MIME-S/MIME Developer's Guide

Version 10.15

October 2022

WEBMETHODS

This document applies to webMethods Integration Server 10.15 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: IS-MIME-DG-1015-20221015

Table of Contents

About this Guide

webMethods Integration Server provides built-in services that let you build secure MIME messages, transport them over the Internet, and extract information from MIME messages that are passed into the pipeline.

This guide is for users who want to use Software AG Designer to:

- Construct MIME messages.
- Secure MIME messages by digitally signing and encrypting them.
- Extract information from MIME messages such as header fields and content, decrypting this information if necessary.
- Transport MIME messages over the Internet.

Note:
This guide describes features and functionality that might or might not be available with your licensed version of webMethods Integration Server. For information about the licensed components for your installation, see the **Server > Licensing** page in the webMethods Integration Server Administrator.

Document Conventions

Convention	Description
Bold	Identifies elements on a screen.
Narrowfont	Identifies service names and locations in the format *folder.subfolder.service*, APIs, Java classes, methods, properties.
Italic	Identifies: Variables for which you must supply values specific to your own situation or environment. New terms the first time they occur in the text. References to other documentation sources.
`Monospace font`	Identifies: Text you must type in. Messages displayed by the system. Program code.
{ }	Indicates a set of choices from which you must choose one. Type only the information inside the curly braces. Do not type the { } symbols.
|	Separates two mutually exclusive choices in a syntax line. Type one of these choices. Do not type the | symbol.
[ ]	Indicates one or more options. Type only the information inside the square brackets. Do not type the [ ] symbols.

Convention	Description
...	Indicates that you can type multiple options of the same type. Type only the information. Do not type the ellipsis (...).

Online Information and Support

Product Documentation

You can find the product documentation on our documentation website at https://documentation.softwareag.com.

In addition, you can also access the cloud product documentation via https://www.softwareag.cloud. Navigate to the desired product and then, depending on your solution, go to "Developer Center", "User Center" or "Documentation".

Product Training

You can find helpful product training material on our Learning Portal at https://knowledge.softwareag.com.

Tech Community

You can collaborate with Software AG experts on our Tech Community website at https://techcommunity.softwareag.com. From here you can, for example:

- Browse through our vast knowledge base.
- Ask questions and find answers in our discussion forums.
- Get the latest Software AG news and announcements.
- Explore our communities.
- Go to our public GitHub and Docker repositories at https://github.com/softwareag and https://hub.docker.com/publishers/softwareag and discover additional Software AG resources.

Product Support

Support for Software AG products is provided to licensed customers via our Empower Portal at https://empower.softwareag.com. Many services on this portal require that you have an account. If you do not yet have one, you can request it at https://empower.softwareag.com/register. Once you have an account, you can, for example:

- Download products, updates and fixes.
- Search the Knowledge Center for technical information and tips.
- Subscribe to early warnings and critical alerts.

- Open and update support incidents.
- Add product feature requests.

Data Protection

Software AG products provide functionality with respect to processing of personal data according to the EU General Data Protection Regulation (GDPR). Where applicable, appropriate steps are documented in the respective administration documentation.

1 Overview of MIME and S/MIME Messages

Overview

This chapter explains what MIME and S/MIME message formats are and identifies their basic structures, describes how digital certificates and signatures work to identify the sender of a message, and explains how message encryption helps to ensure the privacy of a message.

What Is MIME?

Multipurpose Internet Mail Extensions (MIME) is a standard yet flexible message format that is used to represent messages for transmission over the Internet. The MIME extensions were added to the Simple Mail Transport Protocol (SMTP) to allow e-mail transmissions to carry more than simple, 7-bit, textual messages.

The MIME standards allow for the transmission of:

- Non-textual content such as images, audio clips, and other binary files
- Messages in character sets other than US-ASCII
- Multiple files in a single transmission

Although originally developed for the SMTP protocol, MIME can be used by other Internet technologies (such as HTTP) as a standard messaging format.

Basic Structure of a MIME Message

Like a standard mail message, a MIME message has two basic components: a set of *header fields* and a *body*.

A simple MIME message

Header Fields

Header fields provide information about the structure and encoding of a message. They consist of *name*:*value* pairs and appear at the top of the message. A MIME-compliant message must contain the "MIME-Version" header field.

Besides the MIME-Version header field, most messages have additional fields that supply information to the agent, transport, or application that will convey or consume the message. For example, when a MIME message carries anything other than plain, US-ASCII text, it must include the "Content-Type" header field. Messages that are routed over SMTP will also have the "Date," "To," and "From" header fields.

A message may also contain custom header fields that are specific to a particular agent or application. Such application-specific header fields must be prefixed with the characters "X-" to distinguish them from the standard header fields defined by the MIME and/or transport protocols.

This chapter does not attempt to describe the purpose or use of individual header fields. However, to use MIME effectively, you will need to understand which header fields your solution requires and know how to set them or interpret them correctly. For information about header fields, see the following references.

Reference	URL
RFC 2076 – Common Internet Message Headers	http://www.imc.org/rfc2076
RFC 822 – Standard Format of Internet Text Messages	http://www.imc.org/rfc822
RFC 2045 – Multipurpose Internet Mail Extensions	http://www.imc.org/rfc2045
RFC 2046 – MIME Media Types	http://www.imc.org/rfc204
RFC 2047 – MIME Message Header Extensions for Non-ASCII Text	http://www.imc.org/rfc2047
RFC 2048 – MIME Registration Procedures	http://www.imc.org/rfc2048
RFC 2049 – MIME Conformance Criteria	http://www.imc.org/rfc2049

The Body

The body of a MIME message contains the actual content of the message. It is separated from the last header field by a blank line: a two-byte sequence consisting of an ASCII carriage return (CR) and an ASCII linefeed (LF) on a line by itself.

The message body can contain any sequence of data, including additional MIME messages. It is sometimes referred to as the *payload*. When you send an e-mail message, the body of your letter resides in the body of a MIME message. Similarly, when you attach a file to an e-mail message, the content of the file is carried in the body of a MIME message.

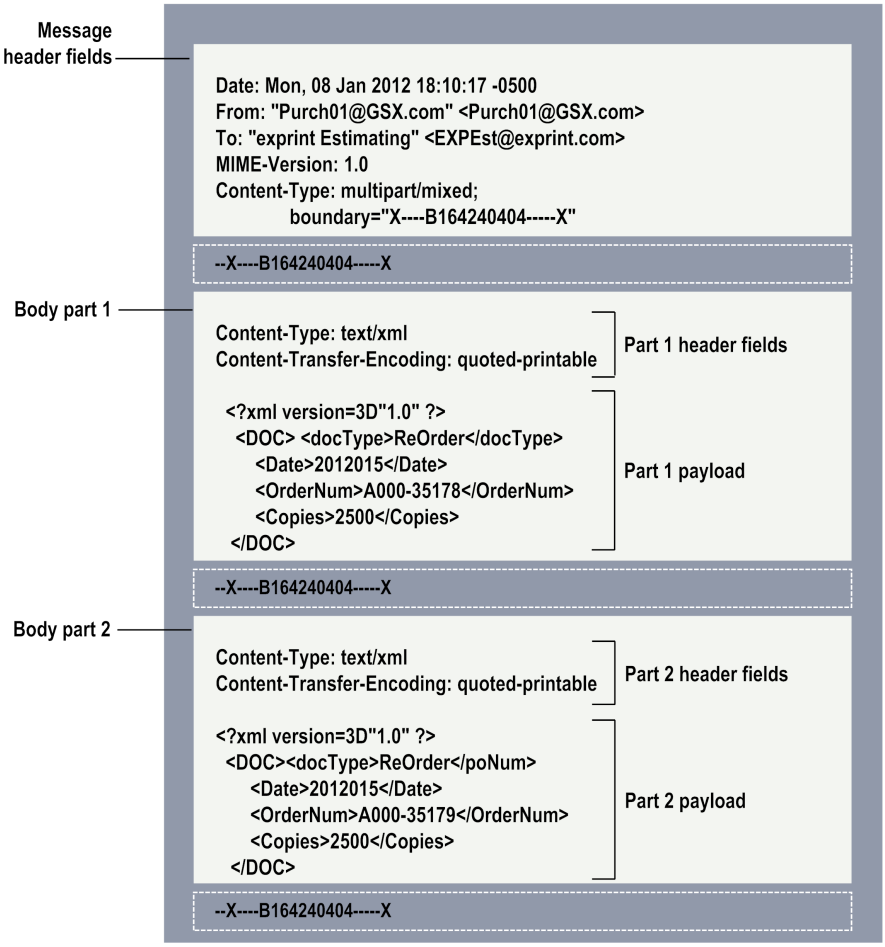
Multipart MIME Messages

One of the key reasons for the development of MIME was to allow the transmission of multiple files (payloads) in a single message. When a MIME message contains multiple payloads, it has

two kinds of header fields: *message headers*, which appear only at the beginning of the message, and *part headers*, which appear at the beginning of each body part.

Message headers apply to the entire message. Part headers apply only to the body part in which they appear. The following example shows a MIME message with two body parts.

A multipart MIME message



If a MIME message has more than one payload, its Content-Type header field must be set to a *multipart* content type (for example, `Content-Type:multipart/mixed` or `Content-Type:multipart/alternative`), and it must declare a boundary separator. The boundary separator is a string that delimits body parts. It must appear before and after each part in the message. (In the example above, the string X----B164240404-----X is the boundary separator.)

Note:
You may have noticed that the string separating the body parts in the preceding example includes a few extra characters that are not part of the separator string declared in the Content-Type header field. This is because the MIME format requires that two dash characters precede each separator in the message and two dash characters follow the last separator string in the message.

When you build a multipart message with webMethods Integration Server, it automatically sets the Content-Type header to "multipart," declares the separator string, and inserts the boundary separators for you.

What Is S/MIME?

Secure Multipurpose Internet Mail Extensions (S/MIME) is a standard message format that allows MIME messages to be exchanged securely between parties over the Internet. It provides two security mechanisms—*digital signatures* and *encryption*—based on RSA technology and the Public Key Infrastructure (PKI).

Digital Certificates

PKI employs a system of credentials known as digital certificates, or electronic documents that represent and identify individual users. A digital certificate is like an electronic identification card. It positively identifies a particular individual, organization, or application.

Besides providing information about the owner of the certificate (name, organization, e-mail address, and so forth), a digital certificate holds the owner's public key. Under public/private key technology, a certificate owner has two keys. Parties that want to exchange messages securely with the certificate owner use the *public key* published on the owner's certificate. Transmissions secured with a public key can only be successfully processed with the corresponding *private key*—a secret key that only the certificate owner has.

Digital certificates are issued and signed by Certificate Authorities (CAs). A CA is similar to a notary public. Its signature vouches for the identity of the individual or organization named on the certificate and attests to the validity of the public key. It also "seals" the certificate with a digital signature, which certifies the certificate's contents and prevents it from ever being altered undetected. VeriSign and Entrust are examples of public CAs. They are considered "root-level" entities. Other intermediaries, such as financial institutions, are also permitted to issue certificates under the authority of a root CA.

You cannot verify the authenticity of a certificate without having the certificate of the CA that issued it. If the issuing CA is an intermediary, you must also have the certificate of its CA. The set of certificates required to trace the authenticity of a certificate back to a trusted CA is called a *certificate chain*.

