

Adding a Custom Message Format

This topic explains how to extend Virtualize's interface and operations to cover message formats not supported by default.

Sections include:

- [About Custom Message Formats](#)
- [Interfaces to Implement for Custom Message Formats](#)
- [Defining parasoft-extension.xml for a Custom Message Format](#)
- [Verifying the New Message Format](#)
- [Tips](#)

About Custom Message Formats

SOAtest and Virtualize include a framework that allows you to extend their built-in message formats. The framework can support any message format that you are working with—for example, mainframe message formats, binary formats, or any other kind of proprietary custom message format. The message formats are defined by creating a conversion between the native format and XML. This conversion allows the user to construct and validate messages using the rich XML tooling that is available. This extension is done using Java.

Once a custom message format has been added, Virtualize will automatically make a new responder available for configuring and sending request or response messages using that format. You can add instances of the new client/responder to your test scenarios or responder suites.

See [Custom Message Responder](#) for details about how to use them. The new message format will also appear in the XML Converter tool, which is described in [XML Converter](#).

In addition to defining a general conversion between formats, you can also optionally define a set of specific message types that define the exact structure of each message within that format. For example, a user might define a general FIX to XML conversion, but then also define a set of specific FIX messages that are used by their application. The structure of each message type is defined with an XML schema and is provided by the SOAtest or Virtualize extension.

Interfaces to Implement for Custom Message Formats

After setting up your environment ([General Procedure of Adding an Extension](#)), implement the following interfaces (described in the Extensibility API documentation):

- `com.parasoft.api.message.ICustomXMLConverter`
- `com.parasoft.api.message.ISchemaGenerator` (optional)

ICustomXMLConverter Implementation

This is a required class that is used to implement the conversion logic to and from the native format and XML. The `toXML()` method should return your message in XML format, while the `toNative()` method should return your message in its native format. The methods take:

- An `INativeMessage` or `IXMLMessage`, which can be used to retrieve a representation of the message to be converted.
- An `IConversionContext`, which provides configuration settings and a way to share data between the different methods. In addition, `IConversionContext` extends `ScriptingContext`, which is a standard Parasoft scripting context that allows for accessing variables, data source values, and setting/getting objects for sharing across tools.

`toXML()` returns an `INativeMessage` while `toNative()` returns an `IXMLMessage`. There are default implementations of these interfaces when can be used for convenience (`DefaultNativeMessage` and `DefaultXMLMessage`).

For `toNative`, implementors must handle the case where the message type for the `xmlMessage` parameter is `{@code null}`. If the particular conversion being defined does not know how to convert from XML to the native message without being passed the specific message type, then a `{@link CustomConversionException}` should be thrown.

For `toXML`, the conversion may optionally be written so that it can auto-detect the message type for the passed-in `nativeMessage`. This auto-detection is used to determine the message type when switching from literal to form views or when using the traffic wizards to generate assets. If the conversion does not support auto-detection, you will need to explicitly choose the message type, or else SOAtest/Virtualize will try to detect the message type (which in some cases may be incorrect). To support auto-detection, implementors should ignore the message type for the passed-in `nativeMessage`. They should instead determine the message type themselves and set it into the returned XML message.

Implementors must handle the case where the message type for the `nativeMessage` parameter is `{@code null}`, because that is how support for auto-detection is determined. If the particular conversion being defined does not support auto-detection and thus does not know how to do the conversion without being passed the message type, then a `{@link CustomConversionException}` should be thrown.

Errors that are encountered during conversion can be handled in two different ways. If the error is a fatal error that must abort the conversion, a `CustomConversionException` with an appropriate error message should be thrown. If the error is non-fatal (meaning the conversion can continue), a message can be reported using `IConversionContext.report(String message)` and a (possibly partial) converted message should still be returned by the method. If an error is reported by throwing an exception or calling `report()`, it will be reported either as a dialog in the client/responder UI, or as an error message when the client/responder is run.

These conversions are executed from clients/responders and the XML Converter tool in the following situations:

- Switching between form and literal views
- Constructing a message from Form Input or Form XML view to send over the wire
- Processing a received message
- Executing the XML Converter tool

ISchemaGenerator Implementation

When defining specific message types for a given message format, you must provide an XML schema for each message type. This is an optional class that provides one method of providing an XML schema for a given specific message type. If you don't implement this class and reference it in `parasoft-extension.xml`, you must instead provide references to schema files (see [Defining parasoft-extension.xml for a Custom Message Format](#) below).

This interface has one method, `generateSchema()`, which takes:

- An `IMessageType`, which provides information about the message type for which a schema should be generated. The instance of `IMessageType` that is passed in will contain the id and name of a message type that was defined in `parasoft-extension.xml`.
- An `ICustomXMLConverterConfiguration`, which provides access to the values that come from the Conversion Options tab of the custom message format clients/responders.

