Progress FUSE

FUSE™ ESB

Using the File Binding Component

Version 4.1
April 2009

PROGRESS SOFTWARE

Using the File Binding Component

Version 4.1

Publication date 22 Jul 2009

Table of Contents

List of Figures

List of Tables

List of Examples

Chapter 1. Introduction to the File Binding Component

The file binding component allows you to create endpoints that read files from a file system and write files out to the file system.

Overview

The file component provides integration to the file system. It can be used to read and write files via URI. It can also be configured to periodically poll directories for new files.

It allows for the creation of two types of endpoint:

poller endpoint

> A poller endpoint polls a specified location on the file system for files. When it finds a file it reads the file and sends it to the NMR for delivery to the appropriate endpoint.
>
> **Important**
>
> A poller endpoint can only create in-only message exchanges.

sender endpoint

> A sender endpoint receives messages from the NMR. It then writes the contents of the message to a specified location on the file system.

Key features

The file component has the following advanced features:

- custom filters for selecting files
- custom marshalers for converting the contents of a file to and from a normalized message
- custom locking mechanism for controlling file access during reads

- archiving of read files

Contents of a file component service unit

A service unit that configures the file binding component will contain two artifacts:

`xbean.xml`

The `xbean.xml` file contains the XML configuration for the endpoint defined by the service unit. The contents of this file are the focus of this guide.

Note

The service unit can define more than one endpoint.

`meta-inf/jbi.xml`

The `jbi.xml` file is the JBI descriptor for the service unit. Example 1.1 on page 12 shows a JBI descriptor for a file component service unit.

Example 1.1. JBI Descriptor for a File Component Service Unit

```
<jbi xmlns="http://java.sun.com/xml/ns/jbi" version="1.0">
  <services binding-component="false" />
</jbi>
```

Tip

The developer typically does not need to hand code this file. It is generated by the FUSE ESB Maven tooling.

OSGi Packaging

You can package file endpoints in an OSGi bundle. To do so you need to make two minor changes:

- you will need to include an OSGi bundle manifest in the `META-INF` folder of the bundle.
- You need to add the following to your service unit's configuration file:

```
<bean class="org.apache.servicemix.common.osgi.EndpointEx
porter" />
```

Important

When you deploy file endpoints in an OSGi bundle, the resulting endpoints are deployed as a JBI service unit.

For more information on using the OSGi packaging see Appendix D on page 65.

Namespace

The elements used to configure file endpoints are defined in the http://servicemix.apache.org/file/1.0 namespace. You will need to add a namespace declaration similar to the one in Example 1.2 on page 13 to your `xbean.xml` file's `beans` element.

Example 1.2. Namespace Declaration for Using File Endpoints

```
<beans ...
       xmlns:file="http://servicemix.apache.org/file/1.0"
       ... >
  ...
</beans>
```

In addition, you need to add the schema location to the Spring `beans` element's `xsi:schemaLocation` as shown in Example 1.3 on page 13.

Example 1.3. Schema Location for Using File Endpoints

```
<beans ...
       xsi:schemaLocation="...
http://servicemix.apache.org/file/1.0 http://service
mix.apache.org/file/1.0/servicemix-file.xsd
...">
  ...
</beans>
```

Chapter 2. Using Poller Endpoints

Poller endpoints poll the file system for files and passes the file to a target endpoint inside an in-only message exchange.

Introduction to Poller Endpoints

Overview

The function of a poller endpoint is to read data, in the form of files, from a location on a file system and pass that information to other endpoints in the ESB. They create an in-only message exchange containing the data read in from a file.

A poller endpoint, as its name implies, works by continually polling the file system to see if a file is present for consumption. The polling interval is completely customizable. By default, poller endpoints check the file system at predefined intervals.

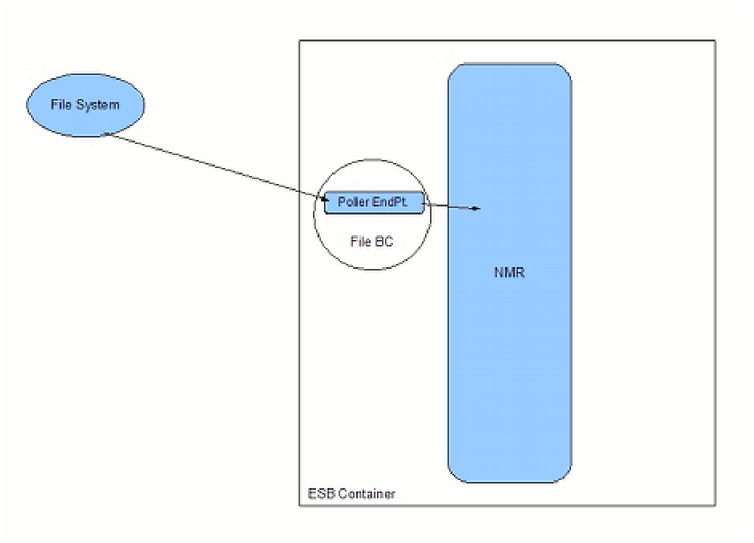
You can also control the files a poller endpoint consumes. Using the basic configuration attributes you can configure the endpoint to poll for a specific file or you can poll it to monitor a specific directory on the file system. In addition you can configure the endpoint to use a custom file filter.

By default, poller endpoints will only consume valid XML files. You can customize this behavior by configuring the endpoint to use a custom marshaler.

Where does a poller endpoint fit into a solution?

Poller endpoints play the role of consumer from the vantage point of the other endpoints in the ESB. As shown in Figure 2.1 on page 17, a poller endpoint watches the file system for files to consume. When it consumes a file, it transfers its contents into a message and starts off an in-only message exchange. Poller endpoints cannot receive messages from the NMR.

Figure 2.1. Poller Endpoint



Configuration element

Poller endpoints are configured using the `poller` element. All its configuration can be specified using attributes of this element.

The more complex features, such as custom marshalers, require the addition of other elements. These can either be separate `bean` elements or child elements of the `poller` element.

Basic Configuration

Overview

The basic requirements for configuring a poller endpoint are straightforward. You need to supply the following information:

- the endpoint's name
- the endpoint's service name
- the file or directory to be monitored
- the ESB endpoint to which the resulting messages will be sent

All of this information is provided using attributes of the `poller` element.

Identifying the endpoint

All endpoints in the ESB need to have a unique identity. An endpoint's identity is made up of two pieces:

- a service name
- an endpoint name

Table 2.1 on page 18 describes the attributes used to identify a poller endpoint.

Table 2.1. Attributes for Identifying a Poller Endpoint

Name	Description
`service`	Specifies the service name of the endpoint. This value must be a valid QName and does not need to be unique across the ESB.
`endpoint`	Specifies the name of the endpoint. This value is a simple string. It must be unique among all of the endpoints associated with a given service name.

Specifying the message source

You specify the location the poller endpoint looks for new messages using the `poller` element's `file` attribute. This attribute takes a URI that identifies a location on the file system.

If you want the endpoint to poll a specific file you use the standard `file:`*`location`* URI. If you do not use the `file` prefix, the endpoint will assume the URI specifies a directory on the file system and will consume all valid XML files placed in the specified directory.

For example, the URI `file:inbox` tells the endpoint to poll for a file called `inbox`. The URI `inbox` instructs the endpoint to poll the directory `inbox`.

Important

Relative URIs are resolved from the directory in which the FUSE ESB container was started.

Specifying the target endpoint

There are a number of attributes available for configuring the endpoint to which the generated messages are sent. The poller endpoint will determine the target endpoint in the following manner:

1. If you explicitly specify an endpoint using both the `targetService` attribute and the `targetEndpoint` attribute, the ESB will use that endpoint.

   The `targetService` attribute specifies the QName of a service deployed into the ESB. The `targetEndpoint` attribute specifies the name of an endpoint deployed by the service specified by the `targetService` attribute.

2. If you only specify a value for the `targetService` attribute, the ESB will attempt to find an appropriate endpoint on the specified service.

3. If you do not specify a service name or an endpoint name, you must specify an the name of an interface that can accept the message using the `targetInterface` attribute. The ESB will attempt to locate an endpoint that implements the specified interface and direct the messages to it.

