Progress FUSE™

FUSE™ ESB

Getting Started with FUSE ESB

Version 4.1
April 2009

PROGRESS SOFTWARE

Getting Started with FUSE ESB

Version 4.1

Publication date 22 Jul 2009

Table of Contents

List of Figures

List of Examples

Chapter 1. Introduction to FUSE ESB

FUSE ESB is an open, standards based integration platform. It reduces the complexity of integrating disparate applications by leveraging the service oriented architecture principles and standardized packaging frameworks.

Overview

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

Integration problems

Enterprise networks commonly deploy disparate applications, platforms, and business processes that need to communicate or exchange data with each other. The applications, platforms and processes have non-compatible data formats and non-compatible communications protocols. If an enterprise needs to interface with external systems, the integration problem extends outside of a company and encompasses its business partners' IT systems and processes as well.

In recent years, there have been several technologies attempting to solve these problems such as Enterprise Application Integration (EAI), Business-to-Business (B2B). These solutions addressed some of the integration problems, but were proprietary, expensive, and time-consuming to implement. These solutions range from expensive vendor solutions (high cost, vendor lock-in) to home-grown custom solutions (high maintenance, high cost). The overwhelming disadvantages of these solutions are high cost and low flexibility due to non-standard implementations.

Most recently, Service Oriented Architecture(SOA) has become the hot integration methodology. SOA attempts to address the shortcomings of the other approaches by advocating the use of standards and the use of loosely coupled interfaces. While, SOA, theoretically, improves the solution, it can be difficult to implement because vendors are offering tools using proprietary technologies and attempting to wrap old solutions in SOA clothing.

The ESB approach

An *Enterprise Service Bus*(ESB) is the back bone of a SOA implementation. There is no canonical definition of an ESB. Wikipedia opens its entry on ESBs (http://en.wikipedia.org/wiki/Enterprise_service_bus) by stating:

> An ESB generally provides an abstraction layer on top of an implementation of an enterprise messaging system,

> which allows integration architects to exploit the value of messaging without writing code. Contrary to the more classical enterprise application integration (EAI) approach of a monolithic stack in a hub and spoke architecture, the foundation of an enterprise service bus is built of base functions broken up into their constituent parts, with distributed deployment where needed, working in harmony as necessary.
>
> —Wikipedia

LooselyCoupled defines an ESB as follows:

> An ESB acts as a shared messaging layer for connecting applications and other services throughout an enterprise computing infrastructure. It supplements its core asynchronous messaging backbone with intelligent transformation and routing to ensure messages are passed reliably. Services participate in the ESB using either web services messaging standards or the Java Message System (JMS). Originally defined by analysts at Gartner, ESB is increasingly seen as a core component in a service-oriented infrastructure.
>
> —looselycoupled.com

The most common way of defining an ESB is by listing the services it provides. These services include:

- transport mediation - not all services that need to be integrated use HTTP or JMS
- dynamic message transformation - not all services are going to use SOAP and are unlikely to require the same message structures
- intelligent routing
- security

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 POs directly into the billing system by plugging the your PO system into the ESB.

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. These types of ESB solution is ideal for distributed computing. They also contribute to vendor lock-in.

  Ideally, an ESB solution should have flexible deployment requirements so that it can be distributed through out an enterprise.

- Container / component model

  Does the ESB solution use a standardized container model, such as J2EE, JBI, or OSGi, for managing deployed services? Or does it use a proprietary model?

  Ideally, an ESB solution should use a standardized container model. Standard models ensure maximum compatibility and lessen the learning curve needed for adoption.

- Coupling to other infrastructure components

  ESB solutions often leave out infrastructure components like orchestration engines and advanced transports like CORBA. Instead they rely on plug-ins or other components to provided the functionality.

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

  Ideally, an ESB solution would provide a loose coupling or provide a standardized interface between the ESB and any added components. This allows the ESB to be extended easily and in a flexible manner.

- 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 implementations. Others rely on standardized implementations as much as possible.

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

The FUSE ESB approach

Based on Apache ServiceMix, FUSE ESB reduces complexity and eliminates vendor lock in because it is standards based and built using best in breed open source technology. It differentiates itself in the following ways:

- The FUSE ESB kernel is lightweight and can run on most platforms.
- The FUSE ESB kernel uses the OSGi framework to simplify componentization of applications.