`generateSchema()` returns a URI that is a reference to a schema file. It can be a reference to a dynamically generated schema, a reference to a schema that lives in the jar file providing the custom message format extension, or a reference to an external resource. An empty URI can be returned if there is an error in generating the schema; this will result in the Form Input view not being populated appropriately for the given message type.

Defining parasoft-extension.xml for a Custom Message Format

After you have implemented the necessary classes, define `parasoft-extension.xml` (introduced in [General Procedure of Adding an Extension in Virtualize](#)) using the following schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<extension xmlns="urn:ocm:parasoft/extensibility-framework/extension"
           type="messageFormat"
           name='the name of your message format, appears in menus' description='A more detailed
description'>
  <class>com.myCompany.MyConverter</class> <!-- implements ICustomXMLConverter -->
  <messageFormat xmlns="http://schemas.parasoft.com/extensibility-framework/messageFormat">
    <defaultMimeType>text/plain</defaultMimeType>
    <messageTypes generatorClass="com.myCompany.SchemaGenerator">
      <messageType id='unique ID' name='name' xsd='path to schema in jar' />
      . . .
    </messageTypes>
    <client icon="myClient.gif" defaultTransport='default transport' />
    <responder icon="myResponder.gif" />
  </messageFormat>
  <version id='your version ID' updaterClass="com.myCompany.myUpdater" />
  <form xmlns="urn:com:parasoft/extensibility-framework/gui">
    <section label="field group 1">
      <field id="key 1" label="field 1" />
      <field id="key 2" label="field 2" />
    </section>
    <section label="field group 2">
      <field id="key 3" label="field 3" />
      . . .
    </section>
    . . .
  </form>
</extension>
```

The field ids under the section elements are used to retrieve values from the `ICustomXMLConverterConfiguration` instance that is passed in to the various API methods, and that allows for the values provided by the user to be passed in to your implementation. They are also used to persist the userprovided values to the responder deployment descriptor file when it is saved. The field labels appear in the GUI that is constructed based on this XML. The section layout does not have a programmatic impact; it merely helps organize the various GUI fields into sections or categories for easy access and usability by the end user.

The fully qualified class name of the class that implements `ICustomXMLConverter`, along with the name of the extension, is used to identify the message format that is being used in clients/responders. Changing the qualified class name or the name of the extension after you have saved `.tst` or `.pva` files will prevent saved clients or responders from being able to resolve the message format that they are using.

<MessageFormat> Element

This element is unique to this extension type. It must be valid and correct for your custom message format to be imported.

- <defaultMimeType>: Specifies the default MIME type for your message format. The MIME type is used to identify the type of information that your message contains. The default MIME type is used as the Content Type in message headers when sending messages.
- <messageTypes>: This is an optional element that defines a set of specific message types. If no messageTypes element is provided, you are defining a general conversion between a native format and XML that has no specific message types defined. An example of such a format is CSV.
 - generatorClass - An optional attribute that holds the fully-qualified name of a class which implements ISchemaGenerator.
- <messageType>: Defines a message type for your custom message format.
 - id - A string identifier for your message type. Must be unique within a given message format. Also is used when saving clients /responders, so changing it will result in saved clients or responders not being able to resolve the message type that they are using.
 - name - Used to display the name of your message type in the GUI.
 - xsd - A URL path to your xsd. This attribute is required if you did not provide a generator class. Otherwise, it is ignored.
- <responder>: Optional element that holds basic information for the custom Virtualize responder generated for your custom message format.
 - icon - An optional attribute that specifies an icon to display in the GUI for the custom Virtualize responder. This is a relative path to an icon that is contained within a jar file or Java project on the Virtualize classpath. If not provided, Virtualize will use its default tool icon for the custom Virtualize responder.

Verifying the New Message Format

1. Build the project (see [General Procedure of Adding an Extension in Virtualize](#)) and restart Virtualize.
2. Verify that a new client/responder that contains the name of your message format is available to add to a .tst or .pva file by choosing **Add New> Test or Responder**.

Your new format should also be listed in the drop-down menu in the XML Converter.

Tips

- GUI fields that are defined in the parasoft-extension.xml file appear in the Conversion Options tab of the custom message format client/responder.
- The values provided to the extension GUI are saved as a name-value String map. As a result, rearranging the fields in the form element in parasoft-extension.xml will not affect how the user values are saved; however, changing the ids will affect this. The ids are used to save/load the values so they need to be unique. If you change them, then previously-saved configurations will not load the previous values and will become empty. However, you can use a version updater to migrate old settings saved with old ids to a new set of ids.
- Only GUI fields with string values are supported in the custom form GUI. If your extension requires integers or other types, then you may convert the string content to the desired type in the extension implementation.
- Tables or lists can be implemented as comma-separated values in the string fields.