Message Preparation Server

version 1.1

User Guide

This product includes software developed by:
the Apache Software Foundation (http://www.apache.org)
Davisor Ltd. (http://www.davisor.com)
the Free Software Foundation (http://www.fsf.org)
Sun Microsystems Inc. (http://www.sun.com)

Modification 200311121427

Contents

Contents v
Figures x
Tables xi

Chapter 1
Overview 1
Introduction 1
Use of MPS 2
Processing 3
The Main Interfaces 4
Application-side Interface 4
Message Assembly Interface 4
Delivery-side Interface 5
Secondary Interfaces 6
Address to Receipient-type Mapping Interface . . . 6

Chapter 2
Message Authoring 7
Overview 7
Messages 7
message Element 7
JSP 7
XML 7
Other MAML Elements 8
MHTML 8
MMS 8
SMS 9
Example Message 9
Repository Information 10
Layouts 10
MHTML 10
MMS 11
SMS 11
Example Message 11
Themes 13

MHTML 13
MMS 13
SMS 13
Example Message 13
Components and Assets 14
MHTML 14
MMS 14
SMS 14
Example Message 14
Media Transcoding 15
Additional Features Over MCS 15
Targeted Audio Assets 15
DeviceAudioAsset Class 16
MCS Features Not Available with MPS 16
Fragmentation 16
Dissection 16

Chapter 3
Programming MPS in Applications 17
Main Steps 17
Creating the Message 17
Establishing the Session 18
Adding Attachments 18
Create a MessageAttachments Object 19
Adding Recipients 20
Resolving Channels and Devices 22
Example Application 24
HTML Page to Supply Details and Invoke the Servlet 24
MpsRecipient.html 25
Example Servlet 29

Chapter 4
Configuration 37
mariner-config.xml 37
Recipient Mapping Interface Configuration . . 38

Within the **application-plugins** element of the configuration file **mariner-config.xml**, can be placed the **mps** element to specify MPS configuration information. See Figure 4.1: "mariner-config.xml <mps> Element" on page 38 for an example. . . 38

Channels . . . 39

SMTP channel adapter for MHTML messages . . 39

SMSC channel adapter for SMS messages 39

MMSC channel adapter for MMS messages . . . 40

Chapter 5

Message Preparation Services API . . . 41

Overview . . . 41

Packaging . . . 41

Exceptions . . . 41

The com.volantis.mps.attachment Package 42

Class MessageAttachments . . . 42

Packaging . . . 42

Constructors . . . 42

MessageAttachments 42

Methods . . . 42

addAttachment 42

removeAttachment 42

iterator 42

Class MessageAttachment . . . 43

Packaging . . . 43

Constructors . . . 43

MessageAttachment 43

MessageAttachment 43

Methods . . . 43

setValue 43

getValue 44

setMimeType 44

getMimeType 44

setValueType 44

getValueType 45

equals 45

Class DeviceMessageAttachment . . . 45

Packaging . . . 45

Constructors . . . 46

DeviceMessageAttachment 46

DeviceMessageAttachment 46
Methods 46
setDeviceName 46
getDeviceName 46
setChannelName 47
getChannelName 47
equals 47
The com.volantis.mps.message Package 48
The MultiChannelMessage Class 48
Packaging 48
Constructors 48
MultiChannelMessage 48
MultiChannelMessage 48
MultiChannelMessage 48
Methods 48
setMessageURL 48
getMessageURL 49
setMessage 49
getMessage 49
setSubject 49
getSubject 49
generateTargetMessageAsString 49
generateTargetMessageAsMimeMultipart 50
addHeader 50
addAttachments 50
removeAttachments 51
clone 51
The com.volantis.mps.recipient Package 51
The DefaultRecipientResolver Class 51
Packaging 51
Constructors 51
DefaultRecipientResolver 51
Methods 51
resolveChannelName 51
resolveDeviceName 52
The MessageRecipient Class 52
Packaging 52
Constructors 52
MessageRecipient 52
MessageRecipient 52
Methods 53
setAddress 53
getAddress 53
setMSISDN 53
getMSISDN 53

setDeviceName 53
getDeviceName 54
setChannelName 54
getChannelName 54
resolveDeviceName 54
resolveChannelName 55
clone 55
equal 55
The MessageRecipients Class 55
Packaging 55
Constructors 55
MessageRecipients 55
Methods 56
addRecipient 56
removeRecipient 56
resolveDeviceNames 56
resolveChannelNames 56
getIterator 57
The MessageRecipientInfo Mapping Interface 57
Packaging 57
Methods 57
resolveDeviceName 57
resolveChannelName 58
Recipient Mapping Interface Configuration . . 58
The com.volantis.message.session Package 58
The Session Class 58
Packaging 58
Constructor 58
Session 58
Methods 59
addRecipients 59
getRecipients 59
removeRecipients 59
send 60
send 61
Copy List Emulation 61
Blind Copy List Emulation 62
Message Transmission Performance 62

Appendix A
MCS Configuration File 63
Index 67

Figures

Figure 1.1: Generation of Messages 2
Figure 2.1: Example MAML JSP Message 10
Figure 2.2: Layout for Example Message 12
Figure 2.3: MIME type for Formats 15
Figure 3.1: MpsRecipient.html Appearance 24
Figure 4.1: mariner-config.xml <mps> Element 38

Tables

Table A.1: mariner-config.xml Elements for MPS 63

Chapter 1: Overview

Introduction

MCS enables content to be adapted for presentation onto any web-enabled device. MCS (Multi-Channel Server) is tailored to “pull”, or browser-based content delivery. Message Preparation Server (MPS) builds on the core functionality of MCS to allow the optimization of message-based or “push” content. 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 and MPS satisfies this need.

The MPS 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 MAML (Multi-Channel Adaptive Mark-up Language) XML documents, combined with layout policy, themes, components/assets, device and protocol policy, can all be used to create rich message content. The Multi-Channel Server supports MPS in enabling message content to be rendered in a way that is optimised according to the capabilities of the target device.

Four kinds of message output are supported:

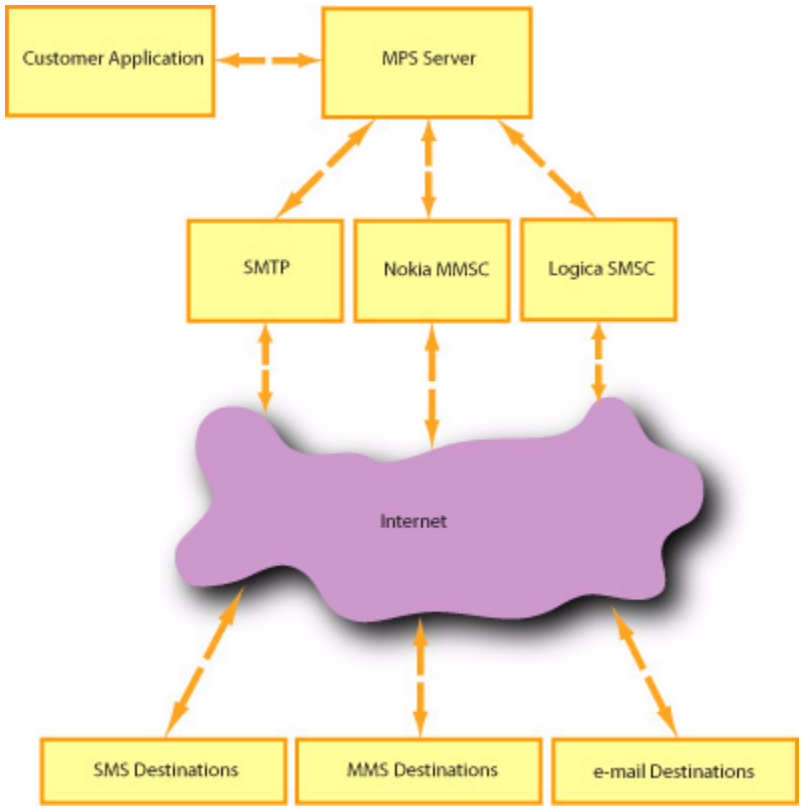
- MMS
- HTML-formatted email (MHTML)
- Plain text (SMS and email)

Only the first three message types may contain rich media content. Plain text message capability is provided as an alternative mechanism to be used when it is determined that a recipient’s device is not capable of accepting MMS messages or where it is uncertain the user will be able to read HTML-formatted email.

Use of MPS

A customer-written application invokes the Message Preparation Server using a public API. The MPS message is authored as a JSP page, or MAML XML file, which uses the MAML *message* element to contain a canvas. The result of processing is a message containing all the content required for rendering on the end-user's device. It is transmitted over the Internet as e-mail traffic using specific e-mail APIs when sending the message. Figure 1.1: "Generation of Messages" on page 2 shows the general arrangement for MPS processing..

Figure 1.1: Generation of Messages



The MPS services are invoked through the Message Preparation Services API (MPSAPI). The Java version of this interface is described in Chapter 5, “Message Preparation Services API” on page 41.

In order to use MPS you must:

1 Write a servlet that will run in the MCS web application and that uses the MPS APIs.

2 Create the device independent message using MAML.

3 Optionally provide an implementation of the mapping from address to device and channel.

Note that callers can request that the messages are returned to them rather than being sent across the network by the 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.

Processing

In order to use MPS is is necessary to create an application which uses the various interfaces which are provided.

The MPS main interfaces consist of the following main elements:

1 An Application-side Interface, as introduced in “Application-side Interface” on page 4.

2 A Message Assembly Interface, as introduced in “Message Assembly Interface” on page 4.

3 A Delivery-side Interface, as introduced in “Delivery-side Interface” on page 5.

In addition there are other, secondary, interfaces. These are introduced in “Secondary Interfaces” on page 6.

The Main Interfaces

Application-side Interface

The application-side interface is in the form of a Java API that 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 a MAML document or a reference to a MAML (Multi-Channel Adaptive Mark-up Language) JSP.

