Fuse ESB Enterprise 7.0

Product Introduction

Introducing Fuse ESB Enterprise, its features, and its capabilities

Edition 1

Legal Notice

1801 Varsity Drive
Raleigh, NC 27606-2072 USA
Phone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701

Abstract

This manual introduces Fuse ESB, and describes its architecture, messaging, routing, and services. It also provides an example use case.

Table of Contents

Preface

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the Liberation Fonts set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

`Mono-spaced Bold`

Used to highlight system input, including shell commands, file names and paths. Also used to highlight keys and key combinations. For example:

> To see the contents of the file **`my_next_bestselling_novel`** in your current working directory, enter the **`cat my_next_bestselling_novel`** command at the shell prompt and press **`Enter`** to execute the command.

The above includes a file name, a shell command and a key, all presented in mono-spaced bold and all distinguishable thanks to context.

Key combinations can be distinguished from an individual key by the plus sign that connects each part of a key combination. For example:

> Press **`Enter`** to execute the command.
>
> Press **`Ctrl`**+**`Alt`**+**`F2`** to switch to a virtual terminal.

The first example highlights a particular key to press. The second example highlights a key combination: a set of three keys pressed simultaneously.

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in **`mono-spaced bold`**. For example:

> File-related classes include **`filesystem`** for file systems, **`file`** for files, and **`dir`** for directories. Each class has its own associated set of permissions.

Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialog box text; labeled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

> Choose **System** → **Preferences** → **Mouse** from the main menu bar to launch **Mouse Preferences**. In the **`Buttons`** tab, click the **`Left-handed mouse`** check box and click **`Close`** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).
>
> To insert a special character into a **gedit** file, choose **Applications** → **Accessories** →

Character Map from the main menu bar. Next, choose **Search** → **Find...** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit** → **Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in proportional bold and all distinguishable by context.

Mono-spaced Bold Italic or ***Proportional Bold Italic***

Whether mono-spaced bold or proportional bold, the addition of italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

To connect to a remote machine using ssh, type **ssh *username@domain.name*** at a shell prompt. If the remote machine is **example.com** and your username on that machine is john, type **ssh john@example.com**.

The **mount -o remount *file-system*** command remounts the named file system. For example, to remount the **/home** file system, the command is **mount -o remount /home**.

To see the version of a currently installed package, use the **rpm -q *package*** command. It will return a result as follows: ***package-version-release***.

Note the words in bold italics above — username, domain.name, file-system, package, version and release. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

Publican is a *DocBook* publishing system.

1.2. Pull-quote Conventions

Terminal output and source code listings are set off visually from the surrounding text.

Output sent to a terminal is set in **mono-spaced roman** and presented thus:

```
books        Desktop   documentation  drafts  mss    photos   stuff  svn
books_tests  Desktop1  downloads      images  notes  scripts  svgs
```

Source-code listings are also set in **mono-spaced roman** but add syntax highlighting as follows:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;

public class ExClient
{
   public static void main(String args[])
       throws Exception
   {
      InitialContext iniCtx = new InitialContext();
      Object         ref    = iniCtx.lookup("EchoBean");
      EchoHome       home   = (EchoHome) ref;
      Echo           echo   = home.create();

      System.out.println("Created Echo");