   Interface names are specified as QNames. They correspond to the value of the `name` attribute of either a WSDL 1.1 `serviceType` element or a WSDL 2.0 `interface` element.

Important

If you specify values for more than one of the target attributes, the poller endpoint will use the most specific information.

Example

Example 2.1 on page 20 shows the configuration for a simple poller endpoint.

Example 2.1. Simple Poller Endpoint

```
<beans xmlns:file="http://servicemix.apache.org/file/1.0"
       xmlns:foo="http://servicemix.org/demo/">

  <file:poller service="foo:filePoller"
               endpoint="filePoller"
               targetService="foo:fileSender"
               file="file:inbox/test.xml" />
  ...
</beans>
```

Configuring How Poller Endpoints Interact with the File System

Overview

Poller endpoints interact with the file system in basic ways. You can configure a number of the aspects of this behavior including:

- if the endpoint creates the directory it is configured to poll
- if the endpoint polls the subdirectories of the configured directory
- if the endpoint deletes the files it consumes
- where the endpoint archives copies of the consumed files

Directory handling

The default behavior of a poller endpoint that is configured to poll a directory on the file system is to create the directory if it does not exist and to poll all of that directory's subdirectories. You can configure an endpoint to not do one or both of these behaviors.

To configure an endpoint to not create the configured directory you set its `autoCreateDirectory` attribute to `false`. If the directory does not exist, the endpoint will do nothing. You will have to create the directory manually.

To configure the endpoint to only poll the configured directory and ignore its subdirectories you set the endpoint's `recursive` attribute to `false`.

Example 2.2 on page 21 shows the configuration for a poller endpoint that does not recurse into the subdirectories of the directory it polls.

Example 2.2. Poller Endpoint that Does Not Check Subdirectories

```
<beans xmlns:file="http://servicemix.apache.org/file/1.0"
       xmlns:foo="http://servicemix.org/demo/">

  <file:poller service="foo:filePoller"
               endpoint="filePoller"
               targetService="foo:fileSender"
               file="inbox"
               recursive="false" />
  ...
</beans>
```

File retention

By default poller endpoints delete a file once it is consumed. To configure the endpoint to leave the file in place after is consumed, set its `deleteFile` attribute to `false`.

Example 2.3 on page 22 shows the configuration for a poller endpoint that does not delete files.

Example 2.3. Poller Endpoint that Leaves Files Behind

```
<beans xmlns:file="http://servicemix.apache.org/file/1.0"
       xmlns:foo="http://servicemix.org/demo/">

  <file:poller service="foo:filePoller"
               endpoint="filePoller"
               targetService="foo:fileSender"
               file="inbox"
               deleteFile="false" />
  ...
</beans>
```

Important

When the poller endpoint does not automatically delete consumed files the list of consumed files is stored in memory. If the FUSE ESB container is stopped and restarted files that have been consumed, but not removed from the polling folder, will be reprocessed. One possible solution is to use a custom lock manager that stores a list of the consumed files to an external data store.

Archiving files

By default, poller endpoints do not archive files after they are consumed. If you want the files consumed by a poller endpoint to be archived you set the endpoint's `archive` attribute. The value of the `archive` attribute is a URI pointing to the directory into which the consumed files will archived.

Important

Relative URIs are resolved from the directory in which the FUSE ESB container was started.

Example 2.4 on page 23 shows the configuration for a poller endpoint that files into a directory called `archives`.

Example 2.4. Poller Endpoint that Archives Files

```
<beans xmlns:file="http://servicemix.apache.org/file/1.0"
       xmlns:foo="http://servicemix.org/demo/">

  <file:poller service="foo:filePoller"
               endpoint="filePoller"
               targetService="foo:fileSender"
               file="inbox"
               archive="archives" />
  ...
</beans>
```

Configuring the Polling Interval

Overview

A default poller endpoint provides limited scheduling facilities. You can configure when the endpoint starts polling and the interval between polling attempts.

Scheduling the first poll

By default, poller endpoints begin polling as soon as they are started. You can control when a poller endpoint first attempts to poll the file system using an attribute that controls the date of the first polling attempt.

You specify a date for the first poll using the endpoint's `firstTime` attribute. The `firstTime` attribute specifies a date using the standard xsd:date format of `YYYY-MM-DD`. For example, you would specify April 1, 2025 as `2025-04-01`. The first polling attempt will be made at 00:00:00 GMT on the specified date.

Note

If you schedule the first polling attempt in the past, the endpoint will begin polling immediately.

Example 2.5 on page 24 shows the configuration for a poller endpoint that starts polling at 1am GMT on April 1, 2010.

Example 2.5. Poller Endpoint with a Scheduled Start Time

```
<beans xmlns:file="http://servicemix.apache.org/file/1.0"
       xmlns:foo="http://servicemix.org/demo/">

  <file:poller service="foo:filePoller"
               endpoint="filePoller"
               targetService="foo:fileSender"
               file="inbox"
               firstTime="2010-04-01" />
  ...
</beans>
```

Delaying the first poll

In addition to controlling the specific data one which polling will start, you can also specify how long to delay the first polling attempt. The delay is specified using the endpoint's `delay` attribute. The `delay` attribute specifies the delay in milliseconds.

Note

If you have specified a date for the first polling attempt, the delay will be added to the date to determine when to make the first polling attempt.

Example 2.6 on page 25 shows the configuration for a poller endpoint that begins polling 5 minutes after it is started.

Example 2.6. Poller Endpoint with a Delayed Start Time

```
<beans xmlns:file="http://servicemix.apache.org/file/1.0"
        xmlns:foo="http://servicemix.org/demo/">

  <file:poller service="foo:filePoller"
               endpoint="filePoller"
               targetService="foo:fileSender"
               file="inbox"
               delay="300000" />
  ...
</beans>
```

Configuring the polling interval

By default, poller endpoints poll the file system every five seconds. You can configure the polling interval by providing a value for the endpoint's `period` attribute. The `period` attribute specifies the number of milliseconds the endpoint waits between polling attempts.

Example 2.7 on page 25 shows the configuration for a poller endpoint that uses a thirty second polling interval.

Example 2.7. Poller Endpoint with a Thirty Second Polling Interval

```
<beans xmlns:file="http://servicemix.apache.org/file/1.0"
        xmlns:foo="http://servicemix.org/demo/">

  <file:poller service="foo:filePoller"
               endpoint="filePoller"
               targetService="foo:fileSender"
               file="inbox"
               period="30000" />
  ...
</beans>
```

File Locking

Overview

It is possible to have multiple instances of a poller endpoint attempting to read a file on the system. To ensure that there are no conflicts in accessing the file, poller endpoints obtain an exclusive lock on a file while it is processing it.

The locking behavior is controlled by an implementation of the `org.apache.servicemix.common.locks.LockManager` interface. By default, poller endpoints use a provided implementation of this interface. If the default behavior is not appropriate for your application, you can implement the `LockManager` interface and configure your endpoints to use your implementation.

Implementing a lock manager

To implement a custom lock manager you need to provide your own implementation of the `org.apache.servicemix.common.locks.LockManager` interface. The `LockManager` has single method, `getLock()` that needs to be implemented. Example 2.8 on page 26 shows the signature for `getLock()`.

Example 2.8. The Lock Manager's Get Lock Method

```
Lock getLock(String id);
```

The `getLock()` method takes a string that represents the URI of the file being processes and it returns a `java.util.concurrent.locks.Lock` object. The returned `Lock` object holds the lock for the specified file.

Example 2.9 on page 26 shows a simple lock manager implementation.

Example 2.9. Simple Lock Manager Implementation

```
package org.apache.servicemix.demo;

import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentMap;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

import org.apache.servicemix.common.locks.LockManager;

public class myLockManager implements LockManager
{
  private ConcurrentMap<String, Lock> locks = new ConcurrentHashMap<String, Lock>();
  public Lock getLock(String id)
  {
    Lock lock = locks.get(id);
    if (lock == null)
    {
      lock = new ReentrantLock();
      Lock oldLock = locks.putIfAbsent(id, lock);
      if (oldLock != null)
      {
        lock = oldLock;
      }
    }
    return lock;
  }

}
```

Configuring the endpoint to use a lock manager

You configure a poller endpoint to use a custom lock manager using its `lockManager` attribute. The `lockManager` attribute's value is a reference to a `bean` element specifying the class of your custom lock manager implementation.

