JBossESB 4.3 GA

SOA Background Concepts

JBESB-SBC-5/20/08


JBoss®
a division of Red Hat

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

Software Version

JBossESB 4.3 GA

Restricted Rights Legend

Contents

Table of Contents

About This Guide

What This Guide Contains

The SOA Background Concepts document contains descriptions on the principles behind Service Oriented Architecture and Enterprise Service Bus, as well as how they relate to JBossESB.

Audience

This guide is most relevant to engineers who are responsible for using JBossESB 4.3 GA 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.

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:	A note highlights important supplemental information.
Caution:	A caution highlights procedures or information that is necessary to avoid damage to equipment, damage to software, loss of data, or

Table 1 Formatting Conventions

Additional Documentation

In addition to this guide, the following guides are available in the JBossESB 4.3 GA documentation set:

1. **JBossESB 4.3 GA** *Trailblazer Guide*: Provides guidance for using the trailblazer example.
2. **JBossESB 4.3 GA** *Programmer's Guide*: Provides guidance for developing applications using JBossESB.
3. **JBossESB 4.3 GA** *Getting Started Guide*: Provides a quick start reference to configuring and using the ESB.
4. **JBossESB 4.3 GA** *Administration Guide*: How to manage JBossESB.
5. **JBossESB 4.3 GA** *Release Notes*: Information on the differences between this release and previous releases.
6. **JBossESB 4.3 GA** *Services Guides*: Various documents related to the services available with the ESB.

Contacting Us

Questions or comments about JBossESB 4.3 GA 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

[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.

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:

- 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.

Statefull and Stateless services

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 statefull 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.

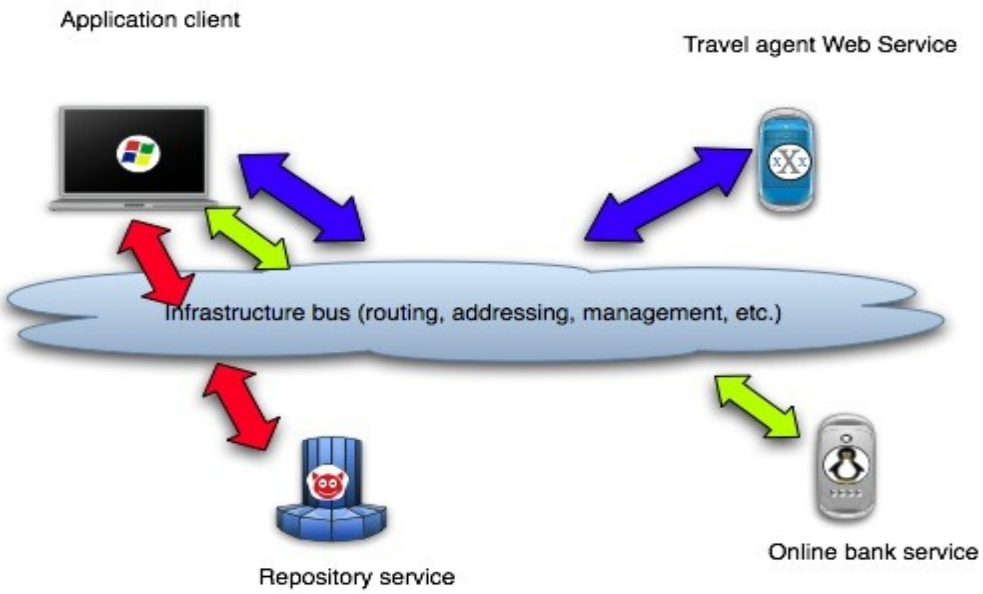
WS-Context provides a session model that is an evolution of the session models found in HTTP servers, transaction, and MOM systems. On the other hand, WS-Context allows a service client to more naturally bind the relationship to the service dynamically and temporarily. The client's communication channel 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 statefull 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.

JBossESB and its relationship with SOA

SOA is more than technology: it does not come in a shrink-wrapped box and requires changes to the way in which people work and interact as much as assistance from underlying infrastructures, such as JBossESB. With JBossESB 4.3 GA, Red Hat is providing a base SOA infrastructure upon which SOA applications can be developed. With the 4.2.1 release, most of the necessary hooks for SOA development are in place and Red Hat is working with its partners to ensure that their higher level platforms leverage these hooks appropriately. However, the baseline platform (JBossESB) will continue to evolve, with out-of-the-box improvements around tooling, runtime management, service life-cycle etc. In

JBossESB 4.3 GA, it may be necessary for developers to leverage these hooks themselves, using low-level API and patterns.

Chapter 2

The Enterprise Service Bus

Overview

The ESB is seen as the next generation of EAI – better and without the vendor-locking 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:

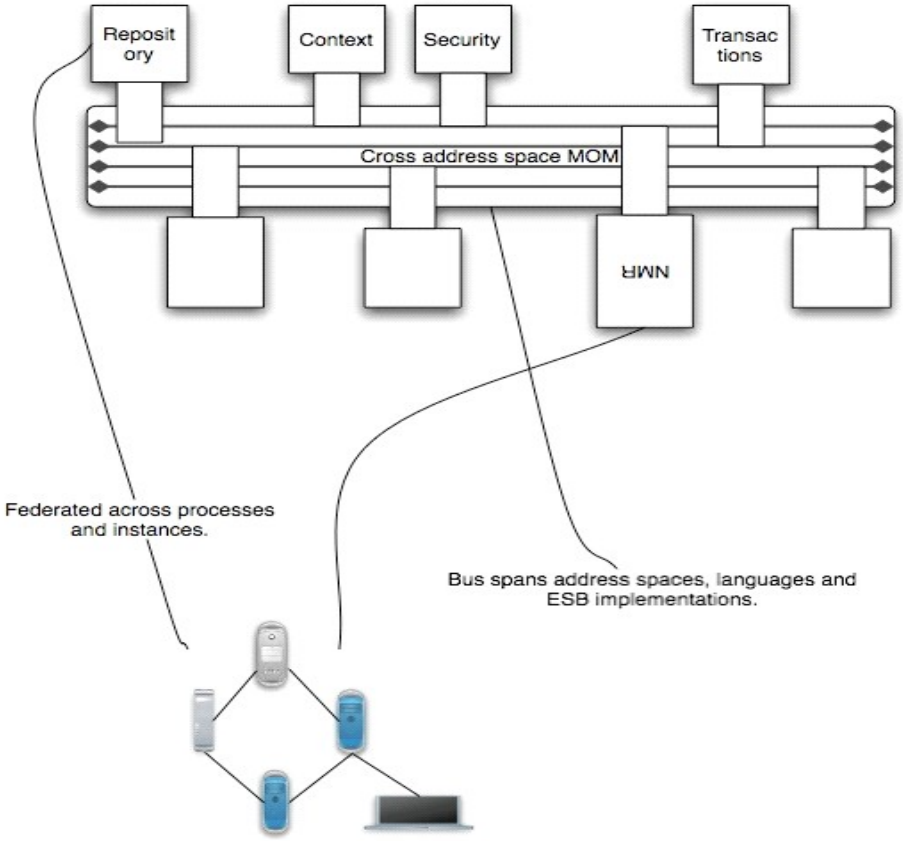
At its core, a good SOA should have a good *messaging infrastructure* (MI), and JMS is a fairly good example of a standards-compliant MI. But it obviously will not be the only implementation supported. Other capabilities that an ESB provides include:

- 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.

Creating services

If you ask 100 people what they mean by SOA applications you'll probably get 100 different answers. However, there are some common requirements:

- they should scale from several to hundreds and thousands of participants/services.
- they should be loosely coupled, so that changes of service implementation at either end of an interaction can occur in relative isolation without breaking the system.
- they need to be highly available.

[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.)

- they need to be able to cope with interactions that span the globe and have connectivity characteristics like the traditional Web (i.e., poor).

- asynchronous (request-request) invocations should be as natural as synchronous request-response.

Scalability and availability are possible with other technologies, such as CORBA. Although (ii) and (iv) can certainly be catered for in those technologies as well, the default paradigm is one based on an implementation choice: objects. Objects have well defined interfaces and although they can change, the languages used to implement them typically place restrictions on the type of changes that can occur. Now although it is true that certain OO architectures, such as CORBA, allow for a loosely coupled, weakly types interaction pattern (e.g., DII/DSI in the case of CORBA), that is not typically the way in which applications are constructed and hence tool support in this area is poor.

There is no objective way in which to approach the question of whether SOAs can be catered for in traditional environments. The answer is obviously yes, because no new language has been invented for SOAs and current tools are used to develop them. However, the real question is what is the best paradigm in which to consider an SOA that allows it to address all 5 points above.

Concentrating on the message and making it the central tenant of the architecture is the key to addressing the 5 points. How this is mapped onto a logical architecture (objects, procedures, etc.) and ultimately onto a physical implementation (objects, methods, state, etc.) is not important. The fact is that many different implementations and sub-architectures could be used. So what is the fundamental concept or mind-set in which to work when considering SOA?

The answer is that this is not about request-response, request-request, asynchrony etc. but it's about events. The fundamental SOA is a unitary event bus which is triggered by receipt of a message: a service registers with this bus to be informed when messages arrive. Next up the chain is a demultiplexing event handler (dispatcher), that allows for sub-services (sub-components) to register for sub-documents (sub-messages) that may be logically or physically embedded in the initially received message. This is an entirely recursive architecture.

Versioning of Services

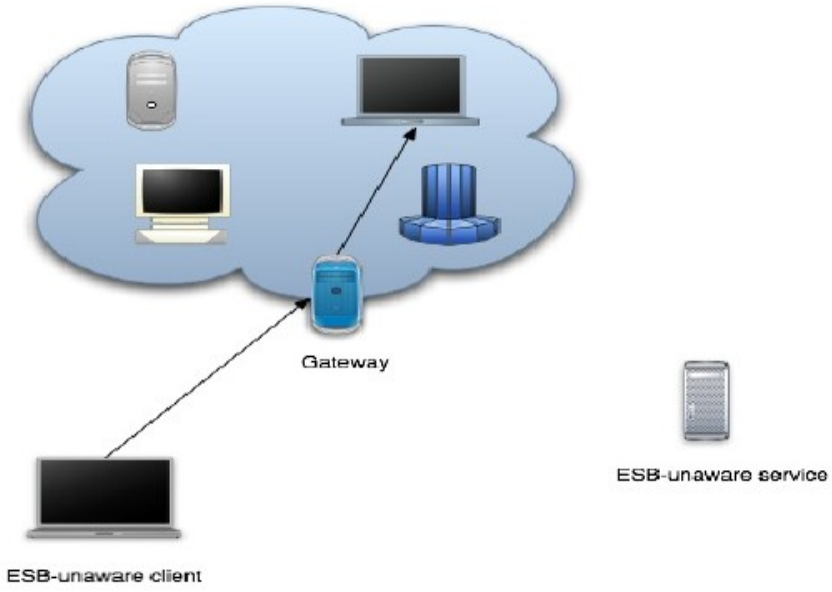
Using the ESB/SOA actually consists of two phases: the initial creation phase and the maintenance phase, which may have different requirements from the creation phase. Services evolve over time and it is often difficult or impossible to find a quiescent period in which to replace a service. As such, 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.

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

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.

Index