Note:
To authenticate a certificate, some recipients require a *complete* certificate chain–one that extends all the way back to a root-level CA–while others are satisfied with a partial chain that goes back to a specific intermediary. Always submit a complete chain unless you know for certain that the recipient accepts partial chains.

Digital Signatures

A digital signature is a special block of data affixed to a message that assures the identity of the sender and the integrity of the message.

A digital signature secures a message in several ways. First, it contains the sender's digital certificate. This allows a recipient to identify the sender and determine whether the sender is a trusted and authorized party. In this way, digital signatures support the identification and authorization processes.

Second, a digital signature assures a recipient that the owner of the enclosed certificate sent the message. A digital signature is produced using the sender's private key. If a recipient can successfully "decode" the signature with the public key from the sender's certificate, the recipient is positively assured that the message is from the person or organization identified on that certificate. This characteristic provides both authentication (the sending party is who it claims to be) and nonrepudiation (the sending party cannot deny issuing the message).

Finally, a digital signature assures the integrity of the message with a *message digest*—a hash code that is mathematically derived from the message itself. When a recipient opens a signed message, it recalculates the hash code and compares its result to the original hash code in the signature. If the values don't match, the recipient knows that the message was deliberately or inadvertently altered after it was signed.

Explicit and Implicit Signatures

There are two types of digital signatures: *explicit signatures* and *implicit signatures*.

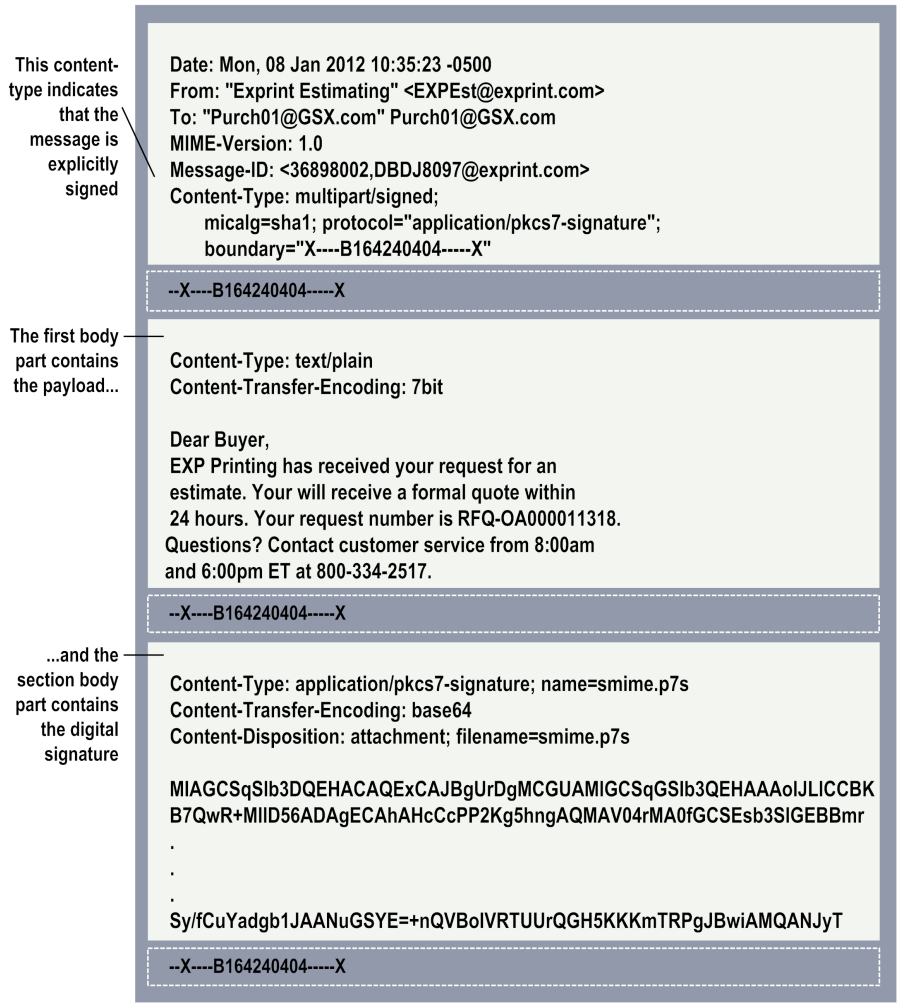
An explicit signature is appended as a separate body part to the end of a MIME message. This format is sometimes referred to as the *clear-signing* or *detached-signature* format. When a MIME entity contains an explicitly signed message, its Content-Type header field is set to "multipart/signed." This field also specifies the protocol and message-integrity algorithm (micalg) used to produce the signature.

Integration Server uses the "pkcs7-signature" protocol and the "SHA-1" integrity algorithm.

Note: Integration Server automatically sets the Content-Type header field when you sign a message using the S/MIME services. Your service does not need to do this.

The following is an example of an explicitly signed MIME message. Notice that the message has two body parts: the first part contains the payload; the second part contains the signature.

An explicitly signed message



A message can also be implicitly signed. When you use this technique, the message is encoded within the signature block, thus preventing the message from being extracted or read unless the signature is processed by a PKCS-enabled recipient. For this reason, explicit signatures are preferred because they also make the message available to non-PKCS recipients.

When a MIME entity contains an implicitly signed message, its Content-Type header field is set to “application/pkcs7-mime.”

The following is an example of a text message that has been implicitly signed. As you can see, the text of the message is not visible.

An implicitly signed message

Encryption

Encryption is a way to ensure privacy by assuring that a message can be read only by the intended recipient.

Encryption is performed using a pair of keys. The sending party encrypts the message using the recipient's public key. The recipient decrypts the message with its private key. Since the owner of the public key is the only one in possession of the private key, only the owner can successfully decrypt the message.

Integration Server supports RC2, TripleDES, and DES encryption algorithms. RC2 lets you specify a key length of 40, 64, or 128. TripleDES uses a key length of 192. DES uses a key length of 64 (in US versions of the product) or 40 (in non-US versions of the product).

The following is an example of an encrypted message. Note that its Content-Type header field is set to "application/pkcs7-mime" (required for encrypted messages) and that the payload contains the encrypted message.

Note: Integration Server automatically sets the Content-Type header field to the appropriate value when you encrypt a MIME message using the S/MIME services. Your service does not need to do this.

An encrypted message

Note:
Although encryption protects a message from being read by an unintended party, it does not assure message integrity, nor does it provide authentication or nonrepudiation. These qualities are guaranteed by digital signatures.

To encrypt a message, you must have the intended recipient's certificate because it contains the public key you use to perform the encryption.

Most sites simply contact the parties with whom they want to exchange encrypted messages and request copies of their certificates (the other parties might e-mail their certificates to you, for example). Then, they store the certificates in their file system, a database, or a special repository for security information.

It does not make any difference where you maintain the certificates of the parties with whom you want to exchange encrypted messages, as long as the certificates are in X.509 format and can be retrieved by Integration Server at run time.

The MIME and S/MIME Services

The MIME and S/MIME services allow you to build secure MIME objects that you can send over the Internet. They also allow you to extract information from MIME messages that are placed in the pipeline, decrypting that information when necessary.

Services Used to Construct MIME and S/MIME Messages

The following table lists services that you use to create MIME messages and optionally secure them using a digital signature and/or encryption. For information about how you use these services to create various kinds of MIME messages, see "Building MIME and S/MIME Messages" on page 21.

Service	Description
createMimeData	Creates an empty MIME object, which you use to compose a MIME message.
addMimeHeader	Adds one or more header fields to a MIME object.
addBodyPart	Adds a body part (headers and content) to a specified MIME object.
getEnvelopeStream	Generates a MIME message from a MIME object.
createSignedData	Digitally signs a MIME message.
createEncryptedData	Encrypts a MIME message.
createSignedAndEncryptedData	Digitally signs a MIME message and then encrypts it.

Services Used to Extract Data from MIME and S/MIME Messages

The following table lists services that you use to extract data from a MIME message. For information about how you use these services to decrypt, authenticate, and extract information from a MIME message, see "Extracting Data from MIME and S/MIME Messages" on page 39.

Service	Description
createMimeData	Produces a MIME object from a MIME message stream.
getMimeHeader	Retrieves the header fields from a MIME object.
getContentType	Retrieves the value of the Content-Type header field from a MIME object.
getNumParts	Reports the number of body parts in a MIME object.
getBodyPartContent	Retrieves the content from a specified body part in a MIME object.
getBodyPartHeader	Retrieves the header fields from a specified body part in a MIME object.
processEncryptedData	Decrypts the contents of an S/MIME object.
processSignedData	Authenticates the digitally signed contents of a specified S/MIME object.

MIME Messages, MIME Entities, and MIME Objects

In this book, the term *MIME message* refers to a complete, top-level MIME message that is made up of a set of message header fields (including the mandatory MIME Version header) and a body.

The term *MIME entity* refers to any block of data composed of header fields and a body. It can mean either a complete MIME message or a single body part within a multipart message.

Most MIME services provided by Integration Server do not operate directly on a MIME message or a MIME entity. Instead, they operate on a *MIME object*. A MIME object is a parsed representation of a MIME message that allows webMethods services to add and/or retrieve the message's constituent elements (header fields and content). By convention, the variable that holds a MIME object is called *mimeData*.

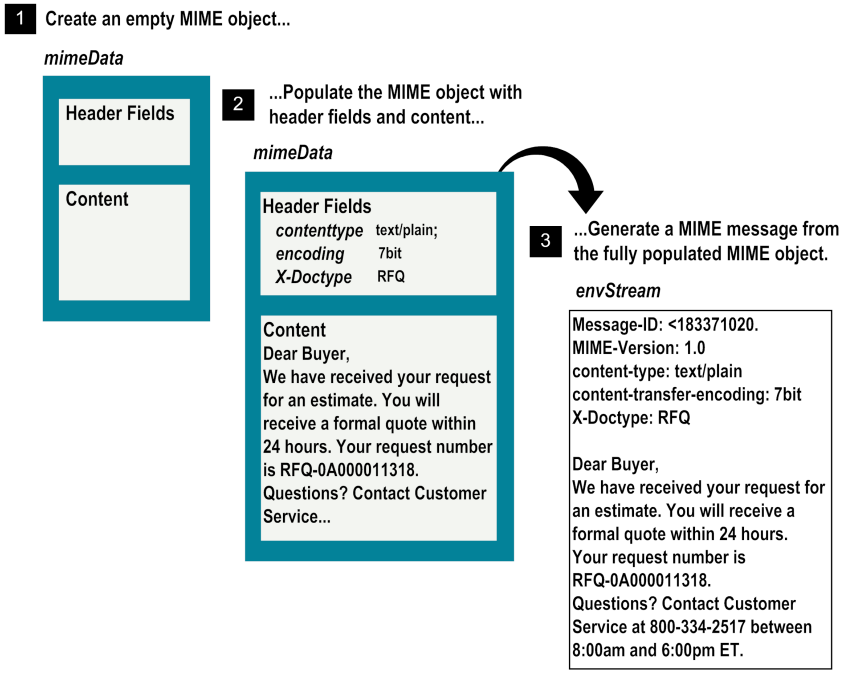
2 Building MIME and S/MIME Messages

Overview

To construct a MIME message with Software AG, you first create an "empty" MIME object and then populate the object with the appropriate header fields and content. After putting the required data into the MIME object, you generate a MIME message from the MIME object.