Message Assembly Interface

The message assembly API accepts the following parameters:

1. A reference to (by local URL) the canvas to be rendered as a JSP or MAML document, or copy of the MAML document.
2. The message type to be rendered (MMS, MHTML, SMS).
3. An indication of whether the generated message instances are to be actually submitted after being rendered and assembled, or returned to the requesting entity without submission to any delivery agent.
4. The destination address(es).
5. For each addressee, the name of the device for which the page is to be rendered (optional - if not provided then MPS resolves addresses to device types using an external mechanism).
6. The subject header and any other headers required for the message.

The message assembly mechanism generates complete, valid multipart/related MIME-encoded message bodies containing a parent document, potentially with media objects (assets). In order to do this, the message assembly mechanism performs several basic tasks. In particular it will:

- Determine the device type for each recipient, using an external source (unless the device type is specified in the request).
- For each unique device type represented in the recipient list, one instance of the message is generated as follows (and addressed to all the recipients that are identified as having that device type).
- Render the markup for the message in the required MPS protocol for the delivery mechanism.
- Retrieve all the required assets from their physical locations.

- Encode the asset content using the appropriate encoding mechanism and specifying the correct content type.
- Assemble the whole collection of generated pages and related objects into a single multipart message body.
- Ensure that the appropriate links are maintained between the related message components so that all URL references within the markup that point to other pages or media assets within the message content bundle, are correctly specified.
- For each URL referenced by the presentation part of the message, it is possible for the content author to specify whether the object referenced by the URL is to be packaged in the message or expressed as a fully qualified Web URL. This is to allow messages to contain links to online content as well as "local" content.
- The complete message is checked to see whether it exceeds the maximum size accepted by the target device. If the size cannot be reduced sufficiently, by resizing any covertible images in the message, the message assembly will fail.
- When delivery has been requested, the appropriate delivery agent for each instance of the message is determined, and each message instance submitted to that delivery mechanism along with the appropriate addressing.
- When delivery has not been requested, the rendered and assembled message instances are returned to the requesting entity, along with the appropriate recipient lists.

Delivery-side Interface

An SMTP interface for message submission is available as a method of submitting MMS and HTML e-mail messages. This provides the following functionality:

1. Support for headers including extension "X-" headers.
2. Support for multipart/related MIME-encoded message bodies.
3. It is possible to configure multiple SMTP servers for load balancing and resilience.
4. The (set of) SMTP relays to which this component connects are separately configurable for each message type. (i.e. MMS or MHTML).
5. Optional SMTP authentication is supported. It is possible to specify the SMTP authentication credentials (username/password) individually for each SMTP server.

6 Support for SMS.

7 Support for MMSC API (Nokia).

Secondary Interfaces

Address to Receipient-type Mapping Interface

This interface allows MPS to 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 supplies interfaces. The logic to provide values for these interfaces must be supplied by the customer.

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 (e.g. 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 services from which it will request information will typically be exposed to it via APIs that will 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.

Chapter 2: Message Authoring

Overview

Messages for MPS are authored as a JSP page or as a MAML (Multi-Channel Mark-up Language) XML file in the same way as they are written for the Multi-Channel Server (MCS). All the MAML elements and the Policy Manager facilities of Components, Layouts and Themes are available for construction of the canvases. However ther are some differences which are outlined in the appropriate section below.

The major difference between MCS pages and MPS messages is that MPS messages are enclosed in a MAML *message* element. This is not the case in MCS.

Messages

MPS Messages represent packages that contain MAML canvasses and any media assets required for rendering. The MAML *message* element defines an MPS message.

message Element

The *message* element contains a MAML canvas to be processed as a message. A message consists of a single canvas. This is the canvas which is displayed when the message is read by its recipient.

JSP

```
<vt:message>
<vt:canvas>
   ....
</vt:canvas>
</vt:message>
```

XML

```
<message>
<canvas>
   ....
</canvas>
</message>
```

Other MAML Elements

Many of the MAML elements, defined in the MAML Element Reference can be used in messages you create for delivery by the Message Preparation Server.

There are, however, some differences.

MHTML

Only simple MAML elements are supported, i.e. no structural items are supported such as *table*, *div*, *float*.

MMS

- Element body content is output with whitespace normalised. MMS output is quite different to HTML/XML markup output in that whitespace is significant, therefore input which has effectively random whitespace is so that it looks acceptable in an MMS message.
- Only simple markup layout is simulated:
  - All linefeeds and unecessary spaces are trimmed from element contents, apart from pre.
  - A line break is placed in the output text before and after most block level elements (*p, h1-6, ul, ol, li, dl, dt, dd, pre, div, blockquote, hr, table, address*), *br* and *tr*.
  - Whitespace is added around text for *td*, *th* and *dt*.
  - Ordered and unordered lists are emulated as appropriate, but only the most simple cases and the attributes of these elements are ignored.

Other issues and features:

- Elements which create newlines (such as *p*) tend to break the emulated formatting.
- Character encoding and trimming to size is handled by the MMS channel adapter.
- All form content is ignored.
- No support is provided for inclusion.
- Any HTML entities in the content will be ignored / passed through.

SMS

- Element body content is output with whitespace normalised. SMS output is quite different to HTML/XML markup output in that whitespace is significant, therefore input which has effectively random whitespace is so that it looks acceptable in an SMS message.
- Only simple markup layout is simulated:
  - All linefeeds and unecessary spaces are trimmed from element contents, apart from pre.
  - A line break is placed in the output text before and after most block level elements (*p, h1-6, ul, ol, li, dl, dt, dd, pre, div, blockquote, hr, table, address*), *br* and *tr*.
  - Whitespace is added around text for *td*, *th* and *dt*.
  - Ordered and unordered lists are emulated as appropriate, but only the most simple cases and the attributes of these elements are ignored.

Other issues and features:

- Elements which create newlines (such as *p*) tend to break the emulated formatting.
- Character encoding and trimming to size is handled by the SMS channel adapter.
- All form content is ignored.
- No support is provided for inclusion.
- Any HTML entities in the content will be ignored / passed through.

Example Message

An example MAML JSP message for SMS is shown in Figure 2.1: "Example MAML JSP Message" on page 10.

The message uses a layout called *"message"*. The *canvas* element is uses the *layoutName* attribute to identify the layout to be used.

```
<vt:canvas layoutName="message" title="Demo Message">
```

Initially, the message will display the text *"Here is a short presentation of the four Galilean moons of Jupiter.."*. The *p* (paragraph) element does this, specifying the layout pane to receive the text as *text* with *pane="text.0"*. The text to be placed in the pane is in the body of the *p* element.

```
<vt:p pane="text.0">Here is a short presentation of the four Galilean moons of
Jupiter..</vt:p>
```

Figure 2.1: Example MAML JSP Message

```
<%@ include file="/VolantisNoError-mcs.jsp" %>

<vt:message>
<vt:canvas layoutName="message" title="Demo Message">

<vt:p pane="text.0">Here is a short presentation of the four Galilean moons of
Jupiter..</vt:p>

<vt:p pane="text.1">..first we have Io</vt:p>
<vt:p pane="image.1"><vt:img src="astro/io"/></vt:p>

<vt:img src="astro/europa" pane="image.2"/>
<vt:pane name="text.2">the second is Europa</vt:pane>

<vt:p pane="text.3">third, we have Ganymede</vt:p>
<vt:img src="astro/ganymede" pane="image.3"/>

<vt:pane name="text.4">And finally Callisto</vt:pane>
<vt:img src="astro/callisto" pane="image.4"/>

</vt:canvas>
</vt:message>
```

You may have noted that the pane name is qualified by *.0*. This shows that a temporal iterator (time iterator) is being used here. More detail will be given under “Layouts” on page 10.

Repository Information

When you create MAML messages you use the facilities of the repository in the same way as the Multi-Channel Server (MCS). You are able to specify layouts, themes and components to be used in the message. Please see the MCS Web Authors’ Guide and the Policy Manager Help systems for more detail.

There are, however, some differences from MCS as detailed below in the appropriate section below.

Layouts

Layouts are built for specific devices or groups of devices in the **Layouts** area of the **Policy Manager**. MPS has some differences from MCS in what is permissible. These differences are as follows.

MHTML

- Layout may be in a single pane or a column of single panes.

- No support is provided for dissecting panes. During page generation, dissecting panes are treated as normal panes and dissection is suppressed.
- No support is provided for segments or fragments. Page generation will fail for layouts that are fragmented.

MMS

- Only two row or two column layout (each containing a single pane) is allowed.
- No support is provided for dissecting panes. During page generation, dissecting panes are treated as normal panes and dissection is suppressed.
- No support is provided for segments or fragments. Page generation will fail for layouts that are fragmented.

SMS

- A single pane only is allowed.
- No support is provided for dissecting panes. During page generation, dissecting panes are treated as normal panes and dissection is suppressed.
- No support is provided for segments or fragments. Page generation will fail for layouts that are fragmented.

Example Message

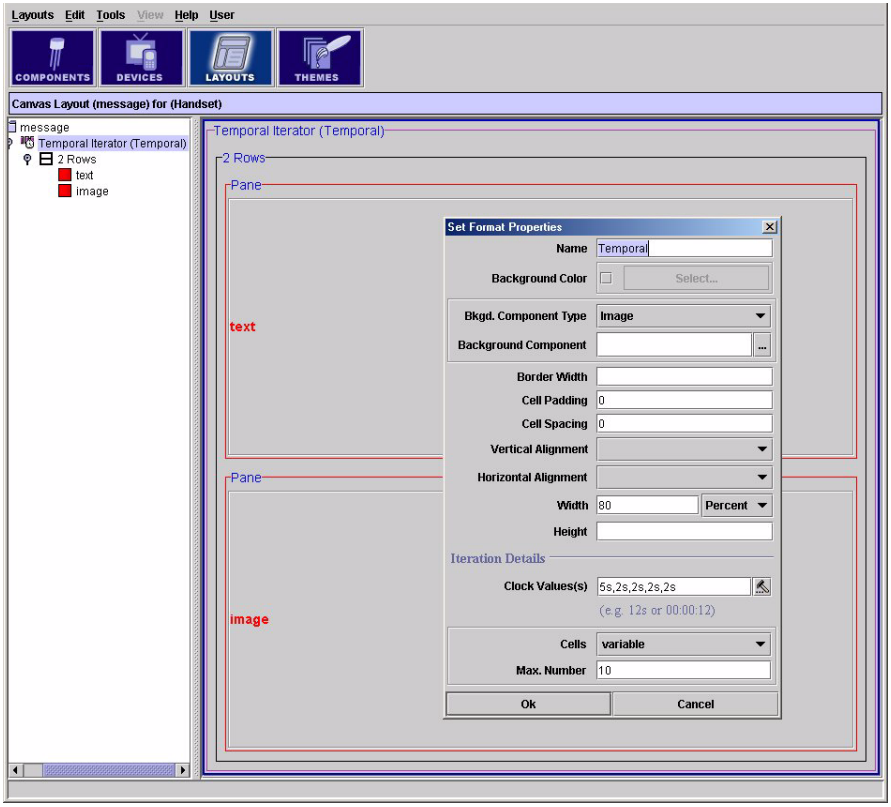
In Figure 2.1: "Example MAML JSP Message" on page 10 the layout to be used is specified in the *canvas* element by the *layoutName* attribute as *message*. This layout can be seen in Figure 2.2: "Layout for Example Message" on page 12.

