The NHapi Beginner's Guide

A practical guide for HL7 messages in .Net

Revision 1.0.1
ISBN E-book: 978-13-011-6935-1
ISBN Printed book: 978-94-021-0582-7

Table of Contents

Introduction

After the implementation of various HL7 versions I wrote a few blog posts on working with NHapi and HL7. Almost instantly these posts got referred to on well know social media sites and forums. Besides those referrals I got a lot of questions through e-mail and the comments on my blog on the subject of HL7, NHapi and .Net. Recently I had an e-mail conversation with a developer who just started working with HL7. At that moment it occurred to me: a structured guide on HL7 and NHapi will help developers more than trying to help solve a part of the puzzle these developers are working on. This led me to writing this e-book.

The NHapi Beginner's Guide is meant for developers who are taking their first steps in the implementation of HL7 communication with .Net using NHapi. Since both HL7 and NHapi have a steep learning curve, this guide will help you take the first plunge. Even if you are more experienced working with NHapi this guide is also a reference guide to fall back upon.

And last but not least: Thanks to Janneke and Kees for your input, corrections and suggestions.

1. What is HL7?

To understand the purpose of NHapi, you have to understand what HL7 is. Furthermore you have to have at least a basic understanding of how to read and interpret HL7 messages. There is a lot of documentation on what HL7 is, on the history of HL7 and the evolution of the HL7 protocol. In this chapter I will limit myself to a short description on the purpose of HL7 and the technical aspects of HL7.

1.1 The HL7 protocol

The HL7 protocol is specifically designed for integrating applications within the health care branch. The HL7 protocol can communicate information like patient data, appointments, lab reports and insurance information. In other words: HL7 is a standard that integrates software by supporting hospital workflows. It was originally designed around 1987 and it is updated regularly. The latest version – at the time of writing - is 2.6. That being said, HL7 versions 2.3 and 2.4 are probably the most widely used versions.

The name of the protocol stands for *Health Level 7*, which is derived from the nature of the protocol (health care) and the abstract layer the protocol plots on in the OSI model (See for more information: https://en.wikipedia.org/wiki/OSI_model#Layer_7:_application_layer). Layer 7 is the *application layer*, including protocols like HTTP and FTP.

“Why use HL7?” you might ask. Well, standardization is a good thing. Yes, it creates complex protocols, because of generalization. On the other hand: imagine a hospital where all the systems use their own interface definition. It would probably quickly become unmanageable.

1.2 HL7 Message structure

The structure of HL7 messages is fairly plain and simple. The messages are made up from segments. Segments consist of fields and fields are data types, for example a number or text or a combination of them, so called components. Segments are divided by a new line (each segment takes up a whole line). Fields are separated by a special character (in most cases this is a pipe character ‘|’). Segments start with a three letter combination, designating the type of the segment and therefore the fields it must contain, according to the HL7 specifications. The segments don’t have to be in any specific order, except for the first one.

Each HL7 message starts with a mandatory segment: the message header (MSH). The MSH segment is a special one: not only does it contain important information on the version, routing and purpose of the message, but it also contains information on how to interpret the message. Let's have a look at an example message:

```
MSH|^~\&|SYS|SYSADT|SMS|SMSADT|199912271408|HIS|ADT^A04|1817457|D|2.5|

PID||0493575^^^2^ID 1|454721||DOE^JOHN^^^^|DOE^JOHN^^^^|19480203|M||B|254
MYSTREET AVE^^MYTOWN^OH^44123^USA||(216)123-4567|||M|NON|400003403~1129086|

NK1||ROE^MARIE^^^^|SPO||(216)123-4567||EC|||||||||||||||||||||||||

PV1||O|168 ~219~C~PMA^^^^^^^^^^||||277^ALLEN MYLASTNAME^BONNIE^^^^|||||||||||||
2688684|||||||||||||||||||||||||199912271408||||||002376853
```

In this message you can see the different segments. If you look at the first line you'll see the mandatory message header segment, the segment starting with MSH. This segment contains fields that refer to the sending system and facility (in this case SYS and SYSADT), the receiving system and facility (here it is SMS and SMSADT), a date/time stamp, etc. There are three quite interesting fields: Encoding characters, Message Type and the Version ID.

The Encoding Characters field is located next to the MSH identifier. In this message it contains

the characters “|^~\&”. This field defines the special characters that are used throughout the message. The characters in the example message are the ones most used. Actually, I’ve never seen a message with different encoding characters. The first character defines the character that separates fields. So actually, the first field separation character is the one that will separate all of the fields. In this way the message header segment is different from other segments. The next character is the component separator (‘^’) followed by the field repeat separator, the escape character and the sub-components separator.

