



# FUSE Services Framework

## Using the SOAP Binding

Version 2.0  
December 2007

Making Software Work Together™

---

# Using the SOAP Binding

IONA Technologies

Version 2.0

Published 19 Mar 2008

Copyright © 2001-2008 IONA Technologies PLC

## Trademark and Disclaimer Notice

IONA Technologies PLC and/or its subsidiaries may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this publication. Except as expressly provided in any written license agreement from IONA Technologies PLC, the furnishing of this publication does not give you any license to these patents, trademarks, copyrights, or other intellectual property. Any rights not expressly granted herein are reserved.

IONA, IONA Technologies, the IONA logo, Orbix, High Performance Integration, Artix, FUSE, and Making Software Work Together are trademarks or registered trademarks of IONA Technologies PLC and/or its subsidiaries.

Java and J2EE are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. CORBA is a trademark or registered trademark of the Object Management Group, Inc. in the United States and other countries. All other trademarks that appear herein are the property of their respective owners.

While the information in this publication is believed to be accurate, IONA Technologies PLC makes no warranty of any kind to this material including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IONA shall not be liable for errors contained herein, or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

## Copyright Notice

No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, photocopying, recording or otherwise, without prior written consent of IONA Technologies PLC. No third-party intellectual property right liability is assumed with respect to the use of the information contained herein. IONA Technologies PLC assumes no responsibility for errors or omissions contained in this publication. This publication and features described herein are subject to change without notice. Portions of this document may include Apache Foundation documentation, all rights reserved.

---

---

# Table of Contents

<b>Using SOAP 1.1 Messages .....</b>	<b>9</b>
Adding a SOAP 1.1 Binding .....	10
Adding SOAP Headers to a SOAP 1.1 Binding .....	13
<b>Using SOAP 1.2 Messages .....</b>	<b>19</b>
Adding a SOAP 1.2 Binding to a WSDL Document .....	20
Adding Headers to a SOAP 1.2 Message .....	23
<b>Sending Binary Data Using SOAP with Attachments .....</b>	<b>29</b>
<b>Sending Binary Data with SOAP MTOM .....</b>	<b>33</b>
Annotating Data Types to use MTOM .....	34
Enabling MTOM .....	38
Using JAX-WS APIs .....	39
Using configuration .....	41
Index .....	43



---

## List of Tables

1. soap12:header Attributes .....	23
2. mime:content Attributes .....	31



---

## List of Examples

1. Ordering System Interface .....	11
2. SOAP 1.1 Binding for <code>orderWidgets</code> .....	12
3. SOAP Header Syntax .....	13
4. SOAP 1.1 Binding with a SOAP Header .....	14
5. SOAP 1.1 Binding for <code>orderWidgets</code> with a SOAP Header .....	16
6. Ordering System Interface .....	21
7. SOAP 1.2 Binding for <code>orderWidgets</code> .....	22
8. SOAP Header Syntax .....	23
9. SOAP 1.2 Binding with a SOAP Header .....	24
10. SOAP 1.2 Binding for <code>orderWidgets</code> with a SOAP Header .....	26
11. MIME Namespace Specification in a Contract .....	29
12. Contract using SOAP with Attachments .....	31
13. Message for MTOM .....	34
14. Binary Data for MTOM .....	36
15. JAXB Class for MTOM .....	37
16. Getting the SOAP Binding from an Endpoint .....	39
17. Setting a Service Provider's MTOM Enabled Property .....	39
18. Getting a SOAP Binding from a <code>BindingProvider</code> .....	40
19. Setting a Consumer's MTOM Enabled Property .....	40
20. Configuration for Enabling MTOM .....	41



---

# Using SOAP 1.1 Messages

## **Summary**

*FUSE Services Framework provides a tool to generate a SOAP 1.1 binding which does not use any SOAP headers. However, you can add SOAP headers to your binding using any text or XML editor.*

## **Table of Contents**

Adding a SOAP 1.1 Binding .....	10
Adding SOAP Headers to a SOAP 1.1 Binding .....	13

## Adding a SOAP 1.1 Binding

---

### Using wsdl2soap

To generate a SOAP 1.1 binding using **wsdl2soap** use the following command:

```
wsdl2soap {-i port-type-name} [-b binding-name] [-d
output-directory] [-o output-file] [-n soap-body-namespace] [-style
(document/rpc)] [-use (literal/encoded)] [-v] [[-verbose] | [-quiet]] wSDLurl
```

The command has the following options:

Option	Interpretation
<code>-i <i>port-type-name</i></code>	Specifies the <code>portType</code> element for which a binding should be generated.
<code><i>wSDLurl</i></code>	The path and name of the WSDL file containing the <code>portType</code> element definition.