      System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
   }
}
```

1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.

Note

Notes are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.

Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled 'Important' will not cause data loss but may cause irritation and frustration.

Warning

Warnings should not be ignored. Ignoring warnings will most likely cause data loss.

Chapter 1. Introducing Fuse ESB Enterprise

1.1. Overview

IT system integration is one of the biggest challenges facing modern enterprises. Fuse ESB Enterprise tackles this problem using a lightweight standards-based, loosely-coupled approach. By relying on standards, Fuse ESB Enterprise reduces the chances of vendor lock-in. By advocating loose coupling, Fuse ESB Enterprise reduces the complexity of integration.

1.2. Integration problems

Enterprise networks commonly deploy disparate applications, platforms, and business processes that need to communicate or exchange data with each other. These entities typically use incompatible data formats and incompatible communications protocols. For an enterprise that needs to interface with external systems, the problem extends outside the company to encompass the IT systems and processes of its business partners.

In recent years, several technologies, such as Enterprise Application Integration (EAI) and Business-to-Business (B2B), have attempted to solve these problems. These solutions addressed some of the problems of integration, but were proprietary, expensive, and time-consuming to implement. They ranged from expensive vendor solutions to home-grown custom solutions. The overwhelming disadvantages of these solutions were high cost and low flexibility due to nonstandard implementations.

Most recently, Service Oriented Architecture (SOA) has become the hot integration methodology. SOA attempts to address the shortcomings of the previous approaches by advocating the use of standards and loosely-coupled interfaces. While SOA theoretically improves on those solutions, it can be difficult to implement because many vendors offer tools that use proprietary technologies and attempt to wrap old solutions in SOA clothing.

1.3. The ESB approach

An *enterprise service bus* (ESB) is the back bone of an SOA implementation. Though no canonical definition of an ESB exists, David Chappell states in his book Enterprise Service Bus:

> An ESB is a standards-based integration platform that combines messaging, web services, data transformation, and intelligent routing to reliably connect and coordinate the interaction of significant numbers of diverse applications across extended enterprises with transactional integrity.
>
> --David A. Chappell

The term extended enterprise describes an organization and its business partners, who are separated by both business and physical boundaries.

An ESB is typically defined by the list of services it provides. Services commonly included are:

- Transport mediation—not all applications and services that need to be integrated use HTTP or JMS.
- Dynamic message transformation—not all services use SOAP and are unlikely to require the same message structures.
- Intelligent routing—not all messages emanating from a source are intended for the same destination. The target destination will likely depend on some criteria inherent in the message.
- Security—only authorized and authenticated users need have administrative access to the Fuse ESB

Enterprise runtime; services and brokers that handle sensitive information may restrict access to unauthorized or unauthenticated clients only; similarly, messages that contain sensitive information may be encrypted as they transit their routes.

An ESB simplifies the complexity of integration by providing a single, standards-based infrastructure into which applications can be plugged. Once plugged into the ESB, an application or service has access to all of the infrastructure services provided by the ESB and can access any other applications that are also plugged into the ESB. For example, you could plug a billing system based on JMS into an ESB and use the ESBs transport mediation features to expose the billing system over the Web using SOAP/HTTP. You could also route internal purchase orders directly into the billing system by plugging the Purchase Order system into the ESB.

1.4. Differentiating between ESB implementations

Most ESB implementations provide all of the services that are used to define an ESB, so it is hard to differentiate ESB implementations based on features. A better way to differentiate between them is to use the following four measures:

- Supported deployment/runtime environments

  Many ESB solutions are designed to be deployed into application servers, other heavy weight containers, or proprietary runtime environments. Though these solutions support distributed computing, they also contribute to vendor lock-in.

  Ideally, an ESB solution should have flexible deployment requirements, so it can be distributed throughout an enterprise.
- Container/component model

  Does the ESB solution use a standardized container model, such as JEE or OSGi for managing deployed services? Or does it use a proprietary model?

  Ideally, an ESB solution should use a standards-based container model. Standard models ensure maximum compatibility and decrease the learning curve needed for adoption.
- Coupling to other infrastructure components

  ESB solutions often omit infrastructure components, such as orchestration engines and advanced transports like CORBA. Instead they rely on plug-ins or other components to provide that functionality.

  Many ESB solutions require a tight coupling between the ESB and added components. This means that you are limited to using only added components supplied by the ESB vendor or that you must learn complex APIs to extend the ESB yourself.

  Ideally, an ESB solution should provide a loose coupling between the ESB and added components or provide a standardized interface between them. This approach allows the ESB to be extended easily and in a flexible way.
- Dependencies

  ESB solutions have a lot of moving parts and complex dependencies. Some ESB solutions handle these dependencies by locking themselves into using proprietary solutions for things like security or JMS. Others rely on standards-based implementations as much as possible.

  Ideally, an ESB solution should depend only on widely available standardized libraries to make dependencies easy to manage.

1.5. The Fuse ESB Enterprise approach

Based on Apache ServiceMix, Fuse ESB Enterprise reduces complexity and eliminates vendor lock-in because it is standards-based and built using best-of-breed, open source technology. It differentiates

itself in these ways:

- Fuse ESB Enterprise is lightweight and can run on most platforms.
- Fuse ESB Enterprise uses the OSGi framework to simplify componentization of applications.
- Fuse ESB Enterprise provides a mechanism, Fuse Application Bundle (FAB), that automates creating and maintaining OSGi bundles.

  FABs free application developers from the complexities associated with creating and maintaining OSGi bundles, enabling them to focus on building their applications. This is especially useful when you are developing large, complex enterprise applications. See Section 1.7, “FABs in a nutshell”.
- Fuse ESB Enterprise supports the Java Business Integration (JBI) specification (JSR 208).
- Fuse ESB Enterprise can be coupled to other infrastructure services over a wide variety of transport protocols and data formats.

  Out of the box, the Fuse ESB Enterprise supports JMS, HTTP, HTTPS, FTP, XMPP, Web services, and a number of other bindings. In addition, you can easily extend its connectivity options using routing and integration components.
- Fuse ESB Enterprise employs standards as much as possible to limit dependencies.
- Fuse ESB Enterprise supports event-driven architectures. Services deployed into the Fuse ESB Enterprise container can be fully decoupled and will simply listen on the bus until an appropriate service request arrives. Fuse ESB Enterprise also supports external events that occur outside the bus. For example, a JMS service can listen on a topic that is hosted outside the bus and act only when an appropriate message arrives.

1.6. OSGi in a nutshell

OSGi is set of open specifications that make it easier to build and deploy complex software applications. The OSGi Framework, the key piece of OSGi technology, is responsible for loading and managing the dynamic modules of functionality, otherwise known as bundles (see *Fuse ESB Enterprise 7.0 Deploying into the Container*: "Building an OSGi Bundle").

In an OSGi environment, applications are packaged into bundles—jar files that contain extra information about the classes and resources included in them. The information supplied in a bundle includes:

- Packages required by classes in the bundle
- Packages the bundle exports
- Version information for the bundle

Using this information, the OSGi framework checks that all of the dependencies required by the bundle are present. If they are, the bundle is activated and made available. The information included in a bundle also enables the OSGi framework to manage multiple versions of it.

Among the advantages OSGI offers over other container and packaging models are:

- Hot deployment of artifacts
- Management of multiple versions of a package, class, or bundle
- Dynamic loading of code (hot deployment and dependency injection via Spring or Blueprint)
- Lightweight footprint

1.7. FABs in a nutshell

A FAB is a jar file created using Apache Maven or similar build tool. It contains a `pom.xml` file located in `META-INF/maven/<groupID>/<artifactID>/`. The `pom.xml` file declares all of the jar's transitive

dependencies. (See Using Apache Maven.)

When a FAB is deployed in Fuse ESB Enterprise, the runtime converts it into a normal OSGi bundle, then extracts and uses the jar's `pom.xml` file to identify its transitive dependencies. This process enables the runtime to generate one or more OSGi bundles according to how the dependencies are listed in the `pom.xml` file. Unless already installed, Fuse ESB Enterprise automatically installs all identified transitive dependencies.

A FAB can depend on regular jars or OSGi bundles, and it can install, start, stop, and uninstall its transitive dependencies.

For details, see *Fuse ESB Enterprise 7.0 Deploying into the Container*: "Building a FAB" and *Fuse ESB Enterprise 7.0 Deploying into the Container*:"Deploying a FAB".

1.8. Using Apache Maven

Apache Maven is a software project management and build tool. It is required for running Fuse supplied tutorials, and it is recommended for developing applications based on Fuse products. When you download Apache Maven, download v3.0.3.

Chapter 2. The Fuse ESB Enterprise Architecture

2.1. Architectural Layers

2.1.1. Overview

As shown in Figure 2.1, “Fuse ESB Enterprise Architecture”, Fuse ESB Enterprise employs a layered architecture that consists of:

- Kernel layer

  A lightweight runtime that provides management features for the runtime and extends OSGi with powerful features for handling and managing OSGi bundles.

  For details, see Section 2.1.2, “The Kernel Layer”.
- Services layer

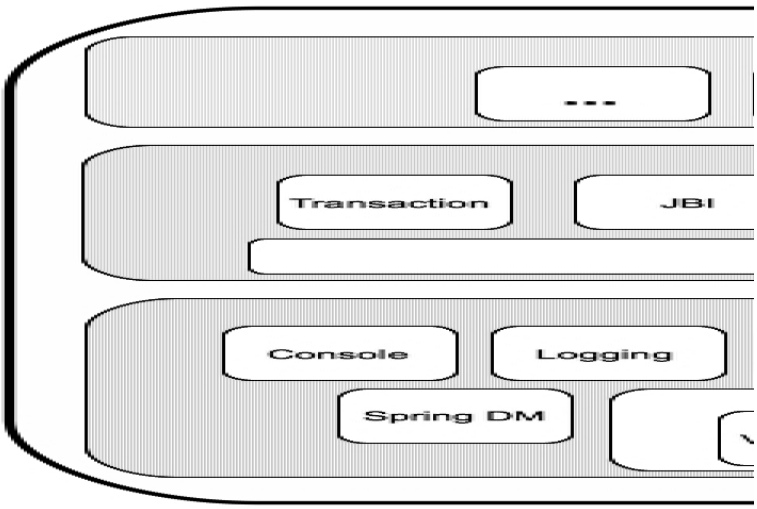
  A layer of embedded services that sits on top of the kernel layer. Each service consists of the interfaces and service implementations required to create an instance of its specific service.

  For details, see Section 2.1.3, “The Services Layer”
- Application layer

  The layer that hosts user-developed applications and sits on top of the services layer. Fuse ESB Enterprise provides numerous APIs that make it easy for user-developed applications to interact with services embedded in the Services layer.

  For details, see Section 2.1.4, “The Application Layer”

Figure 2.1. Fuse ESB Enterprise Architecture



2.1.2. The Kernel Layer

The Fuse ESB Enterprise kernel layer is based on Apache Karaf, an OSGi-based runtime that provides a lightweight container into which you can deploy various components and applications.

The kernel layer interacts with the Services layer to set up, coordinate, and manage logging, and security; and to manage transactions on a per service basis. It also interacts with the Service layer to set up message broker instances with the configuration specified in the supplied **`activemq.xml`** file.

The kernel provides these features:

- Console

  The Fuse ESB Enterprise console is a shell environment that enables you to configure all Fuse ESB Enterprise components and to control the Fuse ESB Enterprise runtime, including brokers and messages, Fuse ESB Enterprise kernel instances, logging, and so on.

