PONTON X/P 3.8 Adapter 2.0 Programming Guide

Version: 3
Date: 24-Sep-2019

Table of Contents

1. Introduction 3
1.1. Message Flow 3

2. SimpleAdapter 4

1. Introduction

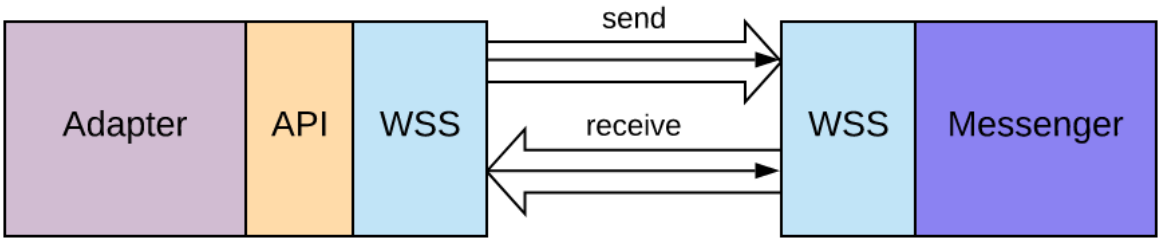
This document describes the basics how to write an adapter for the PONTON X/P Messenger.

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.

Code Block 1 Create Messenger Connection

```
/*
 * Copyright 2019 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 de.ponton.xp.adapter.api.*;
import de.ponton.xp.adapter.api.domainvalues.*;
import de.ponton.xp.adapter.api.messages.*;

public class SimpleAdapter implements MessageHandler,
OutboundMessageStatusUpdateHandler {
    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")//
                .build();
        final int messengerPort = 9090;
        messengerConnection = MessengerConnection.newBuilder()//
                .setWorkFolder(adapterWorkFolder)//
                .setAdapterInfo(adapterInfo)//
                .addMessengerInstance(MessengerInstance.create("messenger.host",
messengerPort))//
                .onMessageReceive(this)//
                .onMessageStatusUpdate(this)//
                .build();

        messengerConnection.start();
    }
```

MessengerConncetion ist the main class to send or receive messages. When an instance of the MessengerConnection is created the adapter is connected to the messenger on host *messenger.host* and port *9090.*
You can use this MessengerConnection instance to initialize your own beans to use it.

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

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

Code Block 2 Send Message

```
    public void sendMessage(final InputStream inputStream, final SenderId
senderId, final ReceiverId receiverId) throws TransmissionException {
        final MetaData metaData = MetaData.newBuilder()//
                .setSenderId(senderId)//
                .setReceiverId(receiverId)//
                .build();
        final OutboundMessage outboundMessage = OutboundMessage.newBuilder()//
                .setInputStream(inputStream)//
                .setMetaData(metaData)//
                .build();
        final TransferId transferId =
messengerConnection.sendMessage(outboundMessage);
        // the transferId is needed to assign status updates to this send process
    }
```

To send a message you have to call *sendMessage(outboundMessage)*. An *OutboundMessage* has to contain a message content as InputStream and optional meta data of the message, like *SenderId*, *ReceiverId*, *MessageId* and other.

As result you get a *TransferId* to assign incoming status updates to this send process.

Code Block 3 Handle Status Updates for Outbound Message

```
    @Override
    public void onOutboundMessageStatusUpdate(final OutboundMessageStatusUpdate
messageStatus) {
        // the send process is finished.
        if (messageStatus.isFinal()) {
            // get transferId to reference the sent outbound message (see
sendMessage())
            final TransferId transferId = messageStatus.getTransferId();

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

To receive status updates the *OutboundMessageStatusUpdateHandler* and set to the MessengerConnection has to be implemented. All status updates come asynchronously and 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.

Code Block 4 Receive Message

```
    @Override
    public InboundMessageStatusUpdate onMessage(final InboundMessage
inboundMessage) {
        // handle inbound message
        final MetaData metaData = inboundMessage.getMetaData();
        final MessageId messageId = metaData.getMessageId();
        final ConversationId conversationId = metaData.getConversationId();
        final SenderId senderId = metaData.getSenderId();
        final ReceiverId receiverId = metaData.getReceiverId();
        final MessageType messageType = metaData.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 result =
InboundMessageStatusUpdate.newBuilder()//
                .setInboundMessage(inboundMessage)//
                .setStatus(InboundStatusEnum.SUCCESS)//
                .setStatusText("message successfully delivered to backend.")//
                .build();
        return result;
    }
}
```

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, 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.

PONTON GmbH

Dorotheenstraße 64

GERMANY 22301 Hamburg

Web: http://www.ponton.de