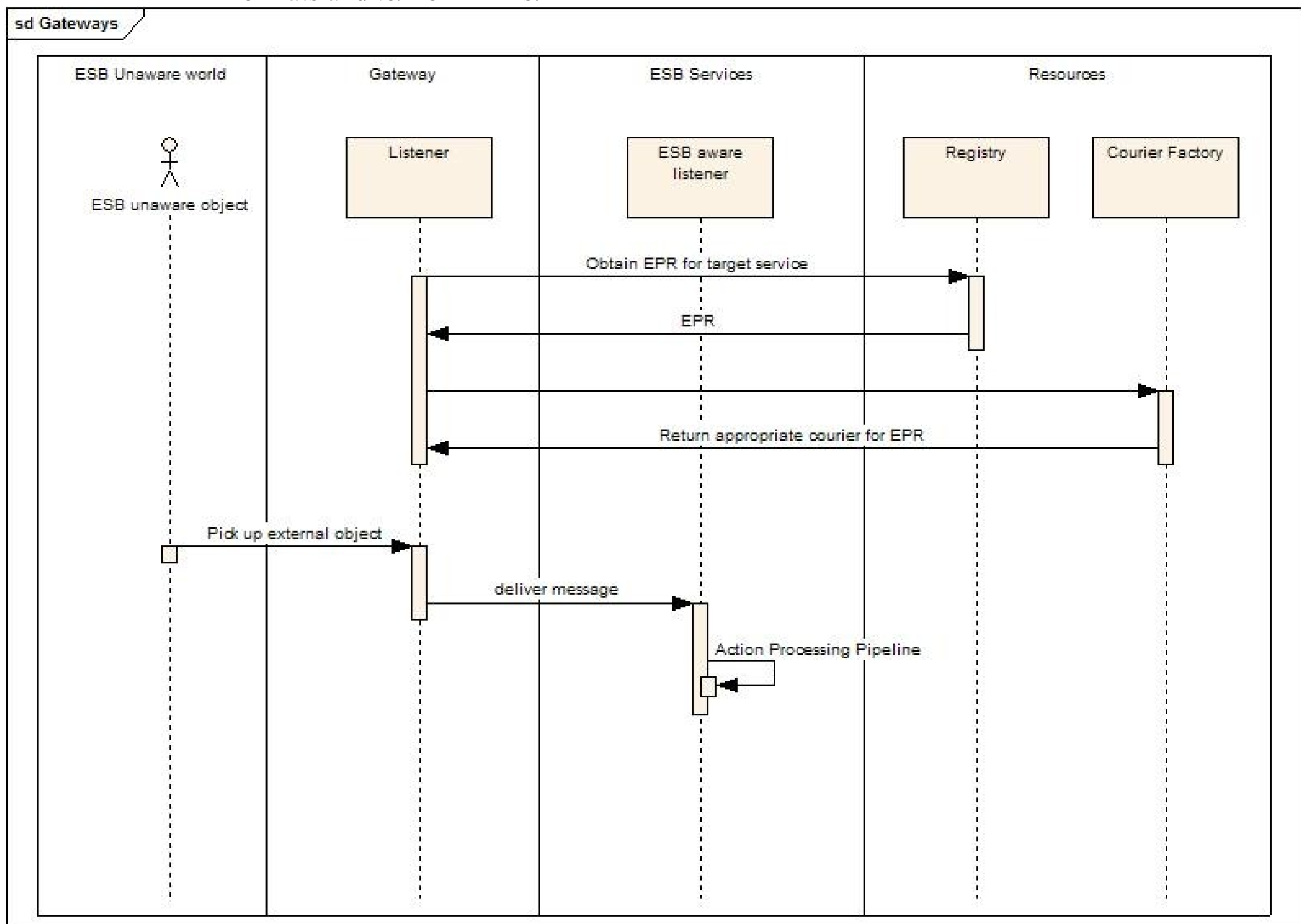
ESB-aware and ESB-unaware users

One of the aims of JBossESB is to allow a wide variety of clients and services to interact. JBossESB does not require that all such clients and services be written using JBossESB or any ESB for that matter. There is an abstract notion of an *Interoperability Bus* within JBossESB, such that endpoints that may not be JBossESB-aware can still be “plugged in to” the bus.

> Note: in what follows, the terms “within the ESB” or “inside the ESB” refer to ESB-aware endpoints.

All JBossESB-aware clients and services communicate with one another using Messages, to be described later. A Message is simply a standardized format for information exchange, containing a header, body (payload), attachments and other data. Furthemore, all JBossESB-aware services are identified using *Endpoint References* (EPRs), to be described later.

It is important for legacy interoperability scenarios that a SOA infrastructure such as JBossESB allow ESB-unaware clients to use ESB-aware services, or ESB-aware clients to use ESB-unaware services. The concept that JBossESB uses to facilitate this interoperability is through *Gateways*. A gateway is a service that can bridge between the ESB-aware and ESB-unaware worlds and translate to/from `Message` formats and to/from EPRs.



Endpoint References

All clients and services within JBossESB are addressed using *Endpoint References* (EPRs). An EPR has the following XML-based composition:

- **[address]** : URI (mandatory). An address URI that identifies the endpoint. This may be a network address or a logical address.
- **[reference properties]** : xs:any (0..unbounded). A reference may contain a number of individual properties that are required to identify the entity or

resource being conveyed. Reference identification properties are element information items that are named by QName and are required to properly dispatch messages to endpoints at the endpoint side of the interaction. Reference properties are provided by the issuer of the endpoint reference and are otherwise assumed to be opaque to consuming applications. The interpretation of these properties (as the use of the endpoint reference in general) is dependent upon the protocol binding and data encoding used to interact with the endpoint. Consuming applications should assume that endpoints represented by endpoint references with different [reference properties] may accept different sets of messages or follow a different set of policies, and consequently may have different associated metadata (e.g., WSDL, XML Schema, and WS-Policy policies).

- **[reference parameters]** : xs:any (0..unbounded). A reference may contain a number of individual parameters which are associated with the endpoint to facilitate a particular interaction. Reference parameters are element information items that are named by QName and are required to properly interact with the endpoint. Reference parameters are also provided by the issuer of the endpoint reference and are otherwise assumed to be opaque to consuming applications. The use of reference parameters is dependent upon the protocol binding and data encoding used to interact with the endpoint. Unlike [reference properties], the [reference parameters] of two endpoint references may differ without an implication that different XML Schema, WSDL or policies apply to the endpoints.

An EPR is essentially an address, to which messages are delivered by the ESB. How the message is delivered (e.g., FTP or JMS) is part of the *binding* of the EPR to messaging infrastructure and is typically reflected within the To component of the EPR, e.g., jms://foo.bar. The binding aspect is important because it imparts important semantic information as to the delivery characteristics for the message. For example, if using HTTP and the ultimate recipient of the message (e.g., business object) is not available, attempts to deliver the message will fail. If using JMS, it may be possible to deposit the message within a queue without delivery to the ultimate destination taking place. Obviously failure to deliver the message may subsequently occur, but unlike in the case of HTTP the sender will not be immediately notified of such a failure.

JBossESB uses the `org.jboss.soa.esb.addressing.EPR` and `org.jboss.soa.esb.addressing.PortReference` classes to represent endpoint references.

```
public class EPR
{
  public EPR ();
  public EPR (PortReference addr);
  public EPR (URI uri);

  public void setAddr (PortReference uri);
  public PortReference getAddr () throws URISyntaxException;

  public void copy (EPR from);

  public boolean equals (Object obj);
}
```

Note: The use of EPRs is based on the WS-Addressing specification from the W3C. However, in the 4.0 release the JBossESB implementation of EPRs is closer to the 2004 version of the specification from IBM, Microsoft et al.

Mapping of EPR to Service

How services map to EPRs can be a very important aspect of any application based on Service Oriented Architecture principles. Too tight a coupling can lead to brittle applications, whereas too loose a coupling can result in more development effort at the higher levels of the application.