The following diagram illustrates this process.

Constructing a MIME message



After you create a MIME message, you can digitally sign it—to identify that the message is being sent by a trusted source—and/or encrypt it.

Note:
If the payload size exceeds the limit set by the `watt.server.mime.largeDataThreshold` parameter (that is set to 25MB by default), the pub.mime:getEnvelopeStream service returns a *mimeMessage* instead of an *envStream* and the pub.mime:addBodyPart service stores data in a temporary location on the disk (Tspace) instead of memory (RAM).

Creating a MIME Message

To create a MIME message, you use services from the pub.mime folder to create an empty MIME object, populate the MIME object with header fields and content, and generate the finished MIME message.

Important:
The MIME object is an IData object whose contents you can examine during testing and debugging. However, the internal structure of this object is subject to change in future versions

of webMethods. *Do not* explicitly set or map data to the elements in the MIME object with the pipeline editor. To add content to a MIME object, you must use *only* the MIME services that Integration Server provides for that purpose.

How to Create a MIME Message

The following procedure describes the general steps you take to create a MIME message.

1. Create an empty MIME object using pub.mime:createMimeData. You do not need to pass any input parameters to this service.

   This service returns an empty MIME object named *mimeData*.

2. Add application-specific message headers with pub.mime:addMimeHeader. If your message requires application-specific (for example, "X- type" fields) or transport-specific message headers (for example, "To" and "From" header fields), use addMimeHeader to specify them. This service takes as input a document called `mimeHeader`, whose fields and values specify message header field names and values.

   For example, this *mimeHeader* document...

Name	Values
To	"xprint mEstimating" <EXPEst@exprint.com>
From	"Purch01@GSX.com" <Purch01@GSX.com>
X-Doctype	RFQ
X-Severity	5

   ...would produce the following message header fields:

```
To: "xprint Estimating" <EXPEst@exprint.com>
From: "Purch01@GSX.com" <Purch01@GSX.com>
X-Doctype: RFQ
X-Severity: 5
```

   Note that you *do not* need to explicitly set the following message headers:

```
Message-ID
MIME-Version
```

   These headers are automatically generated by the service that produces the finished MIME message. If you explicitly set these fields in *mimeHeader*, they will be overwritten when the MIME message is generated.

   You may set message headers before or after adding content (performing the next step 3, below). Either order is permitted, as long as you set them before you generate the finished MIME message.

   Tip:

Instead of using addMimeHeader to add message headers, you may alternatively pass a *mimeHeader* document to createMimeData when you create the MIME object.

Besides a *mimeHeader* document, you must pass to addMimeHeader the *mimeData* object produced in the previous step 1.

The addMimeHeader service does not return an output value. It simply updates the *mimeData* object that you pass to it.

3. Add one or more body parts with the pub.mime:addBodyPart service. This service adds a single body part (both header fields and content) to the MIME object. To add multiple body parts, execute addBodyPart once for each part that you want to add. In the finished message, body parts appear in the order in which you add them—the first body part you add will be the first body part in the message.

   Besides the *mimeData* object that you produced in step 1, addBodyPart takes three other parameters: *content*, *contenttype*, and *encoding*.

   - *content* is an InputStream containing the message content (the payload). Before invoking addBodyPart, your solution must acquire or generate this content and place it in the pipeline as an InputStream.

     The way in which you acquire the content of your message depends on your particular solution. For example, you might acquire it from a file or from a back-end system or manufacture it with a custom-built service. Regardless of how you acquire your content, keep the following points in mind:

     - Your content must exist as an InputStream in the pipeline. If it exists in some other form—for example, a String or a byte[]—you must convert it to an InputStream before adding it to the MIME object.
     - The InputStream should contain only the body of the message (the payload). Do not put header fields in the InputStream. To specify header fields, use the *contenttype*, *encoding*, *description*, and *mimeHeader* input parameters.

     Note:
     If your InputStream already contains header fields, you can set the *isEnvStream* parameter to "true" to tell addBodyPart to pull the header fields out of the InputStream before adding it to the MIME object. For additional information about using the *isEnvStream* parameter, see the addBodyPart description in the *webMethods Integration Server Built-In Services Reference*.

     - Do not encode the content before adding it to the MIME object—simply add it in its original form. If you want to encode the content for transport, set the *encoding* parameter (see below).

   - *contenttype* is a String specifying the value of the entity's Content-Type header field. Besides type and subtype, be sure to include any parameters that the header field requires as shown in the following example:

     ```
     text/plain;charset=UTF8
     ```

For a description of standard content types, see *RFC 2046—MIME Media Types* at http://www.imc.org/rfc2046.

Be aware that when you create a single-part message, the value you specify in *contenttype* is assigned to the Content-Type header for the entire MIME message. In this case, *contenttype* will override any value you may have previously set in *mimeHeader* (using the addMimeHeader service, for example.)

When you create a multipart message, the value you specify in *contenttype* is assigned to the Content-Type header for the body part. The Content-Type header for the entire MIME message is automatically set to multipart/mixed (or to multipart/*subType* if the *subType* parameter was specified when the MIME object was created.)

- *encoding* is a String specifying the value of the entity's Content-Transfer-Encoding header field. This field also specifies the scheme in which you want the entity's content encoded. If you set *encoding* to "base64," for example, the getEnvelopeStream service will base64-encode the data in *content* when it generates the finished MIME message.

  encoding must be one of the following values:

Value	Description
7bit	Specifies that *content* is 7-bit, line-oriented text that needs no encoding. Use this value when *content* contains lines of 7-bit, US-ASCII text (no octets with decimal values greater than 127; no NULs).
8bit	Specifies that *content* is 8-bit, line-oriented text that needs no encoding. Use this value when *content* contains lines of 8-bit text (octets with decimal values greater than 127; no NULs). **Note:** This encoding value is not recommended for messages that will be transported via SMTP over the Internet, because intervening servers that cannot accommodate 8-bit text may alter your content. To safely transport 8-bit text, use quoted-printable encoding instead.
binary	Specifies that *content* contains binary information that needs no encoding. Use this value when *content* contains an arbitrary sequence of octets (binary data). **Note:** This encoding value is not recommended for messages that will be transported via SMTP over the Internet, because intervening servers that cannot accommodate binary data may alter your content. To safely transport binary data, use base64 encoding instead.
quoted-printable	Specifies that *content* contains 7- or 8-bit, line-oriented text that you want to encode using the quoted printable encoding scheme.

Value	Description
base64	Specifies that *content* contains an arbitrary sequence of octets (binary data) that you want to encode using the base64 encoding scheme.
uuencode	Specifies that *content* contains an arbitrary sequence of octets that you want to encode using the uuencode encoding scheme.

Be aware that when you create a single-part message the value you specify in *encoding* is assigned to the Content-Transfer-Encoding header for the entire MIME message. This value will override any value you may have previously set in *mimeHeader* (using the addMimeHeader service, for example).

When you create a multipart message, the value you specify in *encoding* is assigned to the Content-Transfer-Encoding header for the body part. The Content-Transfer-Encoding header in *mimeHeader*, if present, specifies the encoding for the entire MIME message. If Content-Transfer-Encoding is not specified in *mimeHeader*, or if the specified value is not valid for a multipart message, this header defaults to 7bit encoding. (7bit, 8bit, and binary are the only encoding values valid for multipart messages.)

Note:
Besides the *content*, *contenttype*, and *encoding*, parameters described above, the addBodyPart service has a few other optional parameters you can use. For information about these parameters, see the addBodyPart description in the *webMethods Integration Server Built-In Services Reference*.

The addBodyPart service does not return an output value. It simply updates the *mimeData* object that you pass to it.

4. Generate the finished MIME message with the pub.mime:getEnvelopeStream service. After you finish populating the MIME object, invoke getEnvelopeStream to generate a MIME message. This service takes the populated mimeData object and produces an InputStream or a MimeMessage, containing the finished MIME message.

   When getEnvelopeStream generates a MIME message, it does the following:

   - Generates the Message-ID, MIME-Version, Content-Type, and Content-Transfer-Encoding message headers and inserts them at the top of the message.
   - Sets the Content-Type header to "multipart," generates a boundary string, and inserts it between body parts if *mimeData* contains multiple body parts.

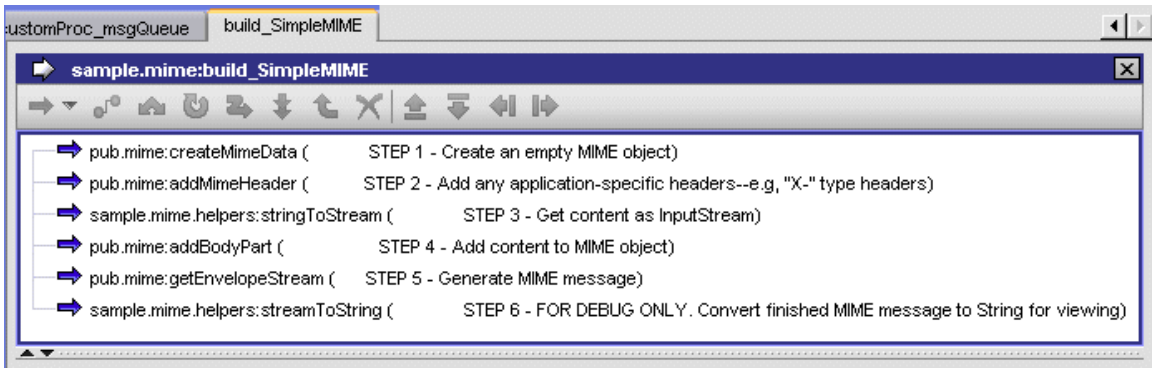
     Note:
     If mimeData contains a single body part, getEnvelopeStream will, by default, create an ordinary, single-part message. Some solutions, however, want a "multipart" message even if the message contains only a single body part. If your solution requires this structure, you can use the createMultipart parameter to tell getEnvelopeStream to generate a multipart message regardless of the number of body parts it finds in mimeData.

   - Encodes the content for each body part according to its *encoding* value.

Example—Creating a Single-Part MIME Message

The following flow service creates a single-part MIME message that contains a simple text message.

