

PONTON X/P 5.0 Adapter 2.0 Programming Guide

Version: 1

Datum: 21-Oct-2025

Copyright Notice

This document is the confidential and proprietary information of PONTON GmbH ("Confidential Information"). You shall not disclose such Confidential Information and shall use it only in accordance with the terms of the license agreement you entered into with PONTON GmbH.



Table of Contents

1	Introduction.....	3
1.1	Message Flow	3
2	SimpleAdapter	3
2.1	Create Messenger Connection	4
2.2	Send Message	7
2.3	Handle Status Updates for Outbound Message	8
2.4	Receive Message	9
2.5	Handle Adapter Status Request	9
2.6	Archive Handler	10
2.7	ConnectionStatusChangeHandler	11

1 Introduction

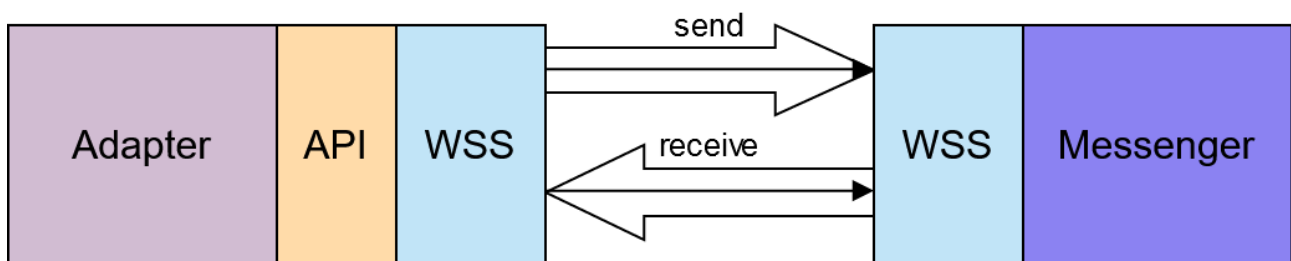
This document describes the basics how to write an adapter for the PONTON X/P Messenger.

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

A Java library is provided that contains the classes that handle the communication with the PONTON X/P Messenger.

To use this library it is required to use at least Java version 11, no third party libraries are needed.

1.1 Message Flow



The communication between the PONTON X/P Messenger and an adapter is based on the HTTPS Websocket protocol.

An adapter establishes permanent Websocket connections to the Messenger. To send or receive messages the adapter must be authorized by the PONTON X/P Messenger.

2 SimpleAdapter

This example shows how to create a simple adapter to send and receive messages.