  The OSGi framework is a newly emerging standard for managing the dependencies between application components. It also provides a standard mechanism for controlling the life-cycle of components.
- FUSE ESB supports the Java Business Integration(JBI) specification (JSR 208).

  JBI is a well defined standard for packaging, deploying, and managing components deployed to the ESB.
- FUSE ESB can be coupled to other infrastructure services over a variety of transport protocols and message formats.

  Out of the box, the FUSE ESB supports JMS, HTTP, HTTPS, FTP, XMPP, Web services, and a number of other bindings. In addition, you can easily extend its connectivity options using other components that conform to either the OSGi or JBI specification.
- FUSE ESB employs standards as much as possible to limit dependencies.

In addition, FUSE ESB supports event driven architectures. Services deployed into the FUSE ESB container can be fully decoupled and will simply listen on the bus until an appropriate service request arrives. FUSE ESB also supports events that occur outside of the bus. For example, a JMS service can listen on a topic that is hosted outside of the bus and only act when an appropriate message arrives.

Chapter 2. Using FUSE ESB

FUSE ESB can be seen as having three main parts: a command console, a runtime container, and a standardized message bus. The command console allows the user to manage the container and the applications deployed into it. The container provides the runtime functionality required by your services. The standardized message bus gives the endpoints running in the container a means for communicating with each other.

The Runtime Container

Overview

The runtime container, based on the Apache ServiceMix kernel, is an OSGi based environment for deploying and managing bundles. It provides facilities for logging, dynamic configuration, and provisioning.

The runtime container also has a JBI adapter layer. This layer allows JBI components, shared libraries, and service assemblies to run inside the OSGi based container.

OSGi in a nutshell

OSGi is set of open specifications aimed at making it easier to build and deploy complex software applications. The key piece of OSGi technology is the OSGi Framework. The framework manages the loading and management of dynamic modules of functionality.

In an OSGi environment applications are packaged into bundles. A bundle is a jar that contains extra information about the classes and resources included in the bundle. The information supplied in the bundle includes:

- packages required by classes in the bundle
- packages being exported by the bundle
- version information for the bundle

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

The OSGi specifications are maintained by the OSGi Alliance. See http://www.osgi.org.

JBI

Java Business Integration (JBI) is a specification developed by the Java Community Process. It defines a packaging, deployment, and runtime model based on service-oriented principles. Functionality can be packaged in a number of different types of packages depending on the scope of functionality they expose. The most common packaging unit used by an application developer is the service assembly.

A service assembly consists of one or more service units. Each service unit defines an endpoint that exposes or consumes a service. Applications typically consist of multiple service units.

The endpoints in a JBI environment communicate using a normalized message bus (NMR). The NMR ensures that endpoints are loosely coupled by shielding them from the physical details of the communication process.

For more information about JBI and using JBI with FUSE ESB see *Using JBI to Develop Applications*.

Benefits of the FUSE ESB container

OSGi offers a number of benefits over other container and packaging models including:

- hot deployment of artifacts
- management of multiple versions of a package, class, or bundle
- dynamic loading of code
- lightweight footprint
- multiple packaging options

The Command Console

Overview

The command console is a shell environment that enables you to control the FUSE ESB runtime container. The console is GShell-based, and includes subshells that provide commands for specific sets of functionality. The command console can be used to control local runtime instances and to securely manage remote containers.

The *Console Reference Guide* provides information about using the console and includes descriptions of the available commands and subshells.

Features

The FUSE ESB command console provides the following features:

- **Modular** — The command console includes modular subshells that provide commands for a specific set of functionality. You can also expand the functionality by writing custom modules.

  See "FUSE ESB Console Root Commands and Subshells" in *Console Reference Guide* for information about using the subshells.

- **Artifact Management** — One of the most important uses of the command console is managing the artifacts deployed into the container. The command console provides subshells to manage artifacts, including OSGi bundles, collections of bundles, JBI artifacts, and OSGi bundle repositories (OBRs).

  See the *Console Reference Guide* for information about the FUSE ESB console subshells available for artifact management.

- **Remote Management** — You will likely have many instances of the FUSE ESB runtime distributed throughout your organization. To address this requirement, the command console includes the commands `ssh` and `sshd`, which enable you to connect to and start a remote secure shell server.

