



PONTON
CONSULTING

Specification

EFET Box+ 3.2 Adapter Programming Guide

Written by
Ponton Consulting GmbH
Dorotheenstr. 60
22301 Hamburg

Contact:
Dr. Michael Merz
Tel.: +49 40 69 213 341
Mob.: +49 40 170 85 22 894
Email: merz@ponton-consulting.de

Content

Content	2
1 Introduction	3
2 Interface ISpecificAdapter	4
3 GenericAdapter	6
4 BackEndMessage	7
5 MessageResult	8
6 Sample Adapter Source Code	9
7 MessageResult Codes	14
7.1 GenericAdapter.sendMessage()	14
7.2 GenericAdapter.sendPing()	16
7.3 GenericAdapter.shutdown()	16
7.4 ISpecificAdapter.receive...()	16

1 Introduction

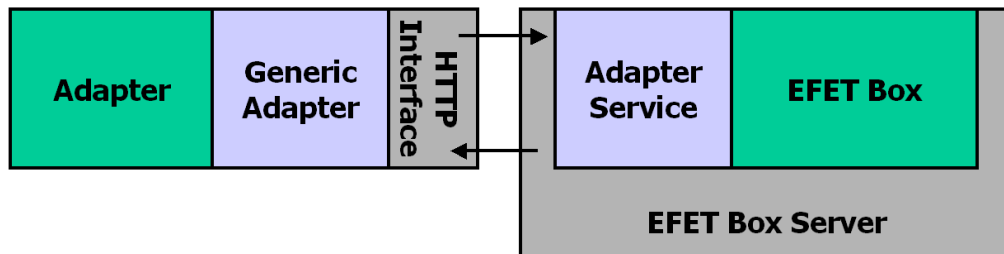
This document describes the basics how to write an Adapter for the EFET Box + version 3.2 which covers the EFET Standard 3.2.

If you plan to implement your own Adapter, you should be familiar with object-oriented programming, inheritance, interface implementation, and the Java programming language.

Every Adapter has to implement the interface `ISpecificAdapter` from the package `de.pontonconsulting.xmlpipe.adapter`. This interface contains several methods that have to be implemented by the Adapter programmer.

Other essential classes are `GenericAdapter`, `MessageResult` from the package `de.pontonconsulting.xmlpipe.adapter` and `BackEndMessage` from the package `de.pontonconsulting.xmlpipe.message`.

1.1.1 Message Flow



The communication between the EFET Box+ and an Adapter are standard HTTP calls.

All the data sent from the Adapter to the EFET Box+ is sent by the `GenericAdapter` to the `AdapterService` servlet running in the same webserver as the EFET Box+. The Adapter calls methods in its `GenericAdapter` instance to invoke the sending of data.

Data from the EFET Box+ to the Adapter is sent to the `HttpInterface` started by the `GenericAdapter` instance of the Adapter. The `GenericAdapter` reconstructs the data and calls the defined methods of the Adapter.

2 Interface ISpecificAdapter

The following methods need to be implemented by the Adapter programmer.

public String getID()

This method should return the identifier of the Adapter. It will be passed to the EFET Box+ on Adapter registration. The identifier should be like 'AdapterName-IdNumber' and has to be unique for a EFET Box+ installation.

public String getStatus()

This method should return the status of the Adapter in human readable text form – like 'The Adapter is ready to receive messages'.

public boolean supportsAcknowledgements()

This option will be ignored by the EFET Box+. Right now all the feedback of the EFET Box+ will be handled through BoxResult documents send to the Adapter.

public boolean supportsAttachments()

This option will be ignored by the EFET Box+. It could be used in future releases to identify the capability of receiving attachments.

public int getNumberOfParallelThreads()

This method should return the maximum number of parallel receiving threads, if the Adapter supports multithreaded receiving. If the Adapter does not support multithreaded receiving, this method should return the int 1.

public MessageResult receiveMessage(BackendMessage message)

This method is called when a new Message is received by the GenericAdapter.

public MessageResult receiveTestMessage(BackendMessage message)

This method is called when a new test message is received by the GenericAdapter. It is not required that the adapter performs any special treatments to test messages – instead the usual `receiveMessage` method might be called. This method just helps to recognise test messages without parsing them.

public MessageResult receiveAcknowledgement(BackendMessage message)

This method is called when an acknowledgement is received by the GenericAdapter. But the method is only called if `supportsAcknowledgements` returns true.

public File getWorkFolder()

This method returns the work-folder of the adapter. When a message contains attachments, the GenericAdapter will save them into a subfolder (named like the message ID) of the work folder. HttpRequests from the EFET Box+ will be saved into this folder temporary.

public boolean doSelfCheck()

This method is called to test the adapters special functions. It should return true, if the adapter is working properly.

public String shutdown()

When this method is called, the adapter should perform a clean shutdown and free all used resources. It can return a shutdown message to the EFET Box+.