It has long been recognized that the World Wide Web is probably the most successful distributed system created. It is inherently loosely coupled (clients and servers frequently interact across the globe) and highly scaleable (many thousands of Web sites). There are a number of factors that can be attributed to the Web's success, but two of the most important are:

> Sessions between clients and servers are maintained only long enough to transfer an HTML page and are dropped immediately afterward. This means that costly resources (e.g., TCP/IP connections, threads, processes) are not maintained for long durations, particularly when there are many users interacting with a service.
>
> Server interactions are either stateless, meaning that any instance of a Web server offering a particular service, e.g., airline reservation, can field the request, or information required to identify a previous user (and possibly state) is propagated with the invocation, e.g., the cookie.

Both of these factors mean that clusters of servers can relatively easily be used to distribute the load and provide improved availability/fault-tolerance to users. Web servers offering critical services are typically deployed over a cluster of machines. A locally distributed cluster of machines with the illusion of a single IP address and capable of working together to host a Web site provides a practical way of scaling up processing power and sharing load at a given site. Commercially available server clusters rely on a specially designed gateway router to distribute the load using a mechanism known as *network address translation* (NAT). The mechanism operates by editing the IP headers of packets so as to change the destination address before the IP to host address translation is performed. Similarly, return packets are edited to change their source IP address. Such translations can be performed on a per session basis so that all IP packets corresponding to a particular session are consistently redirected.

Most proponents of Web Services agree that it is important that its architecture is as scalable and flexible as the Web. As a result, the current interaction pattern for Web Services is based on coarse-grained services or components. The architecture is deliberately not prescriptive about what happens behind service endpoints: Web Services are ultimately only concerned with the transfer of structured data between parties, plus any meta-level information to safeguard such transfers (e.g., by encrypting or digitally signing messages). This gives flexibility of implementation, allowing systems to adapt to changes in requirements, technology etc. without directly affecting users. Furthermore, most businesses will not want to expose their back-end implementation decisions and strategies to users for a variety of reasons.

In distributed systems such as CORBA, J2EE and DCOM, interactions are typically between stateful objects that resided within *containers*. In these architectures, objects are exposed as individually referenceable entities, tied to specific containers and therefore often to specific machines. Because most Web Services applications are written using object-oriented languages, it is natural to think about extending that architecture to Web Services. Therefore a service exposes *Web Services resources* that represent specific states. The result is that such architectures produce tight coupling between clients and services, making it difficult for them to scale to the level of the World Wide Web.

Right now there are two primary models for the session concept that are being defined by companies participating in defining Web services: the WS-Addressing EndpointReference with ReferenceProperties/ReferenceParameters and the WS-Context explicit context structure, both of which are supported within JBossESB. The WS-Addressing session model provides coupling between the web service endpoint information and the session data, which is analogous to object references in distributed object systems.

WS-Context provides a session model that is an evolution of the session models found in HTTP servers, transaction, and MOM systems. On the other hand, WS-Context allows a service client to more naturally bind the relationship to the service dynamically and temporarily. The client's communication channel to the service is not impacted by a specific session relationship.

This has important implications as we consider scaling Web services from intra-domain deployments to general services offered on the Internet. The current interaction pattern for Web Services is based on coarse-grained services or components. The architecture is deliberately not prescriptive about what happens behind service endpoints: Web Services are ultimately only concerned with the transfer of structured data between parties, plus any meta-level information to safeguard such transfers (e.g., by encrypting or digitally signing messages). This gives flexibility of implementation, allowing systems to adapt to changes in requirements, technology etc. without directly affecting users. It also means that issues such as whether or not a service maintains state on behalf of users or their (temporally bounded) interactions, has been an implementation choice not typically exposed to users.

If a session-like model based on WS-Addressing were to be used when interacting with stateful services, then the tight coupling between state and service would impact on clients. As in other distribution environments where this model is used (e.g., CORBA or J2EE), the remote reference (address) that the client has to the service endpoint *must* be remembered by the client for subsequent invocations. If the client application interacts with multiple services within the same logical session, then it is often the case that the state of a service has relevance to the client only when used in conjunction with the associated states of the other services. This necessarily means that the client must remember each service reference and somehow associate them with a specific interaction; multiple interactions will obviously result in different reference sets that may be combined to represent each sessions.

For example, if there are N services used within the same application session, each maintaining m different states, the client application will have to maintain N*m reference endpoints. It is worth remembering that the initial service endpoint references will often be obtained from some bootstrap process such as UDDI. But in

this model, these references are stateless and of no use beyond starting the application interactions. Subsequent visits to these sites that require access to specific states must use different references in the WS-Addressing model.

This obviously does not scale to an environment the size of the Web. However, an alternative approach is to use WS-Context and continue to embrace the inherently loosely-coupled nature of Web Services. As we have shown, each interaction with a set of services can be modeled as a session, and this in turn can be modeled as a WS-Context activity with an associated context. Whenever a client application interacts with a set of services within the same session, the context is propagated to the services and they map this context to the necessary states that the client interaction requires.

How this mapping occurs is an implementation specific choice that need not be exposed to the client. Furthermore, since each service within a specific session gets the same context, upon later revisiting these services and providing the same context again, the client application can be sure to return to a consistent set of states. So for the N services and m states in our previous example, the client need only maintain N endpoint references and as we mentioned earlier, typically these will be obtained from the bootstrap process anyway. Thus, this model scales much better.

Gateways to the ESB

Not all users of JBossESB will be ESB-aware. In order to facilitate those users interacting with services provided by the ESB, JBossESB has the concept of a Gateway: specialised servers that can accept messages from non-ESB clients and services and route them to the required destination.

A Gateway is a specialised listener process, that behaves very similarly to an ESB aware listener. There are some important differences however:

- Gateway classes can pick up arbitrary objects contained in files, JMS messages, SQL tables etc (each 'gateway class' is specialized for a specific transport), whereas JBossESB listeners can only process JBossESB normalized `Messages` as described in "The Message" section of this document. However, those `Messages` can contain arbitrary data.

- Only one action class is invoked to perform the 'message composing' action. ESB listeners are able to execute an action processing pipeline.

- Objects that are 'picked up' will be used to invoke a single 'composer class' (the action) that will return an ESB `Message` object, which will be delivered to a target service that must be an ESB aware service. The target service defined at configuration time, will be translated at runtime into an EPR (or a list of EPRs) by the Registry. The underlying concept is that the EPR returned by the Registry is analogous to the 'toEPR' contained in the header of ESB `Messages`, but because incoming objects are 'ESB unaware' and there is thus no dynamic way to determine the toEPR, this value is provided to the gateway at configuration time and included in all outgoing messages.

There are a few off the shelf composer classes: the default 'file' composer class will just package the file contents into the `Message` body; same idea for JMS messages.

Default message composing class for a SQL table row is to package contents of all columns specified in configuration, into a `java.util.Map`.

Although these default composer classes will be enough for most use cases, it is relatively straightforward for users to provide their own message composing classes. The only requirements are a) they must have a constructor that takes a single `ConfigTree` argument, and b) they must provide a '`Message` composing' method (default name is 'process' but this can be configured differently in the 'process' attribute of the <action> element within the `ConfigTree` provided at constructor time. The processing method must take a single argument of type `Object`, and return a `Message` value.

The Message

All interactions between clients and services within JBossESB occur through the exchange of messages. In order to encourage loose coupling we recommend a message-exchange pattern based on one-way messages, i.e., requests and responses are independent messages, correlated where necessary by the infrastructure or application. Applications constructed in this way are less brittle and can be more tolerant of failures, giving developers more flexibility in their deployment and message delivery requirements.