  For example, using the console subshells, you can manage artifacts deployed in the Fuse ESB Enterprise runtime, including OSGi bundles, collections of bundles grouped into features, JBI artifacts, and OSGi bundle repositories (OBRs). Using the **`ssh`** and **`sshd`** commands, you can connect to and start a remote secure shell server to manage multiple, distributed instances of the

Fuse ESB Enterprise runtime.

See "Using the Command Console" in *Fuse ESB Enterprise 7.0 Console Reference*/> for details on using the console and for descriptions of the available commands.

- Logging

  A dynamic logging back end supports different APIs (JDK 1.4, JCL, SLF4J, Avalon, Tomcat, OSGi). By default, Fuse ESB Enterprise enters all log messages in a single log file, but you can change this behavior by configuring different log files to store log messages generated by specific Fuse ESB Enterprise components.

- Deployment

  You can manually deploy applications using the **`osgi:install`** and **`osgi:start`** commands, or you can automatically deploy them by copying them to the hot deploy folder. When a JAR file, WAR file, OSGi bundle, or FAB file is copied to the ***`InstallDir`*`/deploy`** folder, it's automatically installed on-the-fly inside the Fuse ESB Enterprise runtime.

  For details, see *Fuse ESB Enterprise 7.0 Deploying into the Container*: "Deploying an OSGi Bundle-Hot".

- Fuse Application Bundle (FAB)

  FABs automate the creation and maintenance of OSGi bundles, freeing application developers to focus on building their applications.

  See Section 1.7, “FABs in a nutshell”. See also, *Fuse ESB Enterprise 7.0 Deploying into the Container*: "Building a FAB"/> and *Fuse ESB Enterprise 7.0 Deploying into the Container*: "Deploying a FAB".

- Provisioning

  Applications are provisioned through hot deployment, the Maven repository, and remote downloads.

  For details, see *Fuse ESB Enterprise 7.0 Deploying into the Container*: "Deploying an OSGi Bundle".

- Configuration

  The properties files contained in the ***`InstallDir`*`/etc`** directory are continuously monitored, and changes are automatically propagated to the relevant services at configurable intervals.

  For details, see *Fuse ESB Enterprise 7.0 Managing and Monitoring a Broker*: "Managing and Monitoring a Broker".

- Security

  The security framework is based on Java Authentication and Authorization Service (JAAS). You can secure separately, Fuse ESB Enterprise's OSGi container, deployed instances of the embedded messaging service, and deployed instances of the embedded routing and integration service.

  For details, see *Fuse ESB Enterprise 7.0 Security Guide*: "Security Architecture".

- OSGi container

  For a brief description of OSGi, see Section 1.6, “OSGi in a nutshell”. You can deploy a variety of packages and files into Fuse ESB Enterprise 's OSGi container: FABs, OSGi bundles, Jars, Wars, Blueprint, Spring, JBI, and so on.

  Fuse ESB Enterprise supports two OSGi 4.2 containers, Eclipse Equinox and Apache Felix Framework.

  The OSGi specifications are maintained by the OSGi Alliance. See OSGi.org.

  For more information about Fuse ESB Enterprise's OSGi functionality see *Fuse ESB Enterprise 7.0 Deploying into the Container*: "Introduction to OSGi".

- Dependency injection frameworks

  The supported dependency-injection frameworks facilitate deploying new OSGi applications and updating existing ones dynamically:

  - Blueprint

    An OSGi specification, Blueprint is optimized for the OSGi container. It enables you to define

Camel routes, JMS endpoints, Web service endpoints and OSGi services using XML syntax. OSGi is similar to the Spring framework, but more lightweight.

- Spring

  Adding any Spring configuration file to the **`deploy`** directory causes Fuse ESB Enterprise to generate an OSGi bundle on the fly and to instantiate the Spring application context.

2.1.3. The Services Layer

The Fuse ESB Enterprise services layer consists of all of the interfaces and implementation classes for each of the embedded services. It interacts with the application layer to communicate with user-developed applications that want to access and use these services.

- Messaging

  Using Fuse ESB Enterprise's messaging service, based on Apache ActiveMQ, you can create JMS message brokers and clients, then deploy them as OSGi bundles.

  Fuse ESB Enterprise comes with a default message broker that autostarts when you start up Fuse ESB Enterprise, but you can replace it with your own message broker implementation. For details, see *Fuse ESB Enterprise 7.0 Deploying into the Container*: "JMS Broker".

  For more information on the messaging service, see Section 3.1, “The Embedded Messaging Service” and *Fuse ESB Enterprise 7.0 Deploying into the Container*: "JMS Broker".
- Routing and integration

  Using Fuse ESB Enterprise's routing and integration service, based on Apache Camel, you can define routes and implement enterprise integration patterns, then deploy them as OSGi bundles.

  For more information, see Section 4.1, “The Embedded Routing and Integration Service” and *Fuse ESB Enterprise 7.0 Deploying into the Container*: "Generating and Running an EIP Bundle".
- Web services

  Using Fuse ESB Enterprise's web services framework, based on Apache CXF, you can create JAX-WS web services, then deploy them as OSGi bundles.

  For more information, see Section 5.1, “The Embedded Web and RESTful Services” and *Fuse ESB Enterprise 7.0 Deploying into the Container*: "Generating and Running a Web Services Bundle".
- RESTful services

  Using Fuse ESB Enterprise's RESTful services framework, based on Apache CXF, you can create JAX-RS web services, then deploy them as OSGi bundles.

  For more information, see Section 5.1, “The Embedded Web and RESTful Services” and *Fuse ESB Enterprise 7.0 Deploying into the Container*: "Generating and Running a Web Services Bundle"/>.
- JBI

  Using Fuse ESB Enterprise's JBI service you can create and deploy Java Business Integration (JBI) 1.0 service assemblies and service units in Fuse ESB Enterprise. See *Fuse ESB Enterprise 7.0 Using Java Business Integration*: "Introduction to JBI" for an introduction to Fuse ESB Enterprise's JBI environment.

  Tip

  Using OSGi and Fuse ESB Enterprise's embedded routing and integration service is strongly recommended. OSGi provides many deployment advantages over JBI (see *Fuse ESB Enterprise 7.0 Deploying into the Container*: "Introduction to OSGi").
- Transaction Manager

  Fuse ESB Enterprise's transaction framework employs a JTA transaction manager, based on Apache Aries, to expose various transaction interfaces as OSGi services. The transaction manager enables

you to create and deploy JTA-based or Spring-based transacted applications in Fuse ESB Enterprise. Both the embedded messaging service and the embedded routing and integration service provide easy means for interacting with the transaction manager to implement JMS transactions.

See Section 3.2.4, “JMS transactions” and Section 3.2.5, “XA transactions”.

- Normalized Message Router (NMR)

  The Fuse ESB Enterprise NMR is a general-purpose message bus whose primary role is to transmit messages between the various application bundles deployed into the OSGi container. In this case, no normalization is required because OSGi places no restrictions on the format of message content.

  However, when the JBI container is also deployed, the NMR is also used to transmit messages between OSGi and JBI applications. Normalization must be performed on messages transmitted to a JBI endpoint, because JBI requires that message content be formatted in XML, as defined in a WSDL service description.

  Fuse ESB Enterprise provides a simple Java API for creating NMR endpoints that receive and process messages from the NMR and for writing clients that send messages to NMR endpoints. For more information, see *Fuse ESB Enterprise 7.0 Using Java Business Integration*: "Introduction to JBI"/>.

  In addition, Fuse ESB Enterprise's routing and integration service provides the `nmr:` component to define NMR endpoints for the OSGi container and the `jbi:` component to define NMR endpoints for the JBI container. For details, see *Fuse ESB Enterprise 7.0 Deploying into the Container*: "Nmr".

2.1.4. The Application Layer

The Fuse ESB Enterprise application layer is where user-developed applications reside. Fuse ESB Enterprise provides many APIs with which you can create client and service applications that access and use the embedded services running within Fuse ESB Enterprise. For details, see:

- Messaging—Section 3.5.1.3, “Client-side APIs”
- Routing and integration—Overview
- Web and RESTful web services—Section 5.1.1, “Font end options”.

2.2. How Fuse ESB Enterprise Activates a Deployed Bundle

2.2.1. Overview

Fuse ESB Enterprise's OSGi framework is the runtime that implements and provides OSGi functionality. It provides the execution environment for OSGi-based applications deployed into it.

The OSGi framework consists of three layers:

- The module layer is responsible for managing bundle packaging, resolving dependencies, and class loading.
- Internal to an application, the lifecycle layer defines how bundles access their execution environment, which provides them the means to interact with the OSGi framework and the facilities the framework provides during runtime.
- The service layer, employing the publish, find, and bind service model, provides communication between bundles and the components they contain. It does so by providing an OSGi service registry where service providers register their services and where service requesters find and bind (get a reference) to them.

2.2.2. Bundle bootstrapping process

Fuse ESB Enterprise's kernel acts much like a mini operating system to create and manage a runtime

environment for a deployed application. It does so by setting up an execution environment that meets the requirements specified in the `MANIFEST.MF` file included in each bundle in an application. It then creates the resources each bundle needs by reading the bundle's Blueprint or Spring XML file. If the bundle is a service provider and contains the necessary publishing information, the kernel registers the service in the OSGi service registry, so any service requester bundle that needs to use the registered service can find and access it.

The kernel interacts with the Services layer to create the resources specified within an OSGi bundle. For example, a service provider bundle that instantiates a message broker might define two connections that use different transports, one HTTP and one JMS. The kernel interacts with the messaging service to set up the message broker with the configuration specified in the bundle's `activemq.xml` file and creates the required transports.

Figure 2.2 shows the steps involved in activating an example route bundle. A developer would create **(1)** and deploy **(2)** a route bundle into Fuse ESB Enterprise's OSGi container. The route bundle would contain business logic (in this case a `RouteBuilder` class written in Java code), an OSGi `MANIFEST.MF` file defining the bundle's requirements (dependencies, imports, exports, and so on), and a `Blueprint.xml` file defining the route's resources.

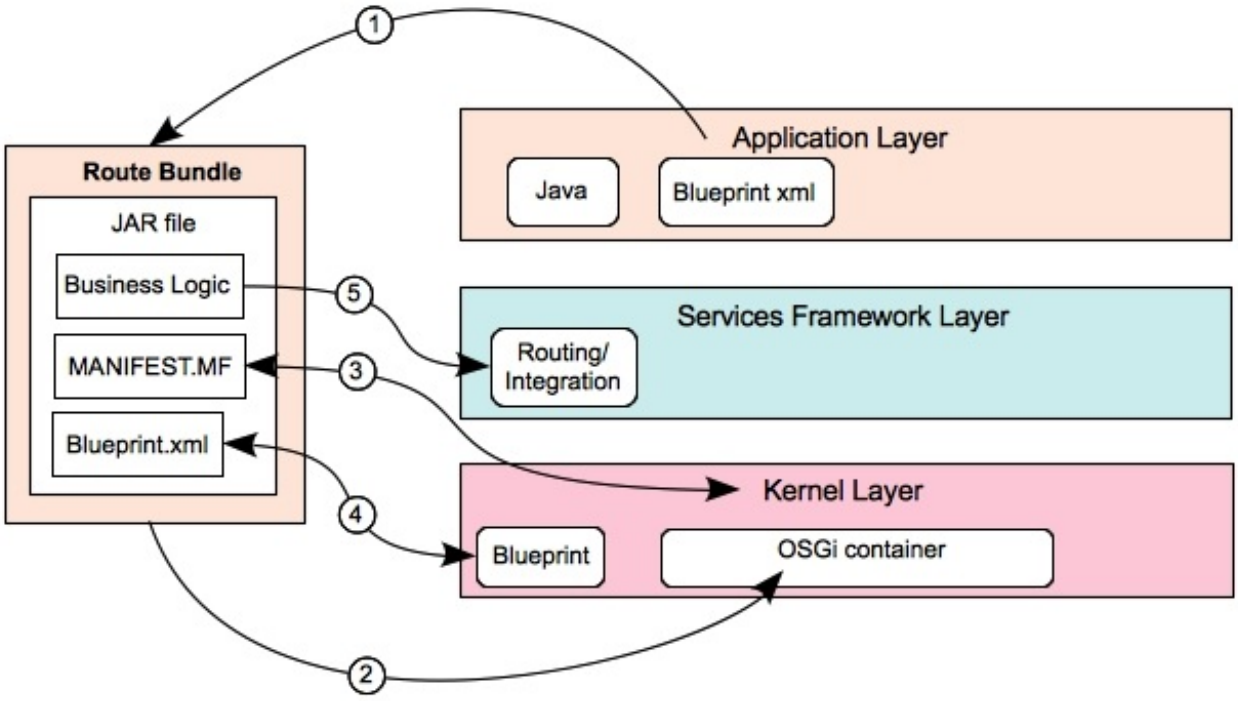


Figure 2.2. Steps in activating a deployed route

First, the kernel would read the bundle's `MANIFEST.MF` file **(3)** to set up the application's execution environment and resolve bundle dependencies. Then it would read the bundle's Blueprint `.xml` configuration file **(4)** to discover and set up the resources needed for the route. The business logic's `RouteBuilder` class would access the routing and integration service **(5)** via the OSGi service registry to define the route's CamelContext (the routing runtime).

In both cases, if the bundles were **(2)** hot deployed, the configured message broker and the route would start up immediately after their resources were in place. Otherwise, the developer or the system administrator would have to issue, at the console command line, `osgi:install` and `osgi:start` commands for each bundle.

Chapter 3. Messaging

3.1. The Embedded Messaging Service

3.1.1. Overview

Based on Apache ActiveMQ, Fuse ESB Enterprise's messaging service provides a JMS 1.1-compliant messaging system, consisting of a messaging broker and client-side libraries that enable remote communication among distributed client applications. The service's messaging broker handles:

- Message exchanges between messaging clients by managing the transport connections used for communicating with messaging clients and the transports used for communicating with other message brokers.
- Connections to database and file system stores for persistent messages.
