JBoss ESB 4.0 Beta 1

Programmers Guide

JBESB-PG-9/22/06

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 4.0 Beta 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 2006 JBoss Inc.

Contents

About This Guide 4

What This Guide Contains 4
Audience 4
Prerequisites 4
Organization 4
Documentation Conventions 5
Additional Documentation 5
Contacting Us 6

Service Oriented Architecture 8

Overview 8
Why SOA? 10
Basics of SOA 11
Advantages of SOA 12

The Enterprise Service Bus 14

Overview 14
Architectural requirements 16

When to use JBossESB 19

Introduction 19

JBossESB 23

Rosetta 23
JBossESB components 25
The Object Store 26
Configuration Table 35

Index 37

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 4.0 Beta 1.

> **Note:** For the beta release, we recommend that you use this manual in conjunction with the trailblazer example, the user forum (http://www.jboss.com/index.html?module=bb&op=viewforum&f=246) and the javadocs associated with the code.

Audience

This guide is most relevant to engineers who are responsible for using JBoss ESB 4.0 Beta 1 installations and want to know how it relates to SOA and ESB principles.

Prerequisites

None.

Organization

This guide contains the following chapters:

1. **Chapter 1, What is SOA?:** JBossESB is a SOA infrastructure. This chapter gives an overview of SOA and the benefits it can provide.
2. **Chapter 2, The Enterprise Service Bus:** an overview of what constitutes an ESB and how JBossESB may differ from traditional ESB definitions.
3. **Chapter 3, JBossESB core:** a description of the core components within JBossESB and how they are intended to be used.
4. **Chapter 4, Configuration:** a description of the configuration options within JBossESB.

Documentation Conventions

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.

The following conventions are used in this guide:

Table 1 Formatting Conventions

Additional Documentation

In addition to this guide, the following guides are available in the JBoss ESB 4.0 Beta 1 documentation set:

1. JBoss ESB 4.0 Beta 1 *Trailblazer Guide*: Provides guidance for using the trailblazer example.
2. JBoss ESB 4.0 Beta 1 *Getting Started Guide*: Provides a quick start reference to configuring and using the ESB.
3. JBoss ESB 4.0 Beta 1 *Configuring Hypersonic Guide*: This is necessary for setting up the Hypersonic database if you want to use it within the trailblazer.

Contacting Us

Questions or comments about JBoss ESB 4.0 Beta 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

[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.

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*.

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:

5. opening the door to new business opportunities by making it easy to connect with partners;
6. saving time and money by cutting software development time and consuming a service created by others;
7. 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:

- 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.
- 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.
- 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:

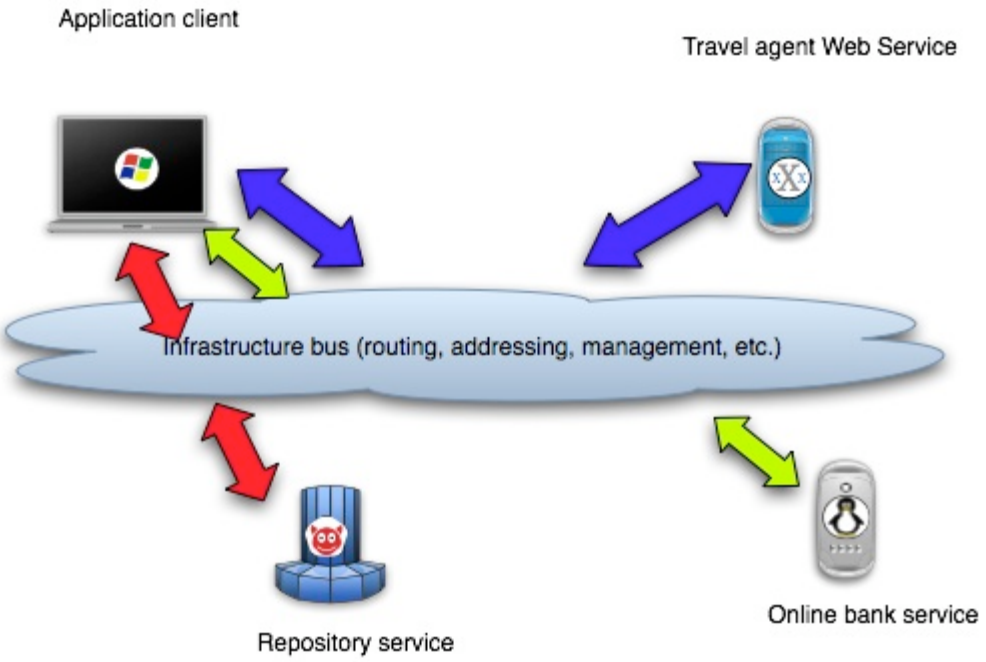
- Java specific.
- Run-time message mediator.
- Message translation.
- Security model translation.

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 is may be done in a traditional ESB, the SOA approach will win within JBossESB.

Therefore, in JBossESB mediation 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).

> **Note:** Content-based routing is not supported in the beta release of JBossESB.

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 realised and JBossESB does not mandate one implementation over another. Therefore, all capabilities will be accesses as services which will give plug-and-play configurability and extensibility options.