  See the "FUSE ESB Console Root Commands and Subshells" in *Console Reference Guide* for information about the console commands available for remote management.

  See "Managing Remote Instances" in *Managing the Container* for more information about remote connections.

The Standardized Message Bus

Overview

The standardized message bus is analogous to the NMR in JBI. It provides a standard interface by which all services deployed into FUSE ESB interact. It normalizes messages and ensures they are delivered to the proper locations.

Message exchange patterns

The message bus uses a WSDL-based messaging model to mediate the message exchanges between endpoints. Using a WSDL-based model provides the needed level of abstraction to ensure that the endpoints are fully decoupled. The WSDL-based model defines operations as a message exchange between a service provider and a service consumer. The message exchanges are defined from the point of view of the service provider and fit one of four message exchange patterns:

in-out

> In this pattern a consumer sends a request message to a provider. The provider responds to the request with a response message. The provider may also respond with a fault message indicating that an error occurred during processing.

in-optional-out

> In this pattern a consumer sends a request message to a provider. The provider may send a response message back to the consumer, but the consumer does not require a response. The provider may also respond with a fault message. In addition, the consumer can send a fault message to the provider.

in-only

> In this pattern a consumer sends a message to a provider, but the provider does not send a response. The provider does not even send fault messages back to the consumer.

robust-in-only

In this pattern a consumer sends a message to a provider. The provider may send a fault message back to the consumer to signal an error condition, but otherwise does not respond to the consumer.

Normalized messages

In order to fully decouple the entities involved in message exchanges the bus uses *normalized messages*. A normalized message is a genericized format used to represent all of message data passed through the bus. It consists of three parts:

meta-data, properties
: The meta-data holds information about the message. This information can include transaction contexts, security information, or other QoS information. The meta-data can also hold transport headers.

payload
: The payload of a message is an XML document that conforms to the XML Schema definition in the WSDL document defining the message exchange. The XML document holds the substance of the message.

attachments
: Attachments hold any binary data associated with the message. For example, an attachment could be an image file sent as an attachment to SOAP message.

security `Subject`
: The security `Subject` holds security information, such as authentication credentials, associated with the message. For more information of the security `Subject`, see Sun's API documentation[1].

JBI binding components automatically normalizing all of the messages placed onto the bus. Binding components normalize messages received from external sources before passing them to the NMR, The binding component will also denormalize the message so that it is in the appropriate format for the external source. Non-JBI endpoints are responsible for normalizing and denormalizing messages on their own.

[1] http://java.sun.com/j2se/1.5.0/docs/api/javax/security/auth/Subject.html

Chapter 3. Deploying a Service into FUSE ESB

You are going to build and deploy simple services using both the native FUSE Services Framework integration.

You are going to build and deploy a simple service that is based on a WSDL document. The source for this example can be found in the `*InstallDir*/examples/cxf-osgi` folder of your FUSE ESB installation.

The example uses the FUSE ESB Maven tooling to build a bundle that contains the service implementation and all of the metadata needed to deploy it into the FUSE ESB container.

The sample code includes a web page, `client.html`, that will allow you to access the service once it is exposed.

Running the Example

Before running the example

To run this example you will need to do the following:

1. Make sure that you have Maven 2.0.6 or greater installed on your system.

   You can download Maven from http://maven.apache.org/.

2. Start an instance of the FUSE ESB.

   You can start an instance of the FUSE ESB container by doing the following:

   1. Change to the root folder of your installation.
   2. Run **.\bin\servicemix.bat** or **./bin/servicemix.sh** depending on your platform.

Running the example

To build and deploy the example do the following:

1. Change to the root folder of the example.

   ```
   cd InstallDir/examples/cxf-osgi
   ```

2. Enter the following command:

   ```
   mvn install
   ```

   This command will build a bundle called `cxf-osgi-4.0.0-fuse.jar`and place it into the `target` folder of the example.

