Volantis Mobility Server

Message Preparation Server™ ver. 5.1

Installation & Administration Guide

Volantis Mobility Server

Message Preparation Server™

Version 5.1

Installation and Administration Guide

Message Preparation Server: Installation and Administration Guide

Version 5.1
Published date 2009-01-29

Included software

Apache Software Foundation
Free Software Foundation
Jaxen Project
Sun Microsystems Inc

Table of contents

About MPS

MCS is tailored to pull, or browser-based content delivery. Multi-media Messaging Services (MMS) technology has created a need for device awareness to be used in a push-based context, by rendering rich media messages in a device-specific way.

Message Preparation Server (MPS) satisfies this need. It builds on the core functionality of MCS to allow the optimization of message-based or WAP push content. It provides the ability to write applications to generate and transmit messages to subscribers' devices. This allows applications to be created that can support mass distribution of messages to provide significant end user function. The messages might, for example, contain information that users had subscribed to.

MPS allows multimedia messages to be authored using the same techniques used to create MCS content for online delivery. This means that JSP pages or XDIME documents, combined with MCS policies, can be used to create rich message content. MCS supports MPS in enabling message content to be rendered in a way that is optimized according to the capabilities of the target device.

MMS, HTML-formatted email (MHTML) and SMS message output are supported. Only the first two message types may contain rich media content. WAP push URLs are sent as SMS messages.

A Message Preparation Services API (MPSAPI) provides a programming interface for developing MPS applications.

Note: Because file locations are set during installation, the configuration examples shown in related topics use placeholders for parts of the directory structure. You should substitute `[path]` with the path preceding the location described, and `[context_root]` with the application root context.

Installing MPS

The MPS installer collects the values required to configure channel adapters for several protocols, the installation location, and the details for locating the files needed by the WAP push channel adapter.

Before you install

You will need to install the following supporting software and MCS policies before you can use MPS.

- You need to **install MCS**. Refer to the topics on installing MCS for details.
- You should have an **MCS repository** installed and working to be able to use MPS with MCS.
- The **JavaMail** package (javax.mail) and the **JavaBeans Activation Framework** (javax.activation) used by the JavaMail API to manage MIME data. Both are available from http://java.sun.com.

Optionally, you can also install:

- The **JavaMail SMTP** protocol provider for the JavaMail API that provides access to an SMTP server, available from http://java.sun.com
- The **Nokia MMSC** protocol used for communication with Nokia MMSC. This is available as the package com.nokia.mms from http://www.forum.nokia.com. The file is `nokia_mmsdriver_1.5.jar`.

Note: If one of the SMTP, MMSC or SMSC JAR files is not in the classpath, and/or one of the channels is not configured correctly this will prevent messages from being delivered and may cause internal problems for the application server. For this reason each `channel` element in the `mcs-config.xml` file is commented out until you specifically enable it.

Running the installer

The installation wizard steps you through a series of pages containing settings that correspond to the `channel` element attributes in the `mcs-config.xml` file. You can edit this file at any time after installation if you need to change the settings.

All the installer values have defaults. When you have completed a section, click **Next** to move to the next page, or click **Previous** to review or modify any values.

When the installation is complete you can choose to save an automated installation file at a convenient location. Then you can repeat the installation on another machine with the path to the configuration file as a parameter.

1. Start the installer with the command
   `java -jar mps_installer-5.1.jar`
2. Accept the license agreement
3. Choose the packs to install
4. Configure the SMTP channel adapter with the *Host* name. If the *Server requires authorization*, check the option and add *Username* and *Password* settings.
5. Configure SMSC, MMSC and WAP Push channel adapters in the same way
6. Click **Browse** to choose the install location. MPS must be installed in the same directory as MCS.
7. Verify the installation settings and click **Next** to complete the installation
8. Optionally save your installation settings by clicking **Generate an automatic installation script**, and naming the file. To repeat the installation run the following command.
   `java -jar mps_installer-5.1.jar [configuration_file]`
9. Click **Done** to complete the installation

Creating a message

You create MPS message content using XDIME markup and the MCS Layout editor.