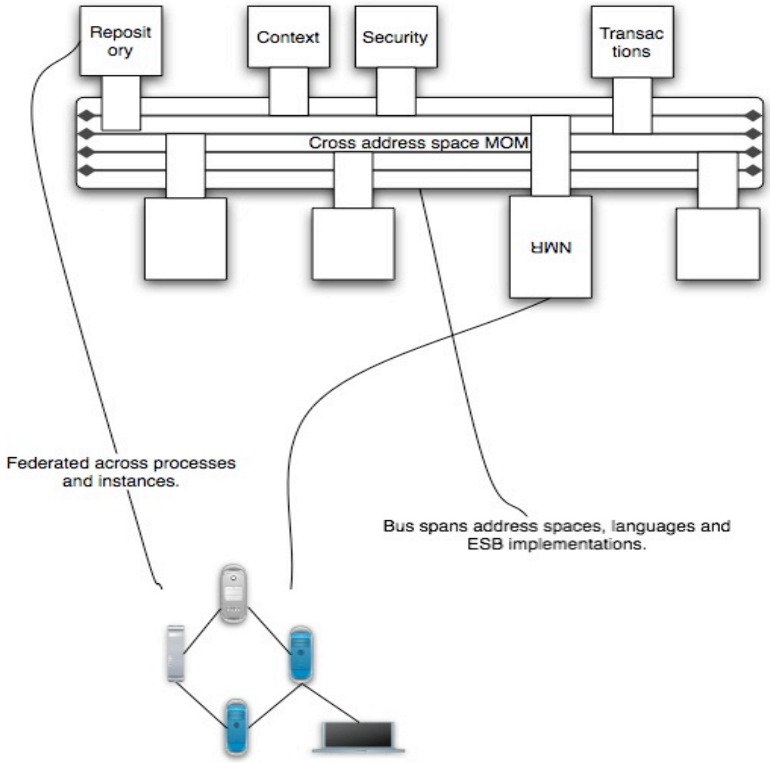


Figure 2: ESB components and multi-bus support.

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 unmarshal 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 will 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).

> **Note:** Support for the multiple bus abstraction will be available in the GA release of JBossESB. However, the beta release supports multiple messaging infrastructure implementations.

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 will allow this mapping to technology to be configurable and dynamic, i.e., it will support 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

[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.)

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.

Chapter 3

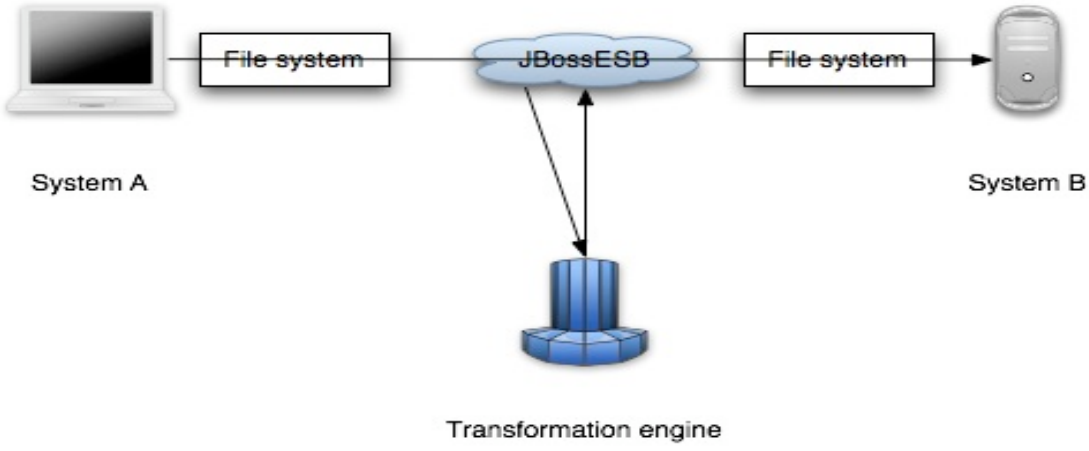
When to use JBossESB

Introduction

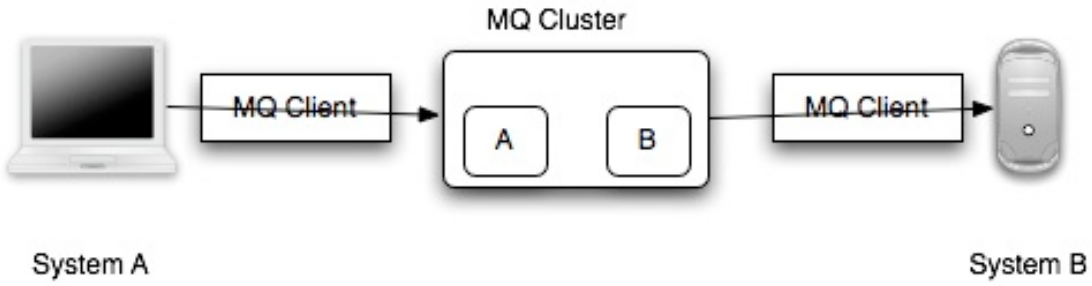
We have already discussed when SOA principles and an ESB implementation may be useful. The table below illustrates some further, concrete examples where JBossESB would be useful. Although these examples are specific to interactions between participants using non-interoperable JMS implementations, the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