- Monitoring and management of various components of the messaging system, and so on.

3.1.2. Message users

In a messaging system, client applications send or receive messages—producers create and send messages, while consumers receive and process them. JMS clients use the JMS API to interact with the messaging service, and non-JMS clients use any of Fuse ESB Enterprise's other client APIs to interact with it.

3.1.3. Messages

Messages are the means by which client applications transmit business data and events. Messages can contain either textual or binary content and metadata, which provides additional information about the message. Applications can use message metadata programmatically to modify or fine tune message delivery or administratively to monitor the health of the messaging system.

3.1.4. Fuse ESB Enterprise's messaging features

Besides providing the features required by the JMS 1.1 specification, Fuse ESB Enterprise's messaging service provides additional features and enhancements that support the special needs of large, complex enterprise messaging applications.

3.2. Standard JMS Features

JMS is a standardized means for transmitting messages between distributed applications. It defines a specific set of required features, a specific message anatomy, and a set of interactions between client applications and the message broker.

3.2.1. Queue- and topic-based messaging

Fuse ESB Enterprise' messaging service supports the two messaging domains defined in the JMS 1.1 specification.

- Point-to-Point (PTP)

  PTP messaging uses queue destinations. Producers are referred to as senders, and consumers are referred to as receivers. Messages are sent and received either synchronously (producer waits until the broker acknowledges receipt of messages) or asynchronously. The queue stores all messages until they either expire or are retrieved by a receiver.

Each message dispatched to a queue is delivered only once to only one receiver. The receiver is responsible for acknowledging receipt of a message. Messages are delivered to the next available, registered consumer, so distribution flows across consumers in a quasi round-robin fashion.

See Section 7.3, “Major Widgets Phase One Solution” for a point-to-point example use case.

- Publish/Subscribe (Pub/Sub)

  Pub/Sub messaging uses topic destinations. Producers are referred to as publishers, and consumers are referred to as subscribers. Topics typically have multiple subscribers concurrently registered to receive messages from it. Messages are sent and received either synchronously (producer waits until the broker acknowledges receipt of messages) or asynchronously.

  Any message dispatched to a topic is automatically delivered to all of the topic's subscribers.

  Topics do not store messages for subscribers, unless explicitly told to do so by the subscriber registering as a durable subscriber (see Section 3.5.3.4, “Durable subscribers”). When a disconnected durable subscriber reconnects to a topic, the message broker delivers all unexpired messages due it.

For more information, see the Java™ Message Service Maintenance Release.

3.2.2. Request/reply messaging

Fuse ESB Enterprise's messaging service supports the request/reply messaging paradigm, which provides a mechanism for a consumer to inform the producer whether it has successfully retrieved and processed a message dispatched to a queue or a topic destination.

The request/reply mechanism implements a two-way conversation in which a temporary destination is typically used for the reply message. The producer specifies the temporary destination, and the consumer identifies the request message to which the reply corresponds.

3.2.3. Persistent and nonpersistent messages

Fuse ESB Enterprise's messaging service supports persistent and nonpersistent messages.

Persistent messaging ensures no message is lost due to a system failure, a broker failure, or an inactive consumer. This is so because the broker always stores persistent messages in its stable message store before dispatching them to their intended destination. However, this guarantee comes at a cost to performance because persistent messages are sent synchronously—producer blocks until broker confirms receipt and storage of message— and writes to disk, typical for a message store, are slow compared to network transmissions.

Nonpersistent messages are significantly faster than persistent messages, but they can be lost when the system or broker fails or when a consumer becomes inactive before the broker dispatches the messages to their destination.

For details, see *Fuse ESB Enterprise 7.0 Configuring Message Broker Persistence*: "Configuring Message Broker Persistence"/> .

3.2.4. JMS transactions

Fuse ESB Enterprise's messaging service supports JMS transactions that occur between a client and broker.

A transaction consists of a number of messages grouped into a single unit. If any one of the messages in the transaction fails, the producer can rollback the entire transaction, so that the broker flushes all of the transacted messages. If all messages in the transaction succeed, the producer can commit the entire transaction, so that the broker dispatches all of the transacted messages.

Transactions improve the efficiency of the broker because they enable it to process messages in batch. A batch consists of all messages a producer sends to the broker before calling **`commit()`**. The broker caches all of these messages in a TransactionStore until it receives the **`commit`** command, then dispatches all of them, in batch, to their destination.

3.2.5. XA transactions

Fuse ESB Enterprise's messaging service supports XA transactions that occur between a client and broker.

XA transactions work similarly to JMS transactions, except XA transactions use a two-phase commit scheme and require an XA Transaction Manager and persistent messaging. This is so because the broker is required to write every message in an XA transaction to a persistent message store, rather than caching them locally, until the producer calls **`commit()`**.

XA transactions are recommended when you are using more than one resource, such as reading a message and writing to a database.

For details, see *Fuse ESB Enterprise 7.0 EIP Transactions Guide*: "TxnIntro". See also What are XA transactions? What is a XA datasource?.

3.3. JMS Message Basics

3.3.1. Overview

Messages are the backbone of a messaging system. A messages is a self-contained autonomous entity that may be resent several times across many clients (and processes) throughout its life span. Each consumer along a message's route will examine it and, depending on its contents (both payload and headers), may execute some business logic, modify it, or create new messages to accomplish a communicated task.

3.3.2. Message anatomy

As shown in Figure 3.1, “Anatomy of a JMS message”, a JMS message consists of three components: headers, properties, and body.

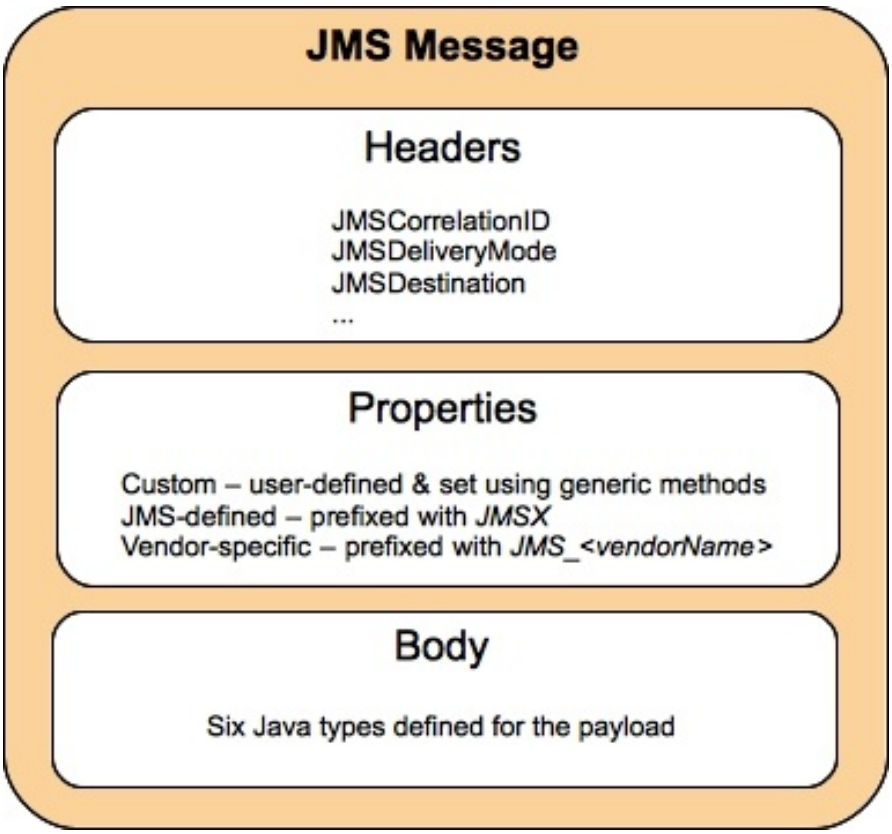


Figure 3.1. Anatomy of a JMS message

3.3.3. Message body

The body component contains the payload, which can be textual or binary data, as specified by using one of the six supported Java message types: **`Message`**, **`TextMessage`**, **`MapMessage`**, **`BytesMessage`**, **`StreamMessage`**, and **`ObjectMessage`**.

3.3.4. Message headers and properties

Header and property components transmit metadata used by clients and by the broker. All of the complexity of JMS messages resides in these two components, which are logically the same, but differ in their semantics.

- Headers

  The JMS API provides a standard list of JMS headers (identified by the **`JMS`** prefix) and methods to work with them. The producer's **`send()`** method automatically assigns a number of these headers to messages, while others are optionally set either by the producer or by the broker.
- Properties

  Client applications can create used-defined properties based on Java primitive types. The JMS API also provides generic methods for working with these user-defined properties.

  Two other types of properties, JMS-defined properties (identified by **`JMSX`** prefix) are reserved by the JMS specification, and the vendor-specific properties (identified by the **`JMSActiveMQBroker`** prefix) are reserved by the broker.

For a list of supported headers and properties, see ActiveMQ in Action (Snyder, Bosanac, and Davies)

3.3.5. Message selectors

Message selectors enable clients to use message headers and properties as filters, fine tuning where

particular messages are sent. They enable a consumer to specify which messages it wants to receive from a destination based on particular header or property values.

3.4. JMS Development

Developing applications using the JMS APIs is a straightforward process. The APIs facilitate the creation of objects that mitigate the interface between client applications and the message broker. Once connected to a broker, clients can create, send, and receive messages.

3.4.1. Basic application components

A JMS broker provides connectivity, message storage, and message delivery functions to its clients. It also provides quality of service features, such as reliability, persistence, security, and availability. A simple broker application typically consist of these basic interfaces and classes:

- Connection factory[1]

  Clients use a connection factory to create connections to a broker instance. You can optimize some messaging characteristics by enabling, disabling, or otherwise modifying the values of certain connection factory properties.
- Connection

  Connections are the technique clients use to specify a transport protocol and credentials for sustained interaction with a broker.
- Session

  Defined by a client on a connection established with a broker, a session defines whether its messages will be transacted and the acknowledgement mode when they are not. Clients can create multiple sessions on a single connection.
- Destinations

  Defined by a client in a session, a destination is either a queue or a topic. The broker maintains destinations.
- Producer

  Producers are client applications that create and send messages to a broker. They use the broker's `MessageProducer` interface to send messages to a destination. The default destination for a producer is set when the producer is created.

  The `MessageProducer` interface provides methods not only for sending messages, but also for setting various message headers (see Section 3.3.4, “Message headers and properties”), which enable you to control many message aspects, such as persistence, delivery priority, life span, and so on.
- Consumer

  Consumers are client applications that process messages retrieved from a broker. They use the broker's `MessageConsumer` interface to receive messages from a destination.

  The `MessageConsumer` interface can consume messages synchronously or asynchronously. Receiving messages asynchronously frees the consumer from having to repeatedly poll the destination for messages.
- Messages

  Messages are the backbone of a messaging system. They are objects that contain not only the data payload, but also the required header and optional properties needed to transfer them from one client application to another. (See, Section 3.3, “JMS Message Basics”.)

3.4.2. Simple broker program

Figure 3.2, “Simple broker program” shows the typical sequence of events involved in sending or receiving messages in a simple broker client application.

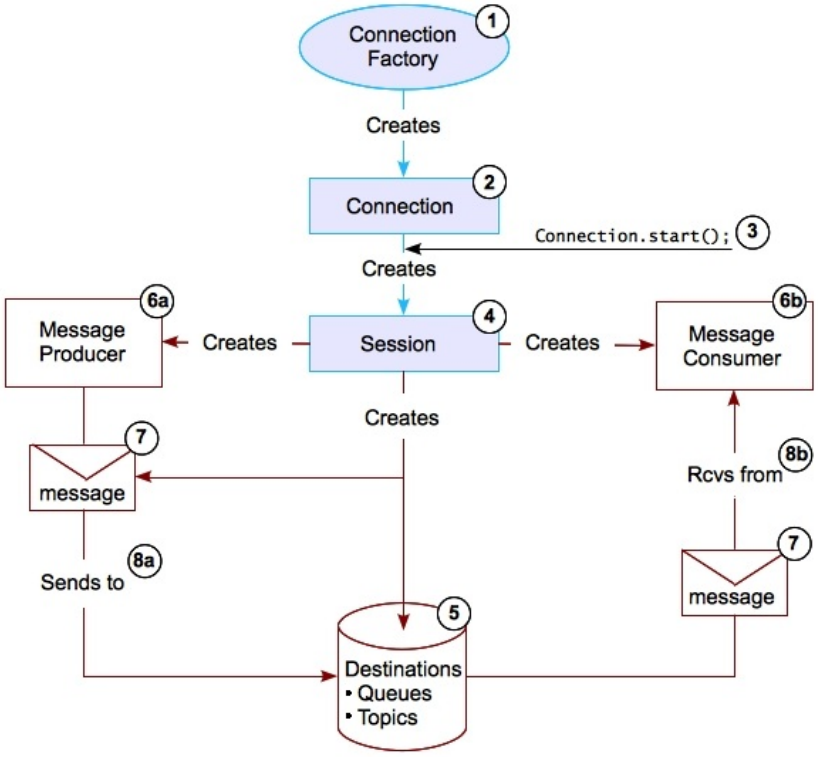


Figure 3.2. Simple broker program

This procedure shows how to implement the sequence of events shown in Figure 3.2, “Simple broker program”:

1. Get a connection factory by looking one up by name in the JNDI.
   ```
   ConnectionFactory=(ConnectionFactory)ctx.lookup("ConnectionFactoryName");
   ```
2. Create a connection.
   ```
   connection=connectionFactory.createConnection();
   ```
3. Start the connection.
   ```
   connection.start();
   ```
4. Create the session.
   ```
   session=connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
   ```
5. Create the destination (in this case a queue called foo.).
   ```
   destination=session.createQueue("FOO.QUEUE");
   ```
6. Do Step 6.a to create a producer, or do Step 6.b to create a consumer.
   a. `producer=session.createProducer(destination);`
   b. `consumer=session.createConsumer(destination);`
7. Create a text message.
   ```
   message=session.createTextMessage("This is a foo test.");
   ```
8. Do Step 8.a to have the producer send a message, or do Step 8.b to have the consumer receive 1000 iterations of the producer's message and print out an enumerated listing.
   a. `producer.send(message);`
   b. `message = (TextMessage) consumer.receive(1000);`