Caution: MPS does not support the use of XDIME 2 content.

The `message` element defines an MPS message, which applies only to MPS. A message consists of a single `canvas` element that is displayed when the message is read by its recipient.

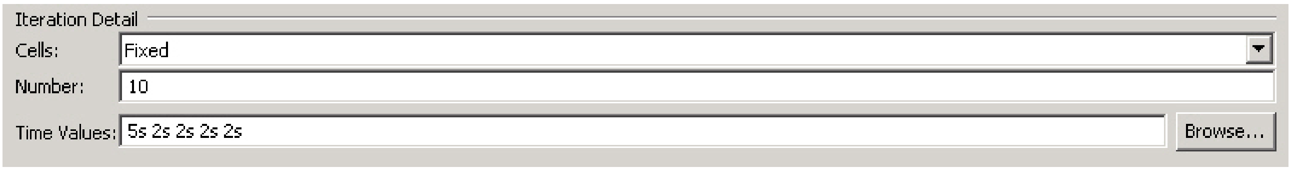
In the example the `canvas` element uses the `layoutName` attribute to identify a layout, created in the Layout editor, with the value 'message'.

A series of temporal iterator panes group text/image pairs. The pane references are qualified by the index values for a temporal iterator, starting from 0. Two methods of referencing are shown.

First a paragraph element with a `pane` attribute contains some titling. Then follow text/image pairs, logically grouped inside the `pane` with the `name` containing the reference. The first image to be used is identified by the `img` element `src` attribute value "/welcome/vol_logo.mimg", which is a reference to an image component named 'vol_logo.mimg' in the folder 'welcome'.

```
<?xml version="1.0" encoding="UTF-8"?>
<message>
  <canvas layoutName="/welcome.mlyt" pageTitle="message">
    <p pane="background">Hello World!!!</p>
    <pane name="logo">
      <img src="/welcome/vol_logo.mimg" alt="Volantis Systems Ltd. Logo"/>
    </pane>
  </canvas>
</message>
```

Timing values for the temporal iterator can be set in the layout, and the indexed panes can be made to appear at the specified intervals.


Iteration Detail

Cells: Fixed

Number: 10

Time Values: 5s 2s 2s 2s 2s

Using XDIME in messages

Only simple XDIME elements are supported in MHTML, so you cannot use structural elements such as `table` or `div`.

In MMS and SMS, whitespace is significant, so MPS renders element body content using whitespace to layout the message.

1. All line feeds and unnecessary spaces are trimmed from element contents, apart from the `pre` element
2. A line break is placed in the output text before and after most block level elements
3. Whitespace is added around text for `td`, `th` and `dt` elements to approximate columns
4. The `ol` and `ul` elements are also emulated, but only in the most simple cases, and any attributes are ignored

Other issues and features:

- Nested elements in table cells such as `p` may break the emulated formatting
- All form content is ignored
- Character encodings not supported in MMS/SMS will produce undefined results
- No support is provided for inclusion
- Any HTML entities are ignored and passed through
- All theme and style information is ignored

Several limitations in the use of layouts, themes and components in MPS messages are given in the table.

Layouts, themes and components

Type	Layout	Themes	Components
MHTML	Layouts should be simple	HTML 3.2 does not support CSS. MCS emulates some CSS properties for HTML 3.2 family protocols by mapping theme policies to individual element attributes.	No differences
MMS	A maximum of two panes, with separate panes for text and images if both are used. *Destination Area* format attribute for the pane must contain the case sensitive literal 'Text' or 'Image'.	All theme information is ignored.	Text fallbacks are used for unsupported non-text components if available
SMS	A single pane only is allowed		

Additional features over MCS

Where a device natively supports the AMR audio format (audio/amr and audio/x-amr), you can use device specific audio assets with the XDIME `audio` element.