Example 2.10 on page 27 shows configuration for a poller endpoint that uses a custom lock manager.

Example 2.10. Poller Endpoint Using a Custom Lock Manager

```
<beans xmlns:file="http://servicemix.apache.org/file/1.0"
        xmlns:foo="http://servicemix.org/demo/">

  <file:poller service="foo:filePoller"
               endpoint="filePoller"
               targetService="foo:fileSender"
               file="inbox"
               lockManager="#myLockManager" />

  <bean id="myLockManager" class="org.apache.servicemix.demo.myLockManager" />
  ...
</beans>
```

Note

You can also configure a poller endpoint to use a custom lock manager by adding a child `lockManager` element to the endpoint's configuration. The `lockManager` element simply wraps the `bean` element that configures the lock manager.

File Filtering

Overview

When a poller endpoint is configured to poll a directory it will attempt to consume any file placed into that directory. If you want to limit the files a poller endpoint will attempt to consume, you can configure the endpoint to filter files based on their names. To do so, you must supply the endpoint with an implementation of the `java.io.FileFilter` interface.

There are several file filter implementation available in open source including the Apache Commons IO implementations and the Apache Jakarta-ORO implementations. You can also implement your own file filter if you need specific filtering capabilities.

Implementing a file filter

To implement a file filter you need to provide and implementation of the `java.io.FileFilter` interface. The `FileFilter` interface has a single method, `accept()` that needs to be implemented. Example 2.11 on page 29 shows the signature of the `accept()` method.

Example 2.11. File Filter's Accept Method

```
public boolean accept()(java.io.File pathname);
```

The `accept()` method takes a `File` object that represents the file being checked against the filter. If the file passes the filter, the `accept()` method should return `true`. If the file does not pass, then the method should return `false`.

Example 2.12 on page 29 shows a file filter implementation that matches against a string passed into its constructor.

Example 2.12. Simple File Filter Implementation

```
package org.apache.servicemix.demo;

import java.io.File;
import java.io.FileFilter;

public class myFileFilter implements FileFilter
{
  String filtername = "joe.xml";

  public myFileFilter()
  {
  }
  public myFileFilter(String filtername)
  {
    this.filtername = filtername;
  }

  public boolean accept(File file)
  {
    String name = file.getName();
    return name.equals(this.filtername);
  }
}
```

Configuring an endpoint to use a file filter

You configure a poller endpoint to use a file filter using its `filter` attribute. The `filter` attribute's value is a reference to a `bean` element specifying the class of the file filter implementation.

Example 2.13 on page 30 shows configuration for a poller endpoint that uses the file filter implemented in Example 2.11 on page 29. The `constructor-arg` element sets the filter's fitlername by passing a value into the constructor.

Example 2.13. Poller Endpoint Using a File Filter

```
<beans xmlns:file="http://servicemix.apache.org/file/1.0"
        xmlns:foo="http://servicemix.org/demo/">

  <file:poller service="foo:filePoller"
               endpoint="filePoller"
               targetService="foo:fileSender"
               file="inbox"
               filter="#myFilter" />

  <bean id="myFilter" class="org.apache.servicemix.demo.myFileFilter">
    <constructor-arg value="joefred.xml" />
  </bean>
  ...
</beans>
```

Note

You can also configure a poller endpoint to use a file filter by adding a child `filter` element to the endpoint's configuration. The `filter` element simply wraps the `bean` element that configures the file filter.

Chapter 3. Using Sender Endpoints

Sender endpoints write messages to the file system.

Introduction to Sender Endpoints

Overview

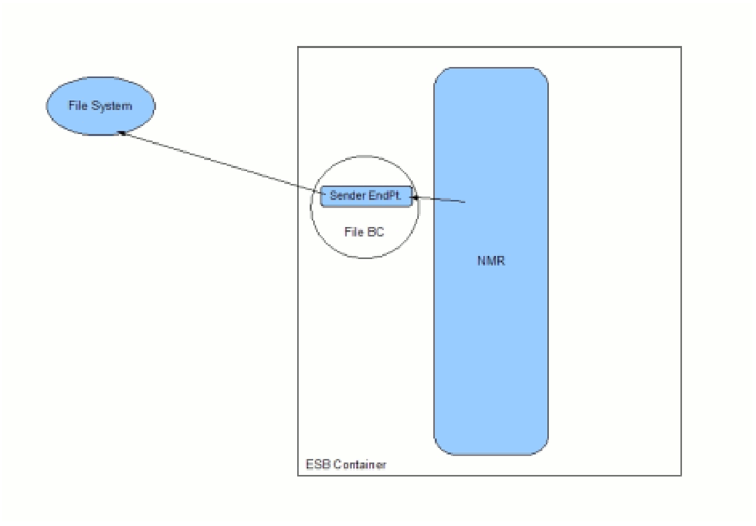
The function of a sender endpoint is to write data, in the form of files, to a location on a file system. You can control the location of the files written to the file system and have some control over the name of the generated files. You can also control if data is appended to existing files or if new copies of a file are created.

By default, sender endpoints write XML data to the file system. You can change this behavior by configuring the endpoint to use a custom marshaler.

Where does a sender endpoint fit into a solution?

Sender endpoints play the role of provider from the vantage point of the other endpoints in the ESB. As shown in Figure 3.1 on page 32, a sender endpoint receives messages from the NMR and writes the message data to the file system.

Figure 3.1. Sender Endpoint



Configuration element

Sender endpoints are configured using the `sender` element. All its configuration can be specified using attributes of this element.

Configuring a sender endpoint to use custom marshalers require the addition of other elements. These can either be separate `bean` elements or child elements of the `sender` element.

Basic Configuration

Overview

The basic requirements for configuring a sender endpoint are straightforward. You need to supply the following information:

- the endpoint's name
- the endpoint's service name
- the file or directory to which files are written

All of this information is provided using attributes of the `sender` element.

Identifying the endpoint

All endpoints in the ESB need to have a unique identity. An endpoint's identity is made up of two pieces:

- a service name
- an endpoint name

Table 3.1 on page 34 describes the attributes used to identify a sender endpoint.

Table 3.1. Attributes for Identifying a Sender Endpoint

Name	Description
`service`	Specifies the service name of the endpoint. This value must be a valid QName and does not need to be unique across the ESB.
`endpoint`	Specifies the name of the endpoint. This value is a simple string. It must be unique among all of the endpoints associated with a given service name.

Specifying the file destination

You specify the location the sender endpoint writes files using the `sender` element's `directory` attribute. This attribute takes a URI that identifies a location on the file system.

Important

Relative URIs are resolved from the directory in which the FUSE ESB container was started.

Using the default marshaler, the name of the file is determined by the org.apache.servicemix.file.name property. This property is set on either the message exchange or the message by the endpoint originating the message exchange.

Important

The marshaler is responsible for determining the name of the file being written. For more information on marshalers see "File Marshalers" on page 39.

Example

Example 3.1 on page 35 shows the configuration for a simple sender endpoint.

Example 3.1. Simple Sender Endpoint

```
<beans xmlns:file="http://servicemix.apache.org/file/1.0"
        xmlns:foo="http://servicemix.org/demo/">

  <file:sender service="foo:fileSender"
               endpoint="sender"
               directory="outbox" />
  ...
</beans>
```

Configuring How Sender Endpoints Interact with the File System

Overview

Sender endpoints interact with the file system in basic ways. You can configure a number of the aspects of this behavior including:

- if the endpoint creates the directory where it writes files
- how the endpoint names temporary files

Directory creation

The default behavior of a sender endpoint that is to automatically create the target directory for its files if that directory does not already exist. To configure an endpoint to not create the target directory you set its `autoCreateDirectory` attribute to `false`. If the directory does not exist, the endpoint will do nothing. You will have to create the directory manually.

Example 3.2 on page 36 shows the configuration for a sender endpoint that does not automatically create its target directory.