The tool has the following optional arguments:

Option	Interpretation
<code>-b <i>binding-name</i></code>	Specifies the name of the generated SOAP binding.
<code>-d <i>output-directory</i></code>	Specifies the directory to place generated WSDL file.
<code>-o <i>output-file</i></code>	Specifies the name of the generated WSDL file.
<code>-n <i>soap-body-namespace</i></code>	Specifies the SOAP body namespace when the style is RPC.
<code>-style (document/rpc)</code>	Specifies the encoding style (document or RPC) to use in the SOAP binding. The default is <code>document</code> .
<code>-use (literal/encoded)</code>	Specifies the binding use (encoded or literal) to use in the SOAP binding. The default is <code>literal</code> .
<code>-v</code>	Displays the version number for the tool.
<code>-verbose</code>	Displays comments during the code generation process.
<code>-quiet</code>	Suppresses comments during the code generation process.

The `-i port-type-name` and `wSDLurl` arguments are required. If the `-style rpc` argument is specified, the `-n soap-body-namespace` argument is also required. All other arguments are optional and may be listed in any order.



## Important

`wSDL2soap` does not support the generation of `document/encoded` SOAP bindings.

---

### Example

If your system had an interface that took orders and offered a single operation to process the orders it would be defined in a WSDL fragment similar to the one shown in Example 1, “Ordering System Interface”.

### Example 1. Ordering System Interface

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
  targetNamespace="http://widgetVendor.com/widgetOrderForm"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://widgetVendor.com/widgetOrderForm"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

  <message name="widgetOrder">
    <part name="numOrdered" type="xsd:int"/>
  </message>
  <message name="widgetOrderBill">
    <part name="price" type="xsd:float"/>
  </message>
  <message name="badSize">
    <part name="numInventory" type="xsd:int"/>
  </message>

  <portType name="orderWidgets">
    <operation name="placeWidgetOrder">
      <input message="tns:widgetOrder" name="order"/>
      <output message="tns:widgetOrderBill" name="bill"/>
      <fault message="tns:badSize" name="sizeFault"/>
    </operation>
  </portType>
  ...
</definitions>
```

The SOAP binding generated for `orderWidgets` is shown in Example 2, “SOAP 1.1 Binding for `orderWidgets`”.

### Example 2. SOAP 1.1 Binding for `orderWidgets`

```
<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="placeWidgetOrder">
    <soap:operation soapAction="" style="document"/>
    <input name="order">
      <soap:body use="literal"/>
    </input>
    <output name="bill">
      <soap:body use="literal"/>
    </output>
    <fault name="sizeFault">
      <soap:body use="literal"/>
    </fault>
  </operation>
</binding>
```

This binding specifies that messages are sent using the `document/literal` message style.

## Adding SOAP Headers to a SOAP 1.1 Binding

---

### Overview

SOAP headers are defined by adding `soap:header` elements to your default SOAP 1.1 binding. The `soap:header` element is an optional child of the `input`, `output`, and `fault` elements of the binding. The SOAP header becomes part of the parent message. A SOAP header is defined by specifying a message and a message part. Each SOAP header can only contain one message part, but you can insert as many SOAP headers as needed.

---

### Syntax

The syntax for defining a SOAP header is shown in Example 3, “SOAP Header Syntax”. The `message` attribute of `soap:header` is the qualified name of the message from which the part being inserted into the header is taken. The `part` attribute is the name of the message part inserted into the SOAP header. Because SOAP headers are always document style, the WSDL message part inserted into the SOAP header must be defined using an element. Together the `message` and the `part` attributes fully describe the data to insert into the SOAP header.

### Example 3. SOAP Header Syntax

```
<binding name="headwig">
  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="weave">
    <soap:operation soapAction="" style="document"/>
    <input name="grain">
      <soap:body .../>
      <soap:header message="QName" part="partName"/>
    </input>
  ...
</binding>
```

As well as the mandatory `message` and `part` attributes, `soap:header` also supports the `namespace`, the `use`, and the `encodingStyle` attributes. These optional attributes function the same for `soap:header` as they do for `soap:body`.

---

### Splitting messages between body and header

The message part inserted into the SOAP header can be any valid message part from the contract. It can even be a part from the parent message which is being used as the SOAP body. Because it is unlikely that you would want

to send information twice in the same message, the SOAP binding provides a means for specifying the message parts that are inserted into the SOAP body.

The `soap:body` element has an optional attribute, `parts`, that takes a space delimited list of part names. When `parts` is defined, only the message parts listed are inserted into the SOAP body. You can then insert the remaining parts into the SOAP header.



## Note

When you define a SOAP header using parts of the parent message, FUSE Services Framework automatically fills in the SOAP headers for you.

---

### Example

Example 4, “SOAP 1.1 Binding with a SOAP Header” shows a modified version of the `orderWidgets` service shown in Example 1, “Ordering System Interface”. This version has been modified so that each order has an `xsd:base64binary` value placed in the SOAP header of the request and response. The SOAP header is defined as being the `keyVal` part from the `widgetKey` message. In this case you would be responsible for adding the SOAP header in your application logic because it is not part of the input or output message.

### Example 4. SOAP 1.1 Binding with a SOAP Header

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
  targetNamespace="http://widgetVendor.com/widgetOrderForm"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://widgetVendor.com/widgetOrderForm"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsdl="http://widgetVendor.com/types/widgetTypes"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
<types>
  <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <element name="keyElem" type="xsd:base64Binary"/>
  </schema>
</types>
```