Within the body of the message are text/image pairs. The first pair is:

```
<vt:p pane="text.1">..first we have Io</vt:p>
<vt:p pane="image.1"><vt:img src="astro/io"/></vt:p>
```

The text is loaded into the *text* pane and the image into the *image* pane. The image to be used is identified by the *img* element *src* attribute. More about the image will be found under “Components and Assets” on page 14.

Figure 2.2: Layout for Example Message



As was said earlier, you may have noted that the pane name is qualified. Here, by *.1*. This shows that a temporal iterator (time iterator) is being used. Looking at Figure 2.2: "Layout for Example Message" on page 12 and you will see the panes are named **text** and **image**, but are within a temporal iterator. This allows the use of qualifiers for the names, starting from 0. Therefore the pane names *text.0, image.0, text.1, image.1, text.2, image.2, text.3, image.3* etc. can be used. By setting appropriate properties for the temporal iterator in the layout, the qualified panes can be made to appear for predetermined periods of time. In Figure 2.2: "Layout for Example Message" on page 12 you will see that the maximum number of cells (qualifiers) has been set to 10. The time for which each of these cells is to be displayed is given as

5, 2, 2, 2, 2 seconds. Note that there are enough clock values for the panes in the message. For details of what would happen if this had not been the case, please see the Policy Manager Help system and the MCS Web Author's Guide.

Themes

The appearance of the canvas on particular devices or groups of devices can be specified in the **Themes** area of the **Policy Manager**. The limitations of the use of Themes with MPS follow.

MHTML

- Themes are treated as for HTML 3.2 without support for stylesheets.

MMS

- All theme/style information is ignored.

SMS

- All theme/style information is ignored.

Example Message

A Theme to be used is specified in the *canvas* element by the *theme* attribute. Here it is *theme4*.

```
<vt:canvas layoutName="layout3" theme=”theme4”>
```

Within the theme, style rules can be created with the Policy Manager which are applicable to particular MAML elements in the message. The style rule to be used from the theme can be specified on the MAML element with the *styleClass* attribute. This can be seen in the following *pane* element:

```
<vt:pane name="cpynews" styleClass=”style1”>
```

and the paragraph, *p*, element:

```
<vt:p pane="trailer" styleClass=”style3”>
```

For more information please see the MCS Web Author's Guide. You should also refer to the Policy Manager on-line help system for more detail.

Components and Assets

Various resources such as images, sound clips and dynamic visuals are given generic component names to be used in MPS messages. This allows MPS to select the most appropriate physical asset to send to a particular target device. The component names and their associated assets with their physical resources are defined in the **Components** section of the **Policy Manager**. The limitations of the use of Components and assets with MPS follow.

MHTML

- No differences.

MMS

- Text fallbacks are used for unsupported non-text components if available.

SMS

- Text fallbacks are used for unsupported non-text components if available.

The generic component to be used in the message is named in the MAML element which handles it. In Figure 2.1: "Example MAML JSP Message" on page 10 the *logo* element uses the *src* attribute to name the component as *stars*.

```
<vt:logo pane="logo1" src="stars" />
```

MPS will look up the *stars* component in the repository and determine which of the assets is most appropriate for the target device and use that one when assembling the message for that device.

Example Message

In Figure 2.1: "Example MAML JSP Message" on page 10 the message contains text/image pairs. The first pair is:

```
<vt:p pane="text.1">..first we have Io</vt:p>
<vt:p pane="image.1"><vt:img src="astro/io"/></vt:p>
```

The text is loaded into the *text* pane and the image into the *image* pane. The image to be used is identified by the *img* element *src* attribute. *"astro/io"* is a component called *io* within the folder *astro*. The component name *io* is an abstract name which is linked to a variety of assets which may be used in

different circumstances. The association between components and their assets is achieved in the Policy Manager. Please see the MCS Web Authors' Guide and the Policy Manager Help system for more information.

Media Transcoding

Instead of relying on an appropriate version of the media being available for the target device, it is possible to use a common resource and to convert it. The Image Conversion Service (ICS) is required for this conversion. For images this conversion involves transforming the size, encoding and color depth to make the resulting asset suitable for the target device.

MCS supports image media transcoding using a variety of media transcoding services, including the Image Conversion Service (ICS) and a software-based mechanism.

For more information please see the MCS Web Author's Guide.

Additional Features Over MCS

Targeted Audio Assets

For details of audio assets please see the MCS Web Authors' Guide and the Policy Manager Help system.

Where a device natively supports an audio format, as with MMS mobile phones for example, targeted audio assets can be used if desired. These assets are associated with a particular device or device family when they are added to the repository. On MMS devices it is possible to associate an individual audio component with each slide in an MMS slide presentation. The media file will be played by the MMS user agent when the corresponding image and/or text are displayed.

Multiple audio assets can be associated with a component referenced by the MAML JSP page, and MCS must select an audio object that is explicitly targeted to the device, or to the closest ancestor in the device hierarchy (provided that the device supports the type of audio asset).

The audio object formats shown in Figure 2.3: "MIME type for Formats" on page 15 are supported.

Figure 2.3: MIME type for Formats

Format	MIME Types
SP-MIDI	audio/sp-midi

Figure 2.3: MIME type for Formats

Format	MIME Types
MIDI	audio/midi, audio/mid, audio/x-midi
SMAF	application/vnd.smaf, application/x-smaf
RMF	audio/rmf, audio/x-beatnic-rmf, audio/x-rmf
iMelody	audio/imelody, audio/x-imy, application/x-imy, text/x-imelody
Nokia Ringtone	application/vnd.nokia.ringing-tone
AMR	audio/amr, audio/x-amr
WAV	audio/x-wav, audio/wav

Where a preference is required (e.g. when there are multiple targeted assets of different types on the same device), the ordering in Figure 2.3: "MIME type for Formats" on page 15 indicates the precedence that is used, from highest (most preferred) to lowest (least preferred).

DeviceAudioAsset Class

There is a new class in the IMDAPI of DeviceAudioAsset. This is used to support the use of audio asssets targeted to devices.

MCS Features Not Available with MPS

Fragmentation

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

Dissection

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.

Chapter 3: Programming MPS in Applications

Main Steps

The main steps in programming MPS applications are:

1. Creating the message,
2. Establishing the session,
3. Adding attachments,
4. Adding recipients,
5. Resolving channels and devices.

These steps will now be demonstrated using the example servlet supplied with MPS. This example differs from an application which is likely to be created by a purchaser of MPS in that:

1. the data about the recipients, their devices and channels, is collected from an HTML form.
2. the details of the message to send is collected from the HTML form as an URL to a MAML XML or the MAML XML message itself.
3. the details of attachments to the message are collected from the HTML form.

It is far more likely that a database of some kind would be used to supply this information.

Creating the Message

The message must be created from MAML elements as decribed in Chapter 2, “Message Authoring” on page 7. Note that either MAML XML, which is classed as an internal request, or a MAML pre-built JSP, which is a servlet request, can be used.

Establishing the Session

The example servlet uses the following code to:

1 Define a MarinerServletApplication called mspExample:

```
MarinerServletApplication mpsExample;
```

2 Initialize mpsExample

```
public void init() throws ServletException {
    super.init();
     mpsExample = MarinerServletApplication.getInstance(
             getServletConfig().getServletContext(),true);
}
```

Adding Attachments

Messages generated for MPS may need to carry arbitrary attachments. Such attachments are not referenced from within the body of the message but do travel with it. The attachments can include things such as:

- Manufacturer-specific ringtones
- Java applications
- Executable content in other formats (e.g. 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 pre-selected by the provisioning engine.

Device dependent attachments are supported through the Java API to MPS.

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 (e.g. the MMS SMIL part in MMS).

The list of attachments object is specifiable explicitly via the MPS Java API using either local or fully-qualified URL(s).

When a message is rendered for a message type that does not support attachments (e.g. 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 with slashes or double quotes.

Create a MessageAttachments Object

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

- @param request

- @return MessageAttachments

```
private MessageAttachments getAttachments(HttpServletRequest request) {

      String attachment[] = request.getParameterValues("attachment");
      String attachmentValueType[] = request.getParameterValues(
                                              "attachmentValueType");
      String attachmentChannel[] = request.getParameterValues(
                                              "attachmentChannel");
      String attachmentDevice[] = request.getParameterValues(
                                              "attachmentDevice");
      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 from the example servlet loads a recipient set from the ServletRequest by looking at parameters "recipients" and "device". If "device" is "" 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.

- @param request - The servletRequest
- @param type - The type of recipients we are trying to load (to,cc,bcc);
- @return MessageRecipients
- @throws RecipientException

- @throws AddressException

```
    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
                                 // address
                                 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 mariner-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.

First, the name of the device may be supplied within each recipient list entry by the application that creates them.

Second, 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.

1. There are default channels for each message type.
2. The name of the channel may be supplied within each recipient list entry by the application that creates them.
3. 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 “The MessageRecipientInfo Mapping Interface” on page 57.

