17 JUN 2018

HL7 Programming using Java and HAPI - Creating HL7 Messages

CONTENTS

Introduction

This is part of my HL7 article series. Before we get started on this tutorial, have a quick look at my earlier article titled "A Very Short Introduction to the HL7 2.x Standard". This tutorial assumes you know Java or any equivalent object-oriented language.

My previous tutorial "HL7 Programming using Java" gave you a foundational understanding of HL7 2.x message transmission, reception and message acknowledgement. Although this was fine for understanding the basics of HL7 message processing, this won't be sufficient for more complex message processing requirements we often encounter in the real world. You will often need a larger programming framework or a library that provides pre-built support to enable message construction, parsing, validation, transmission and reception of a broad set of HL7 2.x standards and its numerous trigger events/message types. One such framework or library that is available for the Java language is the HAPI HL7 Library. This library came out of work undertaken at a Canadian university in the early 2000s, and then grew steadily over the years to support nearly all HL7 Version 2.x Standards (includes 2.1, 2.2, 2.3, 2.3.1, 2.4, 2.5, 2.5.1, and 2.6 at the time of writing this article). The library has been used both directly and "under the covers" by numerous commercial as well as open source health interface applications. I have used this software both directly in some custom applications that I wrote in Java as well as indirectly when I used some popular healthcare interface

engines such as Mirth Connect in the past. The library is also being used to power some of the latest FHIR-based applications and initiatives whose primary goal being to make the task of interoperability between web-enabled healthcare applications even easier. In this introductory tutorial, my task is to introduce you to the very basics of the HAPI toolkit/library by creating a basic patient admit HL7 message using the HL7 2.3 standard.

Tools for Tutorial

- JDK 1.4 SDK or higher
- Eclipse or any other Java IDE (or even a text editor)
- Download HAPI compiled library from the HAPI project website
- View HAPI source code on their GitHub site here
- Download HAPI Test Panel (optional) here
- You can also find all the code demonstrated in the tutorial on GitHub here

> "Courageous people do not fear forgiving, for the sake of peace." ~ Nelson Mandela

Quick Introduction to ADT Messages

HL7 ADT messages are one of the most popular message types used in HL7-enabled systems and are used communicate core patient information captured in the EMR (Electronic Medical Record) to other ancillary/supporting systems such as lab, radiology, etc. This message is often transmitted when the patient information is either captured initially or when it changes because of events such as admission, transfer or discharge in a clinical setting. There are many types of ADT messages including ADT A01 (used during admit), A02 (used during transfer), A03 (used during end of visit or discharge), etc. There are also other HL7 message types that pertain to orders, results, clinical reports, scheduling and billing. The HAPI library provides built-in support to create all these message types. In this particular tutorial, I will use the HAPI library to create a ADT A01 message. This message type or trigger event

that we will create will include the following message segments that are mandatory:

- MSH segment: Every HL7 message will carry this segment if you recall from my earlier tutorial. This segment will carry information such as the sender and receiver of the message, the type of message, and the date and time information of this message
- EVN segment: This segment will include information pertaining to the event that occurred (in our case this is due to an admission of a fictional patient)
- PID segment: This segment carries information pertaining to patient demographics such as such as name of the patient, patient identifier and address, etc
- PV1 segment: This segment typically contains information about the patient's stay, where they are assigned and also the referring doctor

Message construction gets pretty ugly if we don't separate the various steps involved into smaller methods. The approach I show below is a suggestion only. Please feel free to create your own factory methods, builder classes and any other utility classes needed especially if you are wanting to support multiple message types in your application.

```
package com.saravanansubramanian.hapihl7tutorial.create;

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

import ca.uhn.hl7v2.DefaultHapiContext;
import ca.uhn.hl7v2.HL7Exception;
import ca.uhn.hl7v2.HapiContext;
import ca.uhn.hl7v2.model.v24.message.ADT_A01;
import ca.uhn.hl7v2.parser.Parser;

public class HapiCreateMessageSimpleExample {
```

```
        // In HAPI, almost all things revolve around a context
object
        private static HapiContext context = new
DefaultHapiContext();

        public static void main(String[] args) throws
Exception {

            try {

                // create the HL7 message
                // this AdtMessageFactory class is not from
HAPI but my own wrapper
                System.out.println("Creating ADT A01
message...");
                ADT_A01 adtMessage = (ADT_A01)
AdtMessageFactory.createMessage("A01");

                // create these parsers for file operations
                Parser pipeParser = context.getPipeParser();
                Parser xmlParser = context.getXMLParser();

                // print out the message that we constructed
                System.out.println("Message was constructed
successfully..." + "\n");

System.out.println(pipeParser.encode(adtMessage));

                // serialize the message to pipe delimited
output file
                writeMessageToFile(pipeParser, adtMessage,
"testPipeDelimitedOutputFile.txt");

                // serialize the message to XML format output
file
                writeMessageToFile(xmlParser, adtMessage,
"testXmlOutputFile.xml");

                //you can print out the message structure
using a convenient helper method on the message class
                System.out.println("Printing message structure
to console...");

System.out.println(adtMessage.printStructure());

            } catch (Exception e) {
                e.printStackTrace();
            }
```

```
        }

        private static void writeMessageToFile(Parser parser,
ADT_A01 adtMessage, String outputFilename)
                throws IOException, FileNotFoundException,
HL7Exception {
            OutputStream outputStream = null;
            try {

                // Remember that the file may not show special
delimiter characters when using
                // plain text editor
                File file = new File(outputFilename);

                // quick check to create the file before
writing if it does not exist already
                if (!file.exists()) {
                    file.createNewFile();
                }

                System.out.println("Serializing message to
file...");
                outputStream = new FileOutputStream(file);

outputStream.write(parser.encode(adtMessage).getBytes());
                outputStream.flush();

                System.out.printf("Message serialized to file
'%s' successfully", file);
                System.out.println("\n");
            } finally {
                if (outputStream != null) {
                    outputStream.close();
                }
            }
        }

    }
```