On MMS/SMIL devices, it is possible to associate an individual audio component with each slide in an MMS slide presentation. For example, in a message canvas containing several slides timed by a temporal iterator, if there are two panes in the iterator layout (one text, one image) above one another, and an audio component is bound to *either* pane, the audio will be played when the layout is displayed as a single slide. Under MHTML/SMS any alternative text is output.

```
<canvas layoutName="iter8" theme="beaches.mthm">
  <!--slide 1-->
  <p pane="text1">Waikiki beach, Oahu</p>
  <audio pane="text1" src="/sounds/uke1.mauc"/>
  <p pane="image1">
    <img src="/graphics/waikiki.mimg"/>
  </p>
  <!--more slides-->
</canvas>
```

MCS features not available with MPS

Fragmentation, which is available with MCS, is not supported within MPS messages. Page generation will fail for layouts that are fragmented.

Dissection, which is available with MCS, is not supported within MPS messages. During page generation, dissecting panes are treated as normal panes and dissection is suppressed.

Error detection

The error detection features that are available in MPS are channel dependent. If a protocol supports a particular error condition, then the error values will be returned by the appropriate adapter.

Configuring MPS

General configuration of MCS is achieved using the file `mcs-config.xml`. This section explains the part of the configuration file that is specific to MPS. Other sections are explained in *Configuring MCS*.

MCS must be fully configured and working before MPS will work successfully. The MCS installer makes a default set of entries in the `mcs-config.xml` file, including those for MPS. You'll find the configuration file and the schema `mcs-config.xsd` in the directory `WEB-INF`.

mcs-config.xml

The `application-plugins` section shows the default entries for MPS. Comments in the source provide further information.

You must remove the comments around the relevant `channel` elements in the configuration file when you have the appropriate `.jar` files and classpath set up. Notes on the meaning of the arguments are contained in the configuration file.

Note: You should comment out any channel elements for channels which you are no longer using. This is to avoid any possibility messages not being delivered, or of internal problems for the application server, due to an SMTP, MMSC or SMSC `.jar` file not being in the classpath, and/or one of the channels not being configured correctly.

The MSS servlet can be configured on either the same host as the MPS installation or on a different machine. If you wish to run the Wap push service on a different machine, you will need to add the machine URL in another `argument` element in the channel definition.

```
<argument name="message-store-url" value="[remote_url]"/>
```

Note: Not all configurations of MPS have an option to identify this URL during installation, so you may have to add this entry to the `mcs-config.xml` file manually.

```
<mps
  internal-base-url="http://localhost:8080/volantis"
  message-recipient-info="com.volantis.mps.recipient.DefaultRecipientResolver" >
  <!-- The channel definitions for MPS transports must be configured-->
  <channels>
    <!-- Uncomment this section to enable the SMTP channel adapter
    <channel name="smtp"
      class="com.volantis.mps.channels.SMTPChannelAdapter">
      <argument name="host" value="SMTP-HOST"/>
      <argument name="auth" value="false"/>
      <argument name="user" value="SMTP-USERNAME"/>
      <argument name="password" value="SMTP-PASSWORD"/>
    </channel>
    -->
    <!-- Uncomment this section to enable the Logica SMSC channel adapter
    <channel name="smsc"
      class="com.volantis.mps.channels.LogicaSMSChannelAdapter">
      <argument name="smsc-ip" value="SMSC-IP"/>
      <argument name="smsc-port" value="SMSC-PORT"/>
      <argument name="smsc-user" value="SMSC-USERNAME"/>
      <argument name="smsc-password" value="SMSC-PASSWORD"/>
      <argument name="smsc-bindtype" value="async"/>
      <argument name="smsc-supportsmulti" value="no"/>
      <argument name="smsc-pooling" value="yes"/>
      <argument name="smsc-poolsize" value="5"/>
    </channel>
    -->
   <!-- Uncomment this section to enable the Nokia MMSC channel adapter
    <channel name="mmsc"
      class="com.volantis.mps.channels.NokiaMMSChannelAdapter">
      <argument name="url" value="http://MMSC-HOST:MMSC-PORT"/>
      <argument name="default-country-code" value="+44"/>
    </channel>
    -->
    <!-- Uncomment this section to enable the Wap Push channel adapter
    <channel name="wappush"
      class="com.volantis.mps.channels.NowSMSWAPPushChannelAdapter">
      <argument name="url" value="http://WAPPUSH-HOST:WAPPUSH-PORT"/>
      <argument name="default-country-code" value="+44"/>
    </channel>
    -->
  </channels>
</mps>
```