Flow service that creates a simple MIME message



Step	Description
1	This step creates an empty MIME object. It does not take any inputs. It puts an empty MIME object named *mimeData* in the pipeline.
2	This step adds two application-specific message headers in the MIME object. If you view the pipeline, you will see that the *mimeHeader* input variable is set as follows:

Name	Value
X-DocType	alert
X-Severity	9

Step	Description
3	This step generates the content of the message. This example uses a custom Java service to convert a String containing the following text to an InputStream:

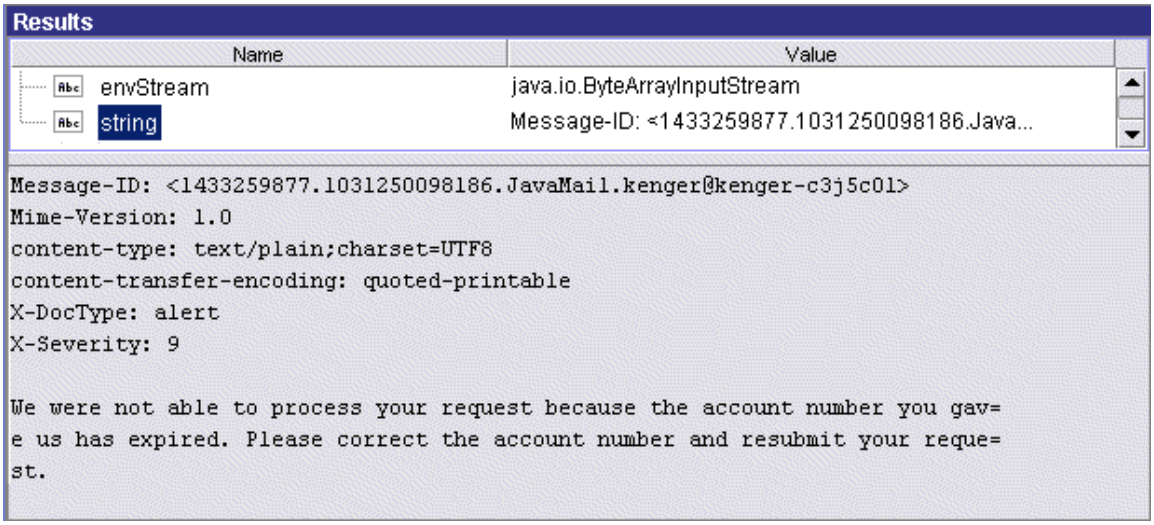
```
We were not able to process your request because the account
number you gave us has expired. Please correct the account number
and resubmit your request
```

In practice, you are more likely to acquire your content from a file, the network, or a back-end system.

Step	Description
4	This step adds the content produced by step 3 to the *mimeData* object. If you view the pipeline, you will note that the *stream* output variable from step 3 is linked to this step's *content* input variable. Because *content* contains a simple text message, the *contenttype* and *encoding* parameters are set as follows:

Parameter	Value
contenttype	text/plain;charset=UTF8
encoding	quoted-printable

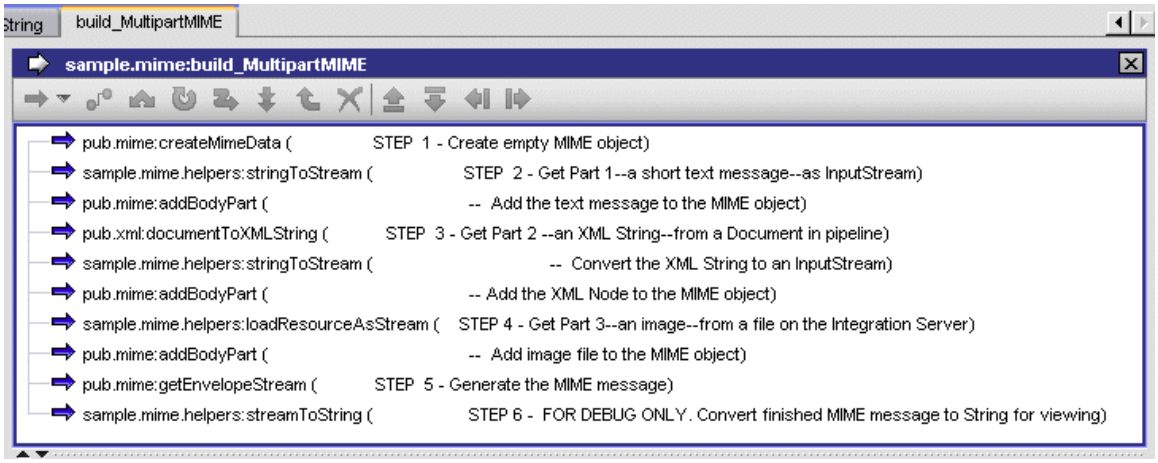
Step	Description
	isEnvStream is set to "no" because the payload is not a MIME entity.
5	This step generates the finished MIME message. It takes the *mimeData* object that was populated in steps 2 and 4 and produces an InputStream or a MimeMessage that contains the MIME message. At this point, you could pass an *envStream* or *mimeMessage* to any process that expects a MIME message as input.
6	Because you cannot view an InputStream, this example includes a step that converts the *envStream* to a String so you can examine the finished message with Designer. This technique is useful for testing and debugging. If you examine the contents of *string* on the Service Result view, you will see a MIME message similar to the following:



Example—Creating a Multipart MIME Message

The following flow service creates a multipart MIME message that contains three parts: a simple text message, an XML document, and an image file. The steps you use to create a multipart message are essentially the same as the ones you use to create a single-part MIME message—the only difference is that you execute addBodyPart multiple times.

Flow service that creates a multipart MIME message



Step	Description

1 This step creates an empty MIME object. It does not take any inputs. It puts an empty MIME object called *mimeData* in the pipeline.

2 This step generates the content of the message and adds it to the *mimeData* object. If you view the pipeline for the addBodyPart service in this step, you will see that the *stream* output variable generated by the stringToStream service is linked to the *content* input variable. Because *content* contains a simple text message, the *contenttype* and *encoding* parameters are set as follows:

Parameter	Value
contenttype	text/plain;charset=UTF8
encoding	quoted-printable

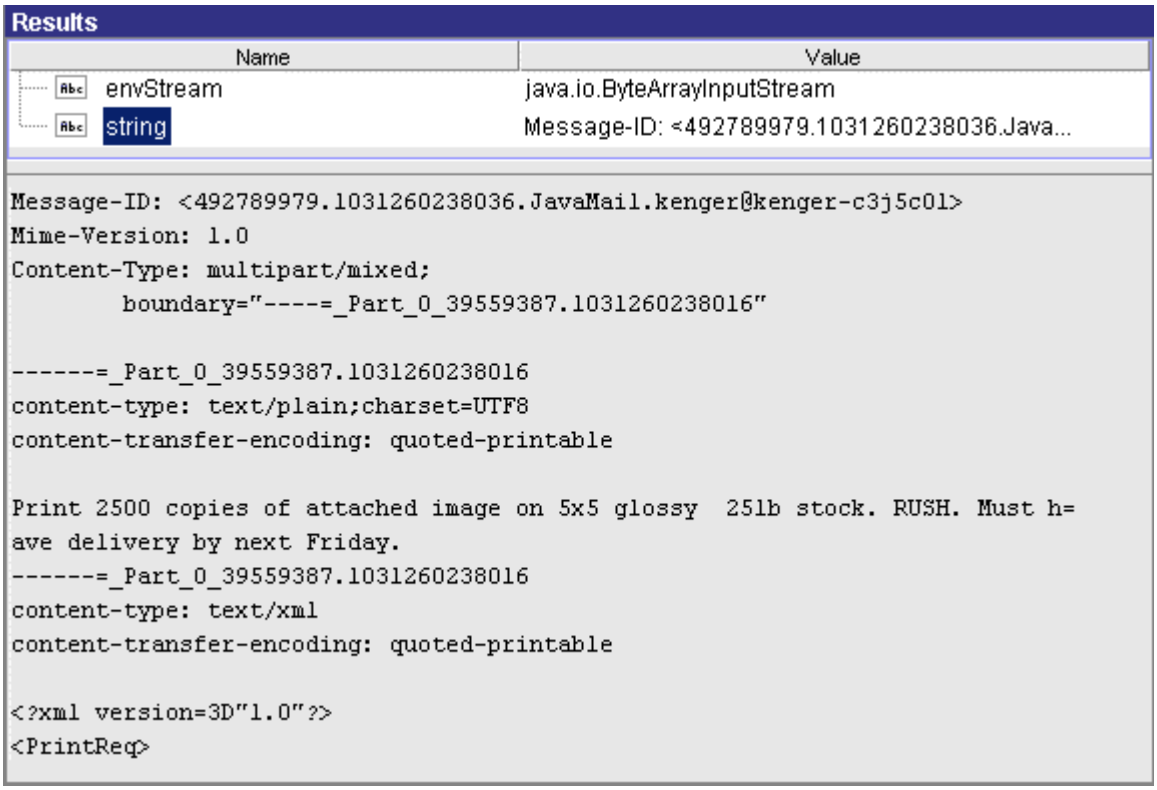
3 This step creates an XML document from a document (IData object) in the pipeline, converts that XML document to an InputStream, and then adds the InputStream to the *mimeData* object. Because *content* contains an XML document, the *contenttype* and *encoding* parameters are set as follows:

Parameter	Value
contenttype	text/xml
encoding	quoted-printable

4 This step gets an image file from disk and adds it to the *mimeData* object. Because the file is retrieved as an InputStream, it can be linked directly to the *mimeData* object. In this case, *content* is an image file (binary data), so the *contenttype* and *encoding* parameters are set as follows:

Parameter	Value
contenttype	image/gif;name="b2b.gif"
encoding	base64

Step	Description
5	This step generates the finished MIME message. It takes the *mimeData* object populated in steps 2–4 and produces an InputStream or a MimeMessage that contains the multipart MIME message. At this point, you could pass an *envStream* or *mimeMessage* to any process that expects a MIME message as input.
6	Because you cannot view an InputStream, this example includes a step that converts *envStream* to a String so you can examine the finished message with Designer. This technique is useful for testing and debugging. If you examine the contents of *string* on the Service Result view, you will see a MIME message similar to the following:



Points to keep in mind when building multipart MIME messages:

- By default, the Content-Type header field is set to "multipart/mixed." If you want to use a different subtype, set the *subtype* parameter when you invoke createMimeData.
- Body parts appear in the message in the order in which you add them to the MIME object—the first part you add appears first in the message.
- If you set message headers (for example, using addMimeHeader) before you add body parts, those header fields will also be inserted into each body part. To prevent this, drop the *mimeHeader* variable from the pipeline before you perform an addBodyPart step or execute the addMimeHeader step after adding the message's body parts.

Signing a MIME Message

To digitally sign a MIME message you must have a keystore that contains the signer's private key and an associated certificate chain. If you know that the recipient trusts an intermediate CA in your chain, the keystore can contain a partial chain that extends back to that CA. However, if you are not sure which CA the recipient trusts, the keystore should contain a complete chain.

Note:
You are not required to have the signer's certificate chain to sign a message; however, if you omit the chain, the recipient must produce the certificate chain when it receives the message. If you do not supply the signer's certificate chain, and the recipient does not have a local copy of it, the signature verification process will fail. By including the certificate chain with a signature, you ensure that the recipient will be able to process the signature.

Important:
You can sign only those messages that are within the threshold value specified by the `watt.server.mime.largeDataThreshold` configuration parameter.

How to Create a Signed S/MIME Message

The following procedure describes the general steps you take to create a signed S/MIME message.

Important:
If you want to create a signed and encrypted MIME message, use the special service that Integration Server provides for this purpose. For more information, see "Signing and Encrypting a MIME Message" on page 36.

Important:
You can sign only those messages that are within the threshold value specified by the `watt.server.mime.largeDataThreshold` configuration parameter.

Before you begin, you must have a keystore alias and a password for at least one key alias. The credentials provided by the keystore alias and the key alias are used to sign the message. If you cannot locate these credentials or do not have direct access to them, consult your Integration Server Administrator.

1. Create an InputStream that contains the MIME message that you want to sign. Use the procedure outlined in "How to Create a MIME Message" on page 23 to create the MIME message.

2. Pass your signing credentials to the pub.smime.keystore:createSignedData service. This service takes an InputStream containing a MIME message and signs it using the private key and the certificates pointed by the keystore alias and key alias that you provide. It produces an InputStream containing the signed message.