3 GenericAdapter

When starting an adapter, it has to create an instance of the GenericAdapter. The GenericAdapter handles the communication between the adapter and the EFET Box+. On start-up it will register the Adapter at the EFET Box+.

To create the GenericAdapter instance it is recommended to use the following constructor:

```
public GenericAdapter(IspecificAdapter endAdapter, String efetBoxHost, int messengerPort, String messengerPath, String logCatefory)
```

```
// Create the GenericAdapter instance which provides the connection between the EFET Box+
// and the Adapter. The Adapter will be registered automatically with the EFET Box+.
// The EFET Box+ is contacted on the given hostname and port number - the EFET Box's
// AdapterService will be reached on the given path.
// The connection string looks like this: http://hostname:portnumber/path
//
// new GenericAdapter(endAdapter, hostname, portnumber, path, logCategory);
```

```
GenericAdapter ga;
```

```
ga =new GenericAdapter(this, "localhost", 8880, "/efet/AdapterService", getID() );
```

To check if a partner ID exists in the EFET Box partner configuration the method `partnerExists(String partnerId)` can be used. It will return a boolean.

To obtain the complete list of partner IDs the method `getFullPartnerList()` can be used. It will return a String-array with all configured partner IDs.

To get only the local partners use the method `getLocalPartnerList()`. To get only the remote partners use the method `getRemotePartnerList()`.

With the method `sendPing(String senderId, String receiverId)` it is possible to send an ebXML-Ping to a remote partner. It does not make sense to send pings, if the adapter does not support acknowledgements, because the resulting ebXML-Pong message will be passed to the adapter as an acknowledgement. This method returns an instance of the `MessageResult` class. See the JavaDoc for information on the `MessageResult` class.

To send a message to a remote partner use the method `sendMessage(BackendMessage message)`. This method will return a `MessageResult`, which contains the information about the local processing. If the message was processed successfully by the receiver is noted in the asynchronously received acknowledgement.

To stop the GenericAdapter use the method `shutdown()`.

4 BackEndMessage

When sending or receiving a message an instance of the `BackEndMessage` class is used. This class contains getter- and setter-Method for all elements and attributes in the `BackEndMessage`.

See the supplied `BackEndEnvelope.xsd` file for the complete structure of the `BackEndMessage`.

To create a `BackEndMessage` there are three constructors available. One takes a `java.io.File` referencing the XML message to be sent, the second takes an `org.dom4j.Document` and the third takes the XML document as a byte-array.

If the given XML file already contains a `BackEndEnvelope`, the `BackEndMessage` class will recognise it. If not an empty `BackEndEnvelope` is wrapped around the message, in this case all necessary elements and attributes have to be set manually.

To add an attachment to the `BackEndMessage`, use the method `addAttachment(java.io.File file)`. This method will determine the mime-type of the attachment automatically. It is also possible to set the mime-type manually with the method `addAttachment(java.io.File file, String type)`. Additionally you can add a description and specify the language of the attachment if you use the method `addAttachment(java.io.File file, String type, String description)` or `addAttachment(java.io.File file, String type, String description, String language)`.

When a message is received, you can access all fields in the `BackEndEnvelope` with the getter-methods. For all elements there are methods available which return an `org.dom4j.Element` and methods which return the element content text, e.g. the method `getConversationID()` returns the `org.dom4j.Element` and the methods `getConversationIDText()` returns the content text of the element as a `String`.

You can obtain the message content as a byte-array by calling the method `getMessageDocumentBytes()`. This method returns only the payload document. If you want to process the `BackEndEnvelope` externally, you can get the whole `BackEndMessage`, including the payload document with the `BackEndEnvelope` wrapped around it, by calling the method `getBackEndMessageBytes()`.

If the received document contains attachments, you can get them by using the method `getAttachment(String filename)`. This method returns the file reference to the attachment in the work folder. The attachment has to be moved or copied out of the work folder before the receive-method of the adapter is returning, because the `GenericAdapter` will remove the temporary folder for this method afterwards. To get all filenames of attached files, use the method `listAttachments()`. You will get a `String`-array with the filenames of all attachments of this message.