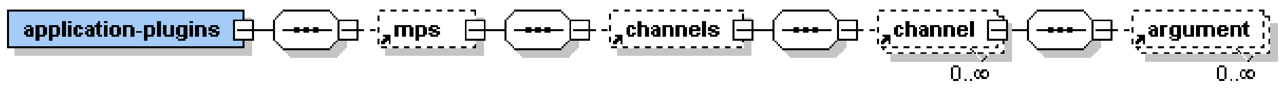
Connection pooling

You can use arguments in the LogicaSMSChannelAdapter definition to configure connection pooling. If pooling is required, then the 'smsc-pooling' argument value must be set to 'yes', and the size of the pool must be specified in the 'smsc-poolsize' argument. If pooling is not required then each request for a connection will cause a new connection to be created and returned.

```
<channel name="smsc"
  class="com.volantis.mps.channels.LogicaSMSChannelAdapter">
  <argument name="smsc-ip" value="SMSC-IP"/>
  <argument name="smsc-port" value="SMSC-PORT"/>
  <argument name="smsc-user" value="SMSC-USERNAME"/>
  <argument name="smsc-password" value="SMSC-PASSWORD"/>
  <argument name="smsc-bindtype" value="async"/>
  <argument name="smsc-supportsmulti" value="no"/>
  <argument name="smsc-pooling" value="yes"/>
  <argument name="smsc-poolsize" value="5"/>
</channel>
```

mcs-config.xsd

This section explains the MPS entries of the schema `mcs-config.xsd`.



The `mps` element contains the MPS configuration. The `internal-base-url` attribute defines the URL to use for MPS asset resolution from internal requests. The `message-recipient-info` attribute specifies the user supplied class used to resolve recipient devices and channels.

The `channels` element gives the channel definition for MPS transports.

The required `name` attribute on the `channel` element gives the name of the channel, and the `class` attribute defines the class implementing the Custom Channel Adapter interface for this channel.

The nested `argument` elements can occur many times, and contain required `name` and `value` attribute pairs to contain the necessary parameters.

```
<xs:element name="application-plugins">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="mps" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="mps">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="channels" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="internal-base-url" type="xs:anyURI"/>
    <xs:attribute name="message-recipient-info" type="JavaClassName"/>
  </xs:complexType>
</xs:element>
<xs:element name="channels">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="channel" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="channel">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="argument" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="name" use="required"/>
    <xs:attribute name="class" type="JavaClassName" use="optional"/>
  </xs:complexType>
</xs:element>
```

Messaging with WAP push

MPS ships with a message store servlet (MSS) that uses the WAP Push channel adapter (WAPPushChannelAdapter). The servlet caches XML messages using an identifier and later retrieves them using the identifier as a parameter to the request. MSS demonstrates the use of the Custom Channel Adapter API, and is described here. For specific WAP push server applications, you will need to adapt the code.

Note: To enable WAP push, you must uncomment the WAP push channel in the `mcs-config.xml` file.

WAP Push does not send content in the message, but instead sends the URL of the page containing the message to the device in the subject of an SMS message. The user of the device can then pull the content down at a convenient time.

MPS handles WAP Push messages according to their type.

Because a JSP message is already in the form of a URL, this URL is used as the subject of the SMS message.