```
System.out.println ("Received message: " + message);
```

9. Close all JMS resources.

3.5. Extended JMS Features

This section provides an overview of just some of the extended JMS features that enable developers to build messaging applications that are flexible, highly available, reliable, scalable, highly performant, and easily administered. (See Fuse ESB Enterprise documentation for detailed descriptions and examples of these and other extended features provided by Fuse ESB Enterprise's embedded messaging service.)

Note

Because these features are not part of the JMS 1.1 specification, to port an application that uses them to another JMS provider, you'd most likely need to rewrite the sections of code that use them.

3.5.1. Flexibility

3.5.1.1. Overview

A flexible messaging system provides a variety of ways for building and deploying messaging applications and for clients to connect and communicate with one another and with message brokers. Fuse ESB Enterprise builds upon this basic flexibility with additional features.

3.5.1.2. Connectivity options

Fuse ESB Enterprise's messaging service provides a variety network protocols that enable producers and consumers to communicate with a broker:[2]

- Extensive Messaging and Presence Protocol (XMPP)—Enables instant messaging clients to communicate with the broker.
- Hypertext Transfer Protocol/Hypertext Transfer Protocol over SSL (HTTP/S)—Favored protocol for dealing with a firewall inserted between the broker and its clients.
- IP multicast—Provides one-to-many communications over an IP network. It enables brokers to discover other brokers in setting up a network of brokers, and clients to discover and establish connections with brokers.
- New I/O API (NIO)—Provides better scalability than TCP for client connections to the broker.
- Secure Sockets Layer (SSL)—Provides secure communications between clients and the broker.
- Streaming Text Oriented Messaging Protocol (STOMP)—A platform-neutral protocol, it supports clients written in scripting languages (Perl, PHP, Python, and Ruby) in addition to clients written in Java, .NET, C, and C++.
- Transmission Control Protocol (TCP)—For most use cases, this is the default network protocol.
- User Datagram Protocol (UDP)—Also deals with a firewall inserted between the broker and its clients. However, UDP guarantees neither packet delivery nor packet order.
- Virtual Machine (VM)—Provides connection between an embedded broker and its clients.

For details, see the *Fuse ESB Enterprise 7.0 Broker Client Connectivity Guide*: "Broker Client Connectivity Guide".

3.5.1.3. Client-side APIs

Fuse ESB Enterprise's messaging service provides client-side APIs for C, C++, and .NET, through OpenWire's binary encoding format. It provides client-side APIs for Perl, PHP, Python, and Ruby, through STOMP's simple text-oriented encoding format.

3.5.1.4. Deployment options

You can deploy a broker as a standalone that hosts clients running in other processes and other locations, or as an embedded broker that can run client and broker functions concurrently in one process, while still enabling connections to clients running in other processes at other locations. You can configure standalone and embedded brokers to work together to provide a more resilient network of brokers (see Section 3.5.4.2, “Network of brokers”).

3.5.1.5. Message routing

For complex messaging scenarios, you often need to implement message routing and mediation logic. Fuse ESB Enterprise's messaging service allows you to deploy routes using any of the Enterprise Integration patterns (EIPs), supplied by Apache Camel, directly into a broker instance. Deploying routes in the broker enables you to optimize performance by using the VM transport for routing messages between JMS destinations and to limit the number of artifacts you need to deploy and manage.

For details, see ActiveMQ in Action (Snyder, Bosanac, and Davies) and Camel in Action (Ibsen and Anstey).

3.5.1.6. Composite destinations

Provide a mechanism for producers to send the same message to multiple destinations at the same time.

A composite destination treats a string of multiple, comma-separated destinations as one destination, but sends messages to each of the individual destinations. You can create composite destinations that are homogeneous—made up of either all topics or all queues—or that are heterogeneous—made up of a combination of topics and queues.

For details, see ActiveMQ in Action (Snyder, Bosanac, and Davies).

3.5.1.7. Virtual destinations

Provide a mechanism for publishers to broadcast messages via a topic to a pool of receivers subscribing through queues.

To do so, consumers register a subscription to a queue that is backed by a virtual topic, like this: **`Consumer.<qname>.VirtualTopic.<tname>`**.

When the producer publishes messages to the virtual topic, the broker pushes the messages out to each consumer's queue, from which the consumers retrieve them. This feature enables you to failover queue subscribers, to load balance messages across competing queue subscribers, and to use message groups on queue subscribers.

For details, see ActiveMQ in Action (Snyder, Bosanac, and Davies).

3.5.2. High Availability

3.5.2.1. Overview

When disaster strikes, a highly available messaging system can continue to deliver messages successfully.

If planned for, power outages and network, hardware, or software failures need not obstruct message delivery. Employing multiple brokers, each configured to persist messages across multiple machines, and configuring consumers to failover to a functioning broker provide a robust solution.

3.5.2.2. Fault tolerance and the failover protocol

To build fault tolerance into a messaging system, you could run multiple standalone brokers, each on separate machines connected together over a network. When one machine or broker failed, clients, using the failover protocol, could automatically reconnect to a broker on another of the networked machines and continue working.

To implement fault tolerance, you need to set up master/slave topologies, which enable master brokers to replicate messages to their slave brokers (see Section 3.5.2.3, “Master/Slave topologies”), and to configure clients to use the failover protocol (see "Failover Protocol" in the *Fuse ESB Enterprise 7.0 Fault Tolerant Messaging* />).

Tip

Stand-alone brokers in this scenario know nothing about the consumers on any of the other brokers. Consequently, if one of these brokers had no consumers, messages sent to it would pile up without being processed. The Network of Brokers feature solves this problem (see Section 3.5.4.2, “Network of brokers”).

3.5.2.3. Master/Slave topologies

A master/slave topology defines an initial master broker and one or more initial slave brokers. When the master broker fails, one of the slave brokers starts up and takes over, becoming the new master broker. Clients using the failover protocol can reconnect to the new master broker and resume processing as normal.

Slave brokers have access to all messages sent to the master broker. Whether all messages are replicated from the master broker to the slaves depends on whether or not the topology includes a shared message store.

- Shared nothing master/slave networks—Both master and slave have their own independent message store. The master replicates all message commands (messages, acknowledgements, subscriptions, transactions, and so on) to the slave via a special connector,`masterConnectorURI`, before acting on them.

  Shared-nothing clusters are very resilient because they have no single point of failure. However, they also have a number of drawbacks:

  - Persistent messaging suffers additional latency because producers must wait for messages to be replicated to the slave and stored in the slave's persistent store.
  - Brokers do not autosynchronize with each other.
  - Only one slave per broker is allowed.
  - Reintroducing a failed master into the cluster involves shutting down the entire cluster and manually synchronizing the databases.
- Shared database or file system master/slave networks—Multiple brokers share the same relational database or file system, but only one broker can be active at any given time. The broker that gets the lock to the database or file system becomes the master. Brokers polling for the lock or attempting to connect to the message store after the master is established automatically become slaves.

  Resynchronization is not an issue, and failback occurs automatically (see Section 7.3, “Major

Widgets Phase One Solution" and Section 7.3.3, "Built-in Broker Fault Tolerance" for an example use case).

For details, see *Fuse ESB Enterprise 7.0 Fault Tolerant Messaging*.

3.5.3. Reliability

3.5.3.1. Overview

A reliable messaging system guarantees that messages are delivered to their intended recipients in the correct order. JMS guaranteed messaging relies on three mechanisms: message autonomy, store and forward delivery for persistent messages, and message acknowledgments.

Fuse ESB Enterprise's messaging service provides additional features for ensuring reliable messaging, including recovery of failed messages.

3.5.3.2. Message redelivery

Provides a mechanism for modifying the maximum number of times and the frequency at which the broker attempts to redeliver expired and undeliverable messages.

You can tune the client's redelivery policy by editing its connection in the broker's configuration file (`activemq.xml`).

For details, see ActiveMQ in Action (Snyder, Bosanac, and Davies).

3.5.3.3. Dead letter queue

When a message cannot be delivered to its destination, the broker sends it to the dead letter queue, ActiveMQ.DLQ. When a message is sent to the dead letter queue, an advisory message is sent to the topic `ActiveMQ.Advisory.MessageDLQd.*`. (See Section 3.5.5.1, "Advisory messages".

Messages held in the dead letter queue can be consumed or reviewed by an administrator at a later time. Reviewing the contents of the dead letter queue helps to debug delivery issues and to prevent messages loss.

For details, see ActiveMQ in Action (Snyder, Bosanac, and Davies).

3.5.3.4. Durable subscribers

This feature preserves the state of a registered durable subscription. A subscriber registers a durable subscription with a topic. A durable subscription instructs the broker to store all messages arriving at the topic while the durable subscriber is disconnected from it. The broker delivers all accumulated messages that have not expired to the durable subscriber when it reconnects to the topic. If the durable subscriber unsubscribes from the topic without first reconnecting, the broker discards all of the subscriber's accumulated messages.

For details, see ActiveMQ in Action (Snyder, Bosanac, and Davies).

3.5.3.5. Exclusive consumers

Provide a mechanism for ensuring that messages are processed in the correct order from a queue connected to multiple receivers.

The broker always dispatches messages in FIFO order. Because the client application has no control over the scheduling of messages, queues with multiple receivers may not consume messages in the same order they were dispatched, even when all consumers use the same connection. The exclusive

consumers feature directs the broker to select one of a queue's multiple consumers to receive all messages from the queue. If that consumer stops or fails, the broker selects another of the queue's consumers to take its place.

For details, see ActiveMQ in Action (Snyder, Bosanac, and Davies).

3.5.3.6. Message groups

Provide a mechanism to group messages for delivery to a single consumer.

This feature enables multiple consumers on the same queue to process, in FIFO order, messages tagged with the same **`JMSXGroupID`**. It also facilitates concurrency as multiple consumers can parallel process different message groups, each identified by a unique **`JMSXGroupID`**.

The producer assigns a message to a group by setting the **`JMSXGroupID`** property in the message header. The broker dispatches all messages tagged with the same **`JMSXGroupID`** value to a single consumer. If that consumer becomes unavailable, the broker dispatches subsequent messages in the group to another consumer.

For details, see ActiveMQ in Action (Snyder, Bosanac, and Davies).

3.5.3.7. Retroactive consumers

Applies only to topics. This feature provides a caching mechanism that improves reliability without the overhead of persistent messaging. It enables consumers to retrieve nonpersistent messages that were sent before the consumer started or that went undelivered because the consumer had to restart.

The broker can cache a configurable number of nonpersistent messages for topic destinations.

For details, see ActiveMQ in Action (Snyder, Bosanac, and Davies).

3.5.4. Scalability and High Performance

3.5.4.1. Overview

In a scalable messaging system, brokers can support an increasing number of concurrently connected clients. In a high-performance messaging system, brokers can process messages through the system, from producers to consumers, at a high rate. In a scalable high-performance messaging system, multiple brokers can concurrently process a large volume of messages for a large number of concurrently connected clients.

You can scale up your broker application to provide connectivity for thousands of concurrent client connections and many more destinations.

- Horizontal scaling

  Create networks of brokers to vastly increase the number of brokers and potentially the number of producers and consumers.

  For details, see *Fuse ESB Enterprise 7.0 Tuning the Message Broker*: "PersTuning-Horizontal".
- Vertical scaling

  Enable individual brokers to handle more connections—for example (when possible), use embedded brokers and VM transports, transactions, nonpersistent messages with the failover transport set to cache asynchronous messages; allocate additional memory; tune the OpenWire protocol; optimize the TCP transport.

  For details, see *Fuse ESB Enterprise 7.0 Tuning the Message Broker*: "PersTuning-Vertical".

Besides the obvious components—network and system hardware, transport protocols, message compression—which you can tune to increase application performance, Fuse ESB Enterprise's messaging service provides means for avoiding bottlenecks caused by large messages and for scheduling message dispatches.

3.5.4.2. Network of brokers

As shown in Figure 3.3, “Network of brokers example”, the brokers in a network of brokers are connected together by network connectors, which define the broker-to-broker links that form the basis of the network. Through network connectors, a network of brokers keeps track of all active consumers[3], receiving notifications whenever a consumer connects to or disconnects from the network. Using the information provided through client registration, brokers can determine where and how to route messages to any consumer in a network of brokers.

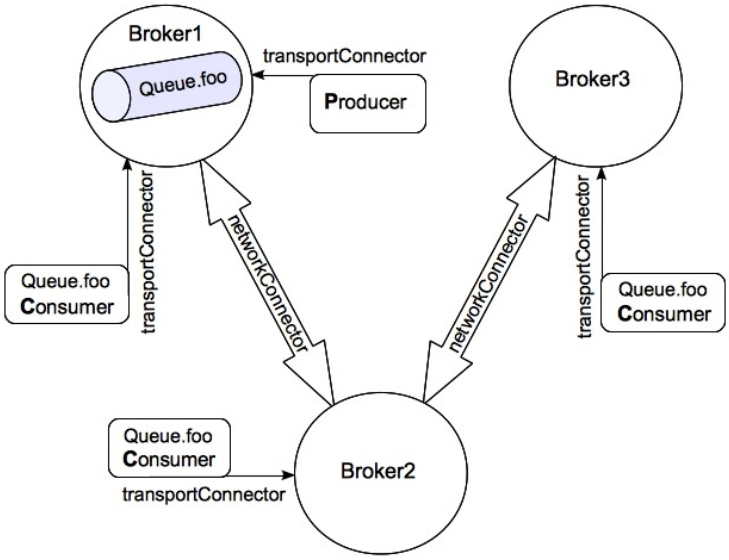


Figure 3.3. Network of brokers example

The brokers use a store-and-forward delivery method to move messages between them. A broker first stores messages locally before passing them on to another broker in its network. This scheme supports the distribution of queues and topics across a network of brokers.

For details, see "Using Networks of Brokers" in the *Fuse ESB Enterprise 7.0 Using Networks of Brokers*.

Tip

You can incorporate multiple master/slave topologies in networks of brokers to ensure a fault tolerant messaging system.

For details, see "Fault Tolerant Messaging Network" in the *Fuse ESB Enterprise 7.0 Fault Tolerant Messaging*.

3.5.4.3. Consumer clusters

Fuse ESB Enterprise's messaging service also supports reliable, high-performance load balancing on queues distributed across multiple consumers. If one of the consumers fails, the broker redelivers any unacknowledged messages to another queue consumer. When one consumer is faster than the others, it receives more messages, and when any consumer slows down, other consumers take up the slack.

This enables a queue to reliably load balance processing across multiple consumer processes.

3.5.4.4. Blob (binary large objects) messages

Blob messages provide a mechanism for robust transfers of very large files, avoiding the bottlenecks often associated with them. Blobs rely on an external server for data storage. Retrieval of a blob message is atomic. Blob messages are transferred out-of-bounds (outside the broker application) via FTP or HTTP. The blob message does not contain the file. It is only a notification that a blob is available for retrieval. The blob message contains the producer-supplied URL of the blob's location and a helper method for acquiring an **`InputStream`** to the actual data.

For details, see ActiveMQ in Action (Snyder, Bosanac, and Davies).

3.5.4.5. Stream messages

Stream messages also provide a mechanism for efficiently transferring very large files, avoiding the bottlenecks often associated with them. A stream message induces a client to function as a Java **`IOStream`**. The broker chunks **`OutputStream`** data received from the producer and dispatches each chunk as a JMS message. On the consumer, a corresponding **`InputStream`** must reassemble the data chunks.

For details, see ActiveMQ in Action (Snyder, Bosanac, and Davies).

3.5.4.6. Scheduled message delivery

Provides a mechanism for scheduling message dispatch for a later time or at regular intervals.

Using properties in the **`org.apache.activemq.ScheduledMessage`** interface, you can schedule when messages are dispatched to a consumer. The broker stores scheduled messages persistently, so they survive broker failure and are dispatched upon broker startup.

For details, see ActiveMQ in Action (Snyder, Bosanac, and Davies).

3.5.5. Simplified Administration

Fuse ESB Enterprise's messaging service provides many ways to manage and administer a messaging system.

3.5.5.1. Advisory messages

Act as administrative channels from which you can receive status on events taking place on the broker and on producers, consumers, and destinations in real time. Using the broker's advisory messages, you can monitor the messaging system using regular JMS messages, which are generated on system-defined topics (see Section 3.5.3.3, “Dead letter queue”). You can also use advisory messages to modify an application's behavior dynamically.

For details, see ActiveMQ in Action (Snyder, Bosanac, and Davies).

3.5.5.2. Interceptor plug-ins

Fuse ESB Enterprise's messaging service provides several plug-ins for visualizing broker components and for logging and retrieving statistics collected on a running broker.

- Authentication—Two plug-ins provide different types of authentication: Simple authentication and JAAS authentication.
- Central timestamp—Updates the timestamp on messages as they arrive at the broker. Useful when

clients' system clocks are out-of-sync with the broker's clock.

- Enhanced logging—Enables you to log messages sent or acknowledged on the broker.
- Statistics—Provides statistics about running brokers and about destinations.
- Visualization—Two plug-ins generate graph files for different broker components. You can display these graphs using several publicly available visualization tools.

For more information, see ActiveMQ in Action (Snyder, Bosanac, and Davies).

3.5.5.3. JMX

Fuse ESB Enterprise's messaging service provides extensive support for monitoring and controlling the broker's behavior from a JMX console, such as jConsole.

For details, see ActiveMQ in Action (Snyder, Bosanac, and Davies).

3.5.5.4. Web Console

The Web Console is a web-based administration tool used to monitor the status of the broker. Combined with Fuse ESB Enterprise's JMX support, the Web Console displays details about a broker, such as system usage, queues, topics and advisory messages, subscribers, connections and protocols, scheduled messages, delayed and scheduled deliveries, and so on.

For details, see ActiveMQ in Action (Snyder, Bosanac, and Davies).

---

[1] Connection factories and destinations are preconfigured JMS objects, known as administered objects. They contain provider-specific configuration data for clients' use. Clients typically access administered objects through the Java Naming and Directory Interface (JNDI).

[2] The default wire protocol used to serialize data for transmission over the network connection is OpenWire (see Section 3.5.1.3, “Client-side APIs”).

[3] Active consumers are those that are connected to a networked broker, have registered interest in a particular topic or queue, and are ready to receive messages from it

Chapter 4. Routing and Integration

4.1. The Embedded Routing and Integration Service

4.1.1. Overview

Based on Apache Camel, Fuse ESB Enterprise's embedded routing and integration service is a framework for building integration and routing solutions using enterprise integration patterns (EIPs). Using Fuse ESB Enterprise's routing and integration service, you can quickly define and implement routes. A route defines an integration path between endpoints—between an input source (consumer) and one or more output destinations (producers). While data is in route between endpoints, you can manipulate it using enterprise integration patterns or custom processors such that the data arrives at each target destination in the format each requires.

Fuse ESB Enterprise's routing and integration service provides two DSLs—one based on Java and one based on XML—which you can use to specify routing rules. The DSLs contain methods named after the EIPs described in *Enterprise Integration Patterns—Designing, Building, and Deploying Messaging Solutions* (Hohpe and Woolf).

4.1.2. Features

Fuse ESB Enterprise's routing and integration service makes it easy to develop integration solutions. Some of its features are:

- Routing and mediation engine

  The routing and mediation engine is the core feature of the service. It moves messages from point to point, according to routes configured using EIPs.
- Enterprise integration patterns

  Within routes, you can use the supported enterprise integration patterns (EIPs) to implement processors, or create your own using plain old Java objects (POJOs). EIPs provide proven solutions for specific integration design problems. The patterns have evolved over time, having been tested and refined through serious use. Out of the box, Fuse ESB Enterprise supports over sixty EIPs. (See Section 4.3.2, “Routes and processors”.)

  For more details, see *Fuse ESB Enterprise 7.0 Implementing Enterprise Integration Patterns*: "Implementing Enterprise Integration Patterns"/>.
- Modular and pluggable architecture

  Fuse ESB Enterprise's routing and integration service has a modular architecture, which enables you to load any component into it, whether the component is provided by Fuse ESB Enterprise, a third party, or custom-built by you.
- Convention Over Configuration

  Fuse ESB Enterprise's routing and integration service adheres to the convention over configuration design paradigm whenever possible to minimize configuration requirements. For example, the routing and integration service uses an easy and intuitive URI scheme to configure endpoints directly within routes.
- Automatic type converters

  Fuse ESB Enterprise has a built-in type-converter mechanism and provides over 150 converters. This mechanism enables Fuse ESB Enterprise's routing and integration service to easily adapt to many protocols, and it also eliminates the need to configure type-converter rules. If you need to convert unsupported types, you can create your own type converter.

4.2. Messages and Message Exchanges

4.2.1. Message basics

Messages deal with data and are the basic structure for moving data over routes. Messages consist of a body (also known as the payload), headers, and attachments (optional). They flow in one direction, from sender to receiver. Each message has a unique ID, generated either by the message creator or by the routing and integration service.

Headers contain metadata, such as sender IDs, content encoding hints, and so on. Attachments can be text, image, audio, or video files, and are typically used with email and web service components.

4.2.2. Message exchange basics

Message exchanges deal with conversations and can flow in both directions. They encapsulate messages in containers while the messages are in route to their target endpoints. Message exchanges incorporate message exchange patterns (MEPs), which determine the messaging mode.

The consumer endpoint sets the MEP element, which has two options for configuring the messaging mode:

- **`InOnly`**

  Specifies one-way messaging (also known as event messaging). When set, the message exchange contains only an In message, and the caller expects no response. (JMS messaging is often one-way.)
- **`InOut`**

  Specifies request-response messaging. When set, the message exchange contains an In message and an Out message, and the caller expects a response. (HTTP-based transports are often **`InOut`**, where clients request and wait to receive a web page from the server.)

Besides the MEP element, a message exchange consists of several other elements: exchange id (identifies the conversation), exception, and properties. You can inspect and modify these elements while a message is in route to its target endpoint.

For more details, see *Fuse ESB Enterprise 7.0 Programing EIP Components*: "MsgFormats"/>.

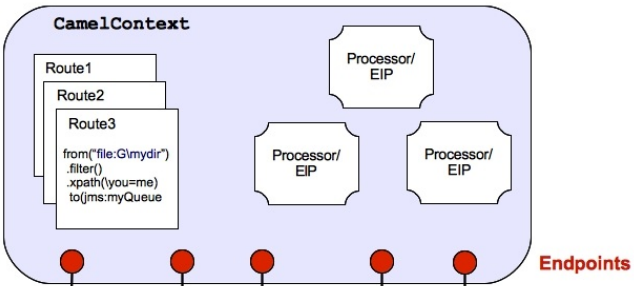
Routing and integration messages differ from JMS messages. For details, see Section 3.3, “JMS Message Basics”.

4.3. Routing Runtime

4.3.1. Overview

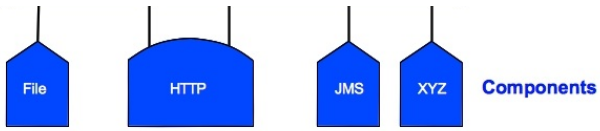
Figure 4.1, “Routing and integration service architecture” shows a high-level overview of the routing and integration service's architecture.

Figure 4.1. Routing and integration service architecture



At a high-level, Fuse ESB Enterprise's routing and integration service consists of routes, processors, components, and endpoints, all of which are contained within a **`CamelContext`**, the routing runtime. In

addition, the routing runtime also contains type converters, data formats, languages, and a registry.



For a detailed description of the routing runtime (**`CamelContext`**), see the *Fuse ESB Enterprise 7.0 Programing EIP Components*: "Configuring a Component".

4.3.2. Routes and processors

Routes specify paths over which messages move. Processors execute actions on messages (such as routing, transformation, mediation, enrichment, validation, interception, and so on) as they move between the route's consumer and producer endpoints.

You create processors using the supported EIPs or by building your own. For more details, see the *Fuse ESB Enterprise 7.0 Programing EIP Components*: "FuseMRProg".

You create routes in routing rules by wiring a string of processors together between a consumer and one or more producer endpoints. The output from one processor in the string is the input into the next one.

Routing rules are declarative statements (written in Java or XML DSL) that define the paths which messages take from their origination (*source*) to their target destination (*sink*). Routing rules start with a consumer endpoint (**`from`**) and typically end with one or more producer endpoints (**`to`**). Between the consumer and producer endpoints, messages can enter various processors, which may transform them or redirect them to other processors or to specific producer endpoints. For example, the Java DSL **`choice()`** method creates a content-based router;