To ensure loose coupling of services and develop SOA applications, it is necessary to:

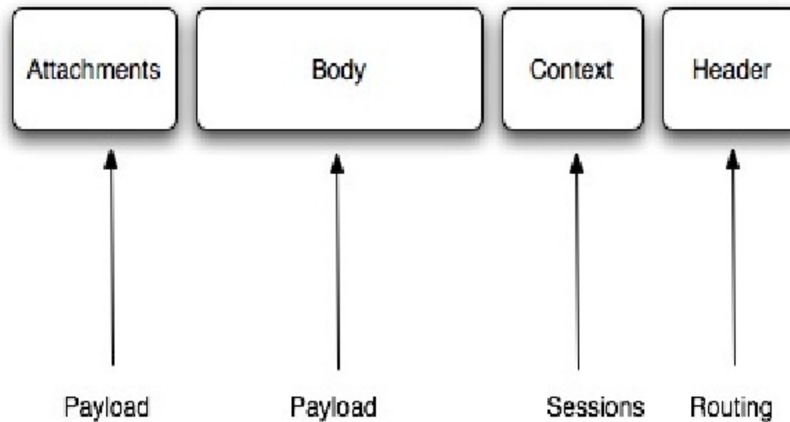
1. Use one-way message exchanges rather than request-response.
2. Keep the contract definition within the exchanged messages. Try not to define a service interface that exposed back-end implementation choices, because that will make changing the implementation more difficult later.
3. Use an extensible message structure for the message payload so that changes to it can be versioned over time, for backward compatibility.
4. Do not develop fine-grained services: this is not a distributed-object paradigm, which can lead to brittle applications.

In order to use a one-way message delivery pattern with requests and responses, it is obviously necessary to encode information about where responses should be sent. That information may be present in the message body (the payload) and hence dealt with solely by the application, or part of the initial request message and typically dealt with by the ESB infrastructure.

Therefore, central to the ESB is the notion of a *message*, whose structure is similar to that found in SOAP:

```
<xs:complexType name="Envelope">
  <xs:attribute ref="Header" use="required"/>
  <xs:attribute ref="Context" use="required"/>
  <xs:attribute ref="Body" use="required"/>
  <xs:attribute ref="Attachment" use="optional"/>
  <xs:attribute ref="Properties" use="optional"/>
  <xs:attribute ref="Fault" use="optional"/>
</xs:complexType>
```

Pictorially the basic structure of the Message can be represented as shown below. In the rest of this section we shall examine each of these components in more detail.



Each message is an implementation of the `org.jboss.soa.esb.message.Message` interface. Within that package are interfaces for the various fields within the `Message` as shown below:

```
public interface Message
{
  public Header getHeader ();
  public Context getContext ();
  public Body getBody ();
  public Fault getFault ();
  public Attachment getAttachment ();
  public URI getType ();
  public Properties getProperties ();
}
```

The `Header` contains routing and addressing information for this message. As we saw earlier, JBossESB uses an addressing scheme based on the WS-Addressing standard from W3C. We shall discuss the `org.jboss.soa.esb.addressing.Call` class in the next section.

```
public interface Header
{
  public Call getCall ();
  public void setCall (Call call);
}
```

The `Context` contains session related information, such as transaction or security contexts.

> Note: The 4.0 release of JBossESB does not support user-enhanced `Contexts`. This will be a feature of the next release.

The `Body` typically contains the payload of the message. It may contain a byte array for arbitrary data. How that array is interpreted by the service is implementation specific and outside the scope of the ESB to enforce. It may also contain a list of

Objects of arbitrary types. How these objects are serialized to/from the message body when it is transmitted is up to the specific Object type.

```
public interface Body
{
  public void add (String name, Object value);
  public Object get (String name);
  public Object remove (String name);
  public void setContents (byte[] content);
  public byte[] getContents ();
  public void replace (Body b);
  public void merge (Body b);
}
```

The `Fault` can be used to convey error information.

```
public interface Fault
{
  public URI getCode ();
  public void setCode (URI code);

  public String getReason ();
  public void setReason (String reason);
}
```

A set of message properties, which can be used to define additional meta-data for the message.

```
public interface Properties
{
  public Object getProperty(String name);
  public Object getProperty(String name, Object defaultVal);

  public Object setProperty(String name, Object value);
  public Object remove(String name);

  public int size();
  public String[] getNames();
}
```

Messages may contain attachments that do not appear in the main payload body. For example, binary document formats, zip files etc. The `Attachment` interface supports both named and unnamed attachments.

```
public interface Attachment
{
  Object get(String name);
  Object put(String name, Object value);

  Object remove(String name);

  String[] getNames();

  Object itemAt (int index) throws IndexOutOfBoundsException;
  Object removeItemAt (int index) throws IndexOutOfBoundsException
  Object replaceItemAt(int index, Object value)
```

```
                                    throws IndexOutOfBoundsException;

  void addItem    (Object value);
  void addItemAt  (int index, Object value)
                             throws IndexOutOfBoundsException;

  public int getNamedCount();
}
```

The Message Header

As we saw above, the `Header` of a `Message` contains a reference to the `org.jboss.soa.esb.addressing.Call` class:

```
public class Call
{
  public Call ();
  public Call (EPR epr);

  public void setTo (EPR epr);
  public EPR getTo () throws URISyntaxException;

  public void setFrom (EPR from);
  public EPR getFrom () throws URISyntaxException;

  public void setReplyTo (EPR replyTo);
  public EPR getReplyTo () throws URISyntaxException;

  public void setFaultTo (EPR uri);
  public EPR getFaultTo () throws URISyntaxException;

  public void setRelatesTo (URI uri);
  public URI getRelatesTo () throws URISyntaxException;

  public void setAction (URI uri);
  public URI getAction () throws URISyntaxException;

  public void setMessageID (URI uri);
  public URI getMessageID () throws URISyntaxException;

  public void copy (Call from);
}
```

The properties below support one way, request reply, and any other interaction pattern:

- **[To]** : URI (mandatory). The address of the intended receiver of this message.
- **[From]** : endpoint reference (0..1). Reference of the endpoint where the message originated from.
- **[ReplyTo]** : endpoint reference (0..1). An endpoint reference that identifies the intended receiver for replies to this message. If a reply is expected, a message must contain a [ReplyTo]. The sender must use the contents of the [ReplyTo] to formulate the reply message. If the [ReplyTo] is absent, the contents of the [From] may be used to formulate a message to the source.

This property may be absent if the message has no meaningful reply. If this property is present, the [MessageID] property is required.

- **[FaultTo]** : endpoint reference (0..1). An endpoint reference that identifies the intended receiver for faults related to this message. When formulating a fault message the sender must use the contents of the [FaultTo] of the message being replied to to formulate the fault message. If the [FaultTo] is absent, the sender may use the contents of the [ReplyTo] to formulate the fault message. If both the [FaultTo] and [ReplyTo] are absent, the sender may use the contents of the [From] to formulate the fault message. This property may be absent if the sender cannot receive fault messages (e.g., is a one-way application message). If this property is present, the [MessageID] property is required.

- **[Action]** : URI (mandatory). An identifier that uniquely (and opaquely) identifies the semantics implied by this message.

- **[MessageID]** : URI (0..1). A URI that uniquely identifies this message in time and space. No two messages with a distinct application intent may share a [MessageID] property. A message may be retransmitted for any purpose including communications failure and may use the same [MessageID] property. The value of this property is an opaque URI whose interpretation beyond equivalence is not defined. If a reply is expected, this property must be present.

> Note: In the 4.0 release of JBossESB not all of the routing and addressing rules are applied by the ESB.

The Message payload

From an application/service perspective the message payload is a combination of the `Body` and `Attachments`. In this section we shall give an overview of best practices when constructing and using the message payload.

The byte array component of the `Body` is a convenience. It allows an unnamed and arbitrary encoding of information to be inserted within the payload. It is neither a recommended nor discouraged practice to use the `setContents/getContents` methods. Neither is the setting of a byte array necessary for inserting named objects within the rest of the payload. The two approaches can be used together or in isolation. The best approach will depend upon the service or application being developed.

More complex content may be added through the `add` method, which supports named `Objects`. Names must be unique on behalf of a given `Message` or an appropriate exception will be thrown. Using `<name, Object>` pairs allows for a finer granularity of data access. The type of `Objects` that can be added to the `Body` can be arbitrary: they do not need to be Java Serializable. However, in the case where non-Serializable `Objects` are added, it is necessary to provide JBossESB with the ability to marshal/unmarshal the `Message` when it flows across the network. See the section of Message Formats for more details.