Example 3.2. Sender Endpoint that Does Create Its Target Directory

```
<beans xmlns:file="http://servicemix.apache.org/file/1.0"
        xmlns:foo="http://servicemix.org/demo/">

  <file:sender service="foo:fileSender"
               endpoint="fileSender"
               directory="outbox"
               autoCreateDirectory="false" />
  ...
</beans>
```

Appending data

By default, sender endpoints overwrite existing files. If a message wants to reuse the name of an existing file, the file on the file system is overwritten. You can configure a sender endpoint to append the message to the existing file by setting the endpoint's `append` attribute to `true`.

Example 3.3 on page 37 shows the configuration for an endpoint that appends messages to a file if it already exists.

Example 3.3. Sender Endpoint that Appends Existing Files

```
<beans xmlns:file="http://servicemix.apache.org/file/1.0"
        xmlns:foo="http://servicemix.org/demo/">

  <file:sender service="foo:fileSender"
               endpoint="fileSender"
               directory="outbox"
               append="true" />
  ...
</beans>
```

Temporary file naming

By default, sender endpoints check the message exchange, or the message itself, for the name to use for the file being written. If the endpoint cannot determine a name for the target file, it will use a temporary file name. Table 3.2 on page 37 describes the attributes used to generate the temporary file name.

Note

Checking for the name of the file to write is handled by the marshaler. For more information on marshalers see "File Marshalers" on page 39.

Table 3.2. Attributes Used to Determine a Temporary File Name

Name	Description	Default
`tempFilePrefix`	Specifies the prefix used when creating output files.	`servicemix-`
`tempFileSuffix`	Specifies the file extension to use when creating output files.	`.xml`

The generated file names will have the form *`tempFilePrefixXXXXXtempFileSuffix`*. The five Xs in the middle of the filename will be filled with randomly generated characters. So given the configuration shown in Example 3.4 on page 37, a possible temporary filename would be `widgets-xy60s.xml`.

Example 3.4. Configuring a Sender Endpoint's Temporary File Prefix

```
<beans xmlns:file="http://servicemix.apache.org/file/1.0"
        xmlns:foo="http://servicemix.org/demo/">

  <file:sender service="foo:fileSender"
               endpoint="fileSender"
               directory="outbox"
               tempFilePrefix="widgets-" />
  ...
</beans>
```

Chapter 4. File Marshalers

When using file component endpoints, you may want to customize how messages are processed as they pass in and out of the ESB. The FUSE ESB file binding component allows you to write custom marshalers for your file component endpoints.

Overview

File component endpoints use a marshaler for processing messages. Poller endpoints rely on the marshaler for reading data off of the file system and normalizing it so it can be passed to the NMR. Sender endpoints rely on the marshaler for determining the name of the file to be written and for converting the normalized messages into the format to be written to the file system.

The default marshaler used by file component endpoints reads and writes valid XML files. It queries the message exchange, and the message, received from the NMR for the name of the outgoing message. The default marshaler expects the file name to be stored in a property called org.apache.servicemix.file.name.

If your application requires different functionality from the marshaler, you can provide a custom marshaler by implementing the `org.apache.servicemix.components.util.FileMarshaler` interface. You can easily configure your endpoints to use your custom marshaler instead of the default one.

Provided file marshalers

In addition to the default file marshaler, FUSE ESB provides two other file marshalers that file component endpoints can use:

Binary File Marshaler

> The binary file marshaler is provided by the class `org.apache.servicemix.components.util.BinaryFileMarshaler`. It reads in binary data and adds it to the normalized message as an attachment. You can set the name of the attachment and specify a content type for the attachment using the properties shown in Table 4.1 on page 39.

Table 4.1. Properties for Configuring the Binary File Marshaler

Name	Description	Default
attachment	Specifies the name of the attachment added to the normalized message.	`content`

Name	Description	Default
contentType	Specifies the content type of the binary data being used. Content types are specified using MIME types. MIME types are specified by RFC 2045[1].	

Flat File Marshaler

The flat file marshaler is provided by the class `org.apache.servicemix.components.util.SimpleFlatFileMarshaler`. It reads in flat text files and converts them into XML messages.

By default the file is wrapped in a `File` element. Each line in the file is wrapped in a `Line` attribute. Each line also has a `number` attribute that represents the position of the line in the original file.

You can control some aspects of the generated XML file using the properties described in Table 4.2 on page 40.

Table 4.2. Properties Used to Control the Flat File Marshaler

Name	Description	Default
docElementname	Specifies the name of the root element generated by a file.	`File`
lineElementname	Specifies the name of the element generated for each line of the file.	`Line`
insertLineNumbers	Specifies if the elements corresponding to a line will use the `number` attribute.	`true`

Implementing a file marshaler

To develop a custom file marshaler you need to implement the `org.apache.servicemix.components.util.FileMarshaler` interface. Example 4.1 on page 40 shows the interface.

Example 4.1. The File Marshaler Interface

```
package org.apache.servicemix.components.util;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

import javax.jbi.JBIException;
import javax.jbi.messaging.MessageExchange;
import javax.jbi.messaging.MessagingException;
```

[1] http://tools.ietf.org/html/rfc2045

```
import javax.jbi.messaging.NormalizedMessage;

public interface FileMarshaler
{
  void readMessage(MessageExchange exchange, NormalizedMessage message, InputStream in,
String path) throws IOException, JBIException;

  String getOutputName(MessageExchange exchange, NormalizedMessage message) throws Mes
sagingException;

  void writeMessage(MessageExchange exchange, NormalizedMessage message, OutputStream out,
 String path) throws IOException, JBIException;
}
```

The `FileMarshaler` interface has three methods that need to be implemented:

`readMessage()`

The `readMessage()` method is responsible for reading a file from the file system and converting the data into a normalized message. Table 4.3 on page 41 describes the parameters used by the method.

Table 4.3. Parameters for Reading Messages from the File System

Name	Description
exchange	Contains the `MessageExchange` object that is going to be passed to the NMR.
message	Contains the `NormalizedMessage` object that is going to be passed to the NMR.
in	Contains the `BufferedInputStream` which points to the file in the file system.
path	Contains the full path to the file on the file system as determined by the Java `getCanonicalPath()` method.

`getOutputName()`

The `getOutputName()` method returns the name of the file to be written to the file system. The message exchange and the message received by the sender endpoint are passed to the method.

Important

The returned file name does not contain a directory path. The sender endpoint uses the directory it was configured to use.

`writeMessage()`

The `writeMessage()` method is responsible for writing messages received from the NMR to the file system as files. Table 4.4 on page 42 describes the parameters used by the method.

Table 4.4. Parameters for Writing Messages to the File System

Name	Description
`exchange`	Contains the `MessageExchange` object received from the ESB.
`message`	Contains the `NormalizedMessage` object received from the ESB.
`out`	Contains the `BufferedOutputStream` which points to the file in the file system.
`path`	Contains the path to the file are returned from the `getOutputName()` method.

Example 4.2 on page 42 shows a simple file mashaler.

Example 4.2. Simple File Marshaler

```
package org.apache.servicemix.demos;

import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.io.ObjectOutputStream;
import java.io.OutputStream;

import javax.jbi.JBIException;
import javax.jbi.messaging.MessageExchange;
import javax.jbi.messaging.MessagingException;
import javax.jbi.messaging.NormalizedMessage;

public class myFileMarshaler implements FileMarshaler
{

  public void readMessage(MessageExchange exchange, NormalizedMessage message,
                                                  InputStream in, String path)
  throws IOException, JBIException
  {
    message.setContent(new StreamSource(in, path));
  }

  public String getOutputName(MessageExchange exchange, NormalizedMessage message)
  throws MessagingException
  {
    return "fred.xml";
  }

  public void writeMessage(MessageExchange exchange, NormalizedMessage message,
                                                OutputStream out, String path)
  throws IOException, JBIException
  {
    Source src = message.getContent();
    if (src == null)
    {
      throw new NoMessageContentAvailableException(exchange);
    }
    try
    {
      ObjectOutputStream objectOut = new ObjectOutputStream(out);
      objectOut.writeObject(src);
    }
  }
}
```

Configuring an endpoint to use a file marshaler

You configure a file component endpoint to use a file marshaler using its `marshaler` attribute. The `marshaler` attribute's value is a reference to a `bean` element specifying the class of the file filter implementation.