```
from("jms:widgetOrders")
 .choice()
   .when(predicate)
     .to("jms:validOrders")
   .otherwise
     .to("jms:deadOrders");
```

which, based on the specified predicate, directs messages to one or the other producer endpoint.

Figure 7.2, “Major Widgets integration solution” shows an example that employs five routes and two processors. The consumer endpoint, **`camel-cxf`** **(3)**, receives part order requests and passes them to a content-based router processor **(4)**, which pushes each onto one of the four store's queue **(5)**, according to the part order's zip code. If a part order's zip code fails to match any of the zip codes on its list, the content-based router processor pushes the part order onto the **deadOrders** queue. In the back end routing runtime, the dynamic router processor **(6)** directs orders to different destinations based on whether the ordered parts are available.

4.3.3. Components

Components are plug-ins that add connectivity to other systems. A component behaves like a factory, creating endpoints of a given type. (For example, you'd use an HTTP component to create HTTP endpoints, and a File component to create file endpoints.)

Fuse ESB Enterprise includes over eighty components that support a wide range of functions, such as data transports, message validation, JMS messaging, and so on.

For more details, see *Fuse ESB Enterprise 7.0 Programing EIP Components*: "FuseMRProg".

4.3.4. Endpoints

Created by components, endpoints represent the end of a message channel through which a system can send and receive messages. You both configure and refer to endpoints using URIs.

Functionally, endpoints are a message source or a message sink, mapping to either a network location or some other resource that can produce or consume a stream of messages. Within a routing rule, endpoints are used in two distinct ways:

- Consumer

  A consumer endpoint is a message source. It appears at the beginning of a routing rule. It receives messages from an external source and creates a message exchange object, which the routing rule processes.

  For example, the **`camel-cxf`** endpoint **(3)** shown in Figure 7.2, “Major Widgets integration solution” is a consumer endpoint.
- Producer

  A producer endpoint is a message sink. It appears at the end of a routing rule. It sends the current message wrapped in a message exchange to an external target destination.

  For example, the **`file`** endpoint **(7)** and the **`mail`** endpoint **(9)**, shown in Figure 7.2, “Major Widgets integration solution” are producer endpoints.

For more details, see *Fuse ESB Enterprise 7.0 Programing EIP Components*: "MsgFormats"/>.

4.4. Integration Development

Developing a routing and integration application involves these high-level steps:

1. *Determine the routes your application needs to implement*.
   - Map out the routes from beginning to end.
   - Determine what the expected inputs and outputs are, such as data types and formats; what connection and transmission protocols must be integrated; and so on.
   - Determine whether and how you need to process messages before they reach their destination.
2. *Define routing rules*.
3. *Implement business logic*.

   When necessary, you can implement custom business logic using plain old Java objects (POJOs).
4. *Configure components*.

   To use components other than those embedded in Fuse ESB Enterprise's messaging service, you must configure them using either Java code or XML.
5. *Choose a deployment method*.
   - FAB-based deployment

     Fab (see Section 1.7, “FABs in a nutshell”) is the recommended deployment method. You can deploy FABs in several ways.

     For details on building and deploying FABs, see *Fuse ESB Enterprise 7.0 Deploying into the Container*: "Building a FAB" and *Fuse ESB Enterprise 7.0 Deploying into the Container*: "Deploying a FAB".
   - OSGi-based deployment

     For details, see *Fuse ESB Enterprise 7.0 Deploying into the Container*: "Deploying an OSGi Bundle".
6. *Deploy the route*.

Chapter 5. Web and RESTful Services

5.1. The Embedded Web and RESTful Services

Fuse ESB Enterprise's embedded Web and RESTful services framework supports a variety of standards and protocols for creating web services:

- Web service standards

  Supported standards include SOAP, WSDL, WS-Addressing, WS-Policy, WS-SecureConversation, WS-Security, WS-SecurityPolicy, and WS-Trust.

- Front end programming standards

  Options include JAX-WS for web services, supporting both contract-first (starting with WSDL) and code-first (starting from Java) development, and JAX-RS for RESTful web services.

- Binary and legacy protocols

  Supported message bindings include SOAP and plain XML, and JSON. Supported transports include HTTP/S, JMS, and CORBA.

5.1.1. Font end options

Front ends provide a programming model for interacting with Fuse ESB Enterprise's Web and RESTful services.

You can create a front end using either of these options:

- JAX-WS

  Develop web services using either a code-first (Java) or contract-first (WSDL) approach. For details, see Section 5.2, “Web Service Development Pattern”.

- JAX-RS

  Develop RESTful web services using the JAX-RS APIs. For details, see Section 5.3, “RESTful Service Development Pattern”.

5.1.2. Data binding options

Data bindings implement the mapping between Java objects and XML by converting data to and from XML. Many data bindings can also produce XML schema.

Each of Fuse ESB Enterprise's data binding options implements a particular method for mapping: JAXB, JSON, or XMLBeans.

Your front end selection determines which data bindings you can use.

- JAXB

  Default for all front ends. See Section 5.2.3, “JAXB data bindings”.

- JSON

  Optional for JAX-RS front ends. For details, see JSON.org.

- XMLBeans

  Optional for JAX-WS front ends. For details, see *Fuse ESB Enterprise 7.0 Developing Applications Using JAX-WS*: "Specifying the Data Binding".

See *Fuse ESB Enterprise 7.0 Developing Applications Using JAX-WS*: "JAXWSDataMapping" and *Fuse ESB Enterprise 7.0 Developing RESTful Web Services*: "RESTIntro" for detailed information on using

Fuse ESB Enterprise's data bindings.

5.1.3. Message binding options

Message bindings map a service's messages to a particular protocol. Fuse ESB Enterprise supports these message bindings:

- SOAP

  Default for JAX-WS front ends. It maps messages to the SOAP protocol and can be used with the various WS-* modules.
- XML

  Avoiding the serialization of the SOAP envelope, the pure XML message binding sends a raw XML message.

5.1.4. Transport options

Fuse ESB Enterprise's Web and RESTful services framework uses a transport abstraction layer to hide transport-specific details from the front end and bindings layers. Fuse ESB Enterprise supports SOAP or XML over HTTP or JMS.

RESTful services use either SOAP or XML over HTTP only.

HTTP, the underlying transport for the Internet, provides a standardized, robust, and flexible platform for communication between endpoints, and is the assumed transport for most WS-* specifications.

5.2. Web Service Development Pattern

5.2.1. Overview

With JAX-WS, you can develop Web services using one of two development approaches:

- Code-first

  Considered the easier approach, code-first (or java-first) is favored for small-scale, tactical integrations. Code-first services use Java annotations in the code, from which WSDL and XSD artifacts are generated on-the-fly.

  The code-first approach is useful when you have existing Java code that implements a set of functionality that you want to expose as part of a service-oriented application, or you just don't want to use WSDL to define your interface. Using JAX-WS annotations in your code, you can add the information required to service-enable a java class and also to create a Service Endpoint Interface (SEI) that can be used in place of a WSDL contract.

  For more information, see "Starting from Java Code" in the *Fuse ESB Enterprise 7.0 Developing Applications Using JAX-WS*.
- Contract-first

  Contract-first (or WSDL-first) is preferred for large-scale, strategic Service Oriented Architecture (SOA) integrations. Contract-first services are typically modular, platform agnostic, and better attend to versioning.

  The contract-first approach uses a WSDL document to define the operations a service exposes and the data that is exchanged with it. Using the WSDL document, you generate starting-point code for the service provider, then add the business logic to it using the Java APIs.

  For more information, see *Fuse ESB Enterprise 7.0 Developing Applications Using JAX-WS*: "JAXWSWSDLFirst"/>.

5.2.2. Data exchange format

Service-oriented design abstracts data into a common exchange format, typically an XML grammar defined in XML schema. The JAX-WS specification calls for marshalling XML schema types into Java objects, in accordance with the Java Architecture for XML Binding (JAXB) specification.

JAXB defines bindings for mapping between XML schema constructs and Java objects and rules for marshalling the data. It also defines an extensive customization framework for controlling how data is handled. For details, see Section 5.2.3, “JAXB data bindings”.

5.2.3. JAXB data bindings

JAXB enables you to store and retrieve data in memory in any XML format, without implementing a set of specific XML loading and saving routines for the program's class structure. It also enables you to map Java classes to XML representations, so you can:

- Marshal Java objects into XML
- Unmarshal XML back into Java objects

JAXB uses Java annotation in combination with files found on the classpath to build the mapping between XML and Java objects. It supports both code-first and contract-first programming.

JAXB is extremely useful when your service specification is complex and changes regularly, because changing the XML schema definitions to maintain synchronization with the java definitions is time consuming and error prone.

For more details, see *Fuse ESB Enterprise 7.0 Developing Applications Using JAX-WS*: "JAXWSDataMapping"/>.

Note

In the Major Widgets use case, JAX-WS is used to create a front-end order entry application on each of the in-store terminals. See Section 7.3, “Major Widgets Phase One Solution”.

5.3. RESTful Service Development Pattern

5.3.1. Overview

Representational State Transfer (REST) describes a stateless client-server architecture in which web services are treated as resources that can be identified by their URLs.

In RESTful systems, servers use URIs to expose resources, and clients use the four HTTP verbs **`GET`**, **`POST`**, **`PUT`**, and **`DELETE`** to access them. As they receive a representation of a resource, client applications enter a state. When they access another resource, typically by following a link, client applications change their state.

Because RESTful systems can take full advantage of HTTP's scalability features, such as caching and proxies, they are highly scalable and highly flexible. Changes made to servers do not impact clients because:

- Resources are accessed and manipulated using the four HTTP verbs.
- Resources are exposed using a URI.
- Resources are represented using standard grammars.

Today's Web architecture is an example of a system designed on RESTful principles. Web browser clients access resources hosted on Web servers. The resources are represented using HTML or XML grammars understood and navigable by all web browsers.

5.3.2. JAX-RS development

Fuse ESB Enterprise supports JAX-RS (JSR-311), the Java API for RESTful Web Services. JAX-RS provides a standard way to build RESTful services in Java, using annotations to simplify the development and deployment of web service clients and endpoints.

Using JAX-RS to develop restful services has several benefits:

- The URI bindings are local to the resource beans, which can be arbitrarily nested. This feature simplifies refactoring.
- Loose coupling between providers and the objects returned by the resource methods makes it easy to drop in support for new representations, without having to change the code of the resource beans (or controllers). Instead, you need only modify an annotation.
- Static typing can be useful when binding URIs and parameters to the controller. For example, using `String`, `integer`, and `Date` fields frees the controller from having to explicitly convert parameter values.

For more information, see "Overview" in the *Fuse ESB Enterprise 7.0 Developing RESTful Web Services*.

5.3.3. JSON data bindings

JavaScript Object Notation (JSON), a lightweight data format for data exchange, is provided as an alternative to JAXB. JSON is a text-based and human-readable format for representing simple data structures and associative arrays (called objects).

For more information, see JSON home.

Note

In the Major Widgets use case, JAX-RS is used to implement a web-based order entry form that customers use to make on-line orders. See Section 7.3, “Major Widgets Phase One Solution”.

Chapter 6. Centralized Configuration, Deployment, and Management

6.1. Overview

Fuse ESB Enterprise allows you deploy a group of containers into a *fabric*. All of the brokers in a fabric are managed by a distributed *Fabric Ensemble* that stores runtime and configuration information for all of the containers in the fabric. Using this information the ensemble enables you to manage large deployments by:

- automating the configuration of message brokers into failover groups, networks of brokers, and master/slave groups

  The ensemble knows the connection details for all of the brokers in the fabric and uses that information to configure brokers into different configurations. Because the information is updated in real time, the broker configurations are always correct.
- automating client connection, failover, and load balancing configuration

  When services, including message brokers, are configured into a fabric, clients can use special discovery protocols that queries the ensemble for an appropriate service instance or message broker. The ensemble uses a round robin algorithm for balancing load among common service instances in the fabric. The fabric discovery connection will also automatically failover to other service instances without needing to know any details about what instances are deployed.
- centralizing configuration information

  All of the configuration for each container in the fabric, including details about which bundles are deployed, are stored in the ensemble. Any of the configurations can be edited from a remote location and directly applied to any container in the fabric.
- simplifying the enforcement of common configuration policies

  The ensemble organizes configuration information into profiles that can inherit from each other. This makes it easy to enforce common policies about how brokers need to be configured using base profiles.
- centralizing management and monitoring functionality

  Using Fuse Management Console, Fuse IDE, or the Fuse ESB Enterprise command console you can update configuration profiles, deploy new containers, shutdown containers, and get runtime metrics for any of the containers in the fabric

6.2. Anatomy of a fabric

Figure 6.1, “A Fabric” shows the components of a fabric with seven brokers and an ensemble consisting of three servers.

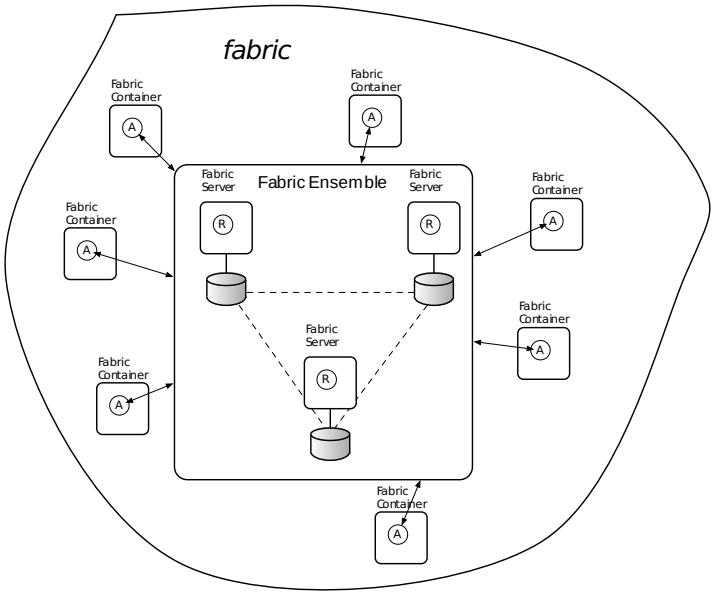


Figure 6.1. A Fabric

The key components of the fabric are:

- Fabric Ensemble—a group of one or more *Fabric Servers* that work together to maintain the registry and other services that provide the glue for the fabric
- Fabric Server—a container that hosts the runtime functionality for the ensemble
- Fabric Container—a Fuse ESB Enterprise container that is managed by a *Fabric Agent*
- Fabric Agent—a process that communicates with the ensemble to manage the container's configuration

6.3. Fabric Ensemble

The Fabric Ensemble, based on Apache Zookeeper, maintains two databases:

- centralized configuration—stores the configuration profiles for all of the containers in the fabric. The configuration profiles are organized by versions and allow for inheritance between profiles.