> Caution: we discourage the general use of Serialized Java objects in messages because it constrains the service implementations. Use with care.

In general you will find it easier to work with the `Message Body` through the named `Object` approach. You can add, remove and inspect individual data items within the `Message` payload without having to decode the entire `Body`. Furthermore, you can combine named `Objects` within the payload with the byte array.

Attachments are additional information that flows with the message but is not embedded within the `Body`. Although ESB meta-data may be encoded within an attachment, e.g., security tokens, this is not their normal use case: attachments should be considered an adjunct to the `Body`. As with the `Body`, `Attachments` can be uniquely named. Furthermore it is possible to iterate over all of a message's attachments.

Attachments can be used to contain data that naturally represents a resource in its own right or which is cumbersome to represent within the primary message body. The latter can be due to the size, type, or format of the data; a secondary part may be an audio clip, an image, or a very large view of a database, for example.

While the attachment relationship is expected to be commonly used, the model makes no assumption about the nature of the semantic relationship between the primary message body and attachments, or between attachments in the same message.

The compound message structure model does not require that a receiver process, dereference, or otherwise verify any attachment parts of a compound message structure. It is up to the receiver to determine, based on the processing context provided by the primary message `Body`, which operations must be performed (if any) on the attachment(s).

> Note: in the 4.0 release of JBossESB only Java Serialized objects may be attachments. This restriction will be removed in the next release.

The MessageFactory

Internally to an ESB component, the message is a collection of Java objects. However, messages need to be serialized for a number of reasons, e.g., transmitted between address spaces (processes) or saved to a persistent datastore for auditing or debugging purposes. The external representation of a message may be influenced by the environment in which the ESB is deployed. Therefore, JBossESB does not impose a specific normalized message format, but supports a range of them.

All implementations of the `org.jboss.soa.esb.message.Message` interface are obtained from the `org.jboss.soa.esb.message.format.MessageFactory` class:

```
public abstract class MessageFactory
{
  public abstract Message getMessage ();
  public abstract Message getMessage (URI type);

  public static MessageFactory getInstance ();
}
```

Message serialization implementations are uniquely identified by a URI. The type of implementation required may be specified when requesting a new instance, or the configured default implementation may be used. Currently JBossESB provides two implementations, which are defined in the `org.jboss.soa.esb.message.format.MessageType` class:

1. `MessageType.JBOSS_XML`: this uses an XML representation of the `Message` on the wire. The schema for the message is defined in the `message.xsd` within the `schemas` directory.

2. `MessageType.JAVA_SERIALIZED`: this implementation requires that all components of a `Message` are `Serializable`. It obviously requires that recipients of this type of `Message` have sufficient information (the Java classes) to be able to de-serialize the `Message`.

Other `Message` implementations may be provided at runtime through the `org.jboss.soa.esb.message.format.MessagePlugin`:

```
public interface MessagePlugin
{
  public static final String MESSAGE_PLUGIN =
                "org.jboss.soa.esb.message.format.plugin";

  public Message getMessage ();
  public URI getType ();
}
```

Each plug-in must uniquely identify the type of `Message` implementation it provides (via `getMessage`), using the `getType` method. Plug-in implementations must be identified to the system via the `jbossesb-properties.xml` file using property names with the `org.jboss.soa.esb.message.format.plugin` extension.

Message Formats

As mentioned previously, JBossESB supports two serialized message formats: `MessageType.JBOSS_XML` and `MessageType.JAVA_SERIALIZED`. In the following sections we shall look at each of these formats in more detail.

MessageType.JAVA_SERIALIZED

This implementation requires that all contents are Java Serializable. Any attempt to add a non-Serializable object to the `Message` will result in a `IllegalParameterException` being thrown.

MessageType.JBOSS_XML

This implementation uses an XML representation of the `Message` on the wire. The schema for the message is defined in the `message.xsd` within the `schemas` directory. Arbitrary objects may be added to the `Message`, i.e., they do not have to be Serializable. Therefore, it may be necessary to provide a mechanism to marshal/unmarshal such objects to/from XML when the `Message` needs to be serialized. This support can be provided through the `org.jboss.soa.esb.message.format.xml.marshal.MarshalUnmarshalPlugin`:

```
public interface MarshalUnmarshalPlugin
{
  public static final String MARSHAL_UNMARSHAL_PLUGIN =
           "org.jboss.soa.esb.message.format.xml.plugin";

  public boolean marshal (Element doc, Object param)
                                   throws MarshalException;

  public Object unmarshal (Element doc) throws UnmarshalException;

  public URI type ();
}
```

Note: Java Serialized objects are supported by default.

Plug-ins can be registered with the system through the `jbossesb-properties.xml` configuration file. They should have attribute names that start with the `MARSHAL_UNMARSHAL_PLUGIN`. When packing objects in XML, JBossESB runs through the list of registered plug-ins until it finds one that can deal with the object type (or faults). When packing, the name (type) of the plug-in that packed the object is also attached to facilitate unpacking at the `Message` receiver.

Data Transformation

Often clients and services will communicate using the same vocabulary. However, there are situations where this is not the case and on-the-fly transformation from one data format to another will be required. It is unrealistic to assume that a single data format will be suitable for all business objects, particularly in a large scale or long running deployment. Therefore, it is necessary to provide a mechanism for transforming from one data format to another.

In JBossESB this is the role the Transformation Service. This version of the ESB is shipped with an out-of-the-box Transformation Service based on Milyn Smooks. Smooks is a Transformation Implementation and Management framework. It allows you implement your transformation logic in XSLT, Java etc and provides a management framework through which you can centrally manage the transformation logic for your message-set.

For more details see the Message Transformation Guide.

Listener, Courier and Action Classes

Listeners encapsulate the endpoints for message reception. Upon receipt of a message, a Listener feeds that message into a “pipeline” of message processors that process the message before routing the result to the “replyTo” endpoint. The action processing that takes place in the pipeline may consist of steps wherein the message gets transformed in one processor, some business logic is applied in the next processor, before the result gets routed to the next step in the pipeline, or to another endpoint. Listeners rely on the Courier interface to pick up and deliver Messages.

The Courier interface encapsulates transport details from listeners.

```
public interface Courier
{
  public boolean deliver(Message message) throws CourierException;
}
```

The TwoWayCourier class that extends Courier, can also pickup Messages from an EPR. It is useful when a response is expected from the target of the outgoing Message (see for example `org.jboss.soa.esb.actions.CbrProxyAction`).

```
public interface TwoWayCourier extends Courier
{
   ...
   public Message pickup(long waitTime, EPR epr) throws
CourierException, CourierTimeoutException;
   ...
}
```

The CourierFactory class will return an appropriate Courier (or TwoWayCourier) class for specific EPRs.

```
public class CourierFactory
{
   ....

   public static Courier getCourier(EPR toEPR) throws
      CourierException
   {




      ...
   }

   public static TwoWayCourier getCourier(EPR toEPR, EPR replyToEPR)
      throws CourierException
   {
      ...
   }
   ...
}
```

The default internal TwoWayCourierImpl checks if the transport specific courier has a public 'void cleanup()' method and if so, invokes it to do housekeeping that need not be implemented for all transports. See `org.jboss.internal.soa.esb.couriers.JmsCourier` for example.

Transport specific classes that implement the Courier or TwoWayCourier interfaces can publish other utility methods that are specific for that particular transport.