5 MessageResult

The class MessageResult is used to pass send/receive results between the EFET Box+ and an Adapter. When the Adapter sends out a message it will get a MessageResult back containing the result of the local processing. Additionally the MessageResult will contain the conversation ID, message ID and other message specific data. See the JavaDoc of the MessageResult class for complete information.

To create a MessageResult you have to use the only constructor of the MessageResult, which takes a MessageResultIdentifier as argument. The MessageResultIdentifier is an inner class of the MessageResult.

```
MessageResult result = new  
MessageResult(MessageResult.MSG_SUCCESSFULLY_RECEIVED);
```

6 Sample Adapter Source Code

```

package de.pontonconsulting.xmlpipe.adapter.sampleadapter;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import de.pontonconsulting.xmlpipe.adapter.AdapterException;
import de.pontonconsulting.xmlpipe.adapter.GenericAdapter;
import de.pontonconsulting.xmlpipe.adapter.ISpecificAdapter;
import de.pontonconsulting.xmlpipe.adapter.MessageResult;
import de.pontonconsulting.xmlpipe.message.BackEndMessage;
import de.pontonconsulting.xmlpipe.message.BackEndMessageException;

/**
 * This is a sample implementation of an adapter for Ponton X/P 2.1.
 *
 * Copyright &copy; 2002-2003 Ponton Consulting GmbH. All rights reserved.
 */
public class SampleAdapter implements ISpecificAdapter
{
    /**
     * CVS Identifier
     */
    private static final String CVS_ID =
        "$RCSfile: SampleAdapter.java,v $ $Revision: 1.11 $ $Date: 2003/06/17 13:28:31 $ $Name:
        $";

    private Log _log;
    private GenericAdapter _ga;

    public static void main(String[] args)
    {
        SampleAdapter adapter = new SampleAdapter();
        adapter.partnerExistenceTest();
        adapter.getPartnerlistTest();
        adapter.sendTest();
        adapter.shutdown();
    }

    /**
     * Create the SampleAdapter instance.
     */
    public SampleAdapter()
    {
        try
        {
            System.setProperty(
                "org.apache.commons.logging.LogFactory",
                "de.pontonconsulting.common.log.PontonLogFactory");
            _log = LogFactory.getFactory().getInstance(getID() + "/Main/console=DEBUG");
            _log.debug("Initializing SampleAdapter");

            // Create the GenericAdapter instance which provides the connection between the EFET
            Box+