- runtime information—stores status for all of the containers in the fabric. The runtime information stored in the database includes the status of the container, the address of any endpoints and messaging destinations exposed by the container, the JMX URL used to monitor the container, the number of client's accessing services deployed in the container, etc.

The ensemble is intended to be distributed across multiple machines to provide a level of high-availability. The servers in the ensemble use a quorum policy to elect the master server and will re-elect a new master when the current master fails. To ensure optimal performance of the ensemble you should deploy an odd number of Fabric Servers to maintain the ensemble. The more servers in the ensemble, the more failures it can endure.

The ensemble is dynamically updated by the Fabric Agents running in the containers that make up the fabric. The agents routinely report the status of their container to the ensemble. Configuration profiles are also updated dynamically when they are edited.

6.4. Profiles

A *profile* is a collection of configuration data that defines the runtime parameters for a container. It contains details about:

- features—XML specifications that define a collection of bundles, jars, and other artifacts needed to implement a set of functionality
- bundles—the OSGi bundles or Fuse Application Bundles to load into a container
- repositories—the URIs from which the container can download artifacts
- configuration files—configuration files OSGI Admin PIDs that are loaded by the container's OSGi admin service and applied to the services running in the container

Profiles are additive and can inherit properties from parent profiles. This makes it possible to create standardized configurations that can be used throughout your deployment. The profile for a specific container will inherit the standard configuration from the base profile and add any specific settings needed to tailor the container for its specific deployment environment.

Profiles are organized into *versions*. A version is a collection of profiles. Creating a new version copies all of the profiles from the parent version and stores them in a new collection. A Fabric Container can access only one version at a time, so changes made to the profiles in one version will only effect the containers that are assigned to the version being edited. This makes it easy to roll out changes to a fabric incrementally.

6.5. Fabric Agents

A Fabric Agent is the link between the ensemble and a Fabric Container. It communicates with the ensemble to:

- manage a container's configuration and provisioning—the agent regularly checks to see what profiles are assigned to the container and if the container is properly configured and running the appropriate set of services. If there is a discrepancy, the agent updates the container to eliminate the discrepancy. It flushes any extraneous bundles and services and install any bundles and services that are missing.
- updating the ensemble's runtime information—the agent reports back any changes to the container's runtime status. This includes information about the containers provisioning status, the endpoints and messaging destination exposed by the container, etc.

6.6. Working with a fabric

One of the advantages of a fabric is that you can manage it from any of the containers in the fabric. The `fabric` command shell interacts directly with the ensemble, so all changes are immediately picked up by all of the containers in the fabric.

You can also use Fuse Management Console to work with a fabric. Fuse Management Console provides a Web-based UI that allows you to:

- add containers to a fabric
- create profiles and versions
- edit profiles
- assign profiles to containers
- monitor the health of the containers in a fabric

Chapter 7. The Major Widgets Use Case

7.1. Introducing Major Widgets

7.1.1. Overview

When Major Widgets, a single Mom and Pop operation that ran an auto parts supply store, decided to buy three more auto part supply stores, they knew they'd have to change their business model and integrate the systems located in all four stores.

7.1.2. Major Widgets business model

Major Widgets, and each of the three stores it bought, routinely supply a number of auto repair shops that are located near them. Each store delivers parts to customers free-of-charge, as long as the customer is located within twenty-five miles of the store. Each store has its own database for storing auto repair customer accounts, store inventory, and part suppliers.

Business was done over the phone, but now Major Widgets wants to implement an online order service to enable their auto repair customers to order parts more quickly and efficiently. The Web-based service will coordinate orders and deliveries, bill customers, track and update each store's inventory, and order parts from suppliers. Customers can use it to check the status of their orders.

All four stores also sell parts over-the-counter to walk-in customers, for whom they do not typically establish customer accounts. Each of the in-store ordering systems will also tie into its store's central order processing system.

7.1.3. Major Widgets future plans

In the long run, Major Widgets wants to centralize order processing and inventory maintenance for all of its stores. Doing so would make it easier to maintain inventory at an optimal level at each store and to analyze business trends in their network of stores. To minimize down time and impact on its resources, Major Widgets' plans to implement these changes in incremental phases.

7.2. Major Widgets Integration Plan: Phase One

Figure 7.1, “Major Widgets integration plan” shows a high-level view of how Fuse ESB Enterprise will provide an integration solution to implement phase one of Major Widgets' new business model.

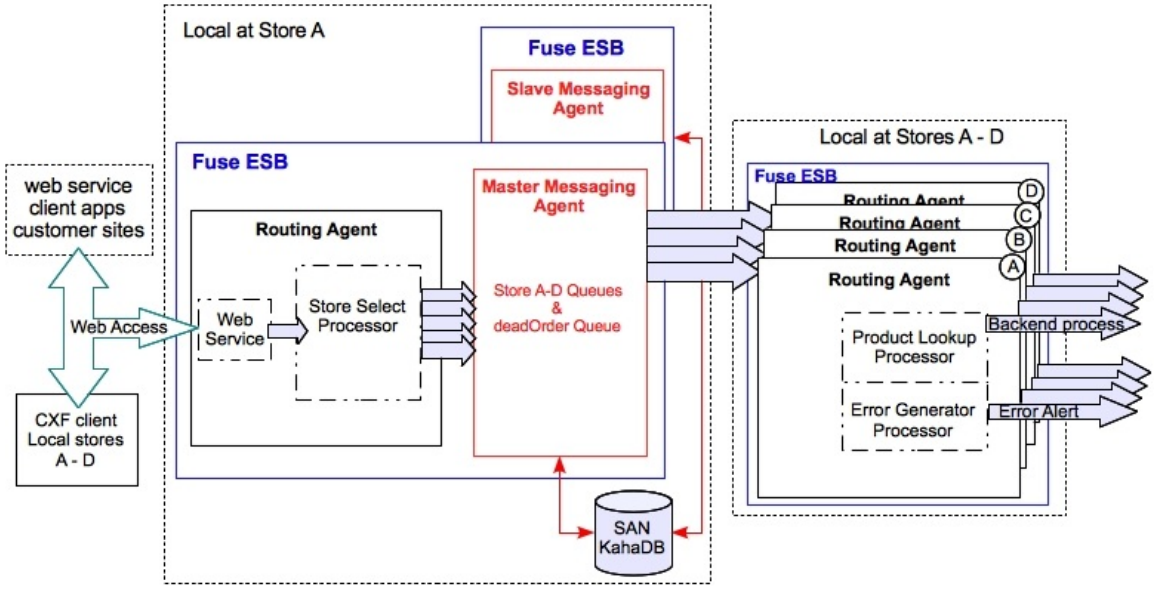


Figure 7.1. Major Widgets integration plan

The phase one plan creates:

- a single order entry point into the order processing system that can be accessed via the Web and by the in-store terminals
- an intelligent order entry system that routes Web-based orders to the store closest to the delivery destination
- an order processing system (instances running locally at each store) that receives and processes orders, maintains customer accounts, and tracks and maintains inventory
- a master/slave broker cluster that provides a highly available, reliable messaging backbone for the integration solution

The phase one plan allows each store to retain their existing internal systems, but enables them to function as a single unit.

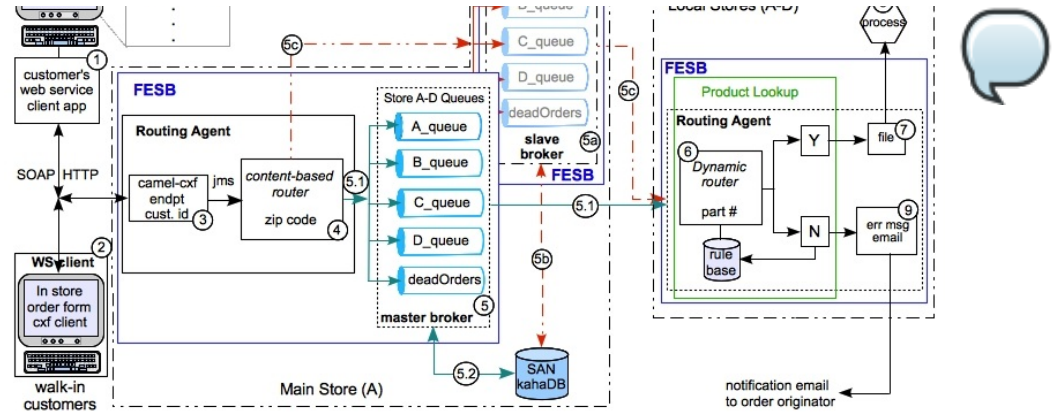
Note

Because Major Widgets requires its systems to remain up and running at all times, the front and back ends must be able to be replaced while the system is running. To fulfill this requirement, the front end and each back end will be deployed as FABs (see Section 1.7, “FABs in a nutshell”).

7.3. Major Widgets Phase One Solution

Figure 7.2, “Major Widgets integration solution” shows how phase one of Major Widgets integration plan might be implemented.

Figure 7.2. Major Widgets integration solution



Note

The encircled numbers in Figure 7.2, “Major Widgets integration solution” serve as reference points for concepts presented in this guide.

For example, (1) in the text references ① where ever it appears in these figures.

7.3.1. Major Widgets solution components

Fuse ESB Enterprise's kernel provides a runtime environment that provides enterprise support (management, logging, provisioning, security) for the main store **(A)**, where most of the integration applications run. Its embedded services provide the frameworks for implementing these components of the solution:

- RESTful service—for creating a JAX-RS application that runs on each auto repair shop terminal **(1)**, enabling customers to input part orders, via an order entry form, over the internet.
- Web service—for creating a JAX-WS front end to implement the order entry functionality on each of the in-store terminals, which receive orders from walk-in customers **(2)** who purchase parts over-the-counter.
- `camel-cxf` component—a routing and integration service component that creates an entry endpoint **(3)** that exposes Major Widgets routing logic to the outside world as a web service or RESTful service.
- Routing and integration service—for creating routes **(4, 6)** that direct orders received from the web/RESTful service entry point through the appropriate store's order processing back end.
- Messaging service—for creating a persistent, fault-tolerant clustered messaging system **(5, 5a)**, which ensures that no order is ever lost due to failure of the system, the message broker, or the connections between the message broker and its various clients—the front end content-based router **(4)** and the back end dynamic router **(6)**.

7.3.2. Major Widgets integration flow

At Major Widgets main store ***(A)***, the order entry front end (routing and messaging agents running inside Fuse ESB Enterprise) is running on that store's main computer system. At each of the four stores ***(A-D)***, an instance of the order entry back end (routing agent and back end processing running in Fuse ESB Enterprise) is running on the local computer system.

When the front end web service **(3)** receives an online order, the routing agent passes it to a content-based router **(4)** to determine which store to route the part order for further processing. Normally, the order then enters the target store's queue **(5)**, where it waits until the target store retrieves it **(6)**. (With fault tolerance built into the system, if the master broker **(5)** fails, the system can continue to function with no loss of orders. For details, see Section 7.3.3, “Built-in Broker Fault Tolerance”.)

In the case of auto repair shops **(1)**, the content-based router routes order requests to the store nearest the customer, based on the submitted zip code. In the case of walk-in customers **(2)**, the auto supply store submits its own zip code to the front end, so the order is always routed to the local store.

When the back end receives the submitted part order, the application employs a dynamic router **(6)** to look up the parts in the store's database to see if they are in stock. Results depend on whether the customer is an auto repair shop or a walk-in:

- Auto repair shop customers

  If the parts are available, the order is submitted to the store's back end processing software **(8)**, which informs and bills the customer **(1)**, schedules delivery, updates inventory, and reorders parts accordingly.

  If the parts are unavailable, the order is submitted to a processor that generates an error message, which is emailed **(9)** to the customer **(1)**.
- Walk-in customers

  If the parts are available, the order is submitted to the store's back end processing software **(8)**, which informs the store clerk **(2)**, updates inventory, and orders parts accordingly. The store clerk retrieves the parts from stock and sells them to the customer over-the-counter.

  If the parts are unavailable, the order is submitted to a processor that generates an error message, which is emailed **(9)** to the local store's email account **(2)**. The store clerk informs the customer, who can then decide whether he wants the store clerk to search the other stores for his parts.

7.3.3. Built-in Broker Fault Tolerance

Figure 7.2, “Major Widgets integration solution” shows how the Major Widgets integration plan implements a fault-tolerant mechanism to protect against loss of orders due to broker failure.

In this scenario, a slave broker **(5a)**, which provides fault tolerance for the messaging system, is running within Fuse ESB Enterprise **(FESB)** on a separate, backup computer system at the main store. Both master broker and slave broker connect **(5.2/5b)** to a shared file system located on a storage area network (**SAN**). In addition, the messaging system has been configured for persistent messaging, storing messages to disk until consumers acknowledge their receipt, to protect against message loss. And all clients have been configured to use the failover transport to enable them to automatically reconnect to the slave broker when the master broker fails.

With the message store set up, the first broker (**(5)** in the Major Widgets scenario) to grab an exclusive lock on the shared file system becomes the master broker. The slave broker loops, waiting to grab the exclusive lock when it becomes available. When the master broker **(5)** fails, it immediately shuts down its transport connections **(5.1)** to all clients and gives up the exclusive lock on the shared file system **(5.2)**. Concurrently, the slave broker **(5a)** activates its transport connections to all clients **(5c)** and grabs the file system lock **(5b)**. Doing so, the slave broker becomes master broker. When the former master broker restarts, it becomes the slave broker and waits to grab the exclusive lock whenever it becomes available.

When the master broker fails, all clients configured to fail over reconnect automatically to the next broker in the supplied list—in this case the initial slave broker **(5a)**. Because the Major Widgets scenario employs a cluster of only two brokers, subsequent failures cause the brokers to switch master/slave roles back and forth between them.

For details on persistent messaging, see *Fuse ESB Enterprise 7.0 Configuring Message Broker Persistence*: "Configuring Message Broker Persistence". For details on the failover protocol, see *Fuse ESB Enterprise 7.0 Fault Tolerant Messaging*: "Failover Protocol".

Chapter 8. Getting More Information

8.1. FuseSource Documentation

All of these documents are part of the Fuse ESB Enterprise library, downloadable from the Fuse ESB Enterprise documentation Web site. A few require that you register with FuseSource before you can view them.

8.1.1. Administration

8.1.1.1. Basics

The books listed here provide information and instructions for installing and setting up Fuse ESB Enterprise in preparation for developing, deploying, and administering enterprise integration solutions.

- *Release Notes*

  You'll find the release notes for the latest version of Fuse ESB Enterprise here. For each version, the release notes list and describe bug fixes and new improvements included in the newly released software.
- *Fuse ESB Enterprise 7.0 Installation Guide*

  Provides detailed instructions for installing the Fuse ESB Enterprise software on Windows, Linux, Unix, and OS X platforms, using the binary distributions or building from source code. It also lays out the prerequisites needed to ensure a successful installation.
- *Fuse ESB Enterprise 7.0 Migration Guide*

  Describes all of the latest changes made to the Fuse ESB Enterprise software and, where necessary, provides instructions for integrating the changes into existing systems.
- *Fuse ESB Enterprise 7.0 Configuring and Running Fuse ESB Enterprise*

  Provides information and instructions for starting/stopping Fuse ESB Enterprise, using remote and child instances of the runtime, configuring Fuse ESB Enterprise, configuring logging for the entire runtime or per component application, configuring where persistent data (messages, log files, OSGi bundles, transaction logs) is stored, and configuring failover deployments.
- *Fuse ESB Enterprise 7.0 Glossary*

  Defines terms used through out FuseSource product documentation.

8.1.1.2. Advanced

The books listed here provide information and instructions for deploying OSGi-based applications and managing their dependencies, for securing Fuse ESB Enterprise and some of its embedded components, and for using each of the console commands.

- *Fuse ESB Enterprise 7.0 Deploying into the Container*

  Describes how to bundle and deploy OSGi-based applications (features, Jars, Wars, routes, web services, brokers, OSGi services, and more) into Fuse ESB Enterprise's OSGi container. It also describes how to perform interbundle communication using the NMR, the Pax-Exam testing framework, URL handlers, and OSGi best practices.
- *Fuse ESB Enterprise 7.0 Managing OSGi Dependencies*

  Describes basic OSGi class loading concepts, the rules for importing and exporting packages in relation to the type of OSGI bundles used in an application, and OSGi version policies so you can avoid incurring incompatible dependencies in your application.