3. Copy the bundle to `InstallDir/deploy` to deploy it to the container.

Testing the example

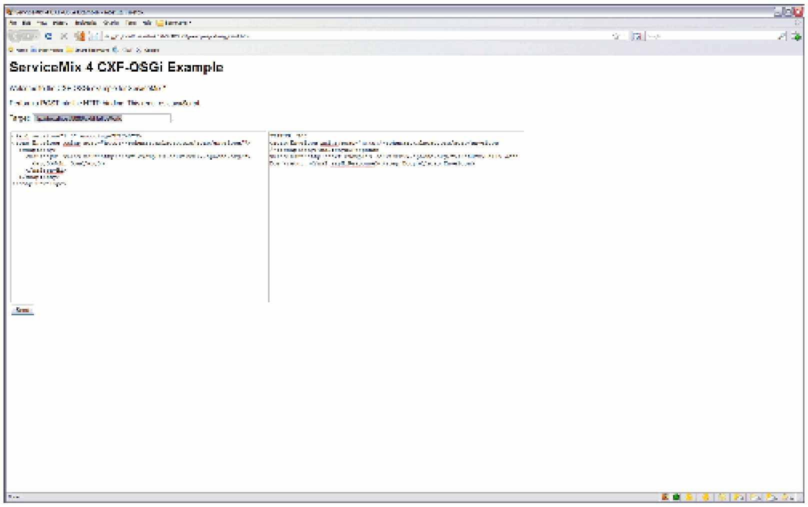
To see if the example is running you can visit http://localhost:8080/cxf/HelloWorld?wsdl in your Web browser. You should see the WSDL shown in Example 3.1 on page 21.

Example 3.1. OSGi Example WSDL

```
<wsdl:definitions name="HelloWorldImplService"
                  targetNamespace="http://cxf.examples.servicemix.apache.org/">
  <wsdl:types>
    <xs:schema attributeFormDefault="unqualified"
               elementFormDefault="unqualified"
               targetNamespace="http://cxf.examples.servicemix.apache.org/">
      <xs:complexType name="sayHi">
        <xs:sequence>
          <xs:element minOccurs="0" name="arg0" type="xs:string"/>
         </xs:sequence>
       </xs:complexType>
       <xs:complexType name="sayHiResponse">
         <xs:sequence>
           <xs:element minOccurs="0" name="return" type="xs:string"/>
         </xs:sequence>
       </xs:complexType>
       <xs:element name="sayHi" nillable="true" type="sayHi"/>
       <xs:element name="sayHiResponse" nillable="true" type="sayHiResponse"/>
     </xs:schema>
  </wsdl:types>
  <wsdl:message name="sayHiResponse">
    <wsdl:part element="tns:sayHiResponse" name="parameters">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="sayHi">
    <wsdl:part element="tns:sayHi" name="parameters">
    </wsdl:part>
  </wsdl:message>
  <wsdl:portType name="HelloWorld">
    <wsdl:operation name="sayHi">
      <wsdl:input message="tns:sayHi" name="sayHi">
      </wsdl:input>
      <wsdl:output message="tns:sayHiResponse" name="sayHiResponse">
       </wsdl:output>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="HelloWorldImplServiceSoapBinding" type="tns:HelloWorld">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
      <wsdl:operation name="sayHi">
        <soap:operation soapAction="" style="document"/>
        <wsdl:input name="sayHi">
          <soap:body use="literal"/>
        </wsdl:input>
        <wsdl:output name="sayHiResponse">
          <soap:body use="literal"/>
        </wsdl:output>
      </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="HelloWorldImplService">
    <wsdl:port binding="tns:HelloWorldImplServiceSoapBinding" name="HelloWorldImplPort">
      <soap:address location="http://localhost:8080/cxf/HelloWorld"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

You can also open `client.html` in your browser. Figure 3.1 on page 22 shows what the screen should look like.

Figure 3.1. Example HTML Screen



ServiceMix 4 CXF-OSGi Example

[illegible]

Understanding the Example

Overview

The example builds a simple HelloWorld service and packages it for deployment into FUSE ESB. The service is written using standard JAX-WS APIs. It implements a single operation `sayHi()`. Once deployed, the service is exposed as a SOAP/HTTP endpoint. The most interesting parts of the example are the Spring configuration used to configure the endpoint and the Maven POM used to build the bundle.

The Spring configuration provides the details needed to expose the service using SOAP/HTTP. It can also contain details used to configure advanced FUSE Services Framework functionality.

The Maven POM, in addition to compiling the code, uses the bundle generation plug-in to package the resulting classes into an OSGi bundle. It contains all of the details needed by the FUSE ESB container to activate the bundle and deploy the packaged service.

Using the Maven tools

The FUSE ESB Maven tooling automates a number of the steps in packaging functionality for deployment into FUSE ESB. In order to use the Maven OSGi tooling, you add the elements shown in Example 3.2 on page 23 to your POM file.

Example 3.2. POM Elements for Using FUSE ESB OSGi Tooling