## Adding SOAP Headers to a SOAP 1.1 Binding

---

```
<message name="widgetOrder">
  <part name="numOrdered" type="xsd:int"/>
</message>
<message name="widgetOrderBill">
  <part name="price" type="xsd:float"/>
</message>
<message name="badSize">
  <part name="numInventory" type="xsd:int"/>
</message>
<message name="widgetKey">
  <part name="keyVal" element="xsd1:keyElem"/>
</message>

<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
</portType>

<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="placeWidgetOrder">
    <soap:operation soapAction="" style="document"/>
    <input name="order">
      <soap:body use="literal"/>
      <soap:header message="tns:widgetKey" part="keyVal"/>
    </input>
    <output name="bill">
      <soap:body use="literal"/>
      <soap:header message="tns:widgetKey" part="keyVal"/>
    </output>
    <fault name="sizeFault">
      <soap:body use="literal"/>
    </fault>
  </operation>
</binding>
...
</definitions>
```

You could modify Example 4, “SOAP 1.1 Binding with a SOAP Header” so that the header value was a part of the input and output messages as shown in Example 5, “SOAP 1.1 Binding for orderWidgets with a SOAP Header”. In this case `keyVal` is a part of the input and output messages. In the `soap:body` element's `parts` attribute specifies that `keyVal` is not to be inserted into the body. However, it is inserted into the SOAP header.

### Example 5. SOAP 1.1 Binding for orderWidgets with a SOAP Header

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
  targetNamespace="http://widgetVendor.com/widgetOrderForm"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://widgetVendor.com/widgetOrderForm"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

  <types>
    <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
      xmlns="http://www.w3.org/2001/XMLSchema"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
      <element name="keyElem" type="xsd:base64Binary"/>
    </schema>
  </types>

  <message name="widgetOrder">
    <part name="numOrdered" type="xsd:int"/>
    <part name="keyVal" element="xsd1:keyElem"/>
  </message>
  <message name="widgetOrderBill">
    <part name="price" type="xsd:float"/>
    <part name="keyVal" element="xsd1:keyElem"/>
  </message>
  <message name="badSize">
    <part name="numInventory" type="xsd:int"/>
  </message>

  <portType name="orderWidgets">
    <operation name="placeWidgetOrder">
      <input message="tns:widgetOrder" name="order"/>
      <output message="tns:widgetOrderBill" name="bill"/>
      <fault message="tns:badSize" name="sizeFault"/>
    </operation>
  </portType>

  <binding name="orderWidgetsBinding" type="tns:orderWidgets">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeWidgetOrder">
      <soap:operation soapAction="" style="document"/>
      <input name="order">
        <soap:body use="literal" parts="numOrdered"/>
        <soap:header message="tns:widgetOrder" part="keyVal"/>
      </input>
      <output name="bill">
```

## Adding SOAP Headers to a SOAP 1.1 Binding

---

```
<soap:body use="literal" parts="bill"/>
  <soap:header message="tns:widgetOrderBill" part="keyVal"/>
</output>
<fault name="sizeFault">
  <soap:body use="literal"/>
</fault>
</operation>
</binding>
...
</definitions>
```



---

# Using SOAP 1.2 Messages

## **Summary**

*FUSE Services Framework provides tools to generate a SOAP 1.2 binding which does not use any SOAP headers. You can add SOAP headers to your binding using any text or XML editor.*

## **Table of Contents**

Adding a SOAP 1.2 Binding to a WSDL Document .....	20
Adding Headers to a SOAP 1.2 Message .....	23

## Adding a SOAP 1.2 Binding to a WSDL Document

---

### Using `wsdl2soap`

To generate a SOAP 1.2 binding using `wsdl2soap` use the following command:

```
wsdl2soap {-i port-type-name} [-b binding-name] {-soap12} [-d  
output-directory] [-o output-file] [-n soap-body-namespace] [-style  
(document/rpc)] [-use (literal/encoded)] [-v] [[-verbose] | [-quiet]] wsdlurl
```

The tool has the following required arguments:

Option	Interpretation
<code>-i port-type-name</code>	Specifies the <code>portType</code> element for which a binding should be generated.
<code>-soap12</code>	Specifies that the generated binding uses SOAP 1.2.
<code>wsdlurl</code>	The path and name of the WSDL file containing the <code>portType</code> element definition.

The tool has the following optional arguments:

Option	Interpretation
<code>-b binding-name</code>	Specifies the name of the generated SOAP binding.
<code>-soap12</code>	Specifies that the generated binding will use SOAP 1.2.
<code>-d output-directory</code>	Specifies the directory to place generated WSDL file.
<code>-o output-file</code>	Specifies the name of the generated WSDL file.
<code>-n soap-body-namespace</code>	Specifies the SOAP body namespace when the style is RPC.
<code>-style (document/rpc)</code>	Specifies the encoding style (document or RPC) to use in the SOAP binding. The default is <code>document</code> .
<code>-use (literal/encoded)</code>	Specifies the binding use (encoded or literal) to use in the SOAP binding. The default is <code>literal</code> .
<code>-v</code>	Displays the version number for the tool.

Option	Interpretation
-verbose	Displays comments during the code generation process.
-quiet	Suppresses comments during the code generation process.

The `-i port-type-name` and `wSDLurl` arguments are required. If the `-style rpc` argument is specified, the `-n soap-body-namespace` argument is also required. All other arguments are optional and may be listed in any order.

### Important

**wSDL2soap** does not support the generation of `document/encoded` SOAP 1.2 bindings.

---

#### Example

If your system had an interface that took orders and offered a single operation to process the orders it would be defined in a WSDL fragment similar to the one shown in Example 6, “Ordering System Interface”.

#### Example 6. Ordering System Interface

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
  targetNamespace="http://widgetVendor.com/widgetOrderForm"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
  xmlns:tns="http://widgetVendor.com/widgetOrderForm"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

  <message name="widgetOrder">
    <part name="numOrdered" type="xsd:int"/>
  </message>
  <message name="widgetOrderBill">
    <part name="price" type="xsd:float"/>
  </message>
  <message name="badSize">
    <part name="numInventory" type="xsd:int"/>
  </message>

  <portType name="orderWidgets">
    <operation name="placeWidgetOrder">
```

```
<input message="tns:widgetOrder" name="order"/>
<output message="tns:widgetOrderBill" name="bill"/>
<fault message="tns:badSize" name="sizeFault"/>
</operation>
</portType>
...
</definitions>
```

The SOAP binding generated for `orderWidgets` is shown in Example 7, “SOAP 1.2 Binding for `orderWidgets`”.

### Example 7. SOAP 1.2 Binding for `orderWidgets`

```
<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="placeWidgetOrder">
    <soap12:operation soapAction="" style="document"/>
    <input name="order">
      <soap12:body use="literal"/>
    </input>
    <output name="bill">
      <wsoap12:body use="literal"/>
    </output>
    <fault name="sizeFault">
      <soap12:body use="literal"/>
    </fault>
  </operation>
</binding>
```

This binding specifies that messages are sent using the `document/literal` message style.

## Adding Headers to a SOAP 1.2 Message

---

### Overview

SOAP message headers are defined by adding `soap12:header` elements to your SOAP 1.2 message. The `soap12:header` element is an optional child of the `input`, `output`, and `fault` elements of the binding. The header becomes part of the parent message. A header is defined by specifying a message and a message part. Each SOAP header can only contain one message part, but you can insert as many headers as needed.

---

### Syntax

The syntax for defining a SOAP header is shown in Example 8, “SOAP Header Syntax”.

### Example 8. SOAP Header Syntax

```
<binding name="headwig">
  <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="weave">
    <soap12:operation soapAction="" style="document"/>
    <input name="grain">
      <soap12:body .../>
      <soap12:header message="QName" part="partName"
        use="literal|encoded"
        encodingStyle="encodingURI"
        namespace="namespaceURI" />
    </input>
    ...
  </binding>
```

The `soap12:header` element’s attributes are described in Table 1, “`soap12:header` Attributes”.

**Table 1. `soap12:header` Attributes**

Attribute	Description
message	A required attribute specifying the qualified name of the message from which the part being inserted into the header is taken.
part	A required attribute specifying the name of the message part inserted into the SOAP header.
use	Specifies whether the message parts are to be encoded using encoding rules. If set to <code>encoded</code> the message parts are encoded using the encoding rules specified by the value of the

## Adding Headers to a SOAP 1.2 Message

---

Attribute	Description
	<code>encodingStyle</code> attribute. If set to <code>literal</code> , then the message parts are defined by the schema types referenced.
<code>encodingStyle</code>	Specifies the encoding rules used to construct the message.
<code>namespace</code>	Defines the namespace to be assigned to the header element serialized with <code>use="encoded"</code> .

---

### Splitting messages between body and header

The message part inserted into the SOAP header can be any valid message part from the contract. It can even be a part from the parent message which is being used as the SOAP body. Because it is unlikely that you would want to send information twice in the same message, the SOAP 1.2 binding provides a means for specifying the message parts that are inserted into the SOAP body.

The `soap12:body` element has an optional attribute, `parts`, that takes a space delimited list of part names. When `parts` is defined, only the message parts listed are inserted into the body of the SOAP 1.2 message. You can then insert the remaining parts into the message's header.



### Note

When you define a SOAP header using parts of the parent message, FUSE Services Framework automatically fills in the SOAP headers for you.