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

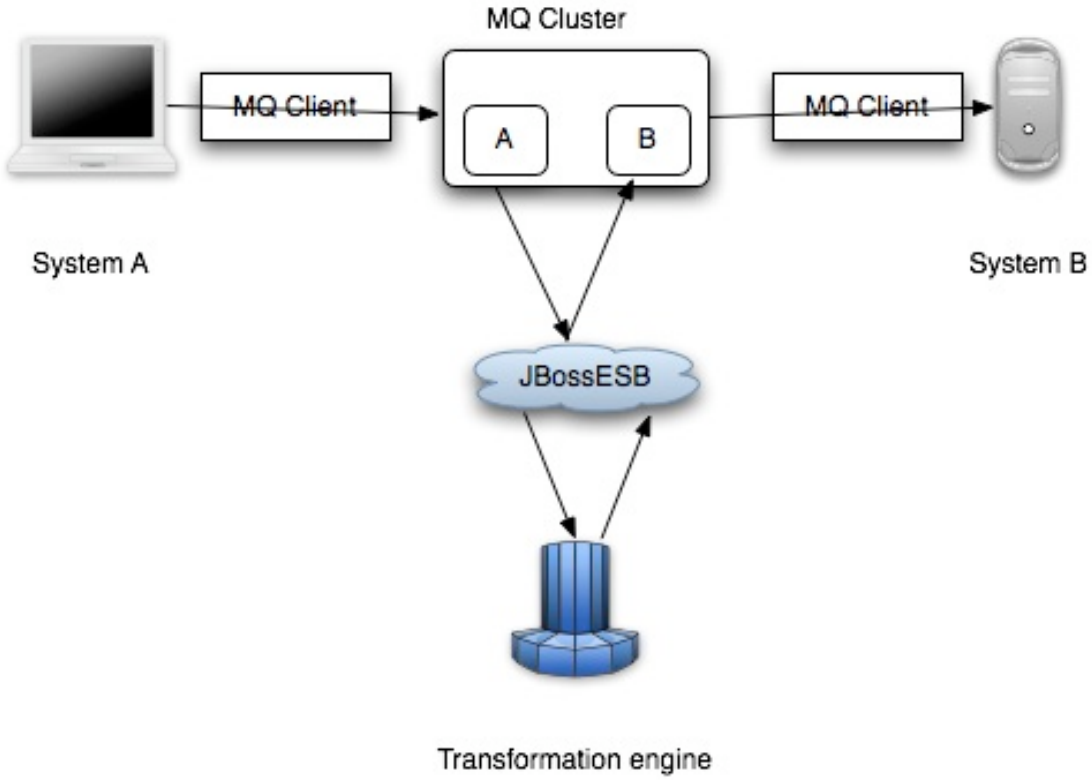
The next diagram illustrates how transformation can be injected into the same scenario using JBossESB.



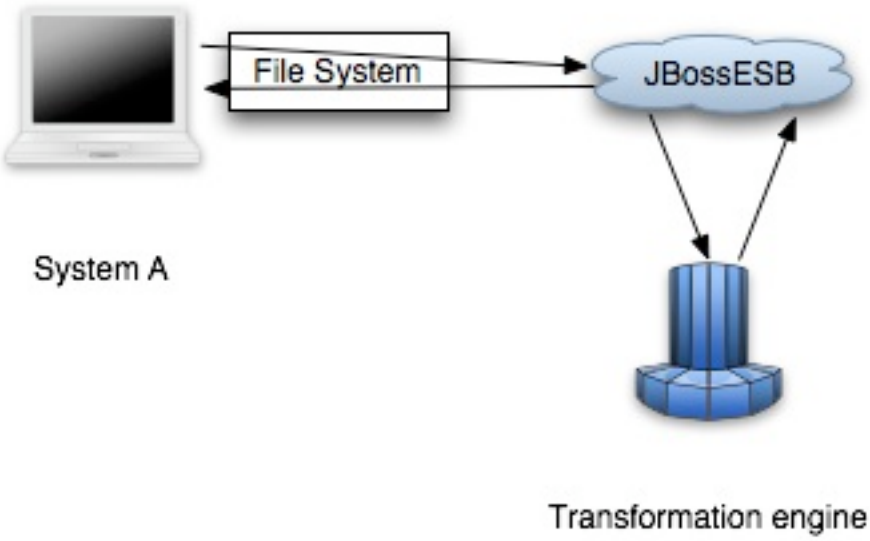
In the next series of examples, we use a queuing system (e.g., a JMS implementation).



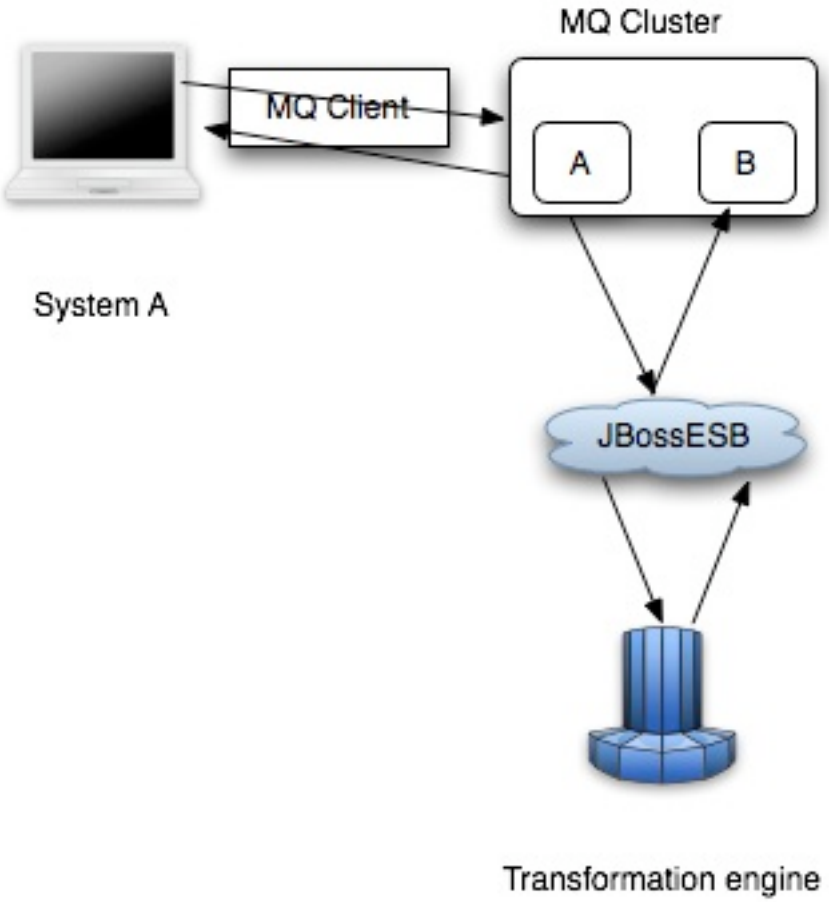
The diagram below shows transformation and queuing in the same situation.