XML Messages have no associated URL, so MSS stores the message and builds a URL that is sent to the device using the WAP push adapter. The URL consists of the fully qualified path to the servlet, and an appended identifier that is unique in this instance of the servlet context. To retrieve a message, MSS uses the identifier in the HTTP request from the device.

One XML message is stored for each device type until the client pulls the message from the message storage servlet.

If you need to push a message to the device using Service Load, you can override a Service Indication by using the `addHeader` method of the `MultiChannelMessage` class to add a special header 'X-WAP-Push-Type' that may have a value of 'SL' or 'SI'. If the header value is invalid then 'SI' is assumed.

```
addHeader(ALL,"X-WAP-Push-Type","SL")
```

Note: Service Indications are alerts with a click-able link. Service Loads are alerts that also automatically deliver the content to the device.

Storing and retrieving messages

To store an XML message, MPS sends an HTTP POST request to the MSS servlet with the single parameter 'xml' containing the XDIME XML to execute.

The MSS servlet generates a page identifier in a secure manner, and returns the page identified in the response header 'xmlid'.

To retrieve a stored message a simple HTTP GET request is made to the MSS servlet with a 'pageId' parameter containing the previously-generated page identifier, for example:

```
http://myhost:8080/[context_root]/mss?pageid=xmlid
```

Configuring the message store

The MSS servlet can be configured on either the same host as the MPS installation or on a different machine.

Because WAP Push messages need to be persisted on a web server until users can retrieve them, MPS provides configurable message caching. All messages will persist in the storage servlet for the period you specify. Messages are stored in the message store directory specified in the configuration of the servlet, and persist over a restart of the servlet until they are expired.

The message store, which MPS uses to cache WAP push messages until they are retrieved by users, is configured in the `mss-config.xml` file. This file, and the related XML schema and log4j files (also prefixed 'mss') will be in the locations you specified when you installed MPS. The default directory is `WEB-INF`.

Two `init-param` elements in the `web.xml` file give the paths to the MSS configuration file and the log4j file. The `param-name` attribute names the file and the `param-value` attribute gives the path information.

The `messageStoreServer` element contains the configuration definition.

```
<messageStoreServer>
  <message-store
    location="[path]\webapps\[context_root]\mss_store"
    timeout="unlimited"
    id-size="12"
    validate="false"/>
</messageStoreServer>
```

The `message-store` element contains the cache settings. The `location` attribute contains the absolute path to the directory used by the message store server. The `timeout` attribute sets the period in seconds that a cache entry will be retained, or the text value 'unlimited'. The `id-size` attribute is the length in characters (minimum 10) of the generated message identifier. The `validate` attribute determines whether the XML message should be validated before storage.

Note: You will probably want to set the `timeout` attribute to a high value, giving users plenty of time to retrieve a message. The message store server checks this value at five-minute intervals, so you should set it as a multiple of 300 seconds.

Note: From MPS 3.3, the `environment` element is obsolete; it will be removed in a later version of the `mss-config.xml` file. If the location is specified here, a warning message will be logged and the location will not be used.

Using MPS interfaces

You invoke MPS through the Message Preparation Services API (MPSAPI). You create an XDIME file, which uses the `message` element to contain a `canvas`. MPS generates a message containing all the content required for rendering on the end-user's device, and transmits the message using channel-specific APIs.

In order to use MPS you must:

- Write a servlet or portlet that will run in the same web application as MCS and MPS, and that uses the MPSAPI
- Create the device independent message using XDIME
- Create the policies referenced by the message and import them into the MCS repository
- Optionally provide an implementation of the mapping from address to device and channel

Note: Callers can request that messages that are not channel-specific are returned to them rather than being sent across the network by MPS itself.

Since a message preparation request normally involves transmission to multiple users, who may be using different devices, each step may be required to generate multiple copies of its output. Caching of static information is an important part of the preparation of the messages.

Matching protocols to channels

MPS imposes no restriction on the type of messages that can be sent down a particular channel, so you must ensure that the protocol for the rendered message is compatible with the delivery mechanism.