The Message Type field contains the definition of the message that follows. So if it contains the value “ADT^A01” the message belongs to the ADT group of messages (Patient Administration) and contains event message A01 (Patient Admin). You can see that ADT and A01 are separated by a component separator. According to the HL7 specs the Message Type field contains the data type CM_MSG. If you look up CM_MSG, you’ll see that it contains two components: Message Type ID and the Trigger Event ID.

The third interesting field is the last field in the MSH segment: Version ID. This field tells you which version of the HL7 specifications is used. The different versions of the specifications define different mandatory or optional segments, fields, etc. If a segment is optional and you don’t need to send it you can leave it out. If a field is optional and you don’t need to send it, you can leave it blank. As you can see in the example message you do have to send the separation characters, otherwise the receiving application can’t understand which field or components are blank. So if you see a few repeating characters like ‘|||||’ or ‘^^^^^’, it will mean that the values are left out.

The other segments and fields work in the same way. As you can see HL7 messages can get quite complex and if you see this for the first time it can be quite overwhelming. After some time you will get used to the HL7 specifications and get a better understanding on the HL7 messages.

The notation that is commonly used to reference a field or component of a message is “[segment abbreviation]-[field number]-[component number]”. So if you are talking about the second component of the message type in the message header segment (these will be explained in the next paragraphs) you can refer to the field “MSH-9-2” (you should start counting at 1 and include the segment abbreviation).

There are two ways of representing HL7 messages. In this paragraph I used the piped message as an example. Originally this is the way HL7 messages are formatted and this is the most common one. The other way is to use XML. The XML representation basically uses the same structure. It is more expressive and readable for humans, since the segment and field names are written out. The biggest difference between the piped and the XML representation is the message header. In the XML version the message header information is the start of the XML message, called the transmission wrapper. After the control information the so called domain content (the rest of the message) will be placed.

1.3 HL7 Communication Patterns

The HL7 communication pattern is (almost) always a request reply pattern. Every request needs to be responded to. In HL7 terms there are two types of messages: unsolicited messages and solicited messages. The latter can be queries or operational requests. The messages are handled in a sequential order. The next message will only be sent if a reply to the message is received. If there is no reply, the sending system will repeat the message (thus blocking the message queue until a reply is received). Note that different systems can have different implementations on how many times the message is sent and the time out for resending messages.

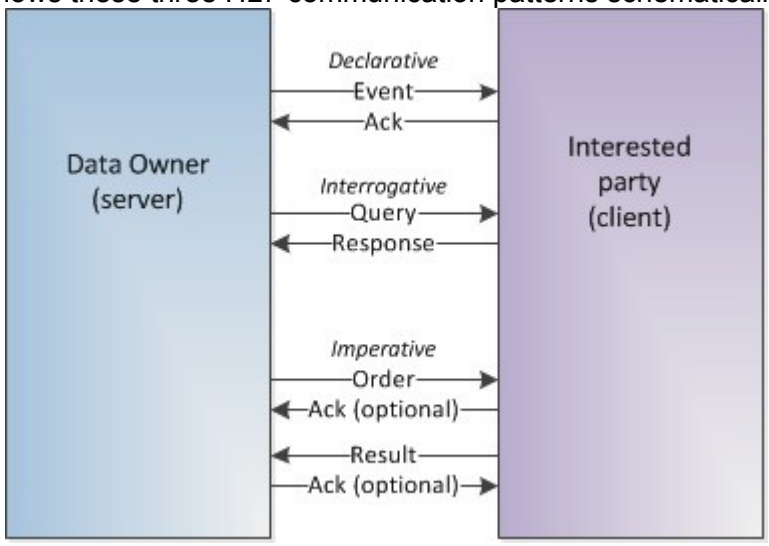
Unsolicited messages are events. If there is a trigger in the server system (the data owner), for example a new patient is admitted, the server will trigger an event. The client system (the interested party) will send an acknowledgment (ACK) message. Interested parties are configured on the server or message hub. Acknowledgments can contain three messages: message accepted (AA), there was an error (AE) or the message is refused (AR). These message pairs are also

known as a *declarative* pair.

The first type of solicited messages is queries. In this case the interested application will send a query to the data owner to get information. For example to find out details on a patient or get a list of patients. The server will respond to the query, either by returning the result of the query or by sending an error message. This is known as an *interrogative* message pair.

The second type of solicited messages is operational requests. In this case the data owner will send an order to the interested party (for example an order for the laboratory). The order must be accepted or rejected by the client application. If the order is accepted the acceptance message will be returned immediately. The order itself can be processed asynchronously, in most cases because the order has to be completed by a human (for example the collected samples have to be transported and examined). When the order is completed the result will (optionally) be sent by the interested party to the data owner. The data owner will send an acknowledge message to confirm that the order result was received. In this case the acknowledgment messages are optional, but if you leave these messages out make sure you have a way to determine if any messages get lost. These are known as *imperative* message pairs.