2.1 Create Messenger Connection

```

/*
 * Copyright 2020 by PONTON GmbH
 * Dorotheenstr. 64, 22301 Hamburg, Germany.
 * All rights reserved.
 *
 * This software is the confidential and proprietary information of
 * PONTON GmbH "Confidential Information". You shall not disclose
 * such Confidential Information and shall use it only in accordance with the
 * terms of the license agreement you entered into with PONTON.
 */

import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.time.Instant;
import java.util.Optional;
import java.util.function.Supplier;

import de.ponton.xp.adapter.api.AdapterStatusRequestHandler;
import de.ponton.xp.adapter.api.ArchiveException;
import de.ponton.xp.adapter.api.ArchiveHandler;
import de.ponton.xp.adapter.api.ArtifactTypeEnum;
import de.ponton.xp.adapter.api.ConnectionException;
import de.ponton.xp.adapter.api.MessageHandler;
import de.ponton.xp.adapter.api.MessengerConnection;
import de.ponton.xp.adapter.api.OutboundMessageStatusUpdateHandler;
import de.ponton.xp.adapter.api.TransmissionException;
import de.ponton.xp.adapter.api.domainvalues.AdapterInfo;
import de.ponton.xp.adapter.api.domainvalues.ArchiveReferenceId;
import de.ponton.xp.adapter.api.domainvalues.ConversationId;
import de.ponton.xp.adapter.api.domainvalues.Filename;
import de.ponton.xp.adapter.api.domainvalues.InboundMetaData;
import de.ponton.xp.adapter.api.domainvalues.InboundStatusEnum;
import de.ponton.xp.adapter.api.domainvalues.MessageId;
import de.ponton.xp.adapter.api.domainvalues.MessageType;
import de.ponton.xp.adapter.api.domainvalues.MessageUniqueId;
import de.ponton.xp.adapter.api.domainvalues.MessengerInstance;
import de.ponton.xp.adapter.api.domainvalues.OutboundMetaData;
import de.ponton.xp.adapter.api.domainvalues.OutboundStatusEnum;
import de.ponton.xp.adapter.api.domainvalues.ReceiverId;
import de.ponton.xp.adapter.api.domainvalues.SenderId;
import de.ponton.xp.adapter.api.domainvalues.TransferId;
import de.ponton.xp.adapter.api.messages.ArchiveDeleteMessage;
import de.ponton.xp.adapter.api.messages.ArchiveGetMessage;
import de.ponton.xp.adapter.api.messages.ArchiveGetResponse;
import de.ponton.xp.adapter.api.messages.ArchiveStoreMessage;
import de.ponton.xp.adapter.api.messages.ArchiveStoreResponse;
import de.ponton.xp.adapter.api.messages.InboundMessageStatusUpdate;
import de.ponton.xp.adapter.api.messages.OutboundMessage;
import de.ponton.xp.adapter.api.messages.OutboundMessageStatusUpdate;

public class SimpleAdapter implements ArchiveHandler {

    private final MessengerConnection messengerConnection;

    public SimpleAdapter() throws ConnectionException {
        final File adapterWorkFolder = new File("work");
        final AdapterInfo adapterInfo = AdapterInfo.newBuilder() //
            .setAdapterId("SimpleAdapterID") //
            .setAdapterVersion("1.0.0") //
            .setSupportArchiving(true) //

```

```

        .setSupportErrorNotification(true)
        .build();

    messengerConnection = connectToMessenger(adapterInfo, adapterWorkFolder);
    startReceivingMessages();
}

private MessengerConnection connectToMessenger(AdapterInfo adapterInfo, File
adapterWorkFolder) throws ConnectionException {
    final int messengerPort = 2600;
    return MessengerConnection.newBuilder() //
        .setWorkFolder(adapterWorkFolder) //
        .setAdapterInfo(adapterInfo) //
        .addMessengerInstance(MessengerInstance.create("messenger.host",
messengerPort, 5, 5, 10)) //
        .onMessageReceive(getMessageHandler()) //
        .onMessageStatusUpdate(getOutboundMessageStatusUpdateHandler()) //
        .onAdapterStatusRequest(getAdapterStatusRequestHandler()) //
        .onArchiveMessageReceive(this) //
        .onConnectionStatusChanged(this::onConnectionStatusChanged)
        .build();
}

private void startReceivingMessages() throws TransmissionException {
    messengerConnection.startReception();
}

```

MessengerConnection is the main class to send or receive messages. When an instance of the *MessengerConnection* is created, the adapter is connected with 5 inbound connections, 5 outbound connections and 10 archive connections to the messenger on host *messenger.host* and port 2600. You can use this *MessengerConnection* instance to initialize your own beans to use it.

NOTE: If you want to use the default messenger archiver (File System Archiver) you don't need to implement *ArchiveHandler* and the count of the archive connections must be 0.

After initialization you have to call *startReception()* to inform the messenger, that the adapter is ready for using.

NOTE: *MessengerConnection* tries to reconnect to the messenger automatically, if the connection is lost.

2.2 Send Message

```

    public void sendMessage(final InputStream inputStream, final SenderId senderId,
final ReceiverId receiverId) throws TransmissionException {
        final OutboundMessage outboundMessage = buildOutboundMessage(inputStream,
senderId, receiverId);
        final TransferId transferId =
messengerConnection.sendMessage(outboundMessage);
        // the transferId is needed to assign status updates to this send process
    }

    public OutboundMessageStatusUpdate sendMessageSynchronously(final InputStream
inputStream, final SenderId senderId, final ReceiverId receiverId) throws
TransmissionException {
        final OutboundMessage outboundMessage = buildOutboundMessage(inputStream,
senderId, receiverId);
        return messengerConnection.sendMessageSynchronously(outboundMessage);
    }

    private static OutboundMessage buildOutboundMessage(InputStream inputStream,
SenderId senderId, ReceiverId receiverId) {
        final OutboundMetaData outboundMetaData = OutboundMetaData.newBuilder() //
            .setSenderId(senderId) //
            .setReceiverId(receiverId) //
            .build();
        final OutboundMessage outboundMessage = OutboundMessage.newBuilder() //
            .setInputStream(inputStream) //
            .setOutboundMetaData(outboundMetaData) //
            .build();
        return outboundMessage;
    }