In the example servlet included with MPS there is a default implementation of the `MessageRecipientInfo` class to assign messages to channels using the *Preferred message protocol* value obtained from the MCS device repository.

A list of relevant message protocol attributes used by MCS is available in the Messages Category on the Policies tab in the Device policy editor in Eclipse, including the *Preferred message protocol* values.

Application-side interface

The application-side interface allows applications to request that messages should be assembled and (optionally) submitted for delivery, by passing or referencing a device-independent message template in the form of an XDIME document.

Message assembly interface

The message assembly API accepts the following parameters:

- A reference to (by local URL) the canvas to be rendered
- The destination address(es)
- For each addressee, the name of the device (or device type)
- The subject header and any other headers required

The message assembly mechanism generates the following types of object on the supported protocols, including attachments in MMS messages:

- Text: SMS
- URL: WAP push
- Multipart MIME: MMS, SMTP

In order to do this, the message assembly mechanism performs several basic tasks.

- Determine the device type for each recipient, using an external source (unless the device type is specified in the request)
- For each device type in the recipient list, one instance of the message is generated as follows:
    - Render the markup for the message in the required protocol
    - Retrieve all the required assets
    - Encode the asset content
    - Assemble the generated pages and related objects into a single multipart message
    - Ensure that all internal and external URL references are correctly specified
    - If the message exceeds the maximum size accepted by the target device, and it cannot be reduced by resizing any convertible images, the assembly will fail. The following limits apply:
      SMS: limited to 160 single-byte characters.

      MMS: the value in the `Max MMS message size` attribute.

      MHTML: no limit is enforced.

WAP push: no limit enforced at message preparation time. Dissection is available to limit the page size when the content is requested by the device. The subject (which includes the URL) is limited by SMS message capacity.

- When delivery has been requested, the delivery agent is determined, and each message instance submitted
- When delivery has not been requested, the message instances are returned, along with the recipient lists

The protocols supported for address modes are:

- To: SMS/WAP push, SMTP, MMS
- Cc: SMTP
- Bcc: SMTP

Note: Owing to an issue with modes in the JavaMail API, recipients of SMTP messages may receive mail that was not intended for them. If all the recipients have the same device, mode information is sent correctly. However, if the recipients have different devices, the message sent does not contain the mode information, and all recipients see the message regardless of the device they are using, and some recipients may receive the message multiple times. Therefore, it is recommended you use the SMTP channel adapter to either send messages to a single recipient (if the message is personalized), or to a list using Bcc addressing.

Delivery-side interface

The delivery-side interface provides:

- An SMTP interface for submitting MMS and MHTML messages with the following features:
  - Support for headers including extension 'X-' headers
  - Support for multipart/related MIME-encoded message bodies
  - Optional SMTP authentication with individual credentials for each SMTP server
- Support for SMS
  - Supported primarily as a fallback to allow some message to be delivered to the recipient
  - SMPP is supported using the Logica SMPP APIs
  - SMS messages are limited to 160 characters and should be contained in a text-only pane.
- Support for MMSC API (Nokia)

Delivery-side support is also available for WAP push. See *Messaging with WAP push* for details.

Address-to-device mapping interface

MPS can perform address-to-device mapping in situations where the content provider making the request via the message assembly API does not know the device type for each user. In this case MPS uses an external mechanism to determine the device type for each recipient in order to provide the device-target version of the message.

Note: MPS only provides interfaces; the logic to provide values for these interfaces must be supplied by the developer

The two main information sources that carriers can use to provide the mapping from address to device type are:

- Mobile subscriber database, typically accessed through LDAP
- Mobile network APIs, for example using JAIN/Parlay

In some cases, a carrier will wish to use a combination of these two sources, for example trying a network API first and then a subscriber database if the device is not currently connected to the network.