The next image shows these three HL7 communication patterns schematically.



1.4 Communication Protocols

As you might have noticed HL7 messages don't have a clear ending. The beginning of a HL7 message could be derived from the MSH segment, but since the segments do not have a specific order and most messages are defined with optional segments, you cannot actually know when you have received the complete HL7 message. Of course you can always wait and give a time out when no extra data is being received, but as you'll understand this doesn't guarantee that you've got the complete message, besides it will undermine the efficiency of your application. So, in order to integrate systems well with HL7 messages, the HL7 communication needs to be wrapped in other protocols. The following paragraphs will explain some protocols that are used.

1.4.1 MLLP

MLLP stands for Minimum Lower Level Protocol. And it is exactly what the acronym says. MLLP is the most widely used (about 90% of the implementations) protocol for the transport of HL7 messages. MLLP wraps the HL7 message in one start character and two closing characters all of which are in the control code section of the ASCII table. The start character is the ASCII value 11 (or hexadecimal 0x0B), the two closing characters are ASCII value 28 (hexadecimal 0x1C) and ASCII value 13 (hexadecimal 0x0D, also known as carriage return). Using the MLLP protocol will give you a clear starting point and end point of the HL7 messages. Every bit of data outside of the

start and end characters should be considered garbage.

1.4.2 HTTP

HTTP (Hypertext Transfer Protocol) is mostly used for websites. It can also be used as a wrapper for HL7 messages, though there is no clear definition on how to implement the HTTP communication when using HL7 messages. There are voices in the HL7 community that are advocating the use of HTTP. There is a proposal document that defines the implementation of HTTP for HL7 and there are software packages, like Mirth, that support the use of HTTP. The use of HTTP can be especially useful when setting up integration between different organizations.

The use of HTTP has several advantages. It is a clear and traceable protocol and there are numerous solutions for the flaws of HTTP. For example if you want to secure the communication you can use encryption by setting up a HTTPS connection. And if you want to have a form of authentication you can easily use the standard HTTP authentication.

1.4.3 Other protocols

Of course other protocols can be used to send HL7 messages, for example SOAP, files, MQ or e-mail. All of these protocols have their upsides and downsides. By implementing SOAP as a carrier for HL7 you are able to make the communication traceable and more secure by adding SOAP headers than implementing HTTP. On the other hand, you'll probably want to stick with the XML representation of HL7 messages. As with any other integration challenge you'll have to choose the protocols and methods that will give your organization the best result. Think about what your business needs in terms of security, traceability, guaranteed delivery, logging, etc.

1.5 Separating channels

The HL7 specifications recommend that the communication of each message group is separated. So if you need to implement messages from the ADT group and the SIU group you'll need to open two different communication channels. Not every developer will adhere to this advice: it is perfectly possible to use one connection. On the other hand regarding the separation of responsibilities it does have advantages in efficiency, security and maintenance. Of course these different channels depend on the protocol you are using: with TCP/IP you need to use different port numbers and with HTTP or the file system using different subdirectories is the way to go.

2. The basics of NHapi

Before you start using NHapi (Find NHapi at http://nhapi.sourceforge.net) it is good to have a clear understanding what NHapi is. NHapi is the .Net port of the open source project HAPI and it is also open source. HAPI is the HL7 implementation for Java. The beauty of that is that the HAPI examples are easily translated to NHapi.

NHapi will give you the tools to parse HL7 messages (piped messages and XML messages) to an object model representation of the HL7 message. This you can use to let your software act on the information provided by the HL7 communication. NHapi also provides the tooling to generate HL7 messages. So if you use NHapi you'll get an object model representing the different versions of the HL7 2.x specifications and the tools to parse and generate messages based on this object model. What NHapi doesn't do is the communication part of the application. In other words: depending on the type of connection and protocols you need or choose you have to build a connection and handle the wrapper protocols by yourself.

There are some alternatives for implementing HL7 messages in .Net. Of course you can always build your own parser. Depending on the scope of your application and the data you'll need from HL7 this will be a lot of work. Using a standard component gives you the advantage of using software that is tested in real life production environments and gives you the complete implementation of the specifications. Besides that the code is maintained and many people are using it, so bugs will be found faster. Another option is to use BizTalk. Microsoft has HL7 support in their BizTalk product. BizTalk, however, isn't cheap and developing BizTalk applications is a discipline on its own.

The biggest downside of NHapi is that there is little documentation and examples on NHapi. As I wrote in the beginning of this chapter, HAPI provides more documentation and examples and these are easily translated to NHapi, but you'll need some understanding on how NHapi works to correctly translate these examples. The learning curve can be quite steep in the beginning, but if you get the hang of it working with the NHapi libraries will get easier.

The next paragraphs will show how to use the parser, to interpret and create HL7 messages, and the terser to manipulate HL7 messages.