The following code from the example servlet loads a recipient set from the ServletRequest by looking at parameters "recipients" and "device". If "device" is "" 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.

- @param request - The servletRequest
- @param type - The type of recipients we are trying to load (to,cc,bcc);
- @return MessageRecipients
- @throws RecipientException

- @throws AddressException

```
    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
                                 // address
                                 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 mariner-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;
    }
```

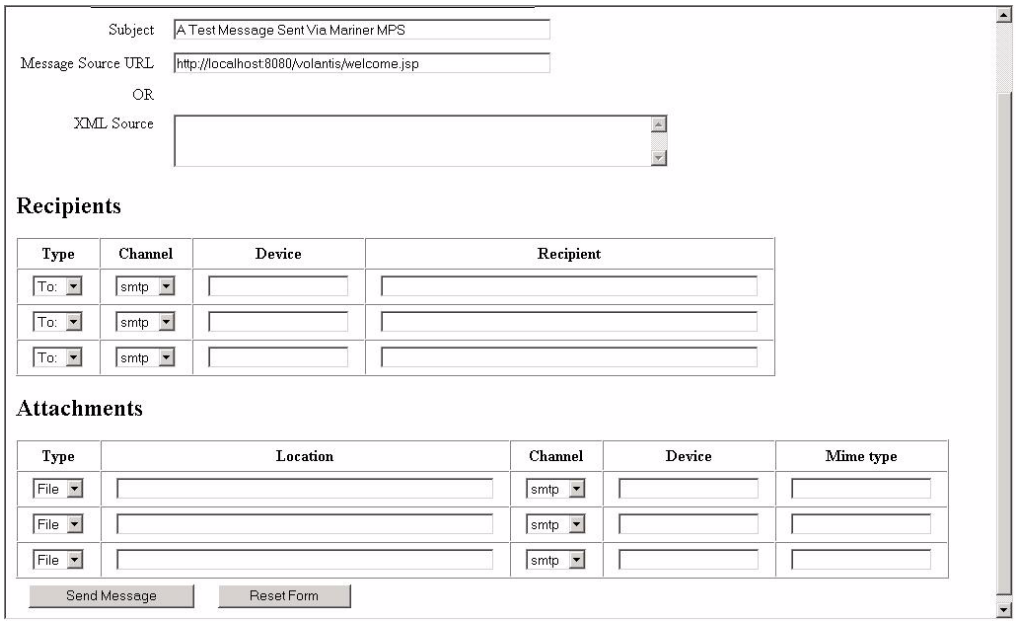
Example Application

The following example uses an example HTML page to deliver information required by the example servlet which uses the MPS API to cause a message to be sent. This is unlikely to be the way in which you would program a live application, as the HTML entry of information is likely to be far too labourious, but the method is used here to illustrate the requirements.

HTML Page to Supply Details and Invoke the Servlet

Under "MpsRecipient.html" on page 25 is the example HTML page **MpsRecipient.html** which creates the form required for the Servlet **RunMps**, shown in "Example Servlet" on page 29. The form's *action* attribute is used to invoke the servlet in **`<form action="RunMps" method="get" name="recipientForm">`**. Load the HTML page using your application server in order to try it out. It should look something like the screen in Figure 3.1: "MpsRecipient.html Appearance" on page 24.

Figure 3.1: MpsRecipient.html Appearance



MpsRecipient.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN" "http://www.w3.org/TR/REC-
html40/loose.dtd">

<html>
<head>
<title>MPS Sample Servlet</title>
<style type="text/css"><!--
body {
   background-color: #FEFEFE;
}
h1 {
   font-weight: bold;
   font-size: 22pt;
}

.action {
   padding-left: 10px;
   padding-right: 10px;
   margin-left: 10px;
   margin-right: 10px;
}
.textmargin {
   margin-left: 10px;
   margin-right: 10px;
}
--></style>
</head>

<body>

<table>
<tr>
<td><img src="images/volantis4.jpg" border="0" alt="MPS"></td>
<td align="left"><h1>MPS Sample Servlet</h1></td>
</tr>
</table>
```

The form to enter values for the messages to be sent. The form's *action* attribute is used to invoke the servlet **RunMps**.

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

The first part of the form is used to collect the message subject and the message source URL or XML source.

```
<table border="0" cellpadding="5" cellspacing="0" >
   <tr>
       <td align="right">Subject</td>
       <td><input class="textmargin" name="subject" size="60" type="text" value="A
Test Message Sent Via MPS" /></td>
   </tr>
   <tr>
       <td align="right">Message Source URL</td>
       <td>
           <input name="vform" type="hidden" value="s0" />
           <input class="textmargin" name="url" size="60" type="text"
value="http://localhost:8080/volantis/welcome.jsp" />
       </td>
   </tr>
   <tr><td align="right">OR</td><td></td></tr>
   <tr>
       <td align="right" valign="top">XML Source</td>
       <td><textarea class="textmargin" cols="60" name="xml"
rows="10"></textarea></td>
   </tr>
</table>
```

The second part of the form collects recipient information of type (to, cc, bcc), channel (smtp, mmsc, smsc), device (MCS device name) and recipient (address).

```
<h2>Recipients</h2>
<table border="1" cellpadding="5" cellspacing="0" >
<th>Type</th>
<th>Channel</th>
<th>Device</th>
<th>Recipient</th>
<tr>
<td><select class="textmargin" name="type"><option value="to">To:</option><option
value="cc">Cc:</option><option value="bcc">Bcc:</option></select></td>
<td><select class="textmargin" name="channel"><option
value="smtp">smtp</option><option value="mmsc">mmsc</option><option
value="smsc">smsc</option></select></td>
<td><input class="textmargin" name="device" size="20" type="text" /></td>
<td><input class="textmargin" name="recipients" size="60" type="text" /></td>
</tr>
<tr>
<td><select class="textmargin" name="type"><option value="to">To:</option><option
value="cc">Cc:</option><option value="bcc">Bcc:</option></select></td>
<td><select class="textmargin" name="channel"><option
value="smtp">smtp</option><option value="mmsc">mmsc</option><option
value="smsc">smsc</option></select></td>
<td><input class="textmargin" name="device" size="20" type="text" /></td>
<td><input class="textmargin" name="recipients" size="60" type="text" /></td>
</tr>
<tr>
<td><select class="textmargin" name="type"><option value="to">To:</option><option
value="cc">Cc:</option><option value="bcc">Bcc:</option></select></td>
<td><select class="textmargin" name="channel"><option
value="smtp">smtp</option><option value="mmsc">mmsc</option><option
value="smsc">smsc</option></select></td>
<td><input class="textmargin" name="device" size="20" type="text" /></td>
<td><input class="textmargin" name="recipients" size="60" type="text" /></td>
</tr>
</table>
```

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).

```
<h2>Attachments</h2>
<table border="1" cellpadding="5" cellspacing="0" >
<th>Type</th>
<th>Location</th>
<th>Channel</th>
<th>Device</th>
<th>Mime type</th>
<tr>
<td><select class="textmargin" name="attachmentValueType"><option
value="1">File</option><option value="2">URL</option></select></td>
<td><input class="textmargin" name="attachment" size="60" type="text" /></td>
<td>
   <select class="textmargin" name="attachmentChannel">
        <option value="smtp">smtp</option><option
value="mmsc">mmsc</option><option value="smsc">smsc</option>
   </select>
</td>
<td><input class="textmargin" name="attachmentDevice" size="20" type="text"
/></td>
<td><input class="textmargin" name="attachmentMimeType" size="20" type="text"
/></td>
</tr>
<tr>
<td><select class="textmargin" name="attachmentValueType"><option
value="1">File</option><option value="2">URL</option></select></td>
<td><input class="textmargin" name="attachment" size="60" type="text" /></td>
<td>
   <select class="textmargin" name="attachmentChannel">
        <option value="smtp">smtp</option><option
value="mmsc">mmsc</option><option value="smsc">smsc</option>
   </select>
</td>
<td><input class="textmargin" name="attachmentDevice" size="20" type="text"
/></td>
<td><input class="textmargin" name="attachmentMimeType" size="20" type="text"
/></td>
</tr>
<tr>
<td><select class="textmargin" name="attachmentValueType"><option
value="1">File</option><option value="2">URL</option></select></td>
<td><input class="textmargin" name="attachment" size="60" type="text" /></td>
<td>
   <select class="textmargin" name="attachmentChannel">
        <option value="smtp">smtp</option><option
value="mmsc">mmsc</option><option value="smsc">smsc</option>
   </select>
</td>
<td><input class="textmargin" name="attachmentDevice" size="20" type="text"
/></td>
<td><input class="textmargin" name="attachmentMimeType" size="20" type="text"
/></td>
</tr>
</table>

<table>
<tr><td></td><td></td></tr>
```

The submit and reset buttons.

```
<tr>
<td><input class="action" type="submit" value="Send Message" /></td>
<td><input class="action" type="reset" value="Reset Form" /></td>
</tr>
</table>

</form>
</body>
</html>
```

Example Servlet

The HTML page **MpsRecipient.html** which collects details and invokes the servlet is shown at "HTML Page to Supply Details and Invoke the Servlet" on page 24.

*Note: You will need to import the sample policies to your repository if you wish to use this sample MPS servlet. The sample policies are in **importsamplempspolicies.xml** in the **repository** directory. Please see the Installation Guide for details of how to do this.*

The initial section deals with the package and imports.