This interface will call other services to get the required information. However, the service APIs will often vary from one carrier to another. MPS therefore includes a generic interface that can be used to invoke implementation-specific code in order to perform a mapping from addresses to device types. Implementation-specific code can be developed and installed by the product supplier.

The class `com.volantis.mps.recipient.DefaultRecipientResolver` that implements this interface can be configured in the `mcs-config.xml` file. See *Configuring MPS* for more information.

Programming MPS

You can use the Message Preparation Services API (MPSAPI) to develop MPS applications. Details of the API are contained in the JavaDoc, and the code fragments in this section illustrate its use.

The main steps in programming MPS applications are:

- Establishing the **session**
- Adding **attachments**
- Adding **recipients**
- Resolving **channels and devices**

Establishing the session

The example servlet uses the following code to define the application and then initialize it.

```
MarinerServletApplication mpsTest;
public RunMps() {
}
public void init() throws ServletException {
  super.init();
  mpsTest = MarinerServletApplication.getInstance(
    getServletConfig().getServletContext());
}
```

If you initialize MCS from a custom application that has its own servlet context you should use something like the following code.

```
ServletContext sc = pageContext.getServletContext().getContext("/mcs")

MarinerServletApplication example =
MarinerServletApplication.getInstance(sc);}
```

Adding attachments

Multipart MIME messages generated for MPS may need to carry arbitrary attachments if the channel adapter supports attachments. Such attachments are not referenced from within the body of the message but travel with it. The attachments can include things such as:

- Manufacturer-specific ringtones
- Java applications
- Executable content in other formats (such as Mophun, BREW)
- vCard, vCalendar entries

Allowing attachments to MPS messages supports the use of MPS as a delivery mechanism for provisioning. For this use, no device independence is required for attachments, since the attachment objects(s) would have been preselected by the provisioning engine.

Device dependent attachments are supported

MPS can include arbitrary objects in the rendered multipart MIME message, such that they will be treated as an attachment by the receiving MMS client. An attachment in this context is an object that is included as part of the multipart/related MIME message but is not referenced by the markup (root) part of the message (for example the MMS SMIL part in MMS).

You specify the list of attachments through MPSAPI using either local or fully-qualified URLs. A local URLs must be the fully qualified file path to the attachment on the server where MPS is installed.

When a message is rendered for a message type that does not support attachments (such as SMS), then any attachments specified are ignored.

The classes provided for handling message attachments are: `MessageAttachments`, `MessageAttachment` and `DeviceMessageAttachment`.

Note: All slashes ('/') in the file attachment path are treated as path separators for compatibility with Windows and Linux platforms. There is no need to escape space characters.

Create a MessageAttachments object

In the example servlet the following code creates a `MessageAttachments` object from the parameters coming in from the HTTP request.

```
private MessageAttachments getAttachments(HttpServletRequest request) {
  String attachment[] = request.getParameterValues("attachment");
  String attachmentValueType[] = request.getParameterValues(
  String attachmentChannel[] = request.getParameterValues(
  String attachmentDevice[] = request.getParameterValues(
  String attachmentMimeType[] = request.getParameterValues(
    "attachmentMimeType");

  MessageAttachments messageAttachments = new MessageAttachments();
  for(int i=0;i<attachment.length;i++){
    if(!attachment[i].equals("")){
      DeviceMessageAttachment dma = new DeviceMessageAttachment();
      try {
        dma.setChannelName(attachmentChannel[i]);
        dma.setDeviceName(attachmentDevice[i]);
        dma.setValue(attachment[i]);
        dma.setValueType(Integer.parseInt(attachmentValueType[i]));
        if(!attachmentMimeType[i].equals("")){
          dma.setMimeType(attachmentMimeType[i]);
        }
        messageAttachments.addAttachment(dma);
      }
      catch(MessageException me){
        log("Failed to create attachment for "+attachment[i],me);
      }
    }
  }
  return messageAttachments;
}
```