---

### Example

Example 9, “SOAP 1.2 Binding with a SOAP Header” shows a modified version of the `orderWidgets` service shown in Example 6, “Ordering System Interface”. This version has been modified so that each order has an `xsd:base64binary` value placed in the header of the request and response. The header is defined as being the `keyVal` part from the `widgetKey` message. In this case you would be responsible for adding the application logic to create the header because it is not part of the input or output message.

### Example 9. SOAP 1.2 Binding with a SOAP Header

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
  targetNamespace="http://widgetVendor.com/widgetOrderForm"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
```

## Adding Headers to a SOAP 1.2 Message

---

```
xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
xmlns:tns="http://widgetVendor.com/widgetOrderForm"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

<types>
  <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <element name="keyElem" type="xsd:base64Binary"/>
  </schema>
</types>

<message name="widgetOrder">
  <part name="numOrdered" type="xsd:int"/>
</message>

<message name="widgetOrderBill">
  <part name="price" type="xsd:float"/>
</message>

<message name="badSize">
  <part name="numInventory" type="xsd:int"/>
</message>

<message name="widgetKey">
  <part name="keyVal" element="xsd1:keyElem"/>
</message>

<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
</portType>

<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="placeWidgetOrder">
    <soap12:operation soapAction="" style="document"/>
    <input name="order">
      <soap12:body use="literal"/>
      <soap12:header message="tns:widgetKey" part="keyVal"/>
    </input>
    <output name="bill">
      <soap12:body use="literal"/>
      <soap12:header message="tns:widgetKey" part="keyVal"/>
    </output>
    <fault name="sizeFault">
      <soap12:body use="literal"/>
    </fault>
  </operation>
</binding>
```

```
    </fault>
  </operation>
</binding>
...
</definitions>
```

You could modify Example 9, “SOAP 1.2 Binding with a SOAP Header” so that the header value was a part of the input and output messages as shown in Example 10, “SOAP 1.2 Binding for orderWidgets with a SOAP Header”. In this case `keyVal` is a part of the input and output messages. In the `soap12:body` elements the `parts` attribute specifies that `keyVal` is not to be inserted into the body. However, it is inserted into the header.

### Example 10. SOAP 1.2 Binding for orderWidgets with a SOAP Header

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
  targetNamespace="http://widgetVendor.com/widgetOrderForm"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
  xmlns:tns="http://widgetVendor.com/widgetOrderForm"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
<types>
  <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <element name="keyElem" type="xsd:base64Binary"/>
  </schema>
</types>
<message name="widgetOrder">
  <part name="numOrdered" type="xsd:int"/>
  <part name="keyVal" element="xsd1:keyElem"/>
</message>
<message name="widgetOrderBill">
  <part name="price" type="xsd:float"/>
  <part name="keyVal" element="xsd1:keyElem"/>
</message>
<message name="badSize">
  <part name="numInventory" type="xsd:int"/>
</message>
<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
```

## Adding Headers to a SOAP 1.2 Message

---

```
<input message="tns:widgetOrder" name="order"/>
<output message="tns:widgetOrderBill" name="bill"/>
<fault message="tns:badSize" name="sizeFault"/>
</operation>
</portType>

<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="placeWidgetOrder">
    <soap12:operation soapAction="" style="document"/>
    <input name="order">
      <soap12:body use="literal" parts="numOrdered"/>
      <soap12:header message="tns:widgetOrder" part="keyVal"/>
    </input>
    <output name="bill">
      <soap12:body use="literal" parts="bill"/>
      <soap12:header message="tns:widgetOrderBill" part="keyVal"/>
    </output>
    <fault name="sizeFault">
      <soap12:body use="literal"/>
    </fault>
  </operation>
</binding>
...
</definitions>
```



---

# Sending Binary Data Using SOAP with Attachments

## Summary

*SOAP attachments provide a mechanism for sending binary data as part of a SOAP message. Using SOAP with attachments requires that you define your SOAP messages as MIME multipart messages.*

---

## Overview

SOAP messages generally do not carry binary data. However, the W3C SOAP 1.1 specification allows for using MIME multipart/related messages to send binary data in SOAP messages. This technique is called using SOAP with attachments. SOAP attachments are defined in the W3C's *SOAP Messages with Attachments Note* (<http://www.w3.org/TR/SOAP-attachments>).

---

## Namespace

The WSDL extensions used to define the MIME multipart/related messages are defined in the namespace `http://schemas.xmlsoap.org/wsdl/mime/`.

In the discussion that follows, it is assumed that this namespace is prefixed with `mime`. The entry in the WSDL `definitions` element to set this up is shown in Example 11, "MIME Namespace Specification in a Contract".

### Example 11. MIME Namespace Specification in a Contract

```
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
```

---

## Changing the message binding

In a default SOAP binding, the first child element of the `input`, `output`, and `fault` elements is a `soap:body` element describing the body of the SOAP message representing the data. When using SOAP with attachments, the `soap:body` element is replaced with a `mime:multipartRelated` element.



### Note

WSDL does not support using `mime:multipartRelated` for `fault` messages.