```
package com.volantis.mps.servlet;

import java.io.IOException;
import java.io.StringWriter;
import java.net.URL;
import java.util.Iterator;

import javax.mail.internet.AddressException;
import javax.mail.internet.InternetAddress;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.volantis.mcs.maml.MamlSAXParser;
import com.volantis.mcs.servlet.MarinerServletApplication;
import com.volantis.mcs.servlet.MarinerServletRequestContext;
import com.volantis.mps.attachment.DeviceMessageAttachment;
import com.volantis.mps.attachment.MessageAttachments;
import com.volantis.mps.message.MessageException;
import com.volantis.mps.message.MultiChannelMessage;
import com.volantis.mps.recipient.MessageRecipient;
import com.volantis.mps.recipient.MessageRecipients;
import com.volantis.mps.recipient.RecipientException;
import com.volantis.mps.session.Session;
```

The following request parameters are used to collect recipient and message information.

- recipients - list of recipients to send to

- device - list of devices for each recipient (empty list item will use default device)
- type - Type of recipient (to,cc,bcc)
- channel - Channel to send to (smtpc, mmsc, smsc or as set up in config file)
- subject - Message subject
- url - The url to send as the message
- xml - The MAML XML to parse and send as message. Only used if url is empty
- attachment - List of attachments either a path to a local file or a URL
- attachmentValueType - List of the type of attachments (1 = File, 2= URL)
- attachmentChannel - Channel this attachment will get attached to
- attachmentDevice - Device message for attaching attachment to
- attachmentMimeType - Content/MIME type of this attachment

```
public class RunMps extends HttpServlet{

    private final String store = "RecipientStore";
    private final String RECIPIENTS = "to";
    private final String CCRECIPIENTS = "cc";
    private final String BCCRECIPIENTS = "bcc";
```

The code used to generate an error

```
String failureStart="<canvas layoutName=\"error\">"+
                    "<pane name=\"error\">";
    String failureEnd="</pane>"+
                      "</canvas>";
```

The MarinerServletApplication to initialise MCS

```
    MarinerServletApplication mpsExample;

    public RunMps() {
    }
```

Initialize the MarinerServletApplication as MPS

```
   public void init() throws ServletException {
       super.init();
        mpsExample = MarinerServletApplication.getInstance(
                getServletConfig().getServletContext(),true);
   }
```

Collect recipient information from the servlet request, set up the MPS recipients and session and then send the message

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    MessageRecipients recipients=null;
    MessageRecipients ccRecipients=null;
    MessageRecipients bccRecipients=null;
```

Get the recipients from the servlet request

```
    try {
            recipients = getRecipients(request,RECIPIENTS);
            ccRecipients = getRecipients(request,CCRECIPIENTS);
            bccRecipients = getRecipients(request,BCCRECIPIENTS);
            if(recipients==null){
                throw new RecipientException("No recipients could be found");
            }
        } catch (Exception ae) {
            log("Error loading recipient set", ae);
            writeError(request, response, ae);
            return;
        }
```

Are we sending a URL or an XML message?

```
    String url = request.getParameter("url");
    MultiChannelMessage message =null;

    if(!(url.equals(""))) {
        message = new MultiChannelMessage(new
URL(url),request.getParameter("subject"));
    } else {
        message = new MultiChannelMessage(request.getParameter("xml"),
                                          request.getParameter("subject"));
    }
    try{
        message.addAttachments(getAttachments(request));
    } catch (Exception e) {
        // log the fact that this failed and carry on.
        // Probably not the best thing to do...
        log("Failed to attach attachments", e);
    }
```

Now set up the “from” recipient and send the messages to each recipient specified in the list

```
        MessageRecipient fromUser=null;
        MessageRecipients failures=null;
        try {
            fromUser = new MessageRecipient();
            fromUser.setAddress(new InternetAddress("mps@volantis.com"));
            Session session = new Session();
            session.addRecipients("toList", recipients);
            session.addRecipients("ccList", ccRecipients);
            session.addRecipients("bccList", bccRecipients);
            // Save failures for display later on
            failures = session.send(message,"toList",
                "ccList", "bccList",fromUser );
        } catch (Exception rec) {
            log("Error sending message ", rec);
            writeError(request, response, rec);
            return;
        }
```

Write out the succcess message

```
        writeMesg(request, response, buildSuccessMessage(failures));

    }
```

Use the canvas defined by failurestart/End to build a message

- @param failures List of MessageRecipients that failed
- @return String String of XML that represent MAML

```
    private String buildSuccessMessage(MessageRecipients failures)
    {
        StringBuffer mesg = new StringBuffer();
        mesg.append("<h1>MPS Test Complete. Messages sent</h1>");
        Iterator itr = failures.getIterator();
        if(itr.hasNext()){
            mesg.append("<h3>Failures are</h3>");
            while(itr.hasNext()) {
                MessageRecipient rec = (MessageRecipient) itr.next();
                try{
                    mesg.append(rec.getAddress().toString());
                } catch(Exception e){
                    //ignore this one, we don't really care
                }
                mesg.append("<br>");
            }
        }
        return mesg.toString();
    }
```

Create a MessageAttachments object from the parameters coming in from the HTTP request.

- @param request

- @return MessageAttachments

```
private MessageAttachments getAttachments(HttpServletRequest request) {

      String attachment[] = request.getParameterValues("attachment");
      String attachmentValueType[] = request.getParameterValues(
                                              "attachmentValueType");
      String attachmentChannel[] = request.getParameterValues(
                                              "attachmentChannel");
      String attachmentDevice[] = request.getParameterValues(
                                              "attachmentDevice");
      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;

}
```

Load a recipient set from the ServletRequest by looking at parameters "recipients" and "device". If "device" is "" 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.

- @param request - The servletRequest
- @param type - The type of recipients we are trying to load (to,cc,bcc);
- @return MessageRecipients
- @throws RecipientException

- @throws AddressException

```
    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
                                 // address
                                 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 mariner-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;
    }
```

Write an error out to the response using the canvas defined in failureStart and failureEnd. The layout "error" with a pne error must be defined in the repository for this to work.

- @param request
- @param response

- @param except The exception to write out

```
    private void writeError(HttpServletRequest request,
                            HttpServletResponse response,
                            Exception except) {
        MamlSAXParser parser = new MamlSAXParser();
        try {
            MarinerServletRequestContext msrc = new MarinerServletRequestContext(
                                          getServletConfig().getServletContext(),
                                            request, response);
            parser.setRequestContext(msrc);
            StringWriter writer = new StringWriter();
            writer.write(failureStart);
            writer.write("<h1>MPS error occured</h1>");
          writer.write("<h3>Check Servlet and MCS log for more information</h3>");
            writer.write(failureEnd);
            parser.parseString(writer.toString());
        } catch(Exception e) {
            log("Failed to write error because of ",e);
        }

    }
```

Write a message out to the HTTP response. Uses the failureStart and failureEnd as MAML for the message. The layout error with a pane called error must be defined in the repository.

- @param request
- @param response
- @param mesg The message to write out

```
    private void writeMesg(HttpServletRequest request,
                           HttpServletResponse response,
                           String mesg) {
        MamlSAXParser parser = new MamlSAXParser();
        try {
            MarinerServletRequestContext msrc = new MarinerServletRequestContext(
                                         getServletConfig().getServletContext(),
                                           request, response);
            parser.setRequestContext(msrc);
            parser.parseString(failureStart+mesg+failureEnd);
        } catch(Exception e) {
            log("Failed to write message because of ",e);
        }
}
}
```

Chapter 4: Configuration

Two files are used to configure Mariner. General configuration is achieved using the file **mariner-config.xml**. Logging configuration requires changes to the **mariner-log4j.xml** file.

In the **mariner-config.xml** file the **mps** section is specific to MPS and is explained here. Other sections of the **mariner-config.xml** file are explained in the **Mariner Multi-Channel Server (MCS) Administrator's Guide**.

Mariner MCS must be fully configured and working before Mariner MPS will work successfully. The installation process for MCS will make a default set of entries in the **mariner-config.xm**l file, including those for the repository to be used. For details of how Mariner MCS is configured please see the **Mariner MCS Installation Guide** and the **Mariner MCS Administrator's Guide**.

Details of the use of the **mariner-log4j.xml** file are also given in the **Mariner Multi-Channel Server (MCS) Administrator's Guide**.

mariner-config.xml

The **mariner-config.xml** configuration file defines system parameters for Mariner. It is XML encoded and it is fully specified by the file **mariner-config.dtd**. You'll find both of these files in the directory **WEB-INF**.

The system parameters within the configuration file are grouped into several sections. The most important settings in the file have already been set by the installer and the configuration file should be in a suitable state for you to start using Mariner without any changes.

The contents of the **mariner-config.xml** file relevant to Mariner MPS are listed in Appendix A, "Mariner Configuration File" on page 59.

For all information relevant to Mariner MCS please see the mariner MCS Administrator's Guide.

Recipient Mapping Interface Configuration

Within the **application-plugins** element of the configuration file **mariner-config.xml**, can be placed the **mps** element to specify MPS configuration information. See Figure 4.1: "mariner-config.xml <mps> Element" on page 38 for an example.

Figure 4.1: mariner-config.xml <mps> Element

```
<application-plugins>
   <mps
       internal-base-url="http://mycomputer:8080/volantis"
       message-recipient-info =
                        "com.volantis.mps.recipient.DefaultRecipientResolver"
   >
       <channels>
           <channel name = "smtp"
                   class="com.volantis.mps.channels.SMTPChannelAdapter"
                   host="www.gomail.fm"
                   auth="true"
                   user="user1"
                   password="password1"
           />
           <channel name = "smsc"
                   class="com.volantis.mps.channels.LogicaSMSChannelAdapter"
                   smsc-ip="199.9.9.9"
                   smsc-port="9090"
                   smsc-user="user2"
                   smsc-password="password2"
                   smsc-bindtype="async"
                   smsc-supportsmulti="no"
           />
           <channel name = "mmsc"
                   class="com.volantis.mps.channels.NokiaMMSChannelAdapter"
                   url="http://199.9.9.9:8189"
                   default-country-code="+44"
           />
       </channels>
   </mps>
</application-plugins>
```

The following attributes are available for the **mps** element:

internal-base-url gives the URL to use for asset resolution from internal requests.

message-recipient-info gives the user supplied class used to resolve recipient devices and channels.

Channels

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

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.

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 channel element has the following attributes for each channel:

SMTP channel adapter for MHTML messages

name ="SMTP"

class The class implementing the ChannelAdapter interface for this channel.

host The SMTP relay through which all messages are sent.

auth Determines if sending requests require SMTP authentication.

user User for SMTP Authentication.

password Password for SMTP Authentication.