There are many variations of the factory method pattern, and most of them tend to implement a static method which is responsible for constructing an object and returning this created object. It is the static method that tends to abstract the construction process of an object. You typically define an interface for creating an object in the superclass and you then let the subclasses decide which classes to instantiate depending on the

parameters passed. However, you can also delegate the construction to another class entirely. So, lots of options and approaches are available to you. Ultimately, the factory method lets the main class (or the "client") defer any instantiation to other classes based on the parameters passed. Here, I am simply using a builder class inside the factory method.

```
    package com.saravanansubramanian.hapihl7tutorial.create;

    import java.io.IOException;
    import ca.uhn.hl7v2.HL7Exception;
    import ca.uhn.hl7v2.model.Message;

    public class AdtMessageFactory {

        public static Message createMessage(String messageType)
throws HL7Exception, IOException {

            //This patterns enables you to build other message
types
            if ( messageType.equals("A01") )
            {
                return new OurAdtA01MessageBuilder().Build();
            }

            //if other types of ADT messages are needed, then
implement your builders here
            throw new RuntimeException(String.format("%s
message type is not supported yet. Extend this if you need to",
messageType));

        }
    }
```

In the code shown below and as described earlier the delegation of the actual construction of our ADT A01 message is left to a *ADT message builder* class called "OurAdtA01MessageBuilder" here for the lack of better imagination on my part. This builder class shown below helps construct the various parts of the ADT A01 message step by step. Using the *builder pattern* will be extremely useful even when using a HL7 library like HAPI since there is just way too many optional segments and components that you will deal with when dealing with the HL7 2.x standard. The other advantage of