As outlined above, the responsibility of a listener is to act as a message delivery endpoint and to deliver messages to an “Action Processing Pipeline”. Each listener configuration needs to supply information for:

- the Registry (see service-category, service-name, service-description and EPR-description tag names)
- instantiation of the listener class (see listenerClass tag name)
- the EPR that the listener will be servicing. This is transport specific. The following example corresponds to a JMS EPR (see connection-

factory, destination-type, destination-name, jndi-type, jndi-URL and message-selector tag names)

- the “action processing pipeline”. One or more <action> elements each that must contain at least the 'class' tagname that will determine which action class will be instantiated for that step in the processing chain

```
<ExampleActionConfig
   service-category="my category"
   service-name="testJmsGateway"
   service-description="My Example Service Name (optional)"
   epr-description="Verbose (optional) description of the EPR"



listenerClass="org.jboss.soa.esb.listeners.message.JmsQueueListener"

   connection-factory="ConnectionFactory"
   destination-type="queue"
   destination-name="queue/A"
   jndi-type="jboss"
   jndi-URL="localhost"
   message-selector="serviceId='xyz'"
   >
   <!--   -->
      <action
class="org.jboss.soa.esb.listeners.gateway.JmsGatewayListenerUnitTest$M
ockMessageAwareAction" process="writeToDisk" />

      <action  class="org.jboss.soa.esb.actions.Notifier"
okMethod="notifyOK">
            <NotificationList type="OK">
                  <target class="NotifyConsole" />
            </NotificationList>
      </action>
      </ExampleActionConfig>
```

This example configuration will instantiate a JmsQueueListener object (listenerClass attribute) that will wait for inconimg ESB Messages, serialized within a `javax.jms.ObjectMessage`, and will deliver each incoming message to an ActionProcessingPipeline consiting of two steps (<action> elements):

1. A MockMessageAwareAction (a trivial example follows)
2. An `org.jboss.soa.esb.actions.Notifier` the child <NotificationList> element is ignored by the pipeline, but passed to and used only by the class specified in the 'class' attribute

The following trivial action class will prove useful for debugging your XML action configuration

```
public class MockMessageAwareAction
{
    ConfigTree _config;
```

```
        public MockMessageAwareAction(ConfigTree config) { _config =
config; }

        public Message process (Message message) throws Exception
        {
         System.out.println(message.getBody().getContents());
         return message;
        }
    }
```

Action classes are the main way in which ESB users can tailor the framework to their specific needs. The ActionProcessingPipeline class will expect any action class to provide at least the following:

- A public constructor that takes a single argument of type ConfigTree
- One or more public methods that take a Message argument, and return a Message result

Optional public callback methods that take a Message argument will be used for notification of the result of the specific step of the processing pipeline (see items 5 and 6 below).

The `org.jboss,soa.esb.listeners.message.ActionProcessingPipeline` class will perform the following steps for all steps configured using <action> elements

1. Instantiate an object of the class specified in the 'class' attibute with a constructor that takes a single argument of type ConfigTree

2. Analyze contents of the 'process' attribute.

   Contents can be a comma separated list of public method names of the instantiated class (step 1), each of which must take a single argument of type Message, and return a Message object that will be passed to the next step in the pipeline

   If the 'process' attribute is not present, the pipeline will assume a single processing method called “process”

   Using a list of method names in a single <action> element has some advantages compared to using successive <action> elements, as the action class is instantiated once, and methods will be invoked on the same instance of the class. This reduces overhead and allows for state information to be kept in the instance objects.

   This approach is useful for user supplied (new) action classes, but the other alternative (list of <action> elements) continues to be a way of reusing other existing action classes.

3. Sequentially invoke each method in the list using the Message returned by the previous step

4. If value returned by any step is null, a warning message will be logged, the pipeline will stop processing, and no Message will be returned to the 'replyToEPR'

5. Callback method for success in each <action> element: If the list of methods in the 'process' attribute was executed successfully, the pipeline will analyze contents of the 'okMethod' attribute. If none is specified, processing will continue with the next <action> element. If a method name is provided in the 'okMethod' attribute, it will be invoked using the Message returned by the last method in step 3

6. Callback method for failure in each <action> element: If an Exception was thrown at any point in the pipeline, processing will be interrupted at that point, an error message will be logged, and contents of the 'exceptionMethod' tag will be analyzed and if present in the current action class, the method will be invoked using the Message returned by the last method in step 3 . At present time, if no exceptionMethod was specified, the only output will be the logged error. Upcoming releases will route the Message to the faultTo EPR contained in the Message.

Action classes supplied by users to tailor behaviour of the ESB to their specific needs, might need extra run time configuration (for example the Notifier class in the XML above needs the <NotificationList> child element). Each <action> element will utilize the attributes mentioned above and will ignore any other attributes and optional child elements. These will be however passed through to the action class constructor in the require ConfigTree argument. Each action class will be instantiated with it's corresponding <action> element and thus does not see (in fact must not see) sibling action elements.

Chapter 5

Process Engine Support

jBPM

Interoperation with jBPM is now possible using the CommandInterpreter and BaseActionHandler classes in the org.jboss.soa.esb.actions.jbpm package. Both use the org.jboss.soa.esb.util.jbpm.CommandVehicle class as a standard to talk each other. CommandVehicle has a constructor that takes a Message as argument, and inherits the toCommandMessage() method (that serializes the object into a Message) from it's parent class.

jBPM api calls that are available can be found in CommandVehicle.java. CommandInterpreter has now basic functionality, and is intended to grow to include more and more of the jBPM api power. Whenever a new call needs to be implemented we should a) add the operation in the 'Operation' enumeration in CommandVehicle and b) modify the process(Message) method in the CommandInterpreter class to do what's necessary to invoke the jBPM method, and place return values in the reply Message. Sometimes new getters/setters might also be needed in the CommandVehicle class.

The jbpm_simple1 quickstart illustrates how to use the jBPM interface classes to deply a process definition, create a process instance, signal a token (and/or process) through it's states, and at any point check if the process has completed (i.e. if it's in an end state).

The BaseActionHandler class extends jBPM ActionHandler, and can be used to send an ESB Message to a registered service from jBPM. It implements an outgoing only message; future versions will include quasi synchronous two way communication. It assumes that two jBPM context variables are set ('esbCategoryName' and 'esbServiceName'), and will include another context variable in the Message payload. The variable name to be included is rendered by this class' getContentVariableName() method and has a default value of "esbUserObjectVariable". Should users wish to include a different context variable in the message, simply extend this class, override the getContentVariableName() method, and use your class as the jBPM ActionHandler.

Using jBPM from within ESB allows (among several other features) persisting process state and handling timers and wait states; powerful features that are needed in (and essential part of) many business processes and don't seem to fall in the realm of ESB itself.

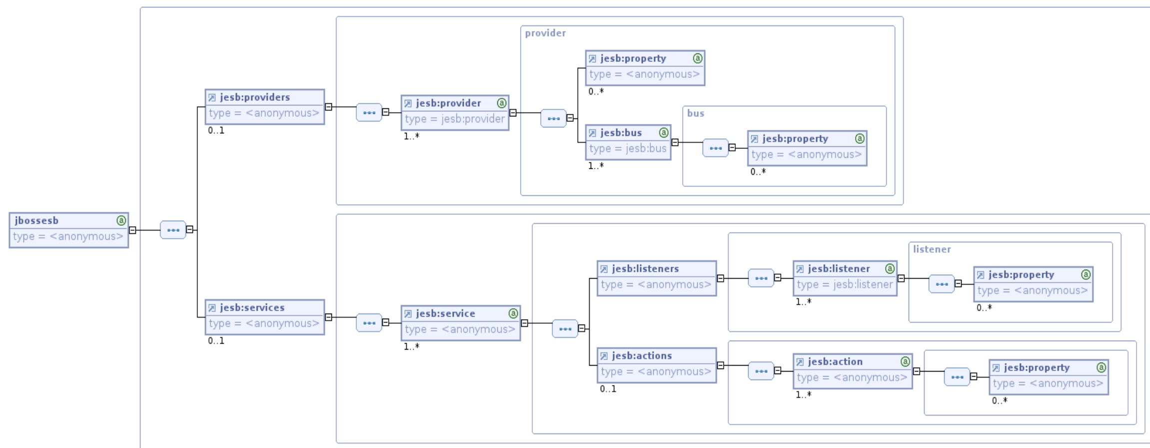
Chapter 6

Configuration

Overview

JBoss ESB 4.0 GA configuration is based on the jbossesb-1.0 XSD.

The basic elements/types of the configuration schema have the following relationships, with the <jbossesb> element/type at the root of the model.