SMSC channel adapter for SMS messages

name ="SMSC"

class The class implementing the ChannelAdapter interface for this channel.

smsc-ip The IP Address of the SMSC.

smsc-port The port on which the SMSC is listening.

smsc-user User for SMTP Authentication.

smsc-password Password for SMTP Authentication.

smsc-bindtype binding type, value can be *async* or *sync*

smsc-svctype Optional service type. Can be used to indicate the SMS Application service associated with the message. Specifying the service type allows the External Short Message Entity (ESME) to:

- Avail of enhanced messaging services such as message “replace_if_present” by service type (generic).
- Control the teleservice used on the air interface (e.g. ANSI-136/TDMA, IS-95/CDMA).

The following generic service types are defined in SMPP protocol specification version 3.4:

- (NULL) - Default
- CMT - Cellular Messaging
- CPT - Cellular Paging
- VMN - Voice Mail Notification
- VMA - Voice Mail Alerting
- WAP - Wireless Application Protocol
- USSD - Unstructured Supplementary Services Data

All other values are carrier specific and are defined by mutual agreement between the SMSC

smsc-svcaddr Optional sender address

smsc-supportsmulti if the smsc supports SUBMIT_MULTI, *true* or *false*.

MMSC channel adapter for MMS messages

name =“MMSC”
class The class implementing the ChannelAdapter interface for this channel.

url The url of the MMSC.

default-country-code The default country code prefix for recipients without fully qualified msISDN numbers.

Chapter 5: Message Preparation Services API

Overview

This section describes the classes, exceptions, methods and attributes that constitute the **public**, local API for use by messaging applications.

Packaging

The Mariner message preparation services are contained in the packages:

- **com.volantis.mps.attachment**
- **com.volantis.mps.message**
- **com.volantis.mps.recipient**
- **com.volantis.mps.session**

Exceptions

The message preparation services API throws two types of exception. These exceptions form part of the API. The specific exceptions thrown by each method are detailed in the sections that follow.

MessageException This exception encapsulates any error occurring during a message-related operation.

RecipientException This exception encapsulates any error occurring during an operation that manipulates or manages recipients.

The com.volantis.mps.attachment Package

Class MessageAttachments

The **MessageAttachments** class encapsulates an ordered list of attachments to be associated with a message. The list is maintained in the order in which individual message attachments are added to it. New attachments are always added at the end.

MessageAttachments is not a generic protocol independent method for adding logical attachments. Rather its is a protocol specific function that allows messages to be attached to protocols that allow it.

Packaging

The MessageAttachments class is **com.volantis.mps.message.MessageAttachments**

Constructors

MessageAttachments

```
public MessageAttachments()
```

Create a **MessageAttachments** object with no attachements defined.

Methods

addAttachment

```
public void addAttachment(DeviceMessageAttachment deviceMessageAttachment)
```

Add a device specific attachment to the attachment list

Parameters: *deviceMessageAttachment* - The attachment to add.

removeAttachment

```
public void removeAttachment(DeviceMessageAttachment deviceMessageAttachment)
```

Remove a device specific attachment from the attachment list

Parameters: *deviceMessageAttachment* - The attachment to remove.

iterator

```
public java.util.Iterator iterator()
```

Return an iterator of the current attachments

Returns: an iterator of ***DeviceMessageAttachment*** objects

Class MessageAttachment

The **MessageAttachment** class is the super class for message attachments.

Packaging

The **MessageAttachment** class is **com.volantis.mps.message.MessageAttachment**

Constructors

MessageAttachment

```
protected MessageAttachment()
```

Creates a **MessageAttachment** with an unknown value type.

MessageAttachment

```
public MessageAttachment(java.lang.String value,
                         java.lang.String mimeType,
                         int valueType)
```

Creates a **MessageAttachment**.

Parameters:

- *value* - The URL or file that points to a resource to use as a location
- *mimeType* - The mimetype of the resource
- *valueType* - Must be either MessageAttachment.FILE or MessageAttachment.URL depending on resource type

Methods

setValue

```
public void setValue(java.lang.String value)
              throws MessageException
```

Sets the value associated with the attachment. It is either a string representing the URL from which the attachment can be retrieved, or a string representing the name of the local file that contains it. Local in this sense means local to the machine on which the Mariner instance is executing. The value type associated with the instance indicates which of these alternatives is represented by the value.

Parameters: *value* - The URL or file name to attach.

Throws: *MessageException* - Not thrown at present

getValue

```
public java.lang.String getValue()
                            throws MessageException
```

Return the URL or file name that this attachment refers to.

Returns: The URL or file name

Throws: *MessageException* - Not thrown at present

setMimeType

```
public void setMimeType(java.lang.String mimeType)
                  throws MessageException
```

Set the mime type for this attachment. For example "image/gif, text/plain, image/jpeg" etc. If the value represents a URL the MIME type will not be necessary if it can be determined from the URL itself. If, however, the value represents a file, the MIME type value must be specified. If the MIME type is required by MPS to process a message and it is not specified, an exception is raised.

Parameters: *mimeType* - The mime type of the attached content.

Throws: *MessageException* - Not thrown at present

getMimeType

```
public java.lang.String getMimeType()
                                throws MessageException
```

Return the mime type of the content of this attachment.

Returns: the attachment mime type.

Throws: *MessageException* - Not thrown at present

setValueType

```
public void setValueType(int valueType)
                   throws MessageException
```

Set the type of the attachment. Attachments can be one of MessageAttachment.FILE if the attachment content is stored in a file or MessageAttachment.URL if the content is accessed through a URL.

- FILE - The value indicates the name of a file on the server on which the Mariner instance is executing
- URL - The value indicates the URL from which the attachment can be retrieved.

Parameters: *valueType* - The type of the attachment.

Throws: *MessageException* - Not thrown at present

getValueType

```
public int getValueType()
                  throws MessageException
```

Returns the type of the attachment. Attachments can be one of `MessageAttachment.FILE` if the attachment content is stored in a file or `MessageAttachment.URL` if the content is accessed through a URL.

Returns: The type of the attachment.

Throws: *MessageException* - Not thrown at present

equals

```
protected boolean equals(MessageAttachment messageAttachment)
```

Test for equality by test equality of each member and the super class. Log any exceptions that might occur

Parameters: *messageAttachment* - The object we are comparing with.

Class DeviceMessageAttachment

The **DeviceMessageAttachment** class represents a device-dependent attachment to be used with a message. This class inherits from the **MessageAttachment**

Packaging

The **MessageAttachment** class is **com.volantis.mps.message.MessageAttachment.**

Constructors

DeviceMessageAttachment

```
public DeviceMessageAttachment()
```

Create an instance of a device specific attachment with unknown type

DeviceMessageAttachment

```
public DeviceMessageAttachment(java.lang.String value,
                               java.lang.String mimeType,
                               int valueType,
                               java.lang.String deviceName,
                               java.lang.String channelName)
```

Create an instance of a device specific attachment of known type.

Parameters:

- *value* - The URL or name of the file that contains the attachment content.
- *mimeType* - The attachment content type.
- *valueType* - The attachment type (URL or File)
- *deviceName* - The name of the device that this attachment is intended for.
- *channelName* - The name of the channel on which the attachment will be sent.

Methods

setDeviceName

```
public void setDeviceName(java.lang.String deviceName)
                  throws MessageException
```

Sets the name of the device for which this attachment is appropriate. Note that if this value is not set at the time that MPS processes the attachment during message transmission, an exception is raised.

Parameters: *deviceName* - The name of the device

Throws: *MessageException* - Not currently thrown

getDeviceName

```
public java.lang.String getDeviceName()
                                throws MessageException
```

Returns the name of the device for which this attachment is intended

Returns: The name of the device

Throws: *MessageException* - Not currently thrown

setChannelName

```
public void setChannelName(java.lang.String channelName)
                    throws MessageException
```

Sets the name of the channel for which this attachment is appropriate. If this value is not set at the time that MPS processes the attachment during message transmission, the attachment will be used for all channels that support attachments.

Parameters: *channelName* - The name of the channel

Throws: *MessageException* - not currently thrown

getChannelName

```
public java.lang.String getChannelName()
                                  throws MessageException
```

Get the name of the channel on which this attachment will be sent.

Returns: The channel name

Throws: *MessageException* - Not currently thrown

equals

```
public boolean equals(MessageAttachment messageAttachment)
```

Test whether this attachment is equal to another. The attachments are considered identical if all of their fields contain the same values.

Overrides: *equals* in class *MessageAttachment*

Parameters: *obj* - The object being compared

Returns: true if the objects equals this attachment, false otherwise

The com.volantis.mps.message Package

The MultiChannelMessage Class

The **MultiChannelMessage** class encapsulates the device independent message from which the targeted messages are created. Messages are created from a Mariner message definition. This can be supplied either as a String variable or as a URL. Where a URL is provided, the message is read before being processed.

If both a message and a message URL are set, the message takes precedence.

Packaging

The **MultiChannelMessage** class is **com.volantis.mps.message.MultiChannelMessage**.

Constructors

There are 3 constructors. The first has no arguments. The second allows a message URL and a subject to be set. The third allows a message and a subject to be set.

MultiChannelMessage

```
public MultiChannelMessage()
```

Creates a new instance of MultiChannelMessage

MultiChannelMessage

```
public MultiChannelMessage(java.net.URL messageURL,
                           java.lang.String subject)
```

MultiChannelMessage

```
public MultiChannelMessage(java.lang.String messageContent,
                           java.lang.String subject)
```

Methods

setMessageURL

```
public void setMessageURL(java.net.URL messageURL)
              throws MessageException
```

This method sets the URL of the message to be used to generate the targeted messages. The URL must be an object of class **java.net.URL.** The URL **must** resolve to a file in the local file system of the server on which Mariner is running. The contents of the file must be **either** valid MAML XML markup or be a valid JSP containing valid MAML markup.

getMessageURL

```
public java.net.URL getMessageURL()
                throws MessageException
```

This method retrieves the URL of the message to be used to generate the targeted messages.

setMessage

```
public void setMessage(java.lang.String messageContent)
                 throws MessageException
```

This method sets the message to be used to generate the targeted messages. The message must be written in valid MAML XML markup.

getMessage

```
public java.lang.String getMessage()
                throws MessageException
```

This method retrieves the message to be used to generate the targeted messages.

setSubject

```
public void setSubject(java.lang.String subject)
                 throws MessageException
```

getSubject

```
public java.lang.String getSubject()
                throws MessageException
```

generateTargetMessageAsString

```
public java.lang.String generateTargetMessageAsString(java.lang.String
deviceName)
                throws MessageException
```

Generates a message targeted for a particular device type. The method returns the message as a string. Attachments are ignored.