Example—Signing a MIME Message

The following flow service signs a single-part MIME message. To sign a MIME message, you must provide the alias of the keystore that contains the signing key, and the alias of the private key to use for signing. When you run the service from Designer, it will prompt you for the following:

Input Parameter	Description
signersKeyStoreAlias	**String** Alias of the keystore.
signersKeyAlias	**String** Alias of the private key of interest in the keystore.

Note:
This example is only for those messages that are within the threshold value specified by the `watt.server.mime.largeDataThreshold` configuration parameter.

Flow service that signs a MIME message

Step	Description
1	This step creates a MIME message containing a simple text message. It produces an InputStream (*envStream*) that contains the MIME message that will be signed.
2	This step generates the signed MIME message. It takes the InputStream from step 1 and the credentials specified in *signersKeyStoreAlias* and *signersKeyAlias* and produces an InputStream called *SMimeEnvStream* that contains the signed message.
3	Because you cannot view the contents of an InputStream, this example includes a step that converts *SMimeEnvStream* to a String so you can examine the finished message with Designer. This technique is useful for testing and debugging. If you examine the contents of *string* on the Service Result view, you will see a signed S/MIME message similar to the following. Note that this example creates an explicitly signed message—the message is in one body part and the digital signature is in another.

Step	Description

Results

Name	Value
SMimeEnvStream	java.io.ByteArrayInputStream
string	Message-ID: <1192584370.1031261087878.Java...

```
Content-Type: multipart/signed; micalg=sha1;
protocol="application/pkcs7-signature";
        boundary="----=_Part_0_1675076786.1031261087828"
X-DocType: alert
X-Severity: 9

------=_Part_0_1675076786.1031261087828
Content-Type: text/plain;charset=UTF8
Content-Transfer-Encoding: quoted-printable

We were not able to process your request because the account number you gav=
e us has expired. Please correct the account number and resubmit your reque=
st.
------=_Part_0_1675076786.1031261087828
Content-Type: application/pkcs7-signature; name=smime.p7s
Content-Transfer-Encoding: base64
Content-Disposition: attachment; filename=smime.p7s

MIAGCSqGSIb3DQEHAqCAMIACAQExCzAJBgUrDgMCGgUAMIAGCSqGSIb3DQEHAQAAMYICEzCCAg8C
AQEwgbgwgbIxCzAJBgNVBAYTAlVTMQswCQYDVQQIEwJWQTEQMA4GA1UEBxMHRmFpcmZheDEYMBYG
A1UEChMPd2ViTWV0aG9kcyBJbmMuMSUwIwYDVQQLExxDZXJ0aWZpY2F0ZSBBdXRob3JpdHkgT2Zm
aWNlMRswGQYDVQQDExJ3ZWJNZXRob2RzIFJvb3QgQ0ExJjAkBgkqhkiG9w0BCQEWF3NlY3VyaXR5
QHdlYmlldGhvZHMuY29tAgEBMAkGBSsOAwIaBQCggbEwGAYJKoZIhvcNAQkDMQsGCSqGSIb3DQEH
ATAcBgkqhkiG9w0BCQUxDxcNMDIwOTAlMjEyNDQ3WjAjBgkqhkiG9w0BCQQxFgQU2twlXYyokZJi
NlXLcij3PHb53J8wUgYJKoZIhvcNAQkPMUUwQzAKBggqhkiG9w0DBzAOBggqhkiG9w0DAgICAIAw
DQYIKoZIhvcNAwICAUAwDQYIKoZIhvcNAwICASgwBwYFKw4DAgcwDQYJKoZIhvcNAQEBBQAEgYAP
4fle5IIW25wHukyxOP3FKHVsHhPkUhOnm4tC5OU+mjswgX+QlMHYXp9W6YVVOYTt42xZqRxLpjDI
nI2J9bSrhPcBNvtnlVPMe63RIBlGSQmRPKCn7b3xCp7WjgNM7ceJxKL3mrQdIRnrc+MBBeRMAp+6
W++vfZOTxi/OaJjl7wAAAAAAAA==
------=_Part_0_1675076786.1031261087828--
```

Encrypting a MIME Message

To encrypt a MIME message you must have the recipient's certificate. It contains the public key required to encrypt the message. To obtain this certificate, ask the parties with whom you want to exchange encrypted messages to send you their X.509 digital certificate in DER file format. For example, you might ask a party to e-mail the .DER file to you or transmit it in a special "certificates only" MIME message, which you can read using the pub.smime:processCertsOnlyData service.

After receiving the certificate, store it in a location (file system, database, or other repository) where it can be retrieved by Integration Server at run time. Once you have access to the recipient's

certificate, you can generate encrypted MIME messages that only the owner of the certificate can read.

Important:
You can encrypt only those messages that are within the threshold value specified by the `watt.server.mime.largeDataThreshold` configuration parameter.

How to Create an Encrypted S/MIME Message

The following procedure describes the general steps you take to create an encrypted S/MIME message.

Important:
If you want to create a signed and encrypted MIME message, use the special service that Integration Server provides for this purpose. For instructions, see "Signing and Encrypting a MIME Message" on page 36.

1. Create an InputStream containing the MIME message that you want to encrypt. You can use the procedure outlined in "How to Create a MIME Message" on page 23 to create the MIME message.

2. Fetch the recipient's certificate as a byte[]. If the message will be sent to multiple recipients, fetch the certificate of each recipient. Load the certificates into a list (a one-dimensional array) of byte[] such that each element in the list holds the certificate of single recipient.

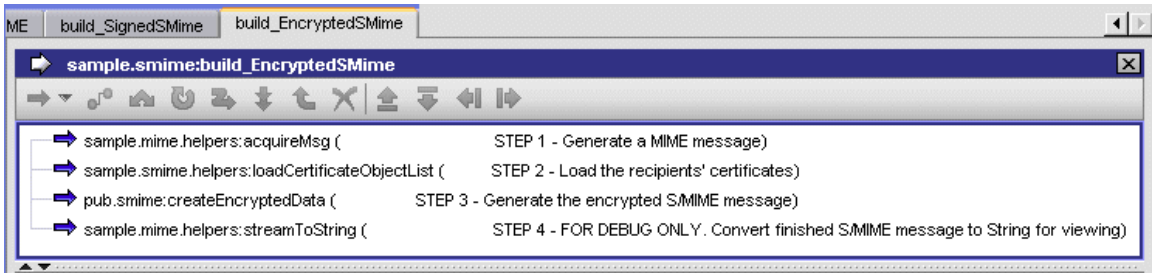
3. Pass the certificate and the MIME message to the pub.smime:createEncryptedData service. This service encrypts the InputStream containing the MIME message and produces a new InputStream containing the encrypted message.

Example—Encrypting a MIME Message

The following flow service encrypts a MIME message. To run this example, you must have at least one certificate file. When you run this service from Designer, it will prompt you for the following:

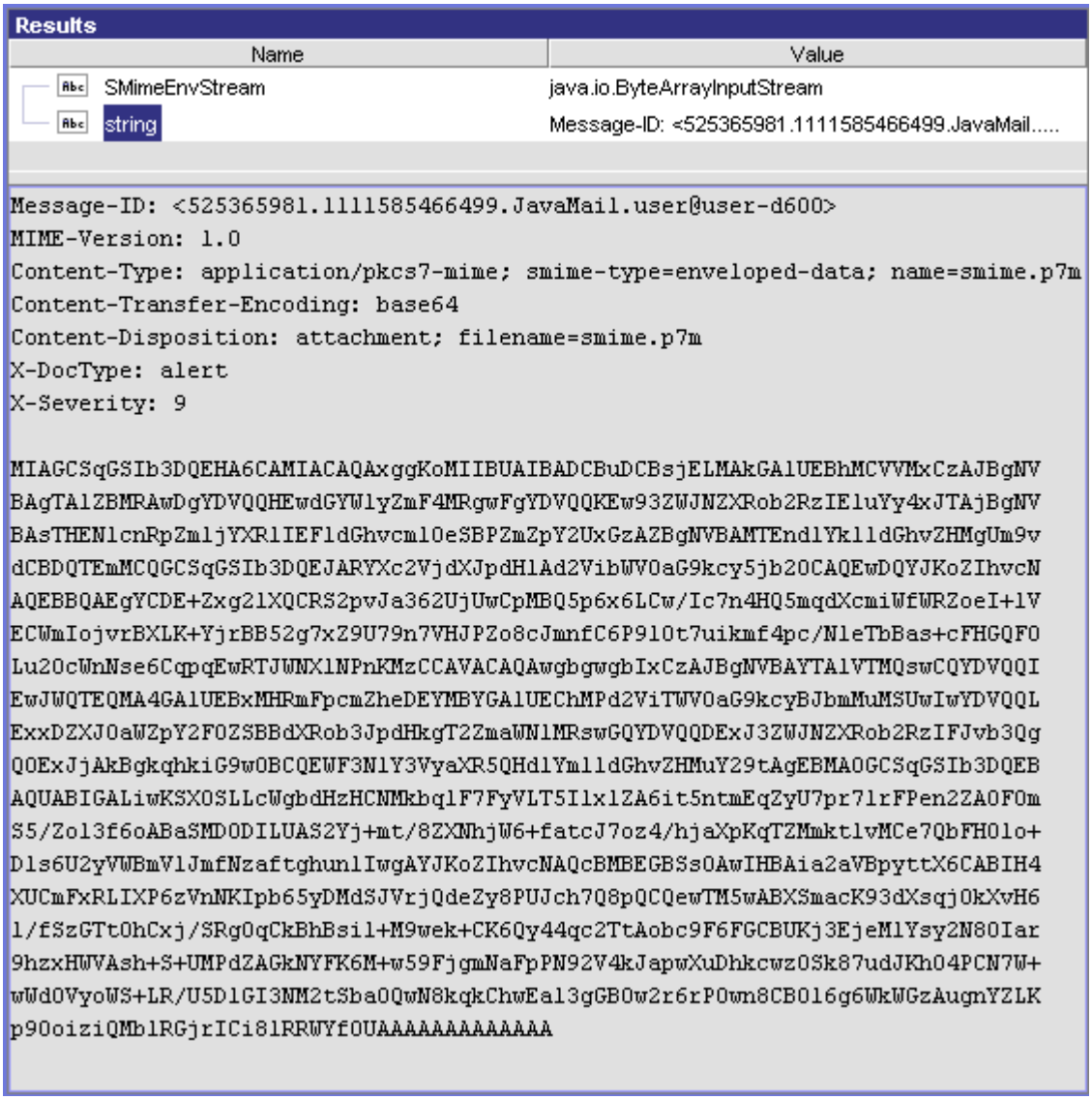
Input Parameter	Description
recipient1CertificateFile	The name of the file containing the certificate of the first intended recipient, for example, d:\netCerts\partner1cert.der.
recipient2CertificateFile	The name of the file containing the certificate of the second intended recipient. If you want to encrypt the message for only one recipient, leave this input parameter empty.

Note:
This example is only for those messages that are within the threshold value specified by the `watt.server.mime.largeDataThreshold` configuration parameter.



Step	Description
1	This step creates a MIME message containing a simple text message. It produces an InputStream that contains the MIME message that will be encrypted.
2	This step loads the recipient's certificates from the files specified in *recipient1CertificateFile* and *recipient2CertificateFile*. This example uses a custom Java service to perform this step. You will need to develop a similar mechanism to load the certificates of the parties to whom you want to send an encrypted message.
3	This step generates the encrypted MIME message using the InputStream from step 1 and the certificates from step 2. It produces a new InputStream called *SMimeEnvStream* that contains the encrypted message.
4	Because you cannot view the contents of an InputStream, this example includes a step that converts *SMimeEnvStream* to a String so you can examine the finished message with Designer. This technique is useful for testing and debugging. If you examine the contents of *string* on the Service Result view, you will see an encrypted S/MIME message similar to the one below.