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.



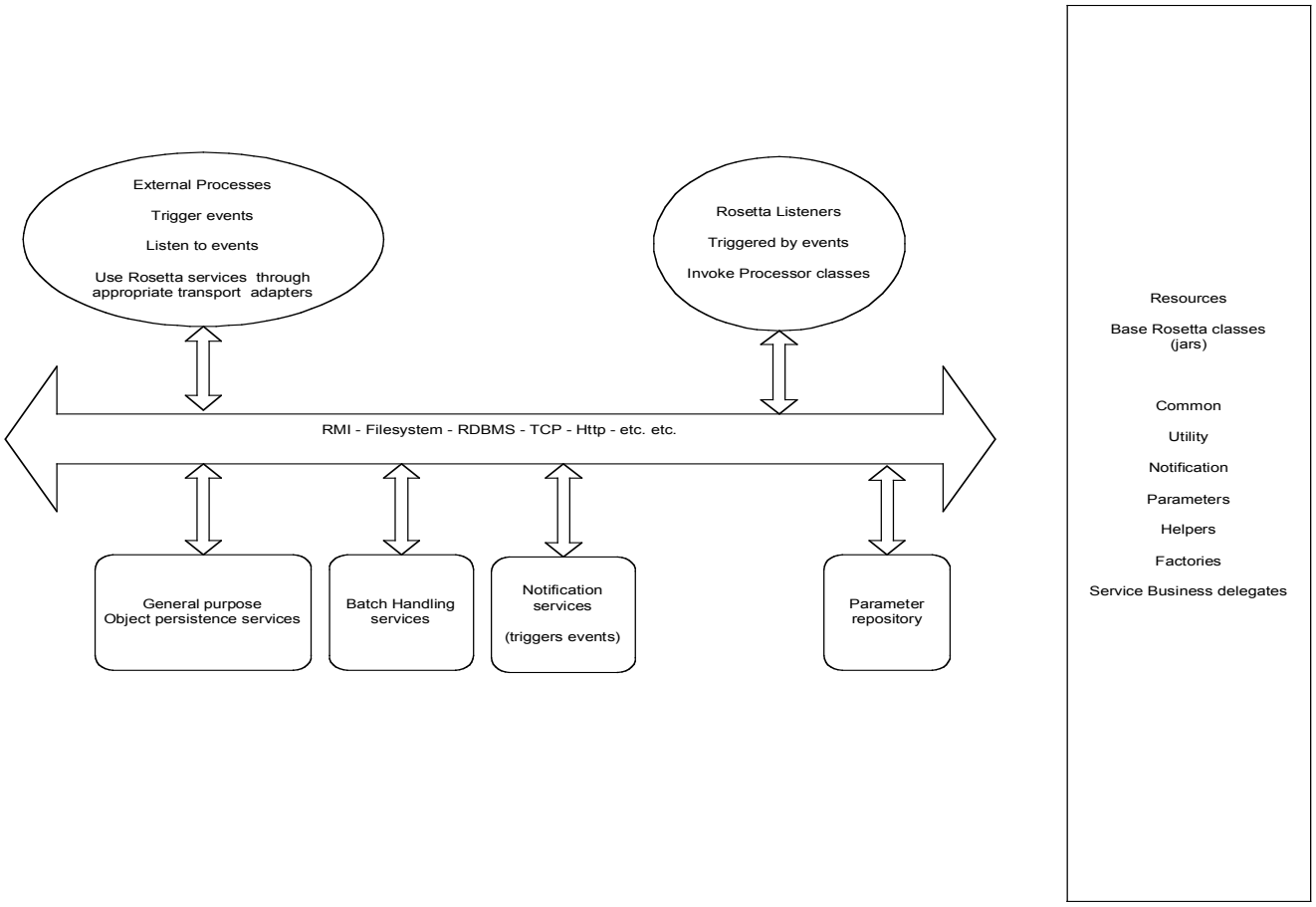
Chapter 4

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, Processors refers to the Action classes within the core which are responsible for processing on triggered events.



[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.

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 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:

- Event handling and process triggering/chaining using the *Action* and *Listener classes*, and the Notification framework.
- Reusable data transformation libraries in public *FormatAdapter* classes.
- A simple general purpose *BusinessObject* repository.

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 Rosetta 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 although can be supported in the beta release of JBossESB, they are not provided.