Example 4.3 on page 43 shows configuration for a poller endpoint that uses the file marshaler implemented in Example 4.2 on page 42.

Example 4.3. Poller Endpoint Using a File Marshaler

```
<beans xmlns:file="http://servicemix.apache.org/file/1.0"
       xmlns:foo="http://servicemix.org/demo/">

  <file:poller service="foo:filePoller"
               endpoint="filePoller"
               targetService="foo:fileSender"
               file="inbox"
               marshaler="#myMarshaler" />

  <bean id="myMarshaler" class="org.apache.servicemix.demo.myFileMarshaler" />
```

Note

You can also configure a file component endpoint to use a file marshaler by adding a child `marshaler` element to the endpoint's configuration. The `marshaler` element simply wraps the `bean` element that configures the file marshaler.

Appendix A. Poller Endpoint Properties

Attributes

Table A.1 on page 45 describes the attributes used to configure a poller endpoint.

Table A.1. Attributes for Configuring a Poller Endpoint

Name	Type	Description	Default
`service`	QName	Specifies the service name of the endpoint.	*required*
`endpoint`	String	Specifies the name of the endpoint.	*required*
`interfaceName`	QName	Specifies the interface name of the endpoint.	
`targetService`	QName	Specifies the service name of the target endpoint.	
`targetEndPoint`	String	Specifies the name of the target endpoint.	
`targetInterface`	QName	Specifies the interface name of the target endpoint.	
`targetUri`	string	Specifies the URI of the target endpoint.	
`autoCreateDirectory`	boolean	Specifies if the endpoint will create the target directory if it does not exist.	`true`
`firstTime`	date	Specifies the date and the time the first poll will take place.	null (The first poll will happen right after start up.)
`delay`	long	Specifies amount of time, in milliseconds, to wait before performing the first poll.	`0`
`period`	long	Specifies the amount of time, in milliseconds, between polls.	`5000`
`file`	String	Speficies the file or directory to poll.	*required*
`deleteFile`	boolean	Specifies if the file is deleted after it is processed.	`true`
`recursive`	boolean	Specifies if the endpoint processes sub directories when polling.	`true`

Name	Type	Description	Default
`archive`	string	Specifies the name of the directory to archive files into before deleting them.	null (no archiving)

Beans

Table A.2 on page 46 describes the beans which can be used to configure a poller endpoint.

Table A.2. Beans for Configuring a Poller Endpoint

Name	Type	Description	Default
`marshaler`	`org.apache.servicemix.components.util.FileMarshaler`	Specifies the class used to marshal data from the file.	`DefaultFileMarshaler`
`lockManager`	`org.apache.servicemix.locks.LockManager`	Specifies the class implementing the file locking.	`SimpleLockManager`
`filter`	`java.io.FileFilter`	Specifies the class implementing the filtering logic to use for selecting files.	

Appendix B. Sender Endpoint Properties

Attributes

Table B.1 on page 47 describes the attributes used to configure a sender endpoint.

Table B.1. Attributes for Configuring a Sender Endpoint

Name	Type	Description	Default
`service`	QName	Specifies the service name of the endpoint.	*required*
`endpoint`	String	Specifies the name of the endpoint.	*required*
`directory`	String	Specifies the name of the directory into which data is written.	*required*
`autoCreateDirectory`	boolean	Specifies if the endpoint creates the output directory if it does not exist.	`true`
`append`	boolean	Specifies if the data is appended to the end of an existing file or if the data is written to a new file.	`false`
`tempFilePrefix`	String	Specifies the prefix used when creating output files.	`servicemix-`
`tempFileSuffix`	String	Specifies the file extension to use when creating output files.	`.xml`

Beans

Table B.2 on page 47 describes the beans used to configure a sender endpoint.

Table B.2. Attributes for Configuring a Sender Endpoint

Name	Type	Description	Default
`marshaler`	`org.apache.servicemix.components.util.FileMarshaler`	Specifies the marshaler to use when writing data from the NMR to the file system.	`DefaultFileMarshaler`

Appendix C. Using the Maven JBI Tooling

Packaging application components so that they conform the JBI specification is a cumbersome job. FUSE ESB includes tooling that automates the process of packaging you applications and creating the required JBI descriptors.

FUSE ESB provides a Maven plug-in and a number of Maven archetypes that make developing, packaging, and deploying JBI artifacts easier. The tooling provides you with a number of benefits. These benefits include:

- automatic generation of JBI descriptors
- dependency checking

Because FUSE ESB only allows you to deploy service assemblies, you will need to do the following when using the Maven JBI tooling:

1. Set up a top-level project on page 50 to build all of the service units and the final service assembly.
2. Create a project for each of your service units. on page 55.
3. Create a project for the service assembly on page 61.

Setting Up a FUSE ESB JBI Project

Overview

When working with the FUSE ESB JBI Maven tooling, you will want to create a top-level project that can build all of the service units and package them into a service assembly. Using a top-level project for this purpose has several advantages. It allows you to control the dependencies for all of the parts of an application in a central location. It limits the number of times you need to specify the proper repositories to load. It also gives you a central location from which to build and deploy the application.

The top-level project is responsible for assembling the application. It will use the Maven assembly plug-in and list your service units and the service assembly as modules of the project.

Directory structure

Your top-level project will contain the following directories:

- a source directory containing the information needed by the Maven assembly plug-in
- a directory to hold the service assembly project
- at least one directory containing a service unit project

Tip

You will need a project folder for each service unit that is to be included in the generated service assembly.

Setting up the Maven tools

In order to use the FUSE ESB JBI Maven tooling, you add the elements shown in Example C.1 on page 50 to your top-level POM file.

Example C.1. POM Elements for Using FUSE ESB 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>
<repositories>
  <repository>
    <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>
  </repository>
  <repository>
    <id>fusesource.m2-snapshot</id>
    <name>FUSE Open Source Community Snapshot Repository</name>
    <url>http://repo.fusesource.com/maven2-snapshot</url>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
    <releases>
      <enabled>false</enabled>
    </releases>
  </repository>
</repositories>
  ...
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.servicemix.tooling</groupId>
      <artifactId>jbi-maven-plugin</artifactId>
      <version>servicemix-version</version>
      <extensions>true</extensions>
    </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 tooling.

Listing the subprojects

Your top-level POM lists all of the service units and the service assembly that will be generated as modules. The modules are contained in a `modules` element. The `modules` element contains one `module` element for each service unit in the assembly. You will also need a `module` element for the service assembly.

The modules should be listed in the order in which they are built. This means that the service assembly module should be listed after all of the service unit modules.

Example JBI Project POM

Example C.2 on page 52 shows a top-level pom for a project that contains a single service unit.

Example C.2. Top-Level POM for a FUSE ESB JBI Project

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://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>com.widgets</groupId>
    <artifactId>demos</artifactId>
    <version>1.0</version>
  </parent>

  <groupId>com.widgets.demo</groupId>
  <artifactId>cxf-wsdl-first</artifactId>
  <name>CXF WSDL Fisrt Demo</name>
  <packaging>pom</packaging>

  <pluginRepositories> ❶
    <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>
  <repositories>
    <repository>
      <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>
    </repository>
    <repository>
      <id>fusesource.m2-snapshot</id>
      <name>FUSE Open Source Community Snapshot Repository</name>
      <url>http://repo.fusesource.com/maven2-snapshot</url>
      <snapshots>
        <enabled>true</enabled>
      </snapshots>
      <releases>
        <enabled>false</enabled>
      </releases>
    </repository>
  </repositories>

  <modules> ❷
    <module>wsdl-first-cxfse-su</module>
     <module>wsdl-first-cxf-sa</module>
  </modules>

  <build>
    <plugins>
      <plugin> ❸
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-assembly-plugin</artifactId>
         <version>2.1</version>
         <inherited>false</inherited>
           <executions>
             <execution>
                <id>src</id>
                <phase>package</phase>
                <goals>
                  <goal>single</goal>
                </goals>
                <configuration>
                  <descriptors>
                    <descriptor>src/main/assembly/src.xml</descriptor>
                  </descriptors>
                 </configuration>
               </execution>
             </executions>
           </plugin>
           <plugin> ❹
```