---

## Describing a MIME multipart message

The `mime:multipartRelated` element tells FUSE Services Framework that the message body is going to be a multipart message that potentially contains binary data. The contents of the element define the parts of the message and their contents. `mime:multipartRelated` elements contain one or more `mime:part` elements that describe the individual parts of the message.

The first `mime:part` element must contain the `soap:body` element that would normally appear in a default SOAP binding. The remaining `mime:part` elements define the attachments that are being sent in the message.

---

MIME multipart messages are described using a `mime:multipartRelated` element that contains a number of `mime:part` elements. To fully describe a MIME multipart message do the following:

1. Inside the `input` or `output` message you want to send as a MIME multipart message, add a `mime:multipartRelated` element as the first child element of the enclosing message.
2. Add a `mime:part` child element to the `mime:multipartRelated` element and set its `name` attribute to a unique string.
3. Add a `soap:body` element as the child of the `mime:part` element and set its attributes appropriately.



### Tip

If the contract had a default SOAP binding, you can copy the `soap:body` element from the corresponding message from the default binding into the MIME multipart message.

4. Add another `mime:part` child element to the `mime:multipartRelated` element and set its `name` attribute to a unique string.
5. Add a `mime:content` child element to the `mime:part` element to describe the contents of this part of the message.

To fully describe the contents of a MIME message part the `mime:content` element has the following attributes:

---

**Table 2. mime:content Attributes**

Attribute	Description
part	Specifies the name of the WSDL message <code>part</code> , from the parent message definition, that is used as the content of this part of the MIME multipart message being placed on the wire.
type	<p>The MIME type of the data in this message part. MIME types are defined as a type and a subtype using the syntax <code>type/subtype</code>.</p> <p>There are a number of predefined MIME types such as <code>image/jpeg</code> and <code>text/plain</code>. The MIME types are maintained by the Internet Assigned Numbers Authority (IANA) and described in detail in <i>Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies</i>(<a href="ftp://ftp.isi.edu/in-notes/rfc2045.txt">ftp://ftp.isi.edu/in-notes/rfc2045.txt</a>) and <i>Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types</i>(<a href="ftp://ftp.isi.edu/in-notes/rfc2046.txt">ftp://ftp.isi.edu/in-notes/rfc2046.txt</a>).</p>

6. For each additional MIME part, repeat steps Step 4 and Step 5.

---

### Example

Example 12, “Contract using SOAP with Attachments” shows a WSDL fragment defining a service that stores X-rays in JPEG format. The image data, `xRay`, is stored as an `xsd:base64binary` and is packed into the MIME multipart message's second part, `imageData`. The remaining two parts of the input message, `patientName` and `patientNumber`, are sent in the first part of the MIME multipart image as part of the SOAP body.

### Example 12. Contract using SOAP with Attachments

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="XrayStorage"
  targetNamespace="http://mediStor.org/x-rays"
  xmlns="http://schemas.xmlsoap.org/wsd/"
  xmlns:tns="http://mediStor.org/x-rays"
  xmlns:mime="http://schemas.xmlsoap.org/wsd/mime/"
  xmlns:soap="http://schemas.xmlsoap.org/wsd/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <message name="storRequest">
    <part name="patientName" type="xsd:string"/>
    <part name="patientNumber" type="xsd:int"/>
    <part name="xRay" type="xsd:base64Binary"/>
  </message>
  <message name="storResponse">
```

---

```
<part name="success" type="xsd:boolean"/>
</message>

<portType name="xRayStorage">
  <operation name="store">
    <input message="tns:storRequest" name="storRequest"/>
    <output message="tns:storResponse" name="storResponse"/>
  </operation>
</portType>

<binding name="xRayStorageBinding" type="tns:xRayStorage">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="store">
    <soap:operation soapAction="" style="document"/>
    <input name="storRequest">
      <mime:multipartRelated>
        <mime:part name="bodyPart">
          <soap:body use="literal"/>
        </mime:part>
        <mime:part name="imageData">
          <mime:content part="xRay" type="image/jpeg"/>
        </mime:part>
      </mime:multipartRelated>
    </input>
    <output name="storResponse">
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>