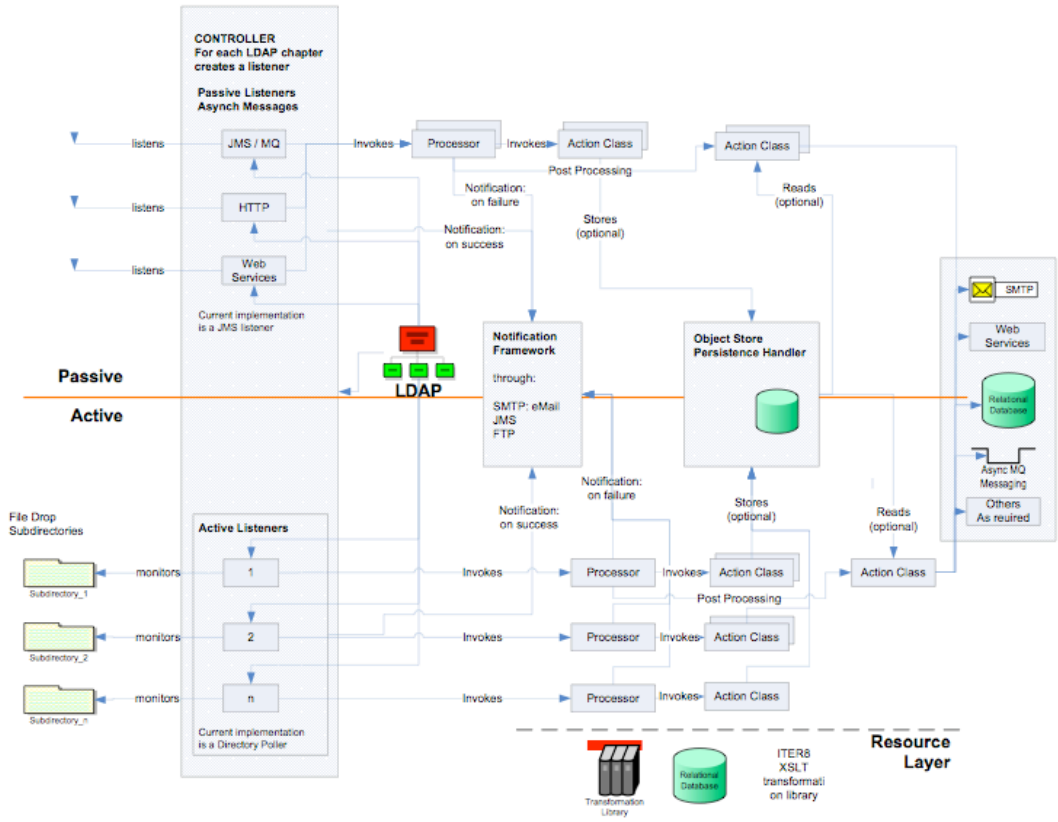


Figure 4: ESB Core components.

> **Note:** The Processor and Action class distinction shown in the above diagram no longer exists in the beta release.

JBossESB components

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

> **Note:** Some class and interface names may change between the beta release and the GA.

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;
}
```

With the beta version of 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: In the beta release the Object Store configuration is handled differently. See the section on the Object Store for more details.

The Object Store

JBossESB has an object store that can be used to support the following requirements:

- Provide persistency of data until all the currently known subscribers have successfully received the data.
- Provide recovery of data to a subscriber; data can be resent from JBossESB as opposed to having to be resent from the publisher.
- Provide historical data to a new or existing subscriber without having to go back to the publisher.

Data is not stored in a manner to provide data mining or reporting capabilities. Stored entities are instances of the `org.jboss.soa.esb.util.BaseBusinessObject` class. The framework expects that all classes that extend `BaseBusinessObject` are able to serialize into a `BobjStdDTO` (data transfer object) and to have a constructor that takes a `BobjStdDTO` as the sole argument.

The `org.jboss.soa.esb.util.BobjStdDTO` class is a specialized tree that has (among other methods) a `toXml()` method and a constructor that takes a valid XML

string as an argument. The persistence interface uses `BobjStdDTO` as the argument to store, retrieve or replace objects in the store and is completely agnostic of the business class that the argument represents. Therefore, the store is able to deal with unknown user objects, as long as they are able to serialize to, and construct from, a `BobjStdDTO`, and that a proper entry is included in the object store's runtime configuration file.

`org.jboss.soa.esb.common.bizclasses.BatchProcess` offers simple functionality to group a series of objects that are handed asynchronously to the `org.jboss.soa.esb.services.IbatchHandler` interface. You can initiate a new batch, add elements to the batch header, add batch, close and eventually commit the batch. The concept of batch strongly relies on the existence of the object store. The basic idea is to provide functionality to batch information that might reside in different sources and store all atomic elements in the object store until a trigger to commit the batch is received. When this happens, all the information resides within the framework and standard methods can be used to process it.

The ability to store `BaseBusinessObjects` and `BatchProcess` in persistent media, provides a simple mechanism for **decoupling successive steps** in a chain of individual asynchronous processing steps. The object store uses a simple yet effective scheme that allows for storage of the serialized object (in the standard XML format), plus index information (Object's UID, batch UID, timestamp, etc.) in a single SQL table (and as many SQL index tables as configured in the Object Store configuration class.

Note: The purpose of this repository is **NOT** to act as a high performance general purpose database. It is used to store *batches* until the batch is commited/rolled back, and/or to store `BusinessObjects` for future retrieval, together with the RDBMS index tables according to the `BusinessObject's locator(int)` methods

The object storage and retrieval functionality is usable only through the business delegate of the actual implementation (currently a J2EE stateless session bean) that can be obtained by the `Processor` classes using the `org.jboss.soa.esb.services.ipersistHandler` interface and `org.jboss.soa.esb.services.PersistHandlerFactory` class:

```
public interface IpersistHandler
{
  public long getUidChunk(int p_iHowMany) throws Exception;
  public ObjLocator[] getLocatorList(Class p_oCls,Properties
p_oProp) throws Exception;
  public long addObject(BaseBusinessObject p_oQ)      throws
Exception;
  public BaseBusinessObject getObject(Class p_oCls,long p_lUid)
throws Exception;
  public void rplObject(BaseBusinessObject p_o)       throws
Exception;
  public void rmvObject(Class p_oCls,long p_lUid) throws Exception;
  public void remove()                            throws Exception;
}
```

Instances of classes that implement this interface represent the contract of the Object Repository service.