2.1 Assembly structure of NHapi

By downloading NHapi you will get a set of assemblies to reference in you .Net solution. These are:

NHapi.Base.dll

NHapi.Model.V21.dll

…

NHapi.Model.V25.dll

NHapi.NUnit.dll

NHapi.NUnit.Additional.dll

The NHapi.Base assembly contains the tools and base classes you'll need for parsing and generating HL7 messages. The base classes are used for the class structures in the implementation of the different HL7 versions. The tooling contains the parsers for piped and XML messages and the validators used with parsing. Besides these tools this assembly contains the exception definitions and logging tooling.

The different NHapi.Model.V2X assemblies contain the classes representing the different HL7 specification versions. It will contain namespaces with the different DataTypes, Groups, Segments

and Messages. These assemblies are used by the parsers in the NHapi.Base namespace and are needed for parsing the messages. After the parsing you can cast the *IMessage* object returned by the parser to the specific message from the NHapi.Model.V2X namespace. The *IMessage* contains enough information (from the MSH segment) to determine the type of message and the HL7 version used. When working with a specific HL7 version you only have to reference the Base assembly and the assembly of that specific version. If you have to support multiple HL7 versions, you'll need to reference all the versions you want to support.

The NHapi.NUnit and Nhapi.NUnit.Additional assemblies contain the unit tests and additional unit tests used by the developers of the NHapi project.

2.2 Using the parser: Parsing a message

After receiving a message you have to parse it. Take a look at the following example.

```
using NHapi.Base.Model;

using NHapi.Base.Parser;

using NHapi.Base;

using NHapi.Model.V23;

using NHapi.Model.V23.Message;

using NHapi.Model.V23.Segment;


// This is the message that will be parsed

// This code example was copied from the Quickstart Guide for NHapi, that can be

// found in the documentation section of the NHapi Sourceforge space.

string message = @"MSH|^~\&|SENDING|SENDER|RECV|INST|20060228155525||
QRY^R02^QRY_R02|1|P|2.3|\n

QRD|20060228155525|R|I|||||10^RD&Records&0126|38923^^^^^^^^&INST|||";


// Get a new instance of the PipeParser to parse this piped message

PipeParser parser = new PipeParser();

// Parsing it will return an abstract message

IMessage m = parser.parse(message);


// Cast the abstract message to the right type

// Other examples will show how to determine the type

// of message if this is unknown

QRY_R02 qryR02 = m as QRY_R02;

// Output one of the field values to the console

Console.WriteLine(qryR02.QRD.GetWhoSubjectFilter(0).IDNumber.Value);
```

What happens here is after receiving a message the *PipeParser* is used to parse the message. There is no error handling, but you should catch a HL7Exception, which will be thrown if anything

went wrong while parsing the message. The *IMessage* interface is the base class of all the specific HL7 message classes. In this case it is pretty clear what type of message is parsed, so the cast to the class QRY_R02 is safe. Note that the QRY_R02 class has the namespace NHapi.Model.V23.Message since this is a HL7 V2.3 message according to the field MSH-12. After casting the object to the right message class you will be able to use the complete object structure to read out any information from the message.

The parser uses reflection to get to the right class definition. What it does is read out the message header. With the information provided there it builds up the complete namespace of the message class and loads this class through reflection. The message class builds the complete message structure, according to the specifications, so the parser is able to read and fill the rest of the message. The namespace is easily determined by getting these values “NHapi.Model.V[MSH-12].Message.[MSH-9-3]”. In this case this will result in the value “NHapi.Model.V23.Message.QRY_R02” (of course the dot in the version number has to be deleted).

2.3 Using the parser: Creating messages

You can also use the parser to create a message based on an object structure of a certain NHapi.Model.V2X message class. Here's an example:

```
using NHapi.Base.Parser;
using NHapi.Base;
using NHapi.Model.V231.Message;


// This code example was copied from the Quickstart Guide for NHapi, that can be
// found in the documentation section of the NHapi Sourceforge space.
// At first create an object of the desired message class
QRY_R02 qry = new QRY_R02();
// And fill the message header segment with the correct values
// Note that certain values are explicitly set, but
// are filled by default.
qry.MSH.MessageType.MessageType.Value ="QRY";
qry.MSH.MessageType.TriggerEvent.Value = "R02";
qry.MSH.MessageType.MessageStructure.Value = "QRY_R02";
qry.MSH.FieldSeparator.Value = "|";
qry.MSH.SendingApplication.NamespaceID.Value="Cohie Central";
qry.MSH.SendingFacility.NamespaceID.Value = "COHIE";
qry.MSH.ReceivingApplication.NamespaceID.Value="Clinical Data Provider";
qry.MSH.ReceivingFacility.NamespaceID.Value=facility;
qry.MSH.