- *Fuse ESB Enterprise 7.0 Security Guide*

Describes how to secure the Fuse ESB Enterprise container, the web console, message brokers, routing and integration components, web and RESTful services, and it provides a tutorial on LDAP authentication.

You must register at fusesource.com/register before you can view this document located on the **Documentation** tab at Fuse ESB Enterprise.

- *Fuse ESB Enterprise 7.0 Console Reference*

  Lists and describes all of Fuse ESB Enterprise's console commands. These commands allow you to manage the Fuse ESB Enterprise runtime and its embedded components.

- *ESB API Reference*

  Includes API references for Fuse ESB Enterprise's kernel (Apache Karaf and Apache Felix), Apache ActiveMQ (messaging), Apache Camel (routing and integration), Apache CXF (Web and RESTful services), the JBI (Java Business Integration) components, and the NMR (Normalized Message Bus).

8.1.2. Messaging

8.1.2.1. Developer—basics

The books listed here provide information about JMS messaging and instructions for developing basic messaging applications—those that do not implement transactions.

- *Fuse ESB Enterprise 7.0 Broker Client Connectivity Guide*

  Describes each of the supported transport options and connectivity protocols in detail and includes code examples.

- *Fuse ESB Enterprise 7.0 Configuring Message Broker Persistence*

  Describes the basic concepts of message persistence and provides detailed information on the supported message stores: KahaDB and JDBC database with/without journaling. It also describes how to use message cursors to improve the scalability of the message store.

- *Fuse ESB Enterprise 7.0 Using Networks of Brokers*

  Describes basic network of brokers concepts and topologies, network connectors, failover and discovery protocols for dynamically discovering and reconnecting to brokers in a network, as well as balancing consumer and producer loads

  You must register at fusesource.com/register before you can view this document.

- *Fuse ESB Enterprise 7.0 Fault Tolerant Messaging*

  Describes how to implement fault tolerance using master/slave broker patterns.

- *Fuse ESB Enterprise 5.5.0 XML Schema Reference*

  Links to *Apache ActiveMQ vx.x XML Schema Reference*, where, for each namespace, all available components are listed. This is the configuration reference for Apache ActiveMQ.

8.1.2.2. Developer—advanced

The books listed here provide information and instructions for implementing transactions and for securing message brokers, and accessing all available message broker classes.

- *Fuse ESB Enterprise 7.0 Tuning the Message Broker*

  Describes general techniques for performance tuning messaging applications and specific techniques for performance tuning persistent messaging.

  You must register at fusesource.com/register before you can download this document from the Fuse ESB Enterprise web page.

- *Fuse ESB Enterprise 7.0 Security Guide*

  Provides tutorials on implementing SSL/TSL security and JAAS authentication for a message broker implementation.

You must register at fusesource.com/register before you can view this document located on the **Documentation** tab at Fuse ESB Enterprise.

- *API Reference*

  Links to the *Apache ActiveMQ 5.5.0 API*, where, for each package, all available classes are listed.

8.1.3. Routing and Integration

8.1.3.1. Developer—Basics

The books listed here provide information and instructions for developing basic integration and routing applications—those that do not implement transactions or require custom-built processors.

- *Fuse ESB Enterprise 7.0 Implementing Enterprise Integration Patterns*: "Implementing Enterprise Integration Patterns"

  Describes how to build routes, from the basic building blocks and principals to the supported Enterprise Integration Patterns (EIPs).
- *Fuse ESB Enterprise 7.0 Programing EIP Components*: "FuseMRProg"

  Chapters 1 through 5 describe routing and integration basics and provide instructions for working with processors, type converters, consumer and producer templates, and components.
- *Fuse ESB Enterprise 7.0 Deploying into the Container*: "Deploying into the Container"

  Describes how to bundle and deploy routes into the OSGi container and how to invoke an OSGi service from within a route.
- *API Reference*

  Links to *FuseSource Distribution of Camel API*, where, for each package, all available classes are listed.
- *Apache Camel XML Schema Reference*

  Links to *Apache Camel 7.0 XML Schema Reference*, where, for each namespace, all available components are listed. This is the reference for the embedded routing and integration service's XML domain-specific language.

8.1.3.2. Developer—Advanced

The books listed here provide information and instructions for implementing transactions and for building custom processors.

- *Fuse ESB Enterprise 7.0 Programing EIP Components*

  Chapters 6 through 11 describe how to work with component, endpoint, consumer, producer, exchange, and message interfaces, so you can create and implement your own custom components and processors.
- *Fuse ESB Enterprise 7.0 EIP Transactions Guide*

  Describes the basic concepts of transactions, how to select and implement a transaction manager, how to access data using Spring, the various ways to demarcate transactions, and JMS transaction semantics.

  You must register at fusesource.com/register before you can view this document from the Apache Camel web page.

8.1.4. Web and RESTful Services

The following reading paths provide quick summaries on how to accomplish many of the development tasks when working with the Web and RESTful services framework:

8.1.4.1. Service consumer developers

- *Fuse ESB Enterprise 7.0 Writing WSDL Contracts* provides a quick overview of the contents of a WSDL document.
- *Fuse ESB Enterprise 7.0 Developing Applications Using JAX-WS* describes how to develop a service consumer using a provided WSDL document. It discusses how to use the command-line tools to generate the stubs and what additional code is required.
- *Fuse ESB Enterprise 7.0 Developing Applications Using JAX-WS* describes how to develop a service consumer if you are provided a Java interface instead of a WSDL document.
- *Fuse ESB Enterprise 7.0 Developing Applications Using JAX-WS* describes how to develop consumers that make asynchronous invocations on remote services.
- "Using XML in a Consumer" in the *Fuse ESB Enterprise 7.0 Developing Applications Using JAX-WS* describes how to use the **`Dispatch`** interface to develop consumers that work with raw XML messages.

8.1.4.2. Java-first service developers

- *Fuse ESB Enterprise 7.0 Developing Applications Using JAX-WS* describes how to develop a SEI for your service.
- *Fuse ESB Enterprise 7.0 Developing Applications Using JAX-WS*: "JAXWSServiceDevJavaFirstAnnotate" describes the annotations used by the JAX-WS framework.
- "Using XML in a Service Provider" in the *Fuse ESB Enterprise 7.0 Developing Applications Using JAX-WS* describes how to use the **`Provider`** interface to develop services that work with raw XML messages.
- "Publishing a Service" in the *Fuse ESB Enterprise 7.0 Developing Applications Using JAX-WS* describes the code needed to publish a service as a standalone application.

8.1.4.3. WSDL-first service developers

- *Fuse ESB Enterprise 7.0 Writing WSDL Contracts* describes the contents of a WSDL document.
- *Fuse ESB Enterprise 7.0 Using the Web Services Bindings and Transports* describes how to add Apache CXF bindings and transports to a WSDL document.
- *Fuse ESB Enterprise 7.0 Developing Applications Using JAX-WS*: "JAXWSServiceDevWSDLFisrtGenCode" describes how to generate the template code needed for a service.
- *Fuse ESB Enterprise 7.0 Developing Applications Using JAX-WS* describes how to implement a service provider.
- "Publishing a Service" in the *Fuse ESB Enterprise 7.0 Developing Applications Using JAX-WS* describes the code needed to publish a service as a standalone application.

8.1.4.4. RESTful service developers

- *Fuse ESB Enterprise 7.0 Developing RESTful Web Services* provides an overview of RESTful design principles.
- *Fuse ESB Enterprise 7.0 Developing RESTful Web Services* describes how to create a root resource class for the resource that represents the top of the service's resource tree.
- *Fuse ESB Enterprise 7.0 Developing RESTful Web Services* describes how to map the service's other resources into sub-resources.
- *Fuse ESB Enterprise 7.0 Developing RESTful Web Services* describes how to create methods to implement each of the HTTP verbs used by each of the resources.
- *Fuse ESB Enterprise 7.0 Developing RESTful Web Services* describes how publish your service.

8.1.4.5. Advanced topics

- "Working with Contexts" in the *Fuse ESB Enterprise 7.0 Developing Applications Using JAX-WS* describes how to use the JAX-WS context mechanism.
- *Fuse ESB Enterprise 7.0 Developing Apache CXF Interceptors*: "Developing Interceptors"/> provides detailed information on writing interceptors that can be added to the Apache CXF message processing chain.

8.1.5. Java Business Integration (JBI)

The books listed here provide information and instructions for creating, building, and deploying JBI applications.

- *Fuse ESB Enterprise 7.0 Using Java Business Integration*

  Provides an overview of JBI, introducing the JBI framework and management structure; describes how to deploy JBI artifacts into the Fuse ESB Enterprise runtime; and how to use the JBI console commands.
- *Fuse ESB Enterprise 7.0 Using the JMS Binding Component*

  Provides an overview of the JBI JMS binding component; describes how to configure the connection factory, how to create and configure various types of endpoints, and how to use the Maven tooling.
- *Fuse ESB Enterprise 7.0 Using the Apache CXF Binding Component*

  Provides an overview of the JBI CXF binding component; describes how to define endpoints in WSDL, how to configure and package endpoints, and how to configure the CXF runtime; describes the properties of consumer and provider endpoints; and describes how to use the Maven tooling.
- *Fuse ESB Enterprise 7.0 Using the File Binding Component*

  Provides an overview of the JBI file binding component; describes configuring and using poller and sender endpoints and file marshallers; describes the properties of poller and sender endpoints; and describes how to use the Maven tooling.

8.2. Other Resources

This section provides links to webinars, video demos, and 3rd party books and articles. Check the FuseSource web site often for new webinars and video demos.

8.2.1. Administration

8.2.1.1. Webinars

The resources listed here are freely available from the FuseSource website at FuseSource Resources.

- Overview of OSGi Enterprise
- Introduction and Demo of Distributed OSGi
- Transactions, Clustering, OSGi and ServiceMix, the Open Source ESB
- Large-Scale Deployments & ServiceMix 4

8.2.2. Messaging

8.2.2.1. Webinars

The resources listed here are freely available from the FuseSource website at FuseSource Resources.

- Getting Started with Apache ActiveMQ

- Get started with ActiveMQ High-Availability
- Performance Tuning for ActiveMQ
- Reliable Messaging to the Browser
- Deploying ActiveMQ in the Enterprise

8.2.2.2. Video demos

The resources listed here are freely available from the FuseSource website at FuseSource Video demos

- Overview of JMS Messaging Concepts

  If you are already familiar with the basics of JMS messaging, you can dive directly into the Guided Tour demo.
- Setting up: The Guided Tour

  The Guided Tour sequentially walks you through setting up and running a number of example applications. Each example in the sequence builds upon the concepts introduced in previous ones.
  - Publish & Subscribe Examples
    - Chat application
    - Durable Chat application
    - Hierarchical Chat application
    - Message Monitor
    - Selector Chat
    - Transacted Chat
  - Point-to-Point Examples
    - Talk
    - Queue Monitor

8.2.2.3. Third party books and articles

The books and reference materials listed here provide detailed information and instructions for designing, building, and deploying enterprise integration solutions and for using the tools that make it easier to do so. The books and reference materials in this list were authored by leading experts in this domain.

- ActiveMQ in Action (Snyder, Bosanac, & Davies)

  Written by the ActiveMQ developers, ActiveMQ in Action is the definitive guide to understanding and working with Apache ActiveMQ.

  Starting off with explaining JMS fundamentals, it uses a running use case to quickly explain and demonstrate how to use ActiveMQ's many features and functionality to build increasingly complex and robust messaging systems.
- ActiveMQ is Ready for Prime Time (Rob Davies)

  This blog describes the many ways ActiveMQ has been deployed in demanding enterprise environments and provides case studies that demonstrate how ActiveMQ provides reliable connectivity not only between remote data centers, but also over unreliable transports, such as dial-up and satellite communications.

  You can read this article on Rob Davies blog.

8.2.3. Routing and Integration

8.2.3.1. Webinars

The resources listed here are freely available from the FuseSource website at FuseSource Resources.

- Getting Started with Apache Camel
- Database Integration with Apache Camel
- Apache Camel: From EIPs to Production
- Enterprise Integration: Patterns and Deployments
- Apache Camel Deployment Options
- The Top Twelve Integration Patterns with Apache Camel

8.2.3.2. Video demos

The resources listed here are freely available from the FuseSource website at FuseSource Video demos

- Introduction to Apache Camel
- Overview of Apache Camel
- Throttler Enterprise Integration Pattern
- Splitter Enterprise Integration Pattern
- Composed Enterprise Integration Pattern

8.2.3.3. Third party books and articles

The books and reference materials listed here provide detailed information and instructions for designing, building, and deploying enterprise integration solutions and for using the tools that make it easier to do so. The books and reference materials in this list were authored by leading experts in this domain.

- Camel in Action (Ibsen & Anstey)

  Written by the Camel developers, Camel in Action is the definitive guide to understanding and working with Apache Camel.

  Tutorial-like and full of small examples, it shows how to work with the integration patterns. Starting with core concepts (sending, receiving, routing, and transforming data), it then shows the entire life cycle—diving into testing, dealing with errors, scaling, deploying, and monitoring applications.
- Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions (Hohpe & Woolf)

  The integration patterns presented in this book provide solutions to specific design problems inherent to enterprise integration. These solutions have evolved over time through use, so that the patterns incorporate the knowledge and experience of many senior integration developers and architects. Apache Camel supports over fifty of these patterns.
- Apache Camel: Integration Nirvana (Jonathan Anstey)

  This article introduces Apache Camel concepts and architecture, and it provides a few code examples in Camel's Java DSL and Spring XML.

  You can find this article at DZone.
- The Top Twelve Integration Patterns for Apache Camel (Claus Ibsen)

  This reference card presents the twelve most popular integration patterns and provides examples for implementing them with Java code or with Spring XML.

  You can find this reference card at DZone.

8.2.4. Web and RESTful Services

The Apache CXF `samples` directory includes samples you can examine and run to more fully explore developing services using JAX-WS and JAX-RS.

8.2.4.1. Webinars

The resources listed here are freely available from the FuseSource website at FuseSource Resources.

- Deploy Web Services into ServiceMix
- How to Secure CXF Web Services with SSL/TSL and WS-Security
- Creating Web Services with Camel CXF

8.2.4.2. Third party books and articles

- Understanding Web Services: XML, WSDL, SOAP, and UDDI (Eric Newcomer)

Index

A

B

F

X