```
public class PersistHandlerFactory
{
  public static IpersistHandler getPersistHandler(String p_sLocRem,
         String p_sJndiType, String p_sJndiServer) throws Exception;

  public static IpersistHandler getPersistHandler(Context p_oCtx)
         throws Exception;
}
```

Through the factory, implementation specific details of the contract are hidden.

As mentioned earlier, the object store is configured separately to other components within Rosetta. The object store is configured through an XML property file, an example of which is shown below:

```
<ObjectStore dataSourceJndiName="java:JbossEsbDS"
uidTable="uid_table"
  batchTable="batches" >
  <Class  name="org.jboss.soa.esb.common.bizclasses.Person"
   table="object_snap" type="Person" encrypt="false" >
    <Index table="people_index" />
  </Class>

  <Class  name="org.jboss.soa.esb.samples.Customer"
table="object_snap"
    type="Customer" encrypt="false" >
    <Index table="customer_index" />
  </Class>
</ObjectStore>
```

The location of the property file may be set through the `org.jboss.soa.esb.objStore.configfile` property.

Serialization of deployed objects

Every class stored in the repository is responsible for knowing how to serialize and deserialize itself from XML. The combination of `BaseBusinessObject` and `BobjStdDTO` classes are used to accomplish this.

```
public abstract class BaseBusinessObject
{
  public BobjStdDTO toDTO () throws Exception;
  public static final BaseBusinessObject getFromDTO (BobjStdDTO
param)
        throws Exception;
}

public class BobjStdDTO implements Serializable
{
   public String toXml() throws Exception;
   public static BobjStdDTO getFromXml(String p_sXml) throws
Exception;
}
```

All instances of `BaseBusinessObject` are thus converted to and from the Date Transfer Object format (`BobjStdDTO`). It is this instance that is then responsible for converting to and from an XML representation.

All classes derived from `BaseBusinessObject` that wish to be stored in the object store must provide a `locator()` method and may optionally provide a `locator(int)` method, as shown below.

```
public abstract class BaseBusinessObject
{
   public String[] locator();
   public String[] locator(int p_i);
}
```

`locator(0)` is assumed to be the same as `locator()`. These operations provide indexing information for the object store searches using standard SQL queries.

> **Note:** The default implementation of `locator` returns a zero-length String array.

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 *of Format Adapters*, whose sole responsibility is data transformation.

Note: In the beta release of JBossESB, Format Adapters are not a service or a component of the core: they are a pattern that we use within the core and recommend for users. This will be rectified in the GA release.

Format adapters should be the **only** place that needs to be aware of coupling between different applications' representation of the same underlying entity.

For example, let us assume we have an application that needs to translate between two different representations of a customer and the developer creates a `CustomerAdapter`:

```
public class CustomerAdapter
{
    public static Customer esbFromWeb (WebCustomer p_o);
    public static WebCustomer webFromEsb (Customer p_o);
}
```

Note: Error handling code has been removed for clarity.

The developer can then use this format adapter when working with the object store:

```
public void RequestLoan(WebCustomer customer)
{
   IpersistHandler esbHandler =
            PersistHandlerFactory.getPersistHandler("remote",
                  EsbSysProps.getJndiServerType(),
                  EsbSysProps.getJndiServerURL());
   long lUid =
esbHandler.addObject(CustomerAdapter.esbFromWeb(customer));
}
```

Listener classes

Listeners encapsulate the endpoints for message reception. Upon a receipt of a message, a Listener triggers an *Action* class to do work based on the content of the message. As illustrated in Figure 4 there are several ways in which triggering of Actions can occur in JBossESB:

- Queue/Topic listeners (in independent processes and/or as MDBs within a J2EE container), raw or protocol specific socket listeners: `org.jboss.soa.esb.listeners.JmsQueueListener`
- Directory pollers (in independent processes and/or as MBeans): `org.jboss.soa.esb.listeners.DirectoryPoller`
- RDBMS table pollers (`org.jboss.soa.esb.listeners.SqlTablePoller`), email listeners, etc.

As mentioned above, the main responsibility of a *Listener* is to trigger *Actions*, which are registered with them via the configuration information. For example:

```
<JBossESB-LoanBroker-TrailBlazer
    commandConnFactoryClass="ConnectionFactory" commandJndiType="jboss"
```

```
    commandJndiURL="localhost" commandIsTopic="false"
    messageSelector="esbApp='esbApp'" commandJndiName="queue/A"
   parameterReloadSecs="60">

  <CreditAgencyJMSInput
    listenerClass="org.jboss.soa.esb.listeners.JmsQueueListener"
actionClass="org.jboss.soa.esb.samples.loanbroker.actions.ProcessCreditReques
t"
    xyzactionClass="org.jboss.soa.esb.actions.DummyAction" maxThreads="1"
    queueConnFactoryClass="ConnectionFactory" listenJndiType="jboss"
    listenJndiURL="localhost" listenQueue="queue/A"
    listenMsgSelector="sample_loanbroker_servicecode='creditRequest'">

    <NotificationList type="OK">
      <target class="NotifyQueues" >
       <queue jndiName="queue/A">
        <messageProp name="sample_loanbroker_servicecode"
                    value="creditResponse" />
       </queue>
      </target>
   </NotificationList>
  </CreditAgencyJMSInput>

  <CreditAgencyJMSOutput
    listenerClass="org.jboss.soa.esb.listeners.JmsQueueListener"
xyzactionClass="org.jboss.soa.esb.samples.loanbroker.actions.ProcessCreditRes
ponse"
    actionClass="org.jboss.soa.esb.actions.DummyAction" maxThreads="1"
    queueConnFactoryClass="ConnectionFactory" listenJndiType="jboss"
    listenJndiURL="localhost" listenQueue="queue/A"
   listenMsgSelector="sample_loanbroker_servicecode='creditResponse'">