```

To send a message you have to build an *OutboundMessage*, which contains a message content as *InputStream* and optional meta data of the message (see *OutboundMetaData*), like *SenderId*, *ReceiverId*, *MessageId* and call one of the following send methods:

- *sendMessage(outboundMessage)* on *MessengerConnection* to send the message asynchronously. As result you get a *TransferId* to assign incoming status updates to this send process.
- *sendMessageSynchronously(outboundMessage)* on *MessengerConnection* to send the message synchronously. As result you get an *OutboundMessageStatusUpdate* with *TransferId* and the status of the process step in the Messenger (e.g. *OutboundStatusEnum.PROCESSED_AND_QUEUED* when message successfully processed).

2.3 Handle Status Updates for Outbound Message

```
private OutboundMessageStatusUpdateHandler
getOutboundMessageStatusUpdateHandler() {
    return outboundMessageStatusUpdate -> {
        // the send process is finished.
        if (outboundMessageStatusUpdate.isFinal()) {
            // get transferId to reference the sent outbound message (see
sendMessage())
            final TransferId transferId =
outboundMessageStatusUpdate.getTransferId();

            // We can send the result of the sent message to backend.
            final OutboundStatusEnum outboundStatusEnum =
outboundMessageStatusUpdate.getResult();
            final String detailText =
outboundMessageStatusUpdate.getDetailText();
        }
    };
}
```

To receive status updates the *OutboundMessageStatusUpdateHandler* has to be implemented and set to the *MessengerConnection*. All status updates have to be assigned to the original send process by using the *TransferId*.

NOTE: the final flag informs you whether the original send process is completed. If the value of the final flag is *true* no other status updates will be sent to the adapter for the send process.

2.4 Receive Message

```
private MessageHandler getMessageHandler() {
    return inboundMessage -> {
        // handle inbound message
        final InboundMetaData inboundMetaData =
inboundMessage.getInboundMetaData();
        final MessageId messageId = inboundMetaData.getMessageId();
        final ConversationId conversationId =
inboundMetaData.getConversationId();
        final SenderId senderId = inboundMetaData.getSenderId();
        final ReceiverId receiverId = inboundMetaData.getReceiverId();
        final MessageType messageType = inboundMetaData.getMessageType();

        try (final InputStream inputStream = inboundMessage.createInputStream())
        {
            // process incoming message content
        } catch (final IOException ioe) {
            // handle IOException here
        }

        // create result for the inbound message
        final InboundMessageStatusUpdate inboundMessageStatusUpdate =
InboundMessageStatusUpdate.newBuilder() //
            .setInboundMessage(inboundMessage) //
            .setStatus(InboundStatusEnum.SUCCESS) //
            .setStatusText("message successfully delivered to backend.") //
            .build();
        return inboundMessageStatusUpdate;
    };
}
```

To receive messages *MessageHandler* has to be implemented and set to *MessengerConnection*. Each *InboundMessage* contains the message content, which can be read by calling *createInputStream()*, and meta data (see *InboundMetaData*), like *SenderId*, *ReceiverId*, *MessageType* and other.

If an incoming message was processed you have to return *InboundMessageStatusUpdate* with the status:

- SUCCESS - if the incoming message was successfully processed.
- REJECTED - if an error occurs while processing the message or forwarding it to a backend system.
- TEMPORARY_ERROR - if a temporary error occurs and the messenger has to send the message again.