```
...
<pluginRepositories>
  <pluginRepository>
     <id>fusesource.m2</id>
      <name>FUSE Open Source Community Release Repository</name>
       <url>http://repo.fusesource.com/maven2</url>
        <snapshots>
           <enabled>false</enabled>
         </snapshots>
           <releases>
              <enabled>true</enabled>
           </releases>
  </pluginRepository>
</pluginRepositories>
 ...
<build>
   <plugins>
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
         ...
       </plugin>
    </plugins>
</build>
  ...
```

These elements point Maven to the correct repositories to download the FUSE ESB Maven tooling and load the plug-in that implements the OSGi tooling.

The Spring configuration

The FUSE ESB container needs some details about a service before it can instantiate and endpoint for it. FUSE Services Framework uses Spring based configuration to define endpoints for services. The configuration shown in Example 3.3 on page 24 is stored in the example's `\src\main\resources\META-INF\spring\beans.xml` file.

Example 3.3. OSGi Example Spring Configuration

```
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:jaxws="http://cxf.apache.org/jaxws"
 xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframe
work.org/schema/beans/spring-beans.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">

    <import resource="classpath:META-INF/cxf/cxf.xml" /> ❶
    <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />
    <import resource="classpath:META-INF/cxf/cxf-extension-http.xml" />
    <import resource="classpath:META-INF/cxf/osgi/cxf-extension-osgi.xml" />

 <jaxws:endpoint id="helloWorld" ❷
                 implementor="org.apache.servicemix.examples.cxf.HelloWorldImpl"
                 address="/HelloWorld"/>

</beans>
```

The configuration shown in Example 3.3 on page 24 does the following:

❶ Imports the required configuration to load the required parts of the FUSE Services Framework runtime.

❷ Configures the endpoint that exposes the service using the `jaxws:endpoint` element and its attributes.

- `id` is an identifier used by the configuration mechanism.

- `implementor` specifies the class that implements the service. It must be on the classpath.
- `address` specifies the address at which the service will be exposed. This address is relative to the containers HTTP address with `cxf` appended to it.

For more information on the FUSE Services Framework Spring configuration see *Developing and Deploying JAX-WS Services*.

The POM

Example 3.4. OSGi POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="ht
tp://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-
v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>org.apache.servicemix.examples</groupId>
        <artifactId>examples</artifactId>
        <version>4.0.0-fuse-SNAPSHOT</version>
    </parent>

    <groupId>org.apache.servicemix.examples</groupId>
    <artifactId>cxf-osgi</artifactId>
    <packaging>bundle</packaging>
    <version>4.0.0-fuse-SNAPSHOT</version>
    <name>Apache ServiceMix Example :: CXF OSGi</name>

<!-- Add ServiceMix repositories for snaphots and releases -->
 ...

    <dependencies>
        <dependency>
            <groupId>org.apache.geronimo.specs</groupId>
            <artifactId>geronimo-ws-metadata_2.0_spec</artifactId>
            <version>${geronimo.wsmetadata.version}</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
```

```
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
                <configuration>
                    <instructions>
                        <Bundle-SymbolicName>${pom.artifactId}</Bundle-SymbolicName>
                        <Import-Package>
                            javax.jws,
                            javax.wsdl,
                            META-INF.cxf,
                            META-INF.cxf.osgi,
                            org.apache.cxf.bus,
                            org.apache.cxf.bus.spring,
                            org.apache.cxf.bus.resource,
                            org.apache.cxf.configuration.spring,
                            org.apache.cxf.resource,
                            org.apache.servicemix.cxf.transport.http_osgi,
                            org.springframework.beans.factory.config
                        </Import-Package>
                     <Private-Package>org.apache.servicemix.examples.cxf</Private-Package>

                        <Require-Bundle>org.apache.cxf.cxf-bundle</Require-Bundle>
                    </instructions>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

Index