JBoss ESB Configuration Model

From this, you can see that the model has 2 main sections:

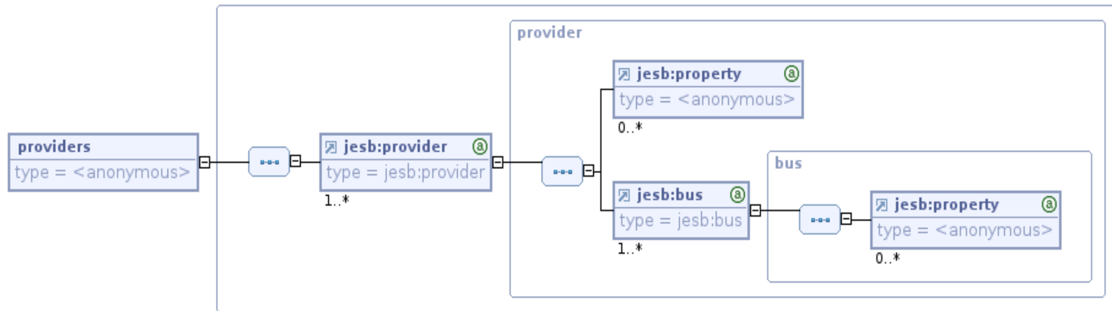
1. <providers>: This part of the model centrally defines all the message <bus>[6] providers used by the message <listener>s, defined within the <services> section of the model.

2. <services>: This part of the model centrally defines all of the services under the control of a single instance of JBoss ESB. Each <service> instance contains either a "Gateway" or "Message Aware" listener definition[7].

By far the easiest way to create configurations based on this model, is to use an XSD aware XML Editor such as the XML Editor in the Eclipse IDE. This provides the author with auto-completion features when editing the configuration. Right mouse-click on the file -> Open With -> XML Editor.

[6] A message bus defines the details of a specific message channel/transport.
[7] The fact that each <service> instance can only contain a single listener definition (Gateway or Message Aware) is a known bug in version 4.0 GA of JBoss ESB. The first patch of this release will contain a fix for this issue such that all of the Gateway Listeners (and the Message Aware Listener) can be defined under a single <service> instance.

Providers



The <providers> part of the configuration defines all of the bus <provider> and <bus> instances for a single instance of the ESB. A <provider> can contain multiple <bus> definitions. The <provider> can also be decorated with <property>[8] instances relating to provider specific properties that are common across all <bus> instances defined on that <provider> (e.g. for JMS - "connection-factory", "jndi-context-factory" etc). Likewise, each <bus> instance can be decorated with <property> instances specific to that <bus> instance (e.g. for JMS - "destination-type", "destination-name" etc).

As an example, a provider configuration for JMS would be as follows[9]:

```
<providers>
    <provider name="JBossMQ">
       <property name="connection-factory" value="ConnectionFactory" />
       <property name="jndi-URL" value="jnp://localhost:1099" />
       <property name="protocol" value="jms" />
       <property name="jndi-pkg-prefix" value="com.xyz" />

       <bus busid="local-jms">
          <property name="destination-type" value="topic" />
          <property name="destination-name" value="queue/B" />
          <property name="message-selector" value="service='Reconciliation'" />
       </bus>
    </provider>
</providers>
```

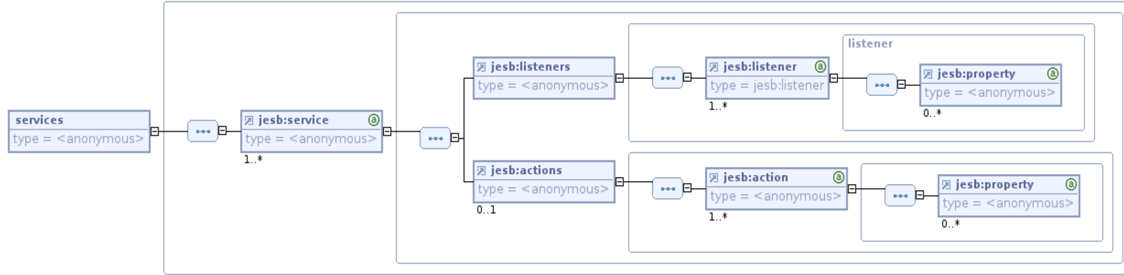
The above example uses the "base" <provider> and <bus> types. This is perfectly legal, but we recommend use of the specialized extensions of these types for creating real configurations, namely <jms-provider> and <jms-bus> for JMS. The most important part of the above configuration is the **busid** attribute defined on the <bus> instance. This is a required attribute on the <bus> element/type (including all of its specializations - <jms-bus> etc). This attribute is used within the <listener> configurations to refer to the <bus> instance on which the listener receives its messages. More on this later.

---

[8]A <property> is typically just a simple name-value-pair. However, it also supports free form (xsd:any) style content.

[9]This JMS example is only for demonstration purposes. We recommend that people use the more strongly typed JMS specific extensions of <provider> and <bus> i.e. <jms-provider> and <jms-bus>.

Services



The <services> part of the configuration defines each of the Services under the management of this instance of the ESB. It defines them as a series of <service> configurations. A <service> can also be decorated with the following attributes.

Name	***Description***	***Type***	***Required***
name	The Service Name under which the Service is Registered in the Service Registry.	xsd:string	true
category	The Service Category under which the Service is Registered in the Service Registry.	xsd:string	true
description	Human readable description of the Service. Stored in the Registry.	xsd:string	true

Service Attributes (<service>)

A <service> may define a set of <listeners>[10] and a set of <actions>. The configuration model defines a "base" <listener> type, as well as specializations for each of the main supported transports i.e. <jms-listener>, <sql-listener> etc.[11]

The "base" <listener> defines the following attribute. These attribute definitions are inherited by all <listener> extensions.

Name	***Description***	***Type***	***Required***
name	The name of the listener. This attribute is required primarily for logging purposes.	xsd:string	true
busrefid	Reference to the **busid** of the <bus> through which the listener instance receives messages.	xsd:string	true

[10]As stated earlier, in the 4.0 GA release of the ESB, each <service> instance can only contain a single listener definition under the <listeners> section (Gateway or Message Aware). This is a known bug and will be fixed in the first 4.0 GA patch releaseof this release will contain a fix for this issue such that all of the Gateway Listeners (and the Message Aware Listener) can be defined under a single <service> instance.

[11]New listener implementations (as well as all existing) can be supported using the "base" listener type. The specializations are only there to aid usability and

Name	Description	Type	Required
maxThreads	The max number of concurrent message processing threads that the listener can have active.	xsd:int	True
is-gateway	Whether or not the listener instance is a "Gateway" or "Message Aware" Listener. *See footnote #5.*	xsd:boolean	true

Listener Attributes (<listener>)

Listeners can define a set of zero or more <property> elements (just like the <provider> and <bus> elements/types). These are used to define listener specific properties.

An example of a <listener> reference to a <bus> can be seen in the following illustration (using "base" types only).

```
1 <?xml version = "1.0" encoding = "UTF-8"?>
2 <jbossesb xmlns="http://anonsvn.labs.jboss.com/labs/jbossesb/trunk/product/etc/schemas/xml/jbossesb-1.0.xsd">
3
4     <providers>
5         <provider name="JBossMQ">
6             <property name="connection-factory" value="ConnectionFactory" />
7             <property name="jndi-URL" value="jnp://localhost:1099" />
8             <property name="protocol" value="jms" />
9
10            <bus busid="local-jms">
11                <property name="destination-type" value="topic" />
12                <property name="destination-name" value="queue/B" />
13                <property name="message-selector" value="service='Reconciliation'" />
14            </bus>
15        </provider>
16    </providers>
17    <services>
18        <service category="Bank" name="Reconciliation"
19                 description="Bank Reconciliation Service" is-gateway="false">
20
21            <listeners>
22                <listener name="Bank-Listener"
23                            busidref="local-jms"
24                            maxThreads="2">
25                </listener>
26            </listeners>
27
28            <actions>
29                ....
30            </actions>
31
32        </service>
33    </services>
34 </jbossesb>
```

A Service will do little without a list of one or more <actions>. The actions are effectively the “meat” of the Service. <action>s typically contain the logic for processing the payload of the messages received by the service (through it's listeners). Alternatively, it may contain the transformation or routing logic for messages to be consumed by an external Service/entity.