this approach is that you can pass in data into the constructor of the builder class to build the ADT message with the required data values from your HL7 application (not shown here). This a style I prefer. However, feel free to choose any style that suits you or your organization. This is after all a tutorial on using HAPI and not on software design. :-)

```
    package com.saravanansubramanian.hapihl7tutorial;

    import java.io.IOException;
    import java.text.SimpleDateFormat;
    import java.util.Date;

    import ca.uhn.hl7v2.HL7Exception;
    import ca.uhn.hl7v2.model.DataTypeException;
    import ca.uhn.hl7v2.model.v24.datatype.PL;
    import ca.uhn.hl7v2.model.v24.datatype.XAD;
    import ca.uhn.hl7v2.model.v24.datatype.XCN;
    import ca.uhn.hl7v2.model.v24.datatype.XPN;
    import ca.uhn.hl7v2.model.v24.message.ADT_A01;
    import ca.uhn.hl7v2.model.v24.segment.EVN;
    import ca.uhn.hl7v2.model.v24.segment.MSH;
    import ca.uhn.hl7v2.model.v24.segment.PID;
    import ca.uhn.hl7v2.model.v24.segment.PV1;

    public class OurAdtA01MessageBuilder {

        private ADT_A01 _adtMessage;

        /*You can pass in a domain object as a parameter
        when integrating with data from your application here
        I will leave that to you to explore on your own
        Using fictional data here for illustration*/

        public ADT_A01 Build() throws HL7Exception,
IOException {
            String currentDateTimeString =
getCurrentTimeStamp();
            _adtMessage = new ADT_A01();
            //you can use the context class's newMessage
method to instantiate a message if you want
            _adtMessage.initQuickstart("ADT", "A01", "P");
            createMshSegment(currentDateTimeString);
            createEvnSegment(currentDateTimeString);
            createPidSegment();
            createPv1Segment();
```

```
            return _adtMessage;
        }

        private void createMshSegment(String
currentDateTimeString) throws DataTypeException {
            MSH mshSegment = _adtMessage.getMSH();
            mshSegment.getFieldSeparator().setValue("|");

mshSegment.getEncodingCharacters().setValue("^~\\&");

mshSegment.getSendingApplication().getNamespaceID().setValue("
Our System");

mshSegment.getSendingFacility().getNamespaceID().setValue("Our
Facility");

mshSegment.getReceivingApplication().getNamespaceID().setValue
("Their Remote System");

mshSegment.getReceivingFacility().getNamespaceID().setValue("T
heir Remote Facility");

mshSegment.getDateTimeOfMessage().getTimeOfAnEvent().setValue(
currentDateTimeString);

mshSegment.getMessageControlID().setValue(getSequenceNumber());

mshSegment.getVersionID().getVersionID().setValue("2.4");
        }

        private void createEvnSegment(String
currentDateTimeString) throws DataTypeException {
            EVN evn = _adtMessage.getEVN();
            evn.getEventTypeCode().setValue("A01");

evn.getRecordedDateTime().getTimeOfAnEvent().setValue(currentD
ateTimeString);
        }

        private void createPidSegment() throws
DataTypeException {
            PID pid = _adtMessage.getPID();
            XPN patientName = pid.getPatientName(0);

patientName.getFamilyName().getSurname().setValue("Mouse");
            patientName.getGivenName().setValue("Mickey");

pid.getPatientIdentifierList(0).getID().setValue("378785433211
");
```

```
            XAD patientAddress = pid.getPatientAddress(0);

patientAddress.getStreetAddress().getStreetOrMailingAddress().
setValue("123 Main Street");
            patientAddress.getCity().setValue("Lake Buena
Vista");
            patientAddress.getStateOrProvince().setValue("FL");
            patientAddress.getCountry().setValue("USA");
        }

        private void createPv1Segment() throws
DataTypeException {
            PV1 pv1 = _adtMessage.getPV1();
            pv1.getPatientClass().setValue("O"); // to
represent an 'Outpatient'
            PL assignedPatientLocation =
pv1.getAssignedPatientLocation();

assignedPatientLocation.getFacility().getNamespaceID().setValu
e("Some Treatment Facility Name");

assignedPatientLocation.getPointOfCare().setValue("Some Point
of Care");
            pv1.getAdmissionType().setValue("ALERT");
            XCN referringDoctor = pv1.getReferringDoctor(0);
            referringDoctor.getIDNumber().setValue("99999999");

referringDoctor.getFamilyName().getSurname().setValue("Smith");
            referringDoctor.getGivenName().setValue("Jack");

referringDoctor.getIdentifierTypeCode().setValue("456789");

pv1.getAdmitDateTime().getTimeOfAnEvent().setValue(getCurrentT
imeStamp());
        }

        private String getCurrentTimeStamp() {
            return new
SimpleDateFormat("yyyyMMddHHmmss").format(new Date());
        }

        private String getSequenceNumber() {
            String facilityNumberPrefix = "1234"; // some
arbitrary prefix for the facility
            return
facilityNumberPrefix.concat(getCurrentTimeStamp());
        }
    }
```

```
Creating ADT A01 message...
Message was constructed successfully...

MSH|^~\&|Our System|Our Facility|Their Remote System|Their Remote Facility|20180630200853||ADT^
EVN|A01|20180630200853
PID|||378785433211||Mouse^Mickey||||||123 Main Street^^Lake Buena Vista^FL^^USA
PV1||O|Some Point of Care^^^Some Treatment Facility Name|ALERT||||99999999^Smith^Jack^^^^^^^^^^

Serializing message to file...
Message serialized to file 'testPipeDelimitedOutputFile.txt' successfully

Serializing message to file...
Message serialized to file 'testXmlOutputFile.xml' successfully
```

The results of running the above program is shown above. Running the program produces a HL7 message that is ready for transmission to a remote HL7 system. You can also inspect the two output files generated by the program to get an idea of what a pipe delimited HL7 file and XML formatted file looks like. Please note that the pipe delimited output file has special unprintable characters within it that help in the "message enveloping" that is required. You can inspect this file using any Hex editor if you are curious to see where these characters are in the file.

Conclusion

That concludes this little tutorial to create our first ADT HL7 message and its various constituent parts using the HAPI library. That was pretty easy, hey? Unlike some other HL7 tools in the market, HAPI provides strongly typed classes to generate various HL7 messages including predefined as well as custom message segments very quickly. The various classes and their methods and other utility functions provided by the library help remove any tedium and guess work when assembling, transmitting, receiving, parsing and validating HL7 messages that we often need to tackle when developing a message processing application. We also used a powerful class called *HapiContext* in this tutorial that helped encode the java message object and makes it ready for message serialization required for message transmission (or message storage). This class forms the foundation of most of the message processing functionality that is provided by the HAPI library We will

encounter this class again when reviewing other features provided by the HAPI library in our subsequent tutorials. See you then!