Step	Description



Signing and Encrypting a MIME Message

If you want to sign and encrypt a message, you must use the special service, pub.smime.keystore:createSignedAndEncryptedData, that Integration Server provides for this purpose.

To use this service, you must provide the credentials for signing a message (the alias of the keystore and the signing key) and the credentials needed to encrypt a message (the recipient's certificate). For information about obtaining these credentials, see "Signing a MIME Message" on page 31 and "Encrypting a MIME Message" on page 33.

Important:
You can sign and encrypt only those messages that are within the threshold value specified by the `watt.server.mime.largeDataThreshold` configuration parameter.

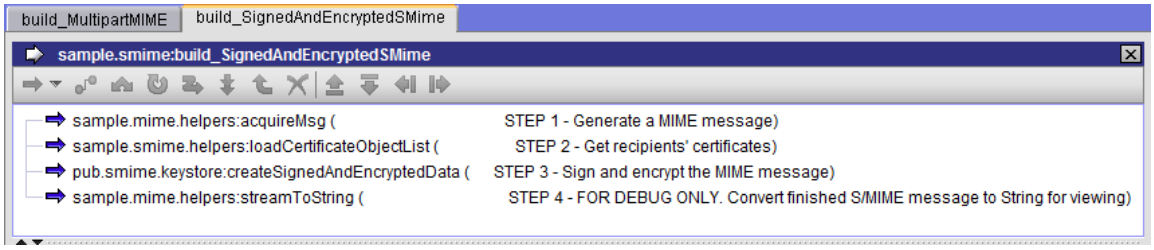
Example—Signing and Encrypting a MIME Message

The following flow service signs and encrypts a MIME message. To run this example, you must provide the alias of the keystore that contains the signing key, and the alias of the private key to use for signing.

Note:
This example is only for those messages that are within the threshold value specified by the `watt.server.mime.largeDataThreshold` configuration parameter.

When you run this service from Designer, it will prompt you for the following:

Input Parameter	Description
signersKeyStoreAlias	**String** Alias of the keystore.
signersKeyAlias	**String** Alias of the private key of interest in the keystore.
recipient1CertificateFile	The name of the file containing the certificate of the first intended recipient, for example, d:\netCerts\partner1cert.der.
recipient2CertificateFile	The name of the file containing the certificate of the second intended recipient. If you want to encrypt the message for only one recipient, leave this input parameter empty.



Step	Description
1	This step creates a MIME message that contains a simple text message. It produces an InputStream containing the MIME message that will be signed and encrypted.
2	This step generates the signed MIME message. It takes the InputStream from step 1 and the credentials specified in *signersKeyStoreAlias* and *signersKeyAlias* and produces an InputStream called *SMimeEnvStream* that contains the signed and encrypted message.
3	Because you cannot view the contents of an InputStream, this example includes a step that converts *SMimeEnvStream* to a String so you can examine the finished message with Designer. This technique is useful for testing and debugging. If you examine the contents of *string* on the Service Result view, you will see an encrypted S/MIME message similar to the following:

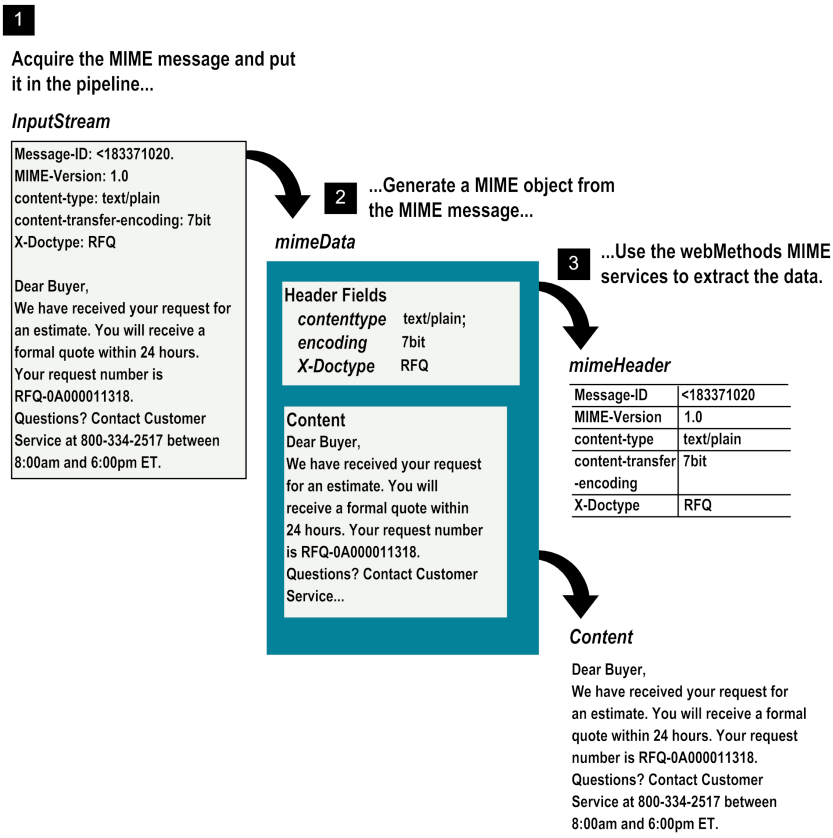
Step Description

3 Extracting Data from MIME and S/MIME Messages

Overview

Besides creating MIME messages, you can also use Integration Server services to extract information (header fields and/or content) from MIME messages that are placed in the pipeline. However, to gain access to the data within a MIME message, you must first convert that message to a MIME object. After you convert the message to a MIME object, you use services such as getMimeHeader, getBodyPartContent, and getBodyPartHeader, to retrieve the information within it.

The following diagram illustrates this process:



Extracting the Payload from a MIME Message

To extract information from a MIME message, you use services from the pub.mime folder to create a MIME object from a MIME message and extract data from it.

How to Extract the Payload from a MIME Message

The following procedure describes the general steps you take to extract data from a MIME message.

1. Place the MIME message in the pipeline as an InputStream or a MimeMessage. For the Integration Server services to work with a MIME message, the message *must* be passed into the pipeline as an InputStream or a MimeMessage. If your solution acquires the MIME message

in another form (such as a String or a byte[]) you must convert it to an InputStream before running the MIME services against it.

Note:
The way in which you acquire a MIME message and put it in the pipeline will depend on your particular solution. You might retrieve it from a back-end system, you might read it from a file, or you might receive it from a custom content handler. Regardless of how you acquire the message, it must be in the form of an InputStream or a MimeMessage to be able to use it with the MIME services.

2. Convert the MIME message to a MIME object using the pub.mime:createMimeData service. Pass the InputStream or MimeMessage containing the MIME message to createMimeData. This service returns a MIME object called *mimeData* that contains the message's constituent elements (header fields and content). It also returns a set of values indicating whether the enclosed message is encrypted or digitally signed. (For information about extracting information from an encrypted and/or signed MIME message, see "Extracting the Payload from a Signed MIME Message" on page 44 and "Extracting the Payload from a MIME Message" on page 40.

   Important:
   The MIME object that the createMimeData service returns is an IData object whose contents you can examine during testing and debugging. However, the internal structure of this object is subject to change in future versions of Integration Server. To extract content from a MIME object, *you must always use* the MIME services that Integration Server provides for this purpose. *Do not* explicitly map data from the elements in the MIME object with the pipeline editor.

3. Extract the payload from the MIME object using the pub.mime:getBodyPartContent service. This service takes as input the MIME object that you created in the previous step 2. If the message contains multiple parts, you can use the *index* or *contentID* parameter to specify which part you want to retrieve, where:

   - *index* is a String that specifies the index number (that is, position number) of the part whose content you want to retrieve. (Index 0 is the first part, Index 1 is the second part, and so forth.)
   - *contentID* is a String that specifies the value of the content-ID header whose content you want to retrieve. For example, if you wanted to retrieve the content of from the part with the "content-ID: AJ9994388-0500," you would set *contentID* to "AJ9994388-0500."

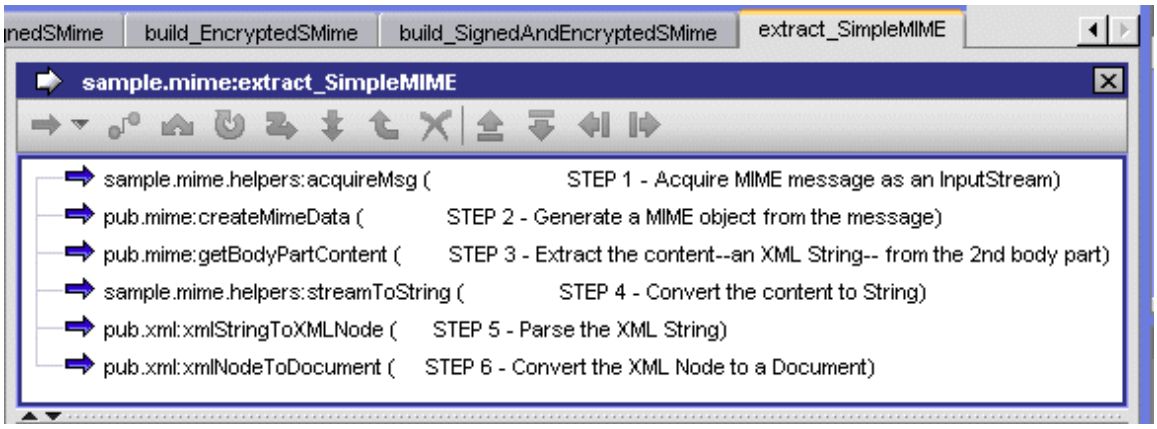
   If you do not specify *index* or *contentID*, getBodyPartContent returns the content from the first body part in the message.

   The content of the requested body part is returned as an InputStream named *content*.

Example—Extracting One Part from a Multipart MIME Message

The following flow service shows how you would extract the content from the second body part in a three-part MIME message. In this example, the message contains an XML document that is extracted, parsed, and put in the pipeline.