<service name="xRayStorageService">
  <port binding="tns:xRayStorageBinding" name="xRayStoragePort">
    <soap:address location="http://localhost:9000"/>
  </port>
</service>
</definitions>
```

---

# Sending Binary Data with SOAP MTOM

## Summary

*SOAP Message Transmission Optimization Mechanism (MTOM) replaces SOAP with attachments as a mechanism for sending binary data as part of an XML message. Using MTOM with FUSE Services Framework requires adding the correct schema types to a service's contract and enabling the MTOM optimizations.*

## Table of Contents

Annotating Data Types to use MTOM .....	34
Enabling MTOM .....	38
Using JAX-WS APIs .....	39
Using configuration .....	41

SOAP Message Transmission Optimization Mechanism (MTOM) specifies an optimized method for sending binary data as part of a SOAP message. Unlike SOAP with Attachments, MTOM requires the use of XML-binary Optimized Packaging (XOP) packages for transmitting binary data. Using MTOM to send binary data does not require you to fully define the MIME Multipart/Related message as part of the SOAP binding. It does, however, require that you do the following:

1. Annotate the data that you are going to send as an attachment.  
  
You can annotate either your WSDL or the Java class that implements your data.
2. Enable the runtime's MTOM support.  
  
This can be done either programmatically or through configuration.
3. Develop a `DataHandler` for the data being passed as an attachment.



## Note

Developing `DataHandler`s is beyond the scope of this book.

## Annotating Data Types to use MTOM

---

### Overview

When defining a data type for passing along a block of binary data, such as an image file or a sound file, in WSDL you define the element for the data to be of type `xsd:base64Binary`. By default, any element of type `xsd:base64Binary` results in the generation of a `byte[]` which can be serialized using MTOM. However, the default behavior of the code generators does not take full advantage of the serialization.

In order to fully take advantage of MTOM you must add annotations to either your service's WSDL document or the JAXB class that implements the binary data structure. Adding the annotations to the WSDL document forces the code generators to generate streaming data handlers for the binary data. Annotating the JAXB class involves specifying the proper content types and may also involve changing the type specification of the field containing the binary data.

---

### WSDL first

Example 13, "Message for MTOM" shows a WSDL document for a Web service that uses a message which contains one string field, one integer field, and a binary field. The binary field is intended to carry a large image file, so it is not appropriate for sending along as part of a normal SOAP message.

### Example 13. Message for MTOM

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="XrayStorage"
  targetNamespace="http://mediStor.org/x-rays"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="http://mediStor.org/x-rays"
  xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
  xmlns:xsd1="http://mediStor.org/types/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <types>
    <schema targetNamespace="http://mediStor.org/types/"
      xmlns="http://www.w3.org/2001/XMLSchema">
      <complexType name="xRayType">
        <sequence>
          <element name="patientName" type="xsd:string" />
          <element name="patientNumber" type="xsd:int" />
          <element name="imageData" type="xsd:base64Binary" />
        </sequence>
      </complexType>
      <element name="xRay" type="xsd1:xRayType" />
    </schema>
  </types>
</definitions>
```

```

</types>

<message name="storRequest">
  <part name="record" element="xsd1:xRay"/>
</message>
<message name="storResponse">
  <part name="success" type="xsd:boolean"/>
</message>

<portType name="xRayStorage">
  <operation name="store">
    <input message="tns:storRequest" name="storRequest"/>
    <output message="tns:storResponse" name="storResponse"/>
  </operation>
</portType>

<binding name="xRayStorageSOAPBinding" type="tns:xRayStorage">
  <soap12:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="store">
    <soap12:operation soapAction="" style="document"/>
    <input name="storRequest">
      <soap12:body use="literal"/>
    </input>
    <output name="storResponse">
      <soap12:body use="literal"/>
    </output>
  </operation>
</binding>
...
</definitions>

```

If you wanted to use MTOM to send the binary part of the message as an optimized attachment you would need to add the `mime:expectedContentTypes` attribute to the element containing the binary data. This attribute is defined in the <http://www.w3.org/2005/05/xmlmime> namespace and specifies the MIME types that the element is expected to contain. You can specify a comma separated list of MIME types. The setting of this attribute will change how the code generators create the JAXB class for the data. For most MIME types, the code generator will create a `DataHandler`. Some MIME types, such as those for images, have defined mappings.



## Note

The MIME types are maintained by the Internet Assigned Numbers Authority (IANA) and described in detail in *Multipurpose Internet*

*Mail Extensions (MIME) Part One: Format of Internet Message Bodies*(<ftp://ftp.isi.edu/in-notes/rfc2045.txt>) and *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*(<ftp://ftp.isi.edu/in-notes/rfc2046.txt>)



## Tip

For most uses you would specify `application/octet-stream`.

Example 14, “Binary Data for MTOM” shows how you would modify `xRayType` from Example 13, “Message for MTOM” for using MTOM.

### Example 14. Binary Data for MTOM

```
...
<types>
  <schema targetNamespace="http://mediStor.org/types/"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:xmime="http://www.w3.org/2005/05/xmlmime">
    <complexType name="xRayType">
      <sequence>
        <element name="patientName" type="xsd:string" />
        <element name="patientNumber" type="xsd:int" />
        <element name="imageData" type="xsd:base64Binary"
          xmime:expectedContentTypes="application/octet-stream"/>
      </sequence>
    </complexType>
    <element name="xRay" type="xsd1:xRayType" />
  </schema>
</types>
...
```

The generated JAXB class generated for `xRayType` will no longer contain a `byte[]`. Instead the code generator will see the `xmime:expectedContentTypes` attribute and generate a `DataHandler` for the `imageData` field.



## Note

You do not need to change the `binding` element to use MTOM. The runtime will make the appropriate changes when the data is sent.