Parameters:

- *deviceName* - The device name to generate this message for
- *channelName* - The channel name to generate this message for

Returns: String The message text

Throws: *MessageException*

generateTargetMessageAsMimeMultipart

```
public javax.mail.internet.MimeMultipart
generateTargetMessageAsMimeMultipart(java.lang.String deviceName)
                throws MessageException
```

Generates a **MimeMultipart** of this message. All attachments are processed for the specified device/channel pair and attached to the MultiPart. The method returns the message as an object of class **javax.mail.internet.MimeMultipart**.

Parameters:

- *deviceName* - The device to generate a message for
- *channelName* - The channel to generate a message for @see MessageChannel

Returns: *MimeMultipart*

Throws: *MessageException*

addHeader

```
public void addHeader(int messageType,
                      java.lang.String name,
                      java.lang.String value)
               throws MessageException
```

Add an additional header to the message. Additional headers can apply to all messages or to messages of a particular type. Control of which message types the header applies to is by the **messageType** parameter. This parameter takes one of a set of values defined within the **MultiChannelMessage** class. The possible values are:

- **MultiChannelMessage.ALL** - The header applies to all types of message.
- **MultiChannelMessage.MHTML** - The header applies to all MHTML messages.
- **MultiChannelMessage.MMS** -The header applies to all MMS messages.
- **MultiChannelMessage.SMS** -The header applies to all SMS messages.

The name and value of the header are specified in the other two parameters to this method.

addAttachments

```
public void addAttachments(MessageAttachments messageAttachments)
                   throws MessageException
```

removeAttachments

```
public void removeAttachments()
                      throws MessageException
```

clone

```
public java.lang.Object clone()
```

The clone returned by this method contains only the following fields setMessageURL setMessage setSubject

Overrides: *clone* in class **java.lang.Object**

The com.volantis.mps.recipient Package

The DefaultRecipientResolver Class

Packaging

The **DefaultRecipientResolver** class is **com.volantis.mps.recipient.DefaultRecipientResolver**

Constructors

DefaultRecipientResolver

```
public DefaultRecipientResolver()
```

Creates a new instance of DefaultRecipientResolver.

Methods

resolveChannelName

```
public java.lang.String resolveChannelName(MessageRecipient recipient)
```

Resolves the channel name for the MessageRecipient.

Specified by: resolveChannelName in interface MessageRecipientInfo

Parameters: *recipient* - The MessageRecipient whos channel needs to be resolved.

Returns: The channel or **null** if unresolved.

resolveDeviceName

```
public java.lang.String resolveDeviceName(MessageRecipient recipient)
```

Resolves the device name for the `MessageRecipient`.

Specified by: resolveDeviceName in interface MessageRecipientInfo

Parameters: *recipient* - The MessageRecipient whos device needs to be resolved.

Returns: •The device or **null** if unresolved.

The MessageRecipient Class

The **MessageRecipient** class encapsulates one destination, or recipient, for the message (MultiChannelMessage). A message recipient includes a mail address and a target device type. This class defines who receives a message and the way in which it is adapted for their device. The recipient should be supplied with an address or phone number as a minimum. If deviceName and channelName are not supplied these will be resolved as needed by the Message Preperation Server.

Packaging

The **MessageRecipient** class is **com.volantis.mps.recipient.MessageRecipient**.

Constructors

There are 2 constructors. The first has no arguments. The second allows an address and a device name to be set. It is valid to specify **null** or an empty string for the device name. However, it must be set or resolved before a message can be sent to it.

MessageRecipient

```
public MessageRecipient()
                    throws RecipientException
```

Creates a new instance of MessageRecipient.

Throws: *RecipientException* - Thrown if the RecipientInfo class can not be found.

MessageRecipient

```
public MessageRecipient(javax.mail.internet.InternetAddress address,
                        java.lang.String deviceName)
                 throws RecipientException
```

Creates a new instance of MessageRecipient.

- Parameters:
- *address* - The recipients internet email address.
- *deviceName* - The name of the device for the recipient.

Throws: *RecipientException* - Thrown if the RecipientInfo class can not be found.

Methods

setAddress

```
public void setAddress(javax.mail.internet.InternetAddress address)
                throws RecipientException
```

This method sets the address of the recipient. The address is an object of class javax.mail.internet.InternetAddress.

Parameters: *address* - The recipients internet email address.

getAddress

```
public javax.mail.internet.InternetAddress getAddress()
                                                    throws RecipientException
```

Gets the internet email address of the recipient.

Returns: The recipients internet email address.

setMSISDN

```
public void setMSISDN(java.lang.String msISDN)
               throws RecipientException
```

Sets the phone number for the recipient.

Parameters: *msISDN* - The recipients phone number.

getMSISDN

```
public java.lang.String getMSISDN()
                           throws RecipientException
```

Gets the phone number of the recipient.

Returns: The recipients phone number.

setDeviceName

```
public void setDeviceName(java.lang.String deviceName)
                  throws RecipientException
```

This method sets the name of the device on which the recipient will receive the message. This name **must** be one of the names of devices or device families in the Mariner repository.

Parameters: *deviceName* - The recipients device.

getDeviceName

```
public java.lang.String getDeviceName()
                                  throws RecipientException
```

Gets the device of the recipient.

Returns: The recipient's device.

setChannelName

```
public void setChannelName(java.lang.String channelName)
                    throws RecipientException
```

This method sets the name of the channel on which the recipient will receive the message. This name **must** be one of the names of channels allowed in Mariner.

Parameters: *deviceName* - The recipients channel.

getChannelName

```
public java.lang.String getChannelName()
                                   throws RecipientException
```

This method retrieves the name of the channel on which the recipient will receive the message.

Returns: The recipients channel.

resolveDeviceName

```
public int resolveDeviceName(boolean force)
                      throws RecipientException
```

This method resolves the name of the device associated with this recipient using external information, the user supplied RecipientInfo, provided via the custom interface described in "The MessageRecipientInfo Mapping Interface" on page 57. If the recipient already has a non-null and non-empty device name associated with it, resolution does **not** occur unless the ***force*** parameter is set to **true**. If resolution does not occur for this reason, the method still returns the **OK** return code.

Parameters: *force* - If true the value resolved will overwrite any existing value for device.

Returns: The success of the resolution either OK or NOT_RESOLVED

resolveChannelName

```
public int resolveChannelName(boolean force)
                        throws RecipientException
```

This method resolves the name of the channel associated with this recipient using external information, the user supplied RecipientInfo, provided via the custom interface described in "The MessageRecipientInfo Mapping Interface" on page 57. If the recipient already has a non-null and non empty channel name associated with it, resolution does **not** occur unless the **force** parameter is set to **true**. If resolution does not occur for this reason, the method still returns the **OK** return code.

Parameters: *force* - If true the value resolved will overwrite any existing value for channel.

Returns: The success of the resolution either OK or NOT_RESOLVED

clone

```
public java.lang.Object clone()
```

Overrides: ***clone*** in class java.lang.Object

equal

```
public boolean equal(java.lang.Object o)
```

The MessageRecipients Class

The **MessageRecipients** class encapsulates a list of recipients. It can be used to represent the recipients, those on the copy list or those on the blind copy list for transmission of a message.

Packaging

The **MessageRecipients** class is **com.volantis.mps.recipient.MessageRecipients**.

Constructors

MessageRecipients

```
public MessageRecipients()
```

Creates a new instance of MessageRecipients

Methods

addRecipient

```
public void addRecipient(MessageRecipient recipient)
                  throws RecipientException
```

Add a MessageRecipient to the MessageRecipients list.

Parameters: *recipient* - The MessageRecipient to add to the list.

removeRecipient

```
public void removeRecipient(MessageRecipient recipient)
                     throws RecipientException
```

Remove a MessageRecipient from the MessageRecipients list.

Parameters: *recipient* - The MessageRecipient to remove from the list.

resolveDeviceNames

```
public int resolveDeviceNames(boolean force)
                       throws RecipientException
```

Resolve the devices for all recipients in the list using the custom interface described in "The MessageRecipientInfo Mapping Interface" on page 57. Resolution does not occur for recipients that already have device names associated with them unless the **force** parameter is set to **true**.

The value returned from this method is the number of recipients whose device could not be resolved. When force is set to **false**, this count **excludes** recipients whose devices were not resolved because they already had associated devices.

Parameters: *force* - If true the value resolved will overwrite any existing value for device.

Returns: The number of recipients with unresolved devices.

resolveChannelNames

```
public int resolveChannelNames(boolean force)
                        throws RecipientException
```

Resolve the channels for all recipients in the list using the the custom interface described in "The MessageRecipientInfo Mapping Interface" on page 57. Resolution does not occur for recipients that already have channel names associated with them unless the **force** parameter is set to **true**.

The value returned from this method is the number of recipients whose channel could not be resolved. When force is set to **false**, this count **excludes** recipients whose channels were not resolved because they already had associated channles.

Parameters: *force* - If true the value resolved will overwrite any existing value for device.

Returns: The number of recipients with unresolved channels.

getIterator

```
public java.util.Iterator getIterator()
```

Get an Iterator<.CODE> for the MessageRecipients. The Iterator is guaranteed to be in channel & device order.

Returns: The Iterator.

The MessageRecipientInfo Mapping Interface

This interface provides a means by which you can provide device name information to MPS programmatically. This interface is invoked by the Java API implementation in response to the **resolveDeviceName()** method on the **MessageRecipient** class being invoked.