Flow service that extracts content from a single part



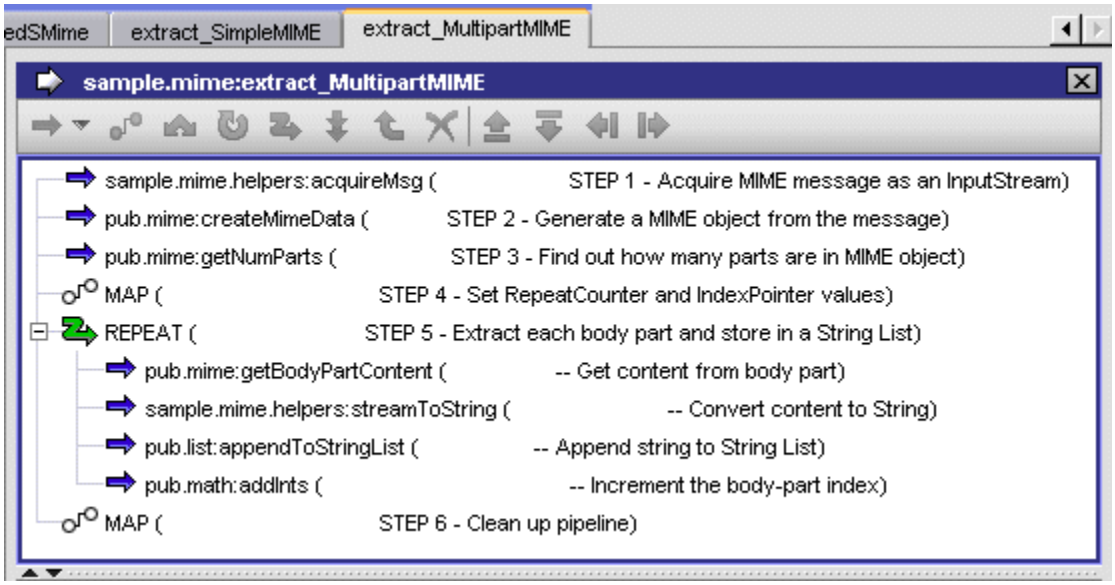
Note:
This example is only for those messages that are within the threshold value specified by the `watt.server.mime.largeDataThreshold` configuration parameter.

Step	Description
1	This step acquires a MIME message. This example calls a helper service that puts a three-part test message in the pipeline as an InputStream. In a production solution, it is more likely that a MIME message would be passed into the pipeline by a content handler or a back-end system.
2	This step takes the MIME message and creates a MIME object (*mimeData*) containing the message's headers and content. If you view the pipeline, you will note that the InputStream produced by step 1 is linked to this step's *input* variable.
3	This step extracts the payload from the second body part in *mimeData*. In this example, the *index* parameter is set to 1 to select the second body part. This step returns the payload (in this case an XML document) as an InputStream named *content*.
4	This step converts the XML document in *content* to a String.
5	This step takes the String containing the XML document and parses it, producing an XML node containing the XML document.
6	This step produces a document (IData object) containing the XML document's elements and values.

Example—Extracting All Parts from a Multipart MIME Message

The following flow service shows how you might process each part in a multipart MIME message sequentially. This example receives a multipart MIME message containing a unknown number of body parts. After discovering the number of body parts, the example uses a REPEAT block to extract the payload from each part.

Flow service that extracts the content from multiple parts



Note:
This example is only for those messages that are within the threshold value specified by the `watt.server.mime.largeDataThreshold` configuration parameter.

Step	Description
1	This step acquires a MIME message. This example uses a helper service that generates a three-part MIME message and puts it in the pipeline as an InputStream call *envStream*. In a production solution, it is more likely that a MIME message would be passed into the pipeline by a content handler or a back-end system.
2	This step takes the MIME message and creates a MIME object (*mimeData*) containing the message's headers and content. If you view the pipeline, you will note that the InputStream produced by step 1 is linked to this step's *input* variable.
3	This step inspects the *mimeData* object and returns the number of body parts (in this case 3) in a String called *numParts*.
4	This step sets the following variables that are used by the REPEAT block:

Variable	Value	Purpose
RepeatCounter	*numParts*-1	Sets the counter that specifies the number of times the REPEAT block needs to re-execute. Since a REPEAT block always executes once, this counter is set to one number less than the total number of body parts in the message.
PartIndex	0	Initializes the pointer that is used to step through the body parts in this message.

Step	Description
5	This REPEAT block does the following for each body part in *mimeData*: 1. Extracts the content 2. Converts the retrieved content to a String 3. Appends that String to a String list The last step in the block increments *PartIndex*. If you view the pipeline, you will see that this variable is linked to the *index* parameter for the getBodyPartContent service.
6	This step drops unnecessary variables, leaving only the populated String list in the pipeline.

Extracting the Payload from a Signed MIME Message

When you pass a signed S/MIME message to createMimeData, it returns an empty MIME object because it cannot parse signed messages. To extract data from a signed message, you must process the message with pub.smime:processSignedData. This service reads an InputStream containing a signed message, verifies the signature, and returns a MIME object containing the message's constituent elements.

Important:
A signer's certificate is authenticated against the set of trusted certificates in the Integration Server truststore. If your site will receive signed messages, you must collect the certificates of CAs that you trust and add them to the truststore. For information about the Integration Server truststore and obtaining CA certificates, see *webMethods Integration Server Administrator's Guide*.

Important:
This procedure is applicable only to InputStream objects and not to MimeMessage objects.

How Do You Know Whether the Message Is Signed?

If your solution always receives signed messages, you can simply pass those messages to processSignedData when you receive them. However, if your solution receives both signed and unsigned messages, you will need to "test" the message to see whether or not it is signed and pass only signed messages to the processSignedData service.

To discover whether a MIME message is signed, pass it to the createMimeData service and check the status of the *signed* variable afterwards. If the value of *signed* is "true," you must pass the message to processSignedData for signature verification.

Working with InputStreams

To work with signed (and encrypted) messages successfully, you need to understand something about the behavior of an InputStream. InputStreams are transient forms of data in a flow service. When a service reads an InputStream, it immediately discards the data within it. This means that

once you process a MIME message with createMimeData, the message no longer exists in the original InputStream object.

This poses a problem if, after running createMimeData on the InputStream, you discover that it contains a signed or encrypted message. Because the original InputStream has been emptied, it cannot be passed to the signature-verification or decryption services. To solve this problem, createMimeData returns a copy of the original InputStream in an output variable called *stream*. This is the variable you pass to processSignedData if, after processing the original message with createMimeData, you discover that it is signed.

What Happens when the Signature is Processed?

When processSignedData processes a signed message, it does the following:

- It verifies the digital signature using the signer's public key.
- It compares the signer's certificate chain to the certificates in Integration Server's truststore to determine whether the credentials are authentic and trustworthy.
- It extracts the message from the S/MIME message stream, parses it, and puts it in the pipeline as a MIME object called *mimeData*.

Error Codes and Messages

If an error prevents the signature from being verified (for example, if the signer's certificate cannot be read or the signature itself is found to be invalid) processSignedData sets the *verify* flag to false and reports the cause of the failure in *errorCode* and *errorMessage* as follows:

errorCode	*errorMessage*	**Signature could not be verified because...**
1	Invalid signer certificate file information.	The signer's certificate could not be read. The variable containing the certificate chain is not an array object.
2	Certificate at index '*i*' is not in recognizable format.	The signer's certificate could not be read. The data at position *i* in the certificate chain does not appear to be a certificate.
3	Invalid certificate input at index '*i*'.	The signer's certificate could not be read. The data at position *i* in the certificate chain is not a byte[ ].
4	Signature cannot be verified.	The signature was invalid. Either the supplied certificate does not belong to the original signer or a message integrity violation has occurred.

If processSignedData is able to verify the signature, but is not able to authenticate the certificate of the signer (that is, the certificate could not be confirmed to be from a trusted source), the *verify* flag will be true and the *errorCode* and *errorMessage* values will be set as follows.

errorCode	*errorMessage*	**Certificate could not be authenticated because...**
5	Expired certificate chain.	One or more certificates in the supplied chain is expired.
6	Error in certificate chain.	The certificate chain is incomplete or invalid. For example, the certificate for an intermediate CA may be missing from the chain.
7	Untrusted certificate.	None of the CAs in the chain are trusted by this server. None of the certificates could be matched to a certificate in the server's truststore.

If processSignedData is able to verify the signature and authenticate the signer's certificate, it does not return *errorCode* or *errorMessage*.

Note:
Regardless of whether processSignedData is able to verify the signature or authenticate the certificate, it always returns a MIME object containing the parsed message.

How to Extract the Payload from a Signed S/MIME Message

The following procedure describes the general steps you take to extract data from a signed S/MIME message.

1. If you do not know whether the message is signed, pass it to the pub.mime:createMimeData service. Afterwards, test the state of the `signed` output parameter. If its value is "true," proceed to the next step 2. Otherwise, check whether the message is encrypted and process it as described in "Extracting the Payload from an Encrypted MIME Message" on page 48. If the message is neither signed nor encrypted, process it as an ordinary MIME message as described in "Extracting the Payload from a MIME Message" on page 40.

2. Pass the message to the pub.smime:processSignedData service to verify the signature. If the signer's certificate chain is included in the signature, you do not need to give this service anything other than the InputStream containing the MIME message. If the signer's certificate chain is not embedded in the signature, you must supply it (this assumes that the signer has given you a certificate chain at some point).

   Keep in mind that if the message was passed to createMimeData before this step, the original InputStream will be empty. In this case, you must pass the *stream* output variable produced by createMimeData to the processSignedData service.

3. Test the verify flag and perform error processing as necessary. If the signature cannot be verified, `verify` will be false. Your service should contain logic to detect this condition and react in a prescribed way. For example, it might send the message to an administrator for a manual inspection or record the event in a log file.

   Note:

Depending on the nature of the messages your service receives, you may want to test the *encrypted* output variable after processing a signature. This will tell you whether the message had been encrypted before it was signed. If *encrypted* is "true," you will need to decrypt the message in *stream*. For procedures, see "Extracting the Payload from an Encrypted MIME Message" on page 48.

4. Extract the payload from the MIME object using the pub.mime:getBodyPartContent service. If the enclosed message is not encrypted, processSignedData returns a MIME object that contains the message's constituent elements (header fields and content). At this point, you can use getBodyPartContent to retrieve the content from the MIME object. For information about using getBodyPartContent, see "Extracting the Payload from a MIME Message" on page 40.

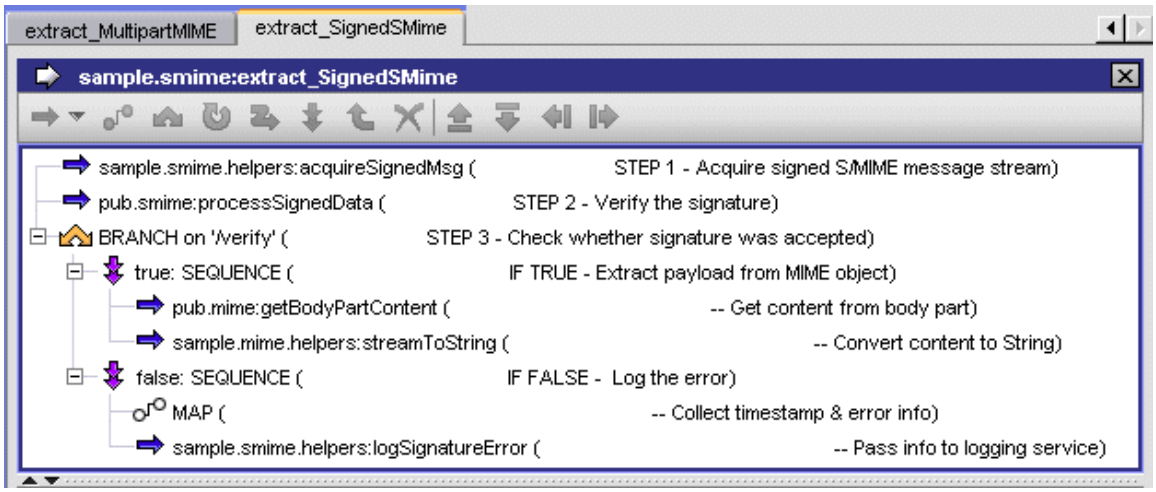
Example—Extracting Content from a Signed S/MIME Message

The following flow service extracts the payload from a signed MIME message.