Adding recipients

The following code loads a recipient set from the ServletRequest by looking at parameters 'recipients' and 'device'. If 'device' = "" or there are fewer devices than recipients then no device is specified for the recipient. If the channel is specified as SMS then the MSISDN of the recipient is set rather than the address.

```
private MessageRecipients getRecipients(HttpServletRequest
  request, String inType) throws RecipientException, AddressException{
  String[] names = request.getParameterValues("recipients");
  String[] devices = request.getParameterValues("device");
  String[] type = request.getParameterValues("type");
  String[] channel = request.getParameterValues("channel");
  MessageRecipients messageRecipients = new MessageRecipients();
  for(int i=0;i<type.length;i++) {
    if(type[i].equals(inType)){
      if(!names[i].equals("")){
        MessageRecipient messageRecipient = new MessageRecipient();
        //Is there a channel for this index? Is it not empty?
        if(channel.length>i&&!channel[i].equals("")) {
          // set the channel for the recipient
          messageRecipient.setChannelName(channel[i]);
          if(channel[i].equals("smsc")) {
            // The channel is smsc so set the MSISDN rather than
            messageRecipient.setMSISDN(names[i]);
          }  else if (channel[i].equals("mmsc") && names[i].charAt(0)=='+') {
            messageRecipient.setMSISDN(names[i]);
          }  else {
            messageRecipient.setAddress(new InternetAddress(names[i]));
          }
        } else {
          // channel is not present or empty so use smtp
          // as default - This will have to be changed if the smtp
          // channel is not present in the mcs-config file
          messageRecipient.setChannelName("smtp");
        }

        // If there is a device then set it otherwise let MPS use the
        // defaults
        if(devices.length>i){
          String device = devices[i];
          if(!device.equals("")){
            messageRecipient.setDeviceName(device);
          }
        }
        // Add recipient to the list of MessageRecipients
        messageRecipients.addRecipient(messageRecipient);
      }
    }
  }
  return messageRecipients;
}
```

Resolving channels and devices

Explicit device identification does *not* occur in message preparation. Instead, each user is identified with a device. When page generation is invoked, the name of the target device is supplied. The name is derived from the entry for the recipient. This may happen in one of two ways:

- The name of the device may be supplied within each recipient list entry by the application that creates them
- The device may be supplied via an API invoked in response to the `ResolveDeviceNames` method of the `MessageRecipients` class. This API is customer-written and implements the `MessageRecipientInfo` interface, described in the following section.

Explicit channel identification does *not* occur in message preparation. Instead, each user is identified with a channel. When page generation is invoked, the name of the target channel is found as follows.

- There are default channels for each message type
- The name of the channel may be supplied within each recipient list entry by the application that creates them
- The channel may be supplied via an API invoked in response to the `ResolveChannelNames` method of the `MessageRecipients` class. This API is customer-written and implements the `MessageRecipientInfo` interface, described in the following section.

MPS example servlet

MPS comes with an example servlet. The example code fragments in *Programming MPS* show part of the code in `RunMps` file, which is commented throughout. An example HTML page `MpsRecipient.html` creates the form required for the servlet `RunMps`.

The HTML page

The form's `action` attribute is used to invoke the servlet.

```
<form action="RunMps" method="get" name="recipientForm"/>
```

Load the HTML page using your application server in order to try it out.

The first part of the form is used to collect the message subject and the message source URL or XML source. The second part of the form collects recipient information of type (To, Cc, Bcc), channel (SMTP, MMS, SMS), device (MCS device name) and recipient (address). The next section deals with the attachments and gets the type (File, URL), location (of attachment), channel (smtp, mmsc, smsc), device (MCS device name), mime-type (of attachment).

Using the example servlet

You will need to import the sample policies to your repository if you wish to use the sample MPS servlet. Refer to *Installing MPS* for details.

References

Apache Software Foundation
    `http://apache.org`
Free Software Foundation
    `http://www.fsf.org`
Jaxen Project
    `http://jaxen.org/`
Sun Microsystems Inc
    `http://www.sun.com`