The <action> element/type defines the following attributes.

Name	***Description***	***Type***	***Required***
name	The name of the action. This attribute is required primarily for logging purposes.	xsd:string	true
class	The *org.jboss.soa.esb.actions.ActionProcessor* implementation class name.	xsd:string	true
process	The name of the “process” method that will be reflectively called for message processing. (Default is the “process” method as defined on the *ActionProcessor* class)[12].	xsd:int	false

In a list of <action> instances within an <actions> set, the actions are called (their “process” method is called) in the order in which the <action> instances appear in the <actions> set. The message returned from each <action> is used as the input message to the next <action> in the list.

Like a number of other elements/types in this model, the <action> type can also contain zero or more <property> element instances. The <property> element/type can define a standard name-value-pair, or contain free form content (xsd:any). According to the XSD, this free form content is valid child content for the <property> element/type no matter where it is in the configuration (on any of <provider>, <bus>, <listener> and any of their derivatives). However, it is only on <action> defined <property> instances that this free form child content is used.

[12] It is recommended to not use the optional “process” attribute on <action> configurations. Instead, stick with the default “process” method as explicitly defined on the *ActionProcessor* implementation. It is very likely that this “process” attribute will be removed from this type in a future release. Reflection is great, but the lack of compile time checking is not adequately repaid in this case. If you find that you need to define more than one “process” method on an *ActionProcessor* implementation, you should consider the possibility that the action in question is really 1+ separate actions.

As stated in the <action> definition above, actions are implemented through implementing the *org.jboss.soa.esb.actions.ActionProcessor* class. All implementations of this interface must contain a public constructor of the following form:

```
public ActionZ(org.jboss.soa.esb.helpers.ConfigTree configuration);
```

It is through this constructor supplied *ConfigTree* instance that all of the action attributes are supplied, including the free form content from the action <property> instances. The free form content is supplied as child content on the *ConfigTree* instance[13].

So an example of an <actions> configuration might be as follows:

```
<actions>
    <action name="MyAction-1" class="com.acme.MyAction1"/>
    <action name="MyAction-2" class="com.acme.MyAction2">
        <property name=”propA” value=”propAVal” />
    </action>
    <action name="MyAction-3" class="com.acme.MyAction3">
        <property name=”propB” value=”propBVal” />
        <property name=”propC”>
            <!-- Free form child content... -->
            <some-free-form-element>zzz<some-free-form-element>
        </property>
    </action>
</actions>
```

[13]In its current implementation, it really only makes sense to supply free form content on one <property> instance within a list of <action> <property> instances. If defined on more than one property, the child content will be appended to the child content of the *ConfigTree* instance supplied to the action.

Transport Specific Type Implementations

The JBoss ESB configuration model defines transport specific specializations of the "base" types <provider>, <bus> and <listener> (JMS, SQL etc). This allows us to have stronger validation on the configuration, as well as making configuration easier for those that use an XSD aware XML Editor (e.g. the Eclipse XML Editor). These specializations explicitly define the configuration requirements for each of the transports supported by JBoss ESB out of the box. It is recommended to use these specialized types over the "base" types when creating JBoss ESB configurations, the only alternative being where a new transport is being supported outside an official JBoss ESB release.

The same basic principals that apply when creating configurations from the "base" types also apply when creating configurations from the transport specific alternatives:

1. Define the provider configuration e.g. <jms-provder>.

2. Add the bus configurations to the new provider (e.g. <jms-bus>), assigning a unique **busid** attribute value.

3. Define your <services> as normal, adding transport specific listener configurations (e.g. <jms-listener> that reference (using **busrefid**) the new bus configurations you just made e.g. <jms-listener> referencing a <jms-bus>.

The only rule that applies when using these transport specific types is that you cannot cross reference from a listener of one type, to a bus of another type i.e. you can only reference a <jms-bus> from a <jms-listener>. A runtime error will result where cross references are made.

So the transport specific implementations that are in place in this release are:

1. JMS: <jms-provider>, <jms-bus>, <jms-listener> and <jms-message-filter>: The <jms-message-filter> can be added to either the <jms-bus> or <jms-listener> elements. Where the <jms-provider> and <jms-bus> specify the JMS connection properties, the <jms-message-filter> specifies the actual message QUEUE/TOPIC and selector details.

2. SQL: <sql-provider>, <sql-bus>, <sql-listener> and <sql-message-filter>: The <sql-message-filter> can be added to either the <sql-bus> or <sql-listener> elements. Where the <sql-provider> and <ftp-bus> specify the JDBC connection properties, the <sql-message-filter> specifies the message/row selection and processing properties[14].

3. FTP: <ftp-provider>, <ftp-bus>, <ftp-listener> and <ftp-message-filter>: The <ftp-message-filter> can be added to either the <ftp-bus> or <ftp-listener> elements. Where the <ftp-provider> and <ftp-bus> specify the FTP access properties, the <ftp-message-filter> specifies the message/file selection and processing properties[15].

[14]The message processing attributes on <sql-message-filter> should really be on the <sql-bus>. This will be rectified in the GA release.

4. File System: <fs-provider>, <fs-bus>, <fs-listener> and <fs-message-filter> The <fs-message-filter> can be added to either the <fs-bus> or <fs-listener> elements. Where the <fs-provider> and <sql-bus> specify the File System access properties, the <fs-message-filter> specifies the message/file selection and processing properties[16].

As you'll notice, all of the currently implemented transport specific types include an additional type not present in the "base" types, that being <*-message-filter>. This element/type can be added inside either the <*-bus> or <*-listener>. Allowing this type to be specified in both places means you can specify message filtering globally for the bus (for all listeners using that bus), or locally on a listener by listener basis.

TODO: List and describe the attributes for each of the transport specific types. For now, readers can use the jbossesb-1.0 XSD, which is fully annotated with descriptions of each of the attributes. Again, using an XSD aware XML Editor such as the Eclipse XML Editor makes working with these types far easier.

[15]The message processing attributes on <ftp-message-filter> should really be on the <ftp-bus>. This will be rectified in the GA release.

[16]The message processing attributes on <fs-message-filter> should really be on the <fs-bus>. This will be rectified in the GA release.

Transitioning From The Old Configuration Model

This section is aimed at developers that are familiar with the old JBoss ESB non-XSD based configuration model.

The old configuration model used a free form (un validateable) XML configuration with ESB components receiving thier configurations via an instance of *org.jboss.soa.esb.helpers.ConfigTree*. The new configuration model is XSD based, however the underlying component configuration pattern is still via an instance of *org.jboss.soa.esb.helpers.ConfigTree*. This means that at the moment, the XSD based configurations are mapped/transformed into *ConfigTree* style configurations.

Developers that were used to using the old model now need to keep the following in mind:

1. Read all of the docs on the new configuration model. Don't assume you can infer the new configurations based on your knowledge of the old.
2. The only location where free-form markup is supported in the new configuration is on the <property> element/type. This type is allowed on <provider>, <bus> and <listener> types (and sub-types). However, the only location in which <property> based free form markup is mapped into the *ConfigTree* configurations is where the <property> exists on an <action>. In this case, the <property> content is mapped into the target *ConfigTree* <action>. Note however, if you have 1+ <property> elements with free form child content on an <action>, all this content will be concatenated together on the target *ConfigTree* <action>.
3. When developing new Listener/Action components, you must ensure that the *ConfigTree* based configuration these components depend on can be mapped from the new XSD based configurations. An example of this is how in the *ConfigTree* configuration model, you could decide to supply the configuration to a listener component via attributes on the listener node, or you could decide to do it based on child nodes within the listener configuration – all depending on how you were feeling on the day. This type of free form configuration on <listener> components is not supports on the XSD to *ConfigTree* mapping i.e. the child content in the above example would not be mapped from the XSD configuration to the *ConfigTree* style configuration. Infact, the XSD configuration simply would not accept the arbitrary content, unless it was in a <property> and even in that case (on a <listener>), it would simply be ignored by the mapping code.