---

### Java first

If you are doing Java first development you can make your JAXB class MTOM ready by doing the following:

1. Make sure the field holding the binary data is a `DataHandler`.
2. Add the `@XmlMimeType()` annotation to the field containing the data you want to be streamed as an MTOM attachment.

Example 15, “JAXB Class for MTOM” shows a JAXB class annotated for using MTOM.

### Example 15. JAXB Class for MTOM

```
@XmlType
public class XRayType {
    protected String patientName;
    protected int patientNumber;
    @XmlMimeType("application/octet-stream")
    protected DataHandler imageData;
    ...
}
```

## Enabling MTOM

### Table of Contents

Using JAX-WS APIs .....	39
Using configuration .....	41

By default the FUSE Services Framework runtime does not enable MTOM support. It will send all binary data as either part of the normal SOAP message or as an unoptimized attachment. You can activate MTOM support either programmatically or through the use of configuration.

## Using JAX-WS APIs

Both service providers and consumers need to have the MTOM optimizations enabled. The JAX-WS APIs offer different mechanisms for each type of endpoint.

---

### Service provider

If you published your service provider using the JAX-WS APIs you enable the runtime's MTOM support as follows:

1. Get access to the `Endpoint` object for your published service.

The easiest way to get the `Endpoint` object is when you publish the endpoint. For more information see *Publishing a Service* in *Developing Applications Using JAX-WS*.

2. Get the SOAP binding from the `Endpoint` using its `getBinding()` method as shown in Example 16, "Getting the SOAP Binding from an Endpoint".

#### Example 16. Getting the SOAP Binding from an Endpoint

```
// Endpoint ep is declared previously
SOAPBinding binding = (SOAPBinding)ep.getBinding();
```

You must cast the returned binding object to a `SOAPBinding` object in order to access the MTOM property.

3. Set the bindings MTOM enabled property to `true` using the binding's `setMTOMEnabled()` method as shown in Example 17, "Setting a Service Provider's MTOM Enabled Property".

#### Example 17. Setting a Service Provider's MTOM Enabled Property

```
binding.setMTOMEnabled(true);
```

---

### Consumer

To MTOM enable a JAX-WS consumer you do the following:

1. Cast the consumer's proxy to a `BindingProvider` object.



## Tip

For information on getting a consumer proxy see *Developing a Consumer Without a WSDL Contract in Developing Applications Using JAX-WS* or *Developing a Consumer From a WSDL Contract in Developing Applications Using JAX-WS*.

2. Get the SOAP binding from the `BindingProvider` using its `getBinding()` method as shown in Example 18, “Getting a SOAP Binding from a `BindingProvider`”.

### Example 18. Getting a SOAP Binding from a `BindingProvider`

```
// BindingProvider bp declared previously
SOAPBinding binding = (SOAPBinding)bp.getBinding();
```

3. Set the binding's MTOM enabled property to `true` using the binding's `setMTOMEnabled()` method as shown in Example 19, “Setting a Consumer's MTOM Enabled Property”.

### Example 19. Setting a Consumer's MTOM Enabled Property

```
binding.setMTOMEnabled(true);
```

## Using configuration

---

### Overview

If you publish your service using XML, such as when deploying into a container, you can enable your endpoint's MTOM support in the endpoint's configuration file. For more information on configuring endpoints see *Configuring and Deploying Artix Java Runtime Endpoints*.

---

### Procedure

The MTOM property is set inside the `jaxws:endpoint` element for your endpoint. To enable MTOM do the following:

1. Add a `jaxws:property` child element to the endpoint's `jaxws:endpoint` element.
  2. Add a `entry` child element to the `jaxws:property` element.
  3. Set the `entry` element's `key` attribute to `mtom-enabled`.
  4. Set the `entry` element's `value` attribute to `true`.
- 

### Example

Example 20, "Configuration for Enabling MTOM" shows an endpoint that is MTOM enabled.

### Example 20. Configuration for Enabling MTOM

```
<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.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://cxf.apache.org/jaxws http://cxf.apache.org/schema/jaxws.xsd">
  <jaxws:endpoint id="xRayStorage"
    implementor="demo.spring.xRayStorImpl"
    address="http://localhost/xRayStorage">
    <jaxws:properties>
      <entry key="mtom-enabled" value="true"/>
    </jaxws:properties>
  </jaxws:endpoint>
</beans>
```



---

# Index

## W

wSDL2soap, 10, 20

## B

bindings

    SOAP with Attachments, 30

## M

mime:content, 30

    part, 31

    type, 31

mime:multipartRelated, 29

mime:part, 29, 30

    name attribute, 30

MTOM, 33

    enabling

        configuration, 41

        consumer, 39

        service provider, 39

    Java first, 36

    WSDL first, 34

## S

SOAP Message Transmission Optimization Mechanism,  
33

soap12:body

    parts, 24

soap12:header, 23

    encodingStyle, 24

    message, 23

    namespace, 24

    part, 23

    use, 23

soap:body

    parts, 13

soap:header, 13

    encodingStyle, 13

    message, 13

    namespace, 13

    part, 13

    use, 13