```

```
// and the Adapter. The Adapter will be registered automatically with the Messenger.
// The EFET Box+ is contacted on the given hostname and port number - the EFET Box's
// AdapterService will be reached on the given path.
// The connection string looks like this: http://hostname:portnumber/path
//
// _ga = new GenericAdapter(endAdapter, hostname, portnumber, path, logCategory);
_ga = new GenericAdapter(this, "localhost", 8880, "/efet/AdapterService", getID());
}
catch (AdapterException ae)
{
    _log.fatal("Error while initializing SampleAdapter", ae);
}
catch (IOException ioe)
{
    _log.fatal("Error while initializing SampleAdapter", ioe);
}
}

public int getNumberOfParallelThreads()
{
    return 1;
}

public MessageResult receiveMessage(BackEndMessage message)
{
    // There are several getter methods to obtain the data from the BackEndMessage.
    // For attributes, these methods return a java.lang.String.
    // For elements, there are methods that return an org.dom4j.Element and methods
    // that return a java.lang.String.
    _log.info("*** Receiving message: " + message.getTransferIDText());
    _log.info(" ** Conversation id: " + message.getConversationIDText());
    _log.info(" ** Local sender id: " + message.getSenderOrganisationText());
    _log.info(" ** Local receiver id: " + message.getReceiverOrganisationText());

    _log.info("*** # of attachments: " + message.getNumberOfAttachments());
    String[] attachments = message.listAttachments();
    for (int i = 0; i < attachments.length; i++)
    {
        File attachment = message.getAttachment(attachments[i]);
        _log.info(" --> Attachment: " + attachment.getAbsolutePath());
        // All attachments have to be moved to a different folder before the MessageResult
        // is returned, because the GenericAdapter will delete these from the work folder.
    }

    /* The XML message data can be accessed via the following command:
    *     byte[] data = message.getMessageDocumentBytes();
    * This will not contain the BackEndEnvelope, however.
    *
    * If the BackEndEnvelope is to be processed, the complete XML message (i.e. the
    * BackEndEnvelope wrapped around the payload message) can be accessed via:
    *     byte[] data = message.getBackEndMessageBytes();
    */

    return new MessageResult(MessageResult.MSG_SUCCESSFULLY_RECEIVED);
}
/**
 * This method will call the method receiveMessage(BackEndMessage), because this
 * Adapter does not distinguish between 'test' and 'production' messages.
 */
```

```
public MessageResult receiveTestMessage(BackEndMessage message)
{
    return receiveMessage(message);
}

/**
 * Get the MessageResult (XML acknowledgement for the message).
 */
public MessageResult receiveAcknowledgement(BackEndMessage message)
{
    /**
     * The XML acknowledgement as a byte array is accessible via the method
     * message.getMessageDocumentBytes()</code>
     */

    return new MessageResult(MessageResult.MSG_SUCCESSFULLY_RECEIVED);
}

public boolean doSelfCheck()
{
    return true;
}

public String getID()
{
    return "sample-adapter";
}

public String getStatus()
{
    return "SampleAdapter is ready to receive Messages.";
}

public String shutdown()
{
    _ga.shutdown();
    return null;
}

public File getWorkFolder()
{
    return new File("");
}

public boolean supportsAcknowledgements()
{
    return false;
}

public boolean supportsAttachments()
{
    return true;
}

/**
 * Send a message to a partner.
 */
public void sendTest()
{
    try
```

```
{
    // Create a new BackEndMessage from the XML file.
    BackEndMessage bem =
        new BackEndMessage(new File("files/TradeConfirmation_3.2_sample.xml"));

    // Set the sender and receiver organisation. These values have to match a local ID in
the
    // messenger's partners.xml file.
    // If the specified XML file already contains a BackEndMessage and the sender and
receiver
    // information is set correctly, these values don't have to be set manually.
    bem.setSenderOrganisation("EIC_CODE_SENDER");
    bem.setReceiverOrganisation("EIC_CODE_RECEIVER");

    _ga.sendMessage(bem);
}
catch (FileNotFoundException e)
{
    e.printStackTrace();
}
catch (AdapterException e)
{
    e.printStackTrace();
}
catch (BackEndMessageException bme)
{
    bme.printStackTrace();
}
}

/**
 * This test checks whether partners exist.
 */
public void partnerExistenceTest()
{
    try
    {
        log.info("Partner 'EIC_OF_PARTNER' exists: " + _ga.partnerExists("EIC_OF_PARTNER "));
    }
    catch (AdapterException e)
    {
        e.printStackTrace();
    }
}

/**
 * This test gets the configured partner list from the Messenger.
 */
public void getPartnerlistTest()
{
    try
    {
        String[] partners = _ga.getFullPartnerList();
        for (int i = 0; i < partners.length; i++)
        {
            _log.info("*** (all) Partner " + i + " has local id: " + partners[i]);
        }

        partners = _ga.getLocalPartnerList();
    }
}
```

```
    for (int i = 0; i < partners.length; i++)
    {
        _log.info("*** (own) Partner " + i + " has local id: " + partners[i]);
    }

    partners = _ga.getRemotePartnerList();
    for (int i = 0; i < partners.length; i++)
    {
        _log.info("*** (remote) Partner " + i + " has local id: " + partners[i]);
    }
}
catch (AdapterException e)
{
    e.printStackTrace();
}
}
```

7 MessageResult Codes

7.1 *GenericAdapter.sendMessage()*

MSG_SUCCESSFULLY_SEND

The Messenger completed all processing steps and the message is queued for delivery. It is not transmitted to the receiver at this point ! From the returned MessageResult object the adapter can get several information:

String getMessageID() this is the message id (transfer id) that is used for transmission. If it was defined in the backend message, then it should be unchanged.

String getConversationID() this is the conversation id that is used for transmission. If it was defined in the backend message, then it should be unchanged.

String getMessageType() the message type that was identified by the messenger

String getSchemaVersion() the schema version that was identified by the messenger.

String getSchemaSet() the schema set that was used for validationString String getMessageTime() the creation timestamp that is transmitted in the transport envelope

String getTransmissionProtocol() the protocol that is used to send the message

ADAPTER_REGISTRY_COULD_NOT_BE_ACCESSED

This error is returned if an adapter tries to send a message. It can only happen if the Messenger was restarted while the adapter kept running. If the adapter cannot be re-registered in the messenger database this error will occur.