The **MessageRecipientInfo** interface provides the means by which customer-written code can provide particular additional information to objects of the **MessageRecipient** class. Implementations of this interface provide address-to-device type mapping, relating users and the devices they use.

This interface should be supported by a user supplied routine to supply device and channel information for a particular MessageRecipient. The use if this routine is configured via the **mariner-config.xml** file.

```
public interface MessageRecipientInfo
```

Packaging

The **MessageRecipientInfo** interface is **com.volantis.mps.recipient.MessageRecipientInfo**.

Methods

resolveDeviceName

public java.lang.String resolveDeviceName(MessageRecipient recipient)

Resolves the device name for the **MessageRecipient**.

Parameters: *recipient* - The MessageRecipient whose device needs to be resolved.

Returns: The device or **null** if unresolved.

resolveChannelName

```
public java.lang.String resolveChannelName(MessageRecipient recipient)
```

Resolves the channel name for the **MessageRecipient**.

Parameters: *recipient* - The MessageRecipient whose channel needs to be resolved.

Returns: The channel or **null** if unresolved.

Recipient Mapping Interface Configuration

Details of the configuration requirements for the receipient mapping interfaces are given in “Recipient Mapping Interface Configuration” on page 38.

The com.volantis.message.session Package

The Session Class

The **session** class encapsulates an application's interaction with Mariner's message preparation services. The session can hold information, such as distribution lists, used in multiple message transmissions. The session is the object that sends messages. In addition, the session is the focus for creation and management of connections with underlying message transport mechanisms. Sessions must be created before messages can be transmitted.

Packaging

The **Session** class is **com.volantis.mps.session.Session**.

Constructor

Session

```
public Session()
        throws MessageException
```

Creates a new instance of Session

Throws: *MessageException* - Thrown if an error occurs whilst initialising channels.

Methods

addRecipients

```
public void addRecipients(java.lang.String name,
                          MessageRecipients recipients)
                   throws RecipientException
```

This method adds a list of recipients to the session under the specified name (a named Messagerecipients list). Subsequent operations such as sending a message can then refer to this named list. This allows lists to be constructed once and reused for multiple messaging operations if desired.

Parameters: •

- *name* - The name of the list
- *recipients* - The MessageRecipients containg the list.

Throws: *RecipientException* - Thrown if an error occured in adding the list to the session.

getRecipients

```
public MessageRecipients getRecipients(java.lang.String name)
                                 throws RecipientException
```

Get the named recipient list from the session, allowing it to be used in a messaging operation.

Parameters: *name* - The name of the list.

Returns: The MessageRecipients containg the list.

Throws: *RecipientException* - Thrown if an error occurs whilst retrieving the list.

removeRecipients

```
public void removeRecipients(java.lang.String name)
                      throws RecipientException
```

Removes a named MessageRecipients list to the session.

Parameters: *name* - The name of the list

Throws: *RecipientException* - Thrown if an error occurs whilst removing the list.

send

There are three methods for sending messages. One sends to a single recipient. The second sends to a single list of recipients. The third sends to three such lists. These three are the usual set of the 'to' list, the 'copy' list and the 'blind copy' list. Notice that lists of recipients are referred to by **name**. The lists must be constructed and added to the session under an appropriate name before being used.

The **send** operation causes generation of target messages for the appropriate devices and their transmission over the appropriate mail provider.

Note that where a mail provider supports copy lists, the list of recipients that appears in each copy of the message will contain **only** those recipients on a common provider.

The **replyTo** parameter represents the recipient for any replies to the message. If set to **null**, generated messages will not contain reply information. Some protocols may throw a **RecipientException** if this is **null**.

The value returned by send operations is a count of the number of destinations in the recipient list for which the message could not be sent. Note that, if any of the recipients in the list is itself a mailing list rather than an individual, this count will not represent the true number of recipients.

send

```
public MessageRecipients send(MultiChannelMessage message,
                              MessageRecipient recipient)
                       throws MessageException
```

Sends a MultiChannelMessage to a single recipient.

Parameters:

- *message* - The MultiChannelMessage to send.
- recipient - The MessageRecipient to send this message to.

Returns: A MessageRecipients list of failed MessageRecipients.

Throws: *MessageException* - Thrown if an error occurs in sending the message.

send

```
public MessageRecipients send(MultiChannelMessage message,
                              java.lang.String toListName,
                              MessageRecipient replyTo)
                       throws MessageException
```

Sends a MultiChannelMessage to a named recipient list.

Parameters:

- *message* - The MultiChannelMessage to send.
- *toListName* - The recipient list previously added to the Session.
- *replyTo* - The MessageRecipient who will be the sender of the message.

Returns: A MessageRecipients list of failed MessageRecipients.

Throws: *MessageException* - Thrown if an error occurs in sending the message.

send

```
public MessageRecipients send(MultiChannelMessage message,
                              java.lang.String toListName,
                              java.lang.String ccListName,
                              java.lang.String bccListName,
                              MessageRecipient replyTo)
                       throws MessageException
```

Sends a MultiChannelMessage to a named recipient list.

Parameters: •

- *message* - The MultiChannelMessage to send.
- *toListName* - The to recipient list previously added to the Session.
- *ccListName* - The cc recipient list previously added to the Session.
- *bccListName* - The bcc recipient list previously added to the Session.
- *replyTo* - The MessageRecipient who will be the sender of the message.

Returns: A MessageRecipients list of failed MessageRecipients.

Throws: *MessageException* - Thrown if an error occurs in sending the message.

Copy List Emulation

Where copy lists are not supported explicitly by a mail provider, the list of recipients and the list of copied recipients must effectively be combined to form the set of destinations for the message.

Blind Copy List Emulation

Where blind copy lists are not supported explicitly by a mail provider, the message must also be sent to all recipients being blind copied. However, in this case it is important that the list of normal recipients and copied recipients are not aware of the blind copied recipients. They must also be unaware of one another. Consequently, blind copied recipients are sent individual copies of the message in separate operations. In addition, the text **bcc:** is added to the subject line, for those recipients, if allowed by the transport.

Since recipients of 'bcc' copies must not know about each other and must not be seen by recipients on the other lists, messages will be sent **individually** to recipients on this list.

Message Transmission Performance

Performance of operations that send messages is extremely important, especially where a large number of recipients is involved, therefore MPS will:

- Minimise the number of times the target message must be generated, by caching the final result and reusing it wherever possible.
- Minimise the number of potentially expensive session or connection operations with mail providers. Operations that send to multiple recipients are used wherever possible.

Appendix A: MCS Configuration File

Table A.1: mariner-config.xml Elements for MPS

Element	Children	Used by	Attribute	Description	Type, optional?	Values, default
application-plugins	mps			Definitions of plugins		
mps			**internal-base-url**	The url to use for asset resolution from internal requests.	string	
			message-recipient-info	the user supplied class used to resolve recipient devices and channels.	string	
channels	channel	mps				
channel		channels				
			name	Name of the channel	string	smtp smsc mmsc
			class	The class implementing the Channel-Adapter interface for this channel.		
			host	If *name="smtp"*, the SMTP relay through which all messages are sent.		
			auth	If *name="smtp"*, determines if sending requests require SMTP auth entication.		
			user	If *name="smtp"*, user for SMTP Authentication.		
			password	If *name="smtp"*, password for SMTP Authentication.		

Table A.1: mariner-config.xml Elements for MPS

Element	Children	Used by	Attribute	Description	Type, optional?	Values, default
			smsc-ip	If *name="smsc"*, the IP Address of the SMSC.		
			smsc-port	If *name="smsc"*, the port on which the SMSC is listening.		
			smsc-user	If *name="smsc"*, user for SMTP authentication.		
			smsc-password	If *name="smsc"*, password for SMTP authentication.		
			smsc-bind-type	If *name="smsc"*, value can be "async" or "sync"		
			smsc-svc-type	If *name="smsc"*, service type (SMPP protocol)	string, optional	(null) - default CMT - Cellular Messaging CPT - Cellular Paging VMN - Voice Mail Notification VMA - Voice Mail Alerting WAP - Wireless Application Protocol USSD - Unstructured Supplementary Services Data
			smsc-svcaddr	If *name="smsc"*, sender address	string, optional	

Table A.1: mariner-config.xml Elements for MPS

Element	Children	Used by	Attribute	Description	Type, optional?	Values, default
			smsc-sup-portsmulti	If *name="smsc"*, if the smsc supports SUBMIT_MULTI	boolean	true false
			url	If *name="mmsc"*, the URL of the MMSC.	url	
			default-contry-code	If *name="mmsc"*, the default country code prefix for recipients without fully qualified msISDN numbers.	string	

Index

A

address-to-device mapping 6
application-side interface 4
assets 4

B

bcc
62
blind copy lists 62

C

canvas 4
channel identification 22
classes 41
com.volantis.mps.message.MultiChannelMessage 48, 51
com.volantis.mps.recipient.MessageRecipientInfo 57
com.volantis.mps.recipient.MessageRecipients 55
com.volantis.mps.recipient.MessageRecipient 52
com.volantis.mps.session.Session 58

D

delivery agent 4, 5
destination address(es) 4
device 4
device identification 22
device type 4
device-independent message template 4
dissection 7

E

e-mail APIs 2
exception 41
extension “X-” headers 5

F

Fragmentation 16
fragmentation 7

G

generated message instances 4
generated pages 5

H

HTML 5

J

JSP 7

L

links 5

M

Marlin message element 7
message assembly 4
message assembly API 4
message element 7
message type 4
MessageRecipient class 52
MessageRecipientInfo interface 22, 57
MessageRecipients class 22, 55
MIME-encoded 4, 5
MMS 5
Mobile Network APIs 6
Mobile subscriber database 6
MPS message 2
MultiChannelMessage class 48
multipart message body 5
multiple SMTP servers 5

P

package com.volantis.mps 41
protocol 4

R

RecipientException 60
ResolveChannelNames() method 22
resolveDeviceName() method 57
ResolveDeviceNames() method 22

S

session class 58
SMTP authentication 5
SMTP interface 5
SMTP relays 5
subject header 4

U

URL 5

X

XML 7