   <NotificationList type="OK">
     <target class="NotifyFiles">
     <file
URI="file:///C:/dev/jbossesb/product/docs/samples/trailblazer/bankloanbrokerd
emo/creditAgency.notifOK" append="true"/>
     </target>
    </NotificationList>
  </CreditAgencyJMSOutput>

</JBossESB-LoanBroker-TrailBlazer>
```

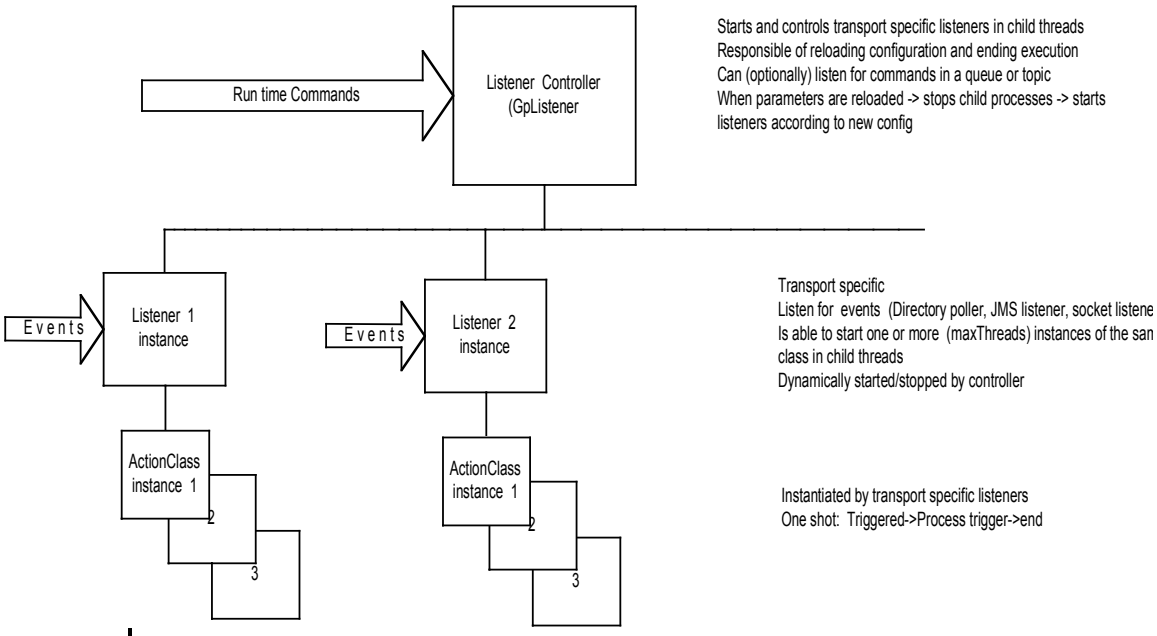
Here we are using a JMS listener and the `Action` is defined in the `actionClass` attribute, along with additional information needed by the processor instance. The configuration file is provided to the ESB when it is started:

```
java MyProgram Configuration.xml
```

Where the code for `MyProgram` in this case would be:

```
JmsQueueListener jms = new JmsQueueListener(args [0]);
```

The relationship between listeners and actions is shown below:



Note: In the beta release of JBossESB, messages are represented as Serializable objects. This gives great flexibility over the content that can be exchanged between endpoints. However, it does mean that endpoints must be able to deserialize received messages. This means that classes representing message objects must be deployed within the application server.

Action classes

`Action` classes are responsible for doing the work requested (or implied) by the receipt of a message. As such, `Actions` are often implemented on a per-application basis, using the provided base classes (all in the `org.jboss.soa.esb.actions` package) `AbstractFileAction`, `AbstractSqlRowAction` and `FileCopier`, for working on file processing, SQL processing and file copying respectively.

`Actions` that are also instances of `BaseBusinessObject` can perform transformations using the format adapters pattern, are triggered by Listeners and can provide operations to learn about the processing outcome.

Common to all `Actions` is the concept of sending a result object. This result may indicate a successful outcome or an error outcome and all `Action` classes inherit the capability from the `AbstractAction` base class:

```
public abstract class AbstractAction extends Observable
                                     implements Runnable
{
    public abstract void processCurrentObject() throws Exception;
    public abstract Serializable getOkNotification();
    public abstract Serializable getErrorNotification();

    protected DomElement m_oParms;
    protected Object  m_oCurr;
```

```
    protected Logger   m_oLogger = Logger.getLogger(this.getClass());
}
```

The `DomElement` contains configuration information that the `AbstractAction` may use at runtime. The `Object` is a reference to the current message. Both parameters are provided to the `AbstractAction` implementation at creation time.

The `processCurrentObject` method is called to instruct the Action implementation to do some work. How often the method is invoked and how the work is performed is implementation specific: it is not mandated by the ESB.

The `getOkNotification` and `getErrorNotification` methods are used by the underlying notification framework. They must be overridden by derived classes to return an application specific object to the recipient. They are frequently used to chain together sequences of interactions in a multi-step process.

For example, consider the scenario where a bank receives a message that requires it to send back a customer's credit rating. We will use the `MsgProcessor` class and as a result redefine its `processMessage` and `getOkNotification` methods to accomplish this:

```
public class ProcessCreditRequest extends AbstractAction
{
   public Integer creditScore;

   public void processCurrentObject() throws Exception
   {
     m_oLogger.info("processObject was called with
          <<"+m_oCurr.toString()+">>");

     if (! (m_oCurr instanceof ObjectMessage))
       throw new Exception("Message must be a ObjectMessage");

     System.out.println(m_oCurr);

     CreditCheckRequest creditRequest =
       (CreditCheckRequest)((ObjectMessage)m_oCurr).getObject();

     //use the notification framework to send back our response
     //use a dummy score between 0 and 10