2.5 Handle Adapter Status Request

```
private AdapterStatusRequestHandler getAdapterStatusRequestHandler() {
    return () -> {
        final String adapterStatus = "Simple adapter is running and received 123
messages.";
        // the adapter status is shown by the messenger on the Adapter Monitor
        return adapterStatus;
    };
}
```

To receive adapter status requests *AdapterStatusRequestHandler* has to be implemented and set to *MessengerConnection*. As adapter status you should return a string with your own information, which is requested and shown by Messenger on the Adapter Monitor.

2.6 Archive Handler

```

@Override
public void onMessage(final ArchiveDeleteMessage message) throws ArchiveException
{
    final String referenceId = message.getArchiveReferenceId().getStringValue();
    // delete the archive entry with the given referenceId
    // throw an ArchiveException if the deletion fails
}

@Override
public ArchiveGetResponse onMessage(final ArchiveGetMessage message) throws
ArchiveException {
    final String referenceId = message.getArchiveReferenceId().getStringValue();
    // get data of the archive entry with the given referenceId
    final Supplier<InputStream> inputStreamSupplier = null;
    final ArchiveGetResponse archiveGetResponse = ArchiveGetResponse.newBuilder()
//
        .setInputStreamSupplier(inputStreamSupplier) //
        .build();

    // throw an ArchiveException if the retrieval fails
    return archiveGetResponse;
}

@Override
public ArchiveStoreResponse onMessage(final ArchiveStoreMessage message) throws
ArchiveException {
    // create a new archive entry by using the message data
    final MessageUniqueId messageUniqueId = message.getMessageUniqueId();
    final MessageId messageId = message.getMessageId();
    final boolean isInboundMessage = message.isInbound();
    final Instant timestamp = message.getTimestamp();
    final ArtifactTypeEnum artifactType = message.getArtifactType();
    final Optional<Filename> filename = message.getFilename();
    final boolean isMessageFailed = message.isFailed();
    try (final InputStream dataInputStream = message.createInputStream()) {
        // store data the archive entry
        // throw an ArchiveException if the storage fails
    } catch (final IOException e) {
        throw new ArchiveException("Could not read the data of the archive
message: " + e.getMessage());
    }

    return ArchiveStoreResponse.newBuilder() //
        .setArchiveReference(ArchiveReferenceId.of("referenceId")) //
        .build();
}
}

```

To connect an own archiver to the messenger the following methods of the interface *ArchiveHandler* have to be implemented:

- **ArchiveStoreResponse onMessage(final ArchiveStoreMessage message) throws ArchiveException**
 - *ArchiveStoreMessage* contains meta data of the message artifact to be archived and the message artifact itself.
 - If the message artifact was successfully stored, *ArchiveStoreResponse* must be returned with the unique *ArchiveReference* of the archived message artifact, otherwise an *ArchiveException* must be thrown.
- **ArchiveGetResponse onMessage(final ArchiveGetMessage message) throws ArchiveException**

- *ArchiveGetMessage* contains the *ArchiveReference* for the searched message artifact.
- If the message artifact exists, *ArchiveGetResponse* must be returned with the message artifact as *Supplier<InputStream>*. In error cases an *ArchiveException* must be thrown.
- *void onMessage(final ArchiveDeleteMessage message)* throws *ArchiveException*
 - *ArchiveDeleteMessage* contains the *ArchiveReference* for the message artifact to be deleted.
 - If the artifact does not exist or could not be deleted, an *ArchiveException* must be thrown.

2.7 ConnectionStatusChangeHandler

When a *ConnectionStatusChangeHandler* is registered on *MessengerConnection* it is called each time a connection is created or closed. The call includes the current *MessengerInstance* and the total number of active connection for Outbound-, Inbound- and Archive-Processing.

```
private void onConnectionStatusChanged(MessengerInstance messengerInstance, int
outboundConnectionCount, int inboundConnectionCount, int archiveConnectionCount) {
    System.out.println("ConnectionStatusChanged of MessengerInstance " +
messengerInstance.getHostname() + ":" + messengerInstance.getPort() + " [out:" +
        outboundConnectionCount + " | in:" + inboundConnectionCount + " |
arch:" + archiveConnectionCount + "]);
}
```

PONTON GmbH
Dorotheenstraße 64
22301 Hamburg
Germany

Web: <http://www.ponton.de>

LinkedIn <https://www.linkedin.com/company/ponton-consulting/>