BACKENDMSG_COULD_NOT_BE_RECONSTRUCTED

When unexpected or corrupted data is send to the Messenger, this error will be returned.

String getResponseMessage() this contains further details about the error

MESSAGE_COULD_NOT_BE_REGISTERED

The message ID is already registered at the Messenger. It is not allowed to send two messages with the same ID. if the previous transmission failed for any reason, then the adapter is allowed to resend the same message (with the same message ID) so that the transmission is triggered again.

A second cause for this error is a database communication problem.

PARTNER_IS_DISABLED

The receiver or sender of this message is deactivated in the partner config of the Messenger.

String getResponseMessage() contains the local id of the blocked partner. And also states if it is the receiver or the sender.

PARTNER_IS_NOT_KNOWN

The local id of the receiver or sender of this message is unknown. The partner settings should be checked for the correct local id.

String getResponseMessage() contains the local id that was not found. It also states if it is the receiver or the sender.

PARTNER_STORE_COULD_NOT_BE_ACCESSED

The partners.xml file got corrupted or is not accessible.

String getResponseMessage() contains additional error information.

TRANSPORT_PROVIDER_NOT_FOUND

The URL defined for the receiver specifies an unknown protocol. Currently only http:// https:// [mailto://](mailto:) smime:// are supported. This has to be fixed in the partners configuration.

COULD_NOT_DECRYPT_PRIVATE_KEY_PASSWORD

The private key to sign messages is not found or cannot be decrypted. This is either because the specified sender is not a local partner or some problem with the private key password exist.

String getResponseMessage() contains additional error information.

COULD_NOT_INITIALIZE_PIPELINE

Problem while initializing the processing pipeline.
String getResponseMessage() contains additional error information.

ENCRYPTION_FAILED

Problem while encrypting the payload.
String getResponseMessage() contains additional error information.

SIGNING_FAILED

Problem while signing the payload.
String getResponseMessage() contains additional error information.

COMPRESSION_FAILED

Problem while compressing the payload.
String getResponseMessage() contains additional error information.

VALIDATION_FAILED

Problem while validating the payload.
String getResponseMessage() contains additional error information.

COULD_NOT_PROCESS_MESSAGE

The message type and or version is not allowed by either the receiver or sender. Check the partner configuration to see if the correct schema-sets are activated for both partners.

Any other problem while processing the message can also result in this error.
String getResponseMessage() contains additional error information.

LOGGING_INTO_DATABASE_FAILED

The message was completely processed but an error occurred when the message was queued for delivery.
String getResponseMessage() contains additional error information.

COULD_NOT_INITIALIZE_PACKAGER

A problem with the messenger configuration caused this internal problem.
String getResponseMessage() contains additional error information.

COULD_NOT_PACKAGE_MESSAGE

The transmission format could not be created. This can be caused by invalid or missing message properties. It can also be caused by file system errors.

String `getResponseMessage()` contains additional error information.

7.2 *GenericAdapter.sendPing()*

Same as for `sendMessage()`

7.3 *GenericAdapter.shutdown()*

ADAPTER_SUCCESSFULLY_UNREGISTERED

Everything ok.

ADAPTER_COULD_NOT_BE_UNREGISTERED

A communication problem with the messenger occurred.

ADAPTER_REGISTRY_COULD_NOT_BE_ACCESSED

A database problem prevented the deregistration

It is not a fatal problem if the adapter deregistration fails. The messenger will automatically deregister it on the next inbound message.

7.4 *ISpecificAdapter.receive...()*

All receive methods can return the same `MessageResults`:

MSG_SUCCESSFULLY_RECEIVED

The Messenger will remove the inbound message from the queue and flag it as successfully completed.

COULD_NOT_ACCESS_ATTACHMENT_FILE

The Messenger will store the reported error but it will try to send the same message again.

CUSTOM_ERROR

The Messenger will store the reported error but it will try to send the same message again.

ADAPTER_REJECTED_MESSAGE

The Messenger will remove the inbound message from the queue and flag it as ERROR.

Another try to deliver this message is possible by selecting "change adapter" on the message monitor.

Before returning the `MessageResult` object it is possible to give additional text to the Messenger using `appendToDescription()` or `setDescription()`.

The later will override the default text that comes with the selected `MessageResult`. This is also possible for the success result.