     Random generator = new Random();

     creditScore = new Integer(generator.nextInt(10));

     //create our Response

     creditResponse =
      new CreditCheckResponse(creditRequest.requestID, creditScore);
    }

  //this is how the response is sent back, through OK notification
  //you could also send back a special ERROR credit score condition

  public Serializable getOkNotification()
  {
      return creditScore;
```

```
  }

  public Serializable getErrorNotification()
  {
      return "error occured in " + this.getClass();
  }
}
```

This implementation of `AbstractAction` is expected to be driven by a JMS listener and so receives an appropriate message format to deal with, which is passed in to the implementation during creation as one of the `AbstractAction` member variables (`m_oCurr`). In this simple example the body of the `processCurrentObject` method simply determines a random credit rating and uses this to initiate the `creditScore`. This score is subsequently returned when the notification infrastructure invokes the `getOkNotification` upon successful execution.

Notification infrastructure

JBossESB uses a notification infrastructure to enable triggering of processing events, which can be subsequently used by `Listeners` to continue/interrupt/branch process steps. At the heart of the notification framework is the `org.jboss.soa.esb.helpers.InotificationHandler` class:

```
public interface InotificationHandler
{
  public void sendNotifications(DomElement p_oP, Serializable p_o)
throws Exception;
  public void sendNotifications(Serializable p_o) throws Exception;
}
```

`sendNotification` is responsible for sending the event/message represented by the `Serializable` parameter to the list of registered notifications (`org.jboss.soa.esb.notification.NotificationTargets`) which are represented in XML form by the first parameters (a serialized `NotificationList`). If no list is specified, then the registered targets are assumed to have been provided statically at deployment time via an appropriate configuration file.

There are many different ways in which components within an ESB may wish to be notified (contacted) by receiving a message. For example, email, ftp, database etc. Therefore, `NotificationTarget` is a base class from which all supported notification implementations inherit:

```
public abstract class NotificationTarget
{

  /**
    * Derived classes must implement this method to do what has to
be done
    * to trigger that specific type of notification event.
    *
    * @param p_o Object - The toString() method of this object will
```

```
be the
     * actual notification content.
     * @throws Exception - invoke Exception.getMessage() at runtime
for this
     * object.
     */

  public abstract void sendNotification(java.io.Serializable p_o)
throws Exception;
}
```

Currently supported implementations include (all in the `org.jboss.soa.esb.notification` package): `NotifyEmail`, `NotifyFiles`, `NotifySqlTable` and `NotifyJMS` (with two sub-classes of `NotifyQueues` and `NotifyTopics`).

As mentioned above, the `NotificationList` is a list of targets (represented in XML) for events. As shown below, the XML representation of targets is passed to the `NotificationList` during construction:

```
public class NotificationList extends DomElement
{
  public NotificationList(DomElement p_oP) throws Exception;

  public void sendNotification(Serializable p_o) throws Exception;
}
```

The `sendNotification` method is used to send the specified message to all registered targets. Messages of arbitrary content are provided through the `Serializable` parameter. These objects are typically provided by `Actions` using either the `getOkNotification` or `getErrorNotification` methods we discussed earlier.

In the beta version of JBossESB the `InotificationHandler` interface is driven in the AS; the `NotificationHandlerFactory` returns a reference to an EJB. A consequence of this is that all URI attributes for the `NotifyFiles` implementation reference the file system as seen by the server (application server) that hosts the instance, as opposed to the client (the environment in which the listeners reside). Similarly, e-mails sent as a result of a `NotifyEmail` implementation will be sent by the server, so SMTP information used to send these mails is as configured for the application server (e.g., `.../conf/jbossEsb.properties`). Similarly for JMS messages (`NotifyQueues`, `NotifyTopics`) use queues/topics in the context of the application server that hosts the `NotificationHandlerBean` instances.

> **Note:** This is irrelevant when the client and server run in the same address space.

Configuration Table

The table below shows the various configuration options available for JBossESB Beta 1.

Property name	Description	Default value
org.jboss.soa.esb.mail.smtp.host	SMTP host name	localhost
org.jboss.soa.esb.mail.smtp.user	SMTP user name	“”
org.jboss.soa.esb.mail.smtp.password	SMTP password	“”
org.jboss.soa.esb.mail.smtp.port	SMTP port	25
org.jboss.soa.esb.jndi.server.type	JNDI server type	jboss
org.jboss.soa.esb.jndi.server.url	JNDI server URL	localhost
org.jboss.soa.esb.paramsRepository.class	Parameter repository implementation	NONE
org.jboss.soa.esb.objStore.configfile	Object store configuration file	NONE
org.jboss.soa.esb.encryption.factory.class	Encryption class	org.jboss.soa.esb.services.DefaultEncryptionFactory

Table 2: Configuration options.

Information concerning the various XML data structures needed to configure JBossESB are provided in separate documents.

Index

actionClass, **31**
`Actions`, **32**
  relationship to notification framework, **33**
  result objects, **32**
  supported implementations, **32**
Architectural components, **24**
Configuring JBossESB, **26**, **28**, **35**
ESB Overview, **14**
Format adapters, **29**
  example, **30**
JBossESB
  Access Control Lists, **15**
  content-based routing, **15**
  contract definition language, **17**
  implementation flexibility, **16**
  multi-bus support, **17**
Listeners, **30**
  actionClass, **31**
  relationship to Actions, **30**
  supported implementations, **30**
Notification framework, **34**
  supported implementations, **35**
Rosetta, 23
  history, **24**
Serialized objects and the object store, **29**
SOA Overview, **8**
  basics, **11**
  benefits, **10**
  Why SOA?, **10**
The Data Transfer Object, **29**