To run this example, you must have a private key, the associated certificate, and the certificate of the CA that signed it. These credentials are needed by the helper service, sample.smime.helpers:acquireSignedMsg, which generates the signed test message used in this example. You will need to edit the first step in the helper service to specify the location of these files on your system.

This service assumes that the signature contains the signer's certificate chain, so you do not need to supply a certificate chain at run time.

Flow service that extracts the content from a signed MIME message



Step	Description
1	This step acquires an InputStream containing a signed MIME message. This example uses a helper service to produce a test message. In a production solution, it is more likely that a MIME message would be passed into the pipeline by a content handler or a back-end system.

Step	Description
2	This step takes the InputStream generated in step 1 and processes the signature. If the signature is valid, this step produces a MIME object called *mimeData*, containing the parsed message. If the signature is invalid, this step returns an empty *mimeData* object and sets the *verify* flag to "false."
3	This step checks whether or not the signature was processed successfully by testing the value of the output variable *verify*. If *verify* is "true," this step extracts the payload and converts it to a String. If *verify* is "false," this step collects the error information in the pipeline and passes it to an error-logging service.

Extracting the Payload from an Encrypted MIME Message

When you pass an encrypted S/MIME message to the createMimeData service, it returns an empty MIME object, because it cannot parse encrypted messages. To extract data from an encrypted message, you must decrypt the message with pub.smime.keystore:processEncryptedData. This service reads an InputStream that contains an encrypted message, decrypts it using a private key pointed by the keystore alias and key alias that you supply, and returns a MIME object containing the message's constituent elements.

Important:
This procedure is applicable only to InputStream objects and not to MimeMessage objects.

How Do You Know Whether the Message Is Encrypted?

If your solution always receives encrypted messages, you can simply pass those messages to processEncryptedData when you receive them. However, if your solution receives both encrypted and clear-text messages, you will need to "test" a message to see whether or not it is encrypted, and pass only encrypted messages to the processEncryptedData service.

To discover whether a MIME message is encrypted, pass it to the createMimeData service and check the status of the *encrypted* variable afterwards. If the value of *encrypted* is "true," you must pass the message to processEncryptedData to be decrypted.

Note:
When you process an InputStream with createMimeData, that InputStream is emptied and is no longer available to other services. For this reason, createMimeData returns a copy of the original message stream in the output variable called *stream*. You pass this variable to processEncryptedData if the original InputStream has been emptied by createMimeData. For additional information about InputStreams, see "Working with InputStreams" on page 44.

How to Extract the Payload from an Encrypted S/MIME Message

The following procedure describes the general steps you take to extract data from an encrypted S/MIME message.

1. If you do not know whether the message is encrypted, pass it to the pub.mime:createMimeData service. Afterwards, test the state of the `encrypted` output parameter. If its value is "true," proceed to the next step 2. Otherwise, test the `signed` variable to see whether the message is signed and process it as described in "Extracting the Payload from a Signed MIME Message" on page 44. If the message is neither signed nor encrypted, process it as an ordinary MIME message as described in "Extracting the Payload from a MIME Message" on page 40.

2. Pass the message to the pub.smime.keystore:processEncryptedData to be decrypted. You must pass three input parameters to this service: the InputStream containing the encrypted MIME message, the keystore alias that points to the recipient's keystore, and the key alias that identifies the recipient's private key in the keystore.

   Keep in mind that if the message was passed to createMimeData prior to this step, the original InputStream will be empty. In this case, you must pass the *stream* output variable produced by createMimeData to the processEncryptedData service.

   Note:
   Depending on the nature of the messages your service receives, you may want to test the *signed* output variable after decrypting the message. This will tell you whether the message had been signed prior to being encrypted. If *signed* is "true," you will need to verify the signature of the message in *stream*. For procedures, see "Extracting the Payload from a Signed MIME Message" on page 44.

3. Extract the payload from the MIME object using the pub.mime:getBodyPartContent service. If the decrypted message is not signed, the MIME object returned by processEncryptedData will contain the message's constituent elements. You use getBodyPartContent to retrieve the content from this MIME object. For information about using getBodyPartContent, see "Extracting the Payload from a MIME Message" on page 40.

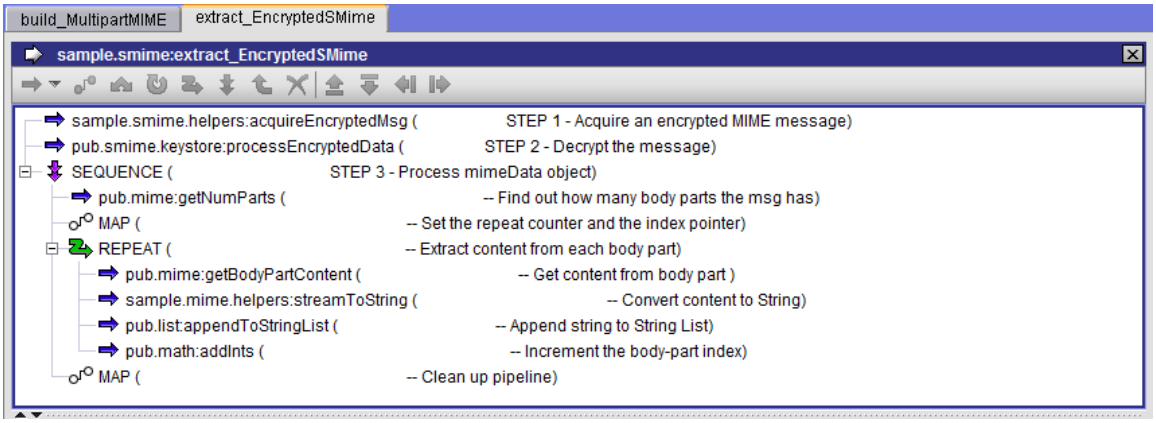
Example—Extracting Content from an Encrypted S/MIME Message

To run this example, you must provide the keystore alias and key alias for the recipient's private key. Some of these credentials are needed by the helper service, sample.smime.helpers:acquireEncryptedMsg, which generates the test message used in this example. You will need to edit the first step in the helper service to specify the location of these files on your system.

When you run this service from Designer, it will prompt you for the following:

Input Parameter	Description
recipientsKeystoreAlias	**String** Alias of the recipient's keystore.
recipientsKeyAlias	**String** Alias of the private key in the recipient's keystore.

Flow service that extracts the content from an encrypted MIME message



Step	Description
1	This step acquires an InputStream containing an encrypted multipart MIME message. This example uses a helper service to produce the test message. In a production solution, it is more likely that a MIME message would be passed into the pipeline by a content handler or a back-end system.
2	This step takes the InputStream from step 1 and decrypts the message. It produces a MIME object (*mimeData*) that contains the decrypted message's constituent elements (header fields and content).
3	This step extracts each body part from *mimeData* and appends it to a String list.

Extracting Data from a Signed and Encrypted MIME Message

If your solution receives messages that are signed and/or encrypted, your flow service must test each incoming message and process it appropriately.

Note:
This procedure is applicable only to InputStream objects and not to MimeMessage objects.

Example—Extracting Content from a Signed and Encrypted S/MIME Message

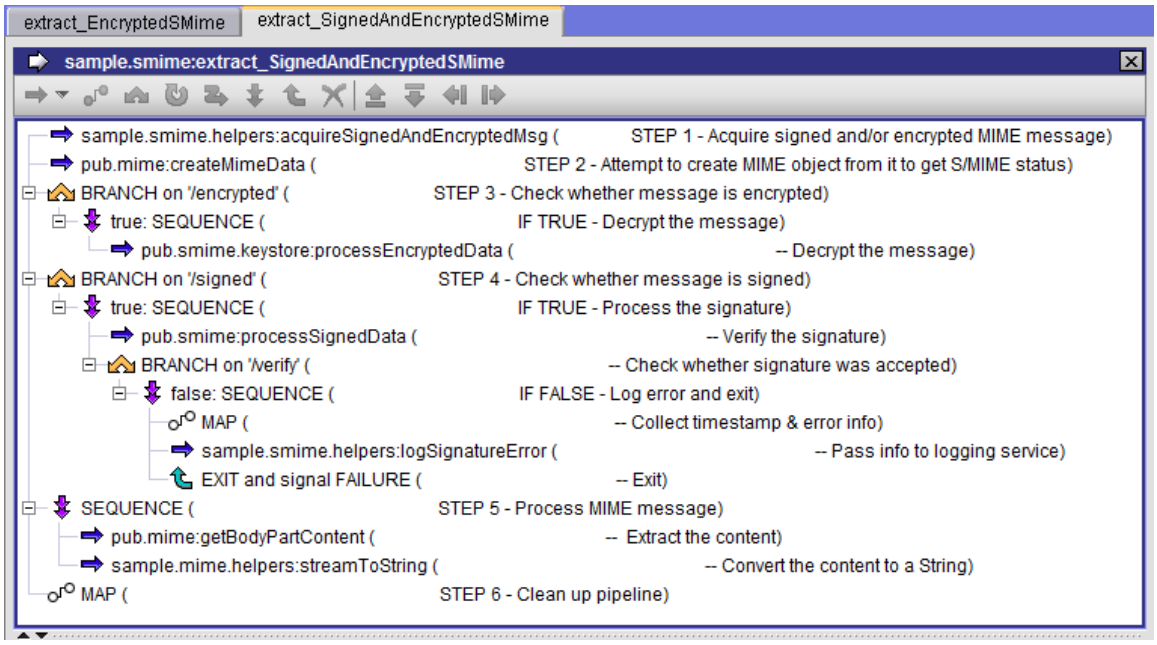
The following flow service extracts data from MIME and S/MIME messages.

To run this example, you must provide the keystore alias and key alias for the recipient's private key. Some of these credentials are needed by the helper service, sample.smime.helpers:acquireSignedAndEncryptedMsg, which generates the test message used in this example. You will need to edit the first step in the helper service to specify the location of these files on your system.

When you run this service from Designer, it will prompt you for the following:

Input Parameter	Description
recipientsKeystoreAlias	**String** Alias of the recipient's keystore.
recipientsKeyAlias	**String** Alias of the private key in the recipient's keystore.

Flow service that extracts the content from a signed and/or encrypted MIME message



Step	Description
1	This step acquires an InputStream containing a signed and encrypted MIME message. This example uses a helper service to produce the test message. In a production solution, it is more likely that a MIME message would be passed into the pipeline by a content handler or a back-end system.
2	This step attempts to create a MIME object from the InputStream produced in step 1.
3	This step tests the *encrypted* flag to see whether the message is encrypted. If it is, it obtains the credentials needed to decrypt the message and passes those credentials and the message to the processEncryptedData service. Note that the *stream* output variable is linked to the *SMimeEnvStream* input parameter, because the original InputStream from step 1 was emptied by step 2. Note that if *encrypted* is "false," execution falls through to step 4.
4	This step tests the *signed* flag to see whether the message is signed. If it is, it passes the message to the processSignedData service. Note that the *stream* output variable is linked to the *SMimeEnvStream* input parameter, because the original InputStream from step 1 was emptied in step 2. (When a message is decrypted in step 3, the processEncryptedData service produces the *stream* used by this step.)

Step	Description
	Note that if *signed* is "false," execution falls through to step 5.
5	This step extracts the data from the *mimeData* object produced by the preceding steps.