Frequently Asked Questions (FAQs)

Question 1: I was used to using the old configuration model. How do I transition to using the new XSD based model?

Answer: See Transitioning From The Old Configuration Model.

Question 2: Can I put whatever markup I like, wherever I like, in the new XSD based configuration?

Answer: No! The new XSD based configuration only supports free-form markup on <property> elements/types and even there, the XSD to *ConfigTree* mapping that's currently in place, only supports mapping from <property> elements contained within an <action> i.e. the free form <property> child content is not mapped from <bus> or <listener> elements.
See Transitioning From The Old Configuration Model.

Question 3: Why does the XSD based configuration specify <listeners> and <actions>, as well as an optional “service-class” attribute on the <service> type?

Answer: Sorry, but the answer to this question is quite convoluted. The reason the “service-class” attribute is on the <service> element is down to 2 factors:

1. A known issue in the ESB (http://jira.jboss.com/jira/browse/JBESB-280).
2. The need to be able to override the default listener class for a Gateway or Message Aware Listener.

In hindsight however, adding this attribute here may not have been the best workaround. Hopefully the “service-class” attribute is only a short term feature of the XSD configuration.

Question 4: Why does the XSD based configuration specify “target-service-name” and “target-service-category” attributes on the <service> type?

Answer: As a workaround to a known issue in the ESB (http://jira.jboss.com/jira/browse/JBESB-280).

Chapter 7

Glossary

▪ ACL	Access Control List. A mean of determining the appropriate access rights to a given object depending on certain aspects of the process that is making the request.
▪ Action Classes	A component that is responsible for doing a certain type of work after a receipt of a message inside the ESB.
▪ Bus	A subsystem that transfers data between computer components inside a computer or between computers. Unlike a point-to-point connection, a bus can logically connect several components over the same structure.
▪ Content Based Router (CBR)	A pluggable service inside the ESB that provides capabilities for message routing based on the content of the message.
▪ CORBA	Common Object Request Broker Architecture. A standard defined by the Object Management Group that enables software components written in multiple computer languages and running on multiple computers to interoperate.
▪ CORBA IDL	CORBA Interface Definition Language. A computer language used to describe a software component's interface. It describes an interface in a language-neutral way, enabling communication between software components written in different languages.
▪ EAI	Enterprise Application Integration. A practice that makes use of software and computer systems architectural principles to integrate a set of different enterprise computer applications.
▪ Endpoint Reference (EPR)	A standard XML structure used to identify and address services inside the ESB. This includes the destination address of the message, any additional parameters (reference properties) necessary to route the message to the destination, and optional metadata (reference parameters) about the service.
▪ ESB	Enterprise Service Bus. An abstraction layer on top of an implementation of an enterprise messaging system that provides the features with which Service Oriented Architectures may be implemented.
▪ Fault	A type of message that express an error condition

	inside a Web Service. Similar to the Exception object in some programming languages.
▪ Gateway	A specialized ESB listener process that can accept messages from non-ESB clients and services and route them to the required destination inside the ESB, taking care of the appropriate bridging of message types and EPRs.
▪ J2EE/JEE	Java Platform Enterprise Edition (formerly known as Java 2 Platform Enterprise Edition). A programming platform, based on the Java language, for developing and running distributed multi-tier Java applications. It is based largely on modular software components running on an application server.
▪ JBI	Java Business Integration. An API that provides a standard pluggable architecture to build integration systems that hosts service producers and consumers components. Components interoperate through mediated normalized message exchanges.
▪ JMS	Java Message Service. An API for sending messages between two or more systems.
▪ JTA	Java Transaction API. An API that allows distributed transactions to be done across multiple XA resources
▪ Listener Classes	A component that encapsulates the endpoints for message reception on the ESB.
▪ Message	A data item that is sent (usually asynchronously) to a communication endpoint. This concept is the higher-level version of a datagram except that messages can be larger than a packet and can optionally be made reliable, durable, secure, and/or transacted.
▪ Message Factory	A service inside the ESB that can build specific types of messages according to their serialization capabilities.
▪ Message Store	A pluggable service inside the ESB that persists messages for auditing and tracking purposes.
▪ MOM	Message Oriented Middleware. A software component that makes possible inter-application communication relying on asynchronous message-passing.
▪ Quality of Service	A term that refers to control mechanisms that can provide different priority to different users or data flows, or guarantee a certain level of performance to a data flow in accordance with requests from the application program.
▪ RPC	Remote Procedure Call. A protocol that allows a computer program running on one computer to call

	a subroutine on another computer without the programmer explicitly coding the details for this interaction.
▪ SCA	Service Component Architecture. A set of specifications that describe a model for building applications and systems using Service-Oriented Architecture. It encourages an SOA organization of applications based on components that offer their capabilities through service-oriented interfaces and which consume functions offered by other components through service-oriented interfaces, called service references.
▪ Service Registry	A persistent repository of Service information. Used by ESB components to publish, discover and consume services.
▪ SOA	Service Oriented Architecture. A perspective of software architecture that defines the use of loosely coupled software services to support the requirements of the business processes and software users. In an SOA environment, resources on a network are made available as independent services that can be accessed without knowledge of their underlying platform implementation.
▪ SOAP	A protocol for exchanging XML-based messages over computer network, normally using HTTP. SOAP forms the foundation layer of the Web services stack, providing the basic messaging framework.
▪ Transformation Service	A pluggable service inside the ESB that provides capabilities for transforming messages from one data format to another.
▪ UDDI	Universal Description, Discovery, and Integration. A platform-independent, XML-based registry and core Web Services standard. It is designed to be interrogated by SOAP messages and to provide access to Web Services Description Language documents describing the protocol bindings and message formats required to interact with the web services listed in its directory.
▪ WS-Addressing	A Web Service specification for addressing web services and messages in a transport-neutral manner. This specification enables messaging systems to support message transmission through networks that include processing nodes such as endpoint managers, firewalls, and gateways.
▪ WS-BPEL	Web Services Business Process Execution Language. A choreography language for the formal specification of business processes and business

	interaction protocols using Web Services. Thus BPEL's messaging facilities depend on the use of Web Services Description Language (WSDL) 1.1 to describe incoming and outgoing messages.
▪ WS-Context	A Web Service specification that provides a definition, a structuring mechanism, and a software service definition for organizing and sharing context across multiple Web Services endpoints. The context contains information (such as a unique identifier) that allows a series of operations to share a common outcome.
▪ WSDL	Web Services Description Language. An XML format for describing the public interface to a Web services based on how to communicate using the web service; namely, the protocol bindings and message formats required to interact with it.
▪ WS-Policy	A Web Service specification that allows web services to advertise their policies (on security, Quality of Service, etc.) and for web service consumers to specify their policy requirements.
▪ WS-Security	A Web Service specification that provides a set of mechanisms to secure SOAP message exchanges. Specifically, it describes enhancements to provide quality of protection through the application of message integrity, message confidentiality, and single message authentication to SOAP messages.
▪ WS-Trust	A Web Service specification that uses the secure messaging mechanisms of WS-Security to define additional primitives and extensions for the issuance, exchange and validation of security tokens.
▪ XA	An X/Open specification for distributed transaction processing. It describes the interface between the global transaction manager and the local resource manager to support a two-phase commit protocol.
▪ XML	Extensible Markup Language. A general-purpose markup language that supports a wide variety of applications. Its primary purpose is to facilitate the sharing of data across different information systems.

Architectural components 25
Configuring JBossESB 27
ESB Overview 15
Format adapters 43
JBossESB
  Access Control Lists 16
  contract definition language 18
  implementation flexibility 16
  multi-bus support 18
Rosetta
  history 25
SOA Overview 9
SOA Overview
  basics 13
  benefts 11
  Why SOA? 11