```
            <groupId>org.apache.servicemix.tooling</groupId>
            <artifactId>jbi-maven-plugin</artifactId>
            <extensions>true</extensions>
          </plugin>
    </plugins>
  </build>
</project>
```

The POM shown in Example C.2 on page 52 does the following:

❶ Configures Maven to use the FUSE repositories for loading the FUSE ESB plug-ins.

❷ Lists the sub-projects used for this application. The `wsdl-first-cxfse-su` module is the module for the service unit. The `wsdl-first-cxf-sa` module is the module for the service assembly

❸ Configures the Maven assembly plug-in.

❹ Loads the FUSE ESB JBI plug-in.

A Service Unit Project

Overview

Each service unit in the service assembly needs to be its own project. These projects are placed at the same level as the service assembly project. The contents of a service unit's project depends on the component at which the service unit is targeted. At a minimum, a service unit project will contain a POM and an XML configuration file.

Seeding a project using a Maven artifact

FUSE ESB provides Maven artifacts for a number of service unit types. You can use them to seed a project with the **smx-arch** command. As shown in Example C.3 on page 55, the **smx-arch** command takes three arguments. The `groupId` value and the `artifactId` values correspond to the project's group ID and artifact ID.

Example C.3. Maven Archetype Command for Service Units

```
smx-arch su suArchetypeName ["-DgroupId=my.group.id"]
["-DartifactId=my.artifact.id"]
```

Important

The double quotes(`"`) are required when using the `-DgroupId` argument and the `-DartifactId` argument.

The *`suArchetypeName`* specifies the type of service unit to seed. Table C.1 on page 55 lists the possible values and describes what type of project will be seeded.

Table C.1. Service Unit Archetypes

Name	Description
camel	Creates a project for using the FUSE Mediation Router service engine.
cxf-se	Creates a project for developing a Java-first service using the FUSE Services Framework service engine.
cxf-se-wsdl-first	Creates a project for developing a WSDL-first service using the FUSE Services Framework service engine.
cxf-bc	Creates an endpoint project targeted at the FUSE Services Framework binding component.

Name	Description
http-consumer	Creates a consumer endpoint project targeted at the HTTP binding component.
http-provider	Creates a provider endpoint project targeted at the HTTP binding component.
jms-consumer	Creates a consumer endpoint project targeted at the JMS binding component. See *Using the JMS Binding Component*.
jms-provider	Creates a provider endpoint project targeted at the JMS binding component. See *Using the JMS Binding Component*.
file-poller	Creates a polling (consumer) endpoint project targeted at the file binding component. See "Using Poller Endpoints" on page 15.
file-sender	Creates a sender (provider) endpoint project targeted at the file binding component. See "Using Sender Endpoints" on page 31.
ftp-poller	Creates a polling (consumer) endpoint project targeted at the FTP binding component.
ftp-sender	Creates a sender (provider) endpoint project targeted at the FTP binding component.
jsr181-annotated	Creates a project for developing an annotated Java service to be run by the JSR181 service engine. [a]
jsr181-wsdl-first	Creates a project for developing a WSDL generated Java service to be run by the JSR181 service engine.[a]
saxon-xquery	Create a project for executing xquery statements using the Saxon service engine.
saxon-xslt	Create a project for executing XSLT scripts using the Saxon service engine.
eip	Creates a project for using the EIP service engine. [b]
lwcontainer	Create a project for deploying functionality into the lightweight container. [c]
bean	Creates a project for deploying a POJO to be executed by the bean service engine.

Name	Description
ode	Create a project for deploying a BPEL process into the ODE service engine.

[a]The JSR181 has been deprecated. The FUSE Services Framework service engine has superseded it.

[b]The EIP service engine has been deprecated. The FUSE Mediation Router service engine has superseded it.

[c]The lightweight container has been deprecated.

Contents of a project

The contents of your service unit project change from service unit to service unit. Different components require different configuration. Some components, such as the FUSE Services Framework service engine, require that you include Java classes.

At a minimum, a service unit project will contain two things:

- a POM file that configures the JBI plug-in to create a service unit
- an XML configuration file stored in `src/main/resources`

  For many of the components the XML configuration file is called `xbean.xml`. The FUSE Mediation Router component uses a file called `camel-context.xml`.

Configuring the Maven plug-in

You configure the Maven plug-in to package the results of the project build as a service unit by changing the value of the project's `packaging` element to `jbi-service-unit` as shown in Example C.4 on page 57.

Example C.4. Configuring the Maven Plug-in to Build a Service Unit

```
<project ...>
  <modelVersion>4.0.0</modelVersion>

  ...
  <groupId>com.widgets.demo.cxf-wsdl-first</groupId>
  <artifactId>cxfse-wsdl-first-su</artifactId>
  <name>CXF WSDL Fisrt Demo :: SE Service Unit</name>
  <packaging>jbi-service-unit</packaging>
  ...
</project>
```

Specifying the target components

In order to properly fill in the metadata required for packaging a service unit, the Maven plug-in needs to be told what component, or components, the service unit is targeting. If your service unit only has a single component dependency, you can specify it in one of two ways:

- list the targeted component as a dependency
- add a `componentName` property specifying the targeted component

If your service unit has more than one component dependency you need to configure the project as follows:

1. Add a `componentName` property specifying the targeted component.
2. Add the remaining components to the list dependencies.

Example C.5 on page 58 shows configuration for a service unit targeting the FUSE Services Framework binding component.

Example C.5. Specifying the Target Components for a Service Unit

```
...
<dependencies>
  <dependency>
    <groupId>org.apache.servicemix</groupId>
    <artifactId>servicemix-cxf-bc</artifactId>
    <version>3.3.1.0-fuse</version>[1]
  </dependency>
>/dependencies>
...
```

The advantage of using the Maven dependency mechanism is that it allows Maven to check if the targeted component is deployed in the container. If one of the components is not deployed, FUSE ESB will not hold off deploying the service unit until all of the required components are deployed.

[1]You replace this with the version of FUSE Services Framework you are using.

Tip

A message identifying the missing component(s) is typically written to the log.

If your service unit targets is not available as a Maven artifact, you can specify the targeted component using the `componentName` element. This element is added to the standard Maven properties block and specifies the name of a targeted component. Example C.6 on page 59 shows how to use the `componentName` element to specify the target component.

Example C.6. Specifying the Target Components for a Service Unit

```
...
<properties>
  <componentName>servicemix-bean</componentName>
>/properties>
...
```

When you use the `componentName` element Maven does not check to see if the component is installed. Maven also cannot download the required component.

Example

Example C.7 on page 59 shows the POM file for a project building a service unit targeted to the FUSE Services Framework binding component.

Example C.7. POM for a Service Unit Project

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://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> ❶
        <groupId>com.widgets.demo</groupId>
        <artifactId>cxf-wsdl-first</artifactId>
        <version>1.0</version>
    </parent>

  <groupId>com.widgets.demo.cxf-wsdl-first</groupId>
  <artifactId>cxfse-wsdl-first-su</artifactId>
  <name>CXF WSDL Fisrt Demo :: SE Service Unit</name>
  <packaging>jbi-service-unit</packaging> ❷
  <dependencies> ❸
    <dependency>
      <groupId>org.apache.servicemix</groupId>
      <artifactId>servicemix-cxf-bc</artifactId>
      <version>3.3.1.0-fuse</version>
    </dependency>
  >/dependencies>

  <build>
    <plugins>
      <plugin> ❹
        <groupId>org.apache.servicemix.tooling</groupId>
        <artifactId>jbi-maven-plugin</artifactId>
        <extensions>true</extensions>
      </plugin>
    </plugins>
  </build>
</project>
```

The POM in Example C.7 on page 59 does the following:

❶ Specifies that it is a part of the top-level project described in Example C.2 on page 52.

❷ Specifies that this project builds a service unit.

❸ Specifies that the service unit targets the FUSE Services Framework binding component.

❹ Specifies that the FUSE ESB Maven plug-in is to be used.

A Service Assembly Project

Overview

FUSE ESB requires that all service units be bundled into a service assembly before they can be deployed into a container. The FUSE ESB Maven plug-in will collect all of the service units to be bundled and the metadata needed for packaging. It will then build a service assembly containing the service units.

Seeding a project using a Maven artifact

FUSE ESB provides a Maven artifact for seeding a service assembly project. You can seed a project with the **smx-arch** command. As shown in Example C.8 on page 61, the **smx-arch** command takes two arguments. The `groupId` value and the `artifactId` values correspond to the project's group ID and artifact ID.

Example C.8. Maven Archetype Command for Service Assemblies

`smx-arch` sa ["-DgroupId=*`my.group.id`*"] ["-DartifactId=*`my.artifact.id`*"]

Important

The double quotes(") are required when using the `-DgroupId` argument and the `-DartifactId` argument.

Contents of a project

A service assembly project typically only contains the POM file used by Maven.

Configuring the Maven plug-in

You configure the Maven plug-in to package the results of the project build as a service assembly by changing the value of the project's `packaging` element to `jbi-service-assembly` as shown in Example C.9 on page 61.

Example C.9. Configuring the Maven Plug-in to Build a Service Assembly

```
<project ...>
  <modelVersion>4.0.0</modelVersion>

  ...
  <groupId>com.widgets.demo.cxf-wsdl-first</groupId>
  <artifactId>cxf-wsdl-first-sa</artifactId>
  <name>CXF WSDL Fisrt Demo :: Service Assembly</name>
  <packaging>jbi-service-assembly</packaging>
  ...
</project>
```

Specifying the target components

The Maven plug-in needs to be told what service units are being bundled into the service assembly. You do this by specifying the service units as a dependencies using the standard Maven `dependencies` element. You add a `dependency` child element for each service unit. Example C.10 on page 62 shows configuration for a service assembly that bundles two service units.

Example C.10. Specifying the Target Components for a Service Unit

```
...
<dependencies>
  <dependency>
    <groupId>com.widgets.demo.cxf-wsdl-first</groupId>
    <artifactId>cxfse-wsdl-first-su</artifactId>
    <version>1.0</version>
  </dependency>
  <dependency>
    <groupId>com.widgets.demo.cxf-wsdl-first</groupId>
    <artifactId>cxfbc-wsdl-first-su</artifactId>
    <version>1.0</version>
  </dependency>
</dependencies>
...
```

Example

Example C.11 on page 62 shows the POM file for a project building a service assembly.

Example C.11. POM for a Service Assembly Project

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://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> ❶
        <groupId>com.widgets.demo</groupId>
        <artifactId>cxf-wsdl-first</artifactId>
        <version>1.0</version>
    </parent>

  <groupId>com.widgets.demo.cxf-wsdl-first</groupId>
  <artifactId>cxf-wsdl-first-sa</artifactId>
  <name>CXF WSDL Fisrt Demo ::  Service Assemby</name>
  <packaging>jbi-service-assembly</packaging> ❷

  <dependencies> ❸
    <dependency>
      <groupId>com.widgets.demo.cxf-wsdl-first</groupId>
      <artifactId>cxfse-wsdl-first-su</artifactId>
      <version>1.0</version>
    </dependency>
    <dependency>
      <groupId>com.widgets.demo.cxf-wsdl-first</groupId>
      <artifactId>cxfbc-wsdl-first-su</artifactId>
      <version>1.0</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin> ❹
        <groupId>org.apache.servicemix.tooling</groupId>
        <artifactId>jbi-maven-plugin</artifactId>
        <extensions>true</extensions>
      </plugin>
    </plugins>
  </build>
</project>
```

The POM in Example C.11 on page 62 does the following:

❶ Specifies that it is a part of the top-level project described in Example C.2 on page 52.

❷ Specifies that this project builds a service assembly.

❸ Specifies the service units the service assembly bundles.

❹ Specifies that the FUSE ESB Maven plug-in is to be used.

Appendix D. Using the Maven OSGi Tooling

Manually creating a bundle, or a collection of bundles, for a large project can be cumbersome. The Maven bundle plug-in makes the job easier by automating the process and providing a number of shortcuts for specifying the contents of the bundle manifest.

The FUSE ESB OSGi tooling uses the Maven bundle plug-in[1] from Apache Felix. The bundle plug-in is based on the **bnd**[2] tool from Peter Kriens. It automates the construction of OSGi bundle manifests by introspecting the contents of the classes being packaged in the bundle. Using the knowledge of the classes contained in the bundle, the plug-in can calculate the proper values to populate the Import-Packages and the Export-Package properties in the bundle manifest. The plug-in also has default values that are used for other required properties in the bundle manifest.

To use the bundle plug-in you will need to do the following:

1. Add the bundle plug-in to your project's POM file.

2. Configure the plug-in to correctly populate your bundle's manifest.

[1] http://cwiki.apache.org/FELIX/apache-felix-maven-bundle-plugin-bnd.html
[2] http://www.aqute.biz/Code/Bnd

Setting Up a FUSE ESB OSGi Project

Overview

A Maven project for building an OSGi bundle can be a simple single level project. It does not require any sub-projects. It does, however, require that you do the following:

1. Add the bundle plug-in to your POM.
2. Instruct Maven to package the results as an OSGi bundle.

Tip

There are several Maven archetypes to set up your project with the appropriate settings.

Directory structure

A project that constructs an OSGi bundle can be a single level project. It only requires that you have a top-level POM file and a `src` folder. As in all Maven projects, you place all Java source code in the `src/java` folder. You place any non-Java resources into the `src/resources` folder.

Non-Java resources include Spring configuration files, JBI endpoint configuration files, WSDL contracts, etc.

Note

FUSE ESB OSGi projects that use FUSE Services Framework, FUSE Mediation Router, or another Spring configured bean also include a `beans.xml` file located in the `src/resources/META-INF/spring` folder.

Adding a bundle plug-in

Before you can use the bundle plug-in you must add a dependency on Apache Felix. After you add the dependency, you can add the bundle plug-in to the plug-in portion of the POM.

Example D.1 on page 67 shows the POM entries required to add the bundle plug-in to your project.

Example D.1. Adding an OSGi Bundle Plug-in to a POM

```
...
<dependencies>
  <dependency> ❶
    <groupId>org.apache.felix</groupId>
    <artifactId>org.osgi.core</artifactId>
    <version>1.0.0</version>
  </dependency>
...
</dependencies>
...
<build>
  <plugins>
    <plugin> ❷
      <groupId>org.apache.felix</groupId>
      <artifactId>maven-bundle-plugin</artifactId>
      <configuration>
        <instructions>
          <Bundle-SymbolicName>${pom.artifactId}</Bundle-SymbolicName> ❸
          <Import-Package>*,org.apache.camel.osgi</Import-Package> ❹
          <Private-Package>org.apache.servicemix.examples.camel</Private-Package> ❺
        </instructions>
      </configuration>
    </plugin>
  </plugins>
</build>
...
```

The entries in Example D.1 on page 67 do the following:

❶ Adds the dependency on Apache Felix.

❷ Adds the bundle plug-in to your project.

❸ Configures the plug-in to use the project's artifact ID as the bundle's symbolic name.

❹ Configures the plug-in to include all Java packages imported by the bundled classes and also import the `org.apache.camel.osgi` package.

❺ Configures the plug-in to bundle the listed class, but not include them in the list of exported packages.

Note

You should edit the configuration to meet the requirements of your project.

For more information on configuring the bundle plug-in, see "Configuring a Bundle Plug-in" on page 71.

Activating a bundle plug-in

To instruct Maven to use the bundle plug-in, you instruct it to package the results of the project as a bundle. You do this by setting the POM file's `packaging` element to `bundle`.

Useful Maven archetypes

There are several Maven archetypes to generate a project that is preconfigured to use the bundle plug-in:

- "Spring OSGi archetype"
- "FUSE Services Framework code-first archetype"
- "FUSE Services Framework wsdl-first archetype"
- "FUSE Mediation Router archetype"

Spring OSGi archetype

The Spring OSGi archetype creates a generic project for building an OSGi project using Spring DM:

```
org.springframework.osgi/spring-bundle-osgi-archetype/1.1.2
```

You invoke the archetype using the following command:

```
mvn archetype:create
-DarchetypeGroupId=org.springframework.osgi
-DarchetypeArtifactId=spring-osgi-bundle-archetype
-DarchetypeVersion=1.12
-DgroupId=groupId
-DartifactId=artifactId
-Dversion=version
```

FUSE Services Framework code-first archetype

The FUSE Services Framework code-first archetype creates a project for building a service from Java:

```
org.apache.servicemix.tooling/servicemix-osgi-cxf-code-first-archetype/2008.01.0.3-fuse
```

You invoke the archetype using the following command:

```
mvn archetype:create
-DarchetypeGroupId=org.apache.servicemix.tooling
-DarchetypeArtifactId=spring-osgi-bundle-archetype
-DarchetypeVersion=2008.01.0.3-fuse
-DgroupId=groupId
-DartifactId=artifactId
-Dversion=version
```

FUSE Services Framework wsdl-first archetype

The FUSE Services Framework wsdl-first archetype creates a project for creating a service from WSDL:

```
org.apache.servicemix.tooling/servicemix-osgi-cxf-wsdl-first-archetype/2008.01.0.3-fuse
```

You invoke the archetype using the following command:

```
mvn archetype:create
-DarchetypeGroupId=org.apache.servicemix.tooling
-DarchetypeArtifactId=servicemix-osgi-cxf-wsdl-first-archetype
-DarchetypeVersion=2008.01.0.3-fuse
-DgroupId=groupId
-DartifactId=artifactId
-Dversion=version
```

FUSE Mediation Router archetype

The FUSE Mediation Router archetype creates a project for building a route that is deployed into FUSE ESB:

```
org.apache.servicemix.tooling/servicemix-osgi-camel-archetype/2008.01.0.3-fuse
```

You invoke the archetype using the following command:

```
mvn archetype:create
-DarchetypeGroupId=org.apache.servicemix.tooling
-DarchetypeArtifactId=servicemix-osgi-camel-archetype
-DarchetypeVersion=2008.01.0.3-fuse
-DgroupId=groupId
-DartifactId=artifactId
-Dversion=version
```

Configuring a Bundle Plug-in

Overview

A bundle plug-in requires very little information to function. All of the required properties have default settings to generate a valid OSGi bundle.

While you can create a valid bundle using just the default values, you will likely want to modify some of the values. You can specify most of the properties inside the plug-in's `instructions` element.

Configuration properties

Some of the commonly used configuration properties are:

- Bundle-SymbolicName
- Bundle-Name
- Bundle-Version
- Export-Package
- Private-Package
- Import-Package

Setting a bundle's symbolic name

By default, the bundle plug-in sets the value for the Bundle-SymbolicName property to *groupId*+ "." + *artifactId*, with the following exceptions:

- If *groupId* has only one section (no dots), the first package name with classes is returned.

  For example, if the groupId is `commons-logging:commons-logging`, the bundle's symbolic name is `org.apache.commons.logging`.

- If *artifactId* is equal to the last section of *groupId*, then *groupId* is used.

  For example, if the POM specifies the group ID and artifact ID as `org.apache.maven:maven`, the bundle's symbolic name is `org.apache.maven`.

- If *artifactId* starts with the last section of *groupId*, that portion is removed.

  For example, if the POM specifies the group ID and artifact ID as `org.apache.maven:maven-core`, the bundle's symbolic name is `org.apache.maven.core`.

To specify your own value for the bundle's symbolic name, add a `Bundle-SymbolicName` child in the plug-in's `instructions` element, as shown in Example D.2.

Example D.2. Setting a Bundle's Symbolic Name

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Bundle-SymbolicName>${pom.artifactId}</Bundle-SymbolicName>
     ...
    </instructions>
  </configuration>
</plugin>
```

Setting a bundle's name

By default, a bundle's name is set to `${pom.name}`.

To specify your own value for the bundle's name, add a `Bundle-Name` child to the plug-in's `instructions` element, as shown in Example D.3.

Example D.3. Setting a Bundle's Name

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Bundle-Name>JoeFred</Bundle-Name>
     ...
    </instructions>
  </configuration>
</plugin>
```

Setting a bundle's version

By default, a bundle's version is set to `${pom.version}`. Any dashes (`-`) are replaced with dots (`.`).

To specify your own value for the bundle's version, add a `Bundle-Version` child to the plug-in's `instructions` element, as shown in Example D.4.

Example D.4. Setting a Bundle's Version

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Bundle-Version>1.0.3.1</Bundle-Version>
     ...
    </instructions>
  </configuration>
</plugin>
```

Specifying exported packages

By default, the OSGi manifest's `Export-Package` list is populated by all of the packages in your project's class path that match the pattern *`Bundle-SymbolicName`*`.*`. These packages are also included in the bundle.

Important

If you use a `Private-Package` element in your plug-in configuration and do not specify a list of packages to export, the default behavior is to assume that no packages are exported. Only the packages listed in the `Private-Package` element are included in the bundle and none of them are exported.

The default behavior can result in very large packages as well as exporting packages that should be kept private. To change the list of exported packages you can add a `Export-Package` child to the plug-in's `instructions` element.

The `Export-Package` element specifies a list of packages that are to be included in the bundle and be exported. The package names can be specified using the `*` wildcard. For example, the entry `com.fuse.demo.*`, includes all packages on the project's classpath that start with `com.fuse.demo`.

You can specify packages to be excluded be prefixing the entry with `!`. For example, the entry, `!com.fuse.demo.private`, excludes the package `com.fuse.demo.private`.

When attempting to exclude packages, the order of entries in the list is important. The list is processed in order from the start and subsequent contradicting entries are ignored.

For example, to include all packages starting with `com.fuse.demo` except the package `com.fuse.demo.private`, list the packages in the following way:

```
!com.fuse.demo.private,com.fuse.demo.*
```

However, if you list the packages as:

```
com.fuse.demo.*,!com.fuse.demo.private
```

Then `com.fuse.demo.private` is included in the bundle because it matches the first pattern.

Specifying private packages

By default, all packages included in a bundle are exported. You can include packages in the bundle without exporting them. To specify a list of packages to be included in a bundle, but not exported, add a `Private-Package` child to the plug-in's `instructions` element.

The `Private-Package` element works similarly to the `Export-Package` element. You specify a list of packages to be included in the bundle. The bundle plug-in uses the list to find all classes on the project's classpath to be included in the bundle. These packages are packaged in the bundle, but not exported.

Important

If a package matches an entry in both the `Private-Package` element and the `Export-Package` element, the `Export-Package` element takes precedent. The package is added to the bundle and exported.

Example D.5 shows the configuration for including a private package in a bundle

Example D.5. Including a Private Package in a Bundle

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Private-Package>org.apache.cxf.wsdlFirst.impl</Private-Package>
     ...
    </instructions>
  </configuration>
</plugin>
```

Specifying imported packages

By default, the bundle plug-in populates the OSGi manifest's Import-Package property with a list of all the packages referred to by the contents of the bundle and not included in the bundle.

While the default behavior is typically sufficient for most projects, you might find instances where you want to import packages that are not automatically added to the list. The default behavior can also result in unwanted packages being imported.

To specify a list of packages to be imported by the bundle, add a `Import-Package` child to the plug-in's `instructions` element. The syntax for the package list is the same as for both the `Export-Package` and `Private-Package` elements.

Important

When you use the `Import-Package` element, the plug-in does not automatically scan the bundle's contents to determine if there are any required imports. To ensure that the contents of the bundle are scanned, you must place `*` as the last entry in the package list.

Example D.6 shows the configuration for including a private package in a bundle

Example D.6. Specifying the Packages Imported by a Bundle

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Import-Package>javax.jws,
         javax.wsdl,
         org.apache.cxf.bus,
         org.apache.cxf.bus.spring,
         org.apache.cxf.bus.resource,
         org.apache.cxf.configuration.spring,
         org.apache.cxf.resource,
         org.springframework.beans.factory.config,
         *
     </Import-Package>
     ...
   </instructions>
  </configuration>
</plugin>
```

More information

For more information on configuring a bundle plug-in, see:

- Apache Felix documentation[3]
- Peter Kriens' aQute Software Consultancy web site[4]

[3] http://cwiki.apache.org/FELIX/apache-felix-maven-bundle-plugin-bnd.html
[4] http://www.aqute.biz/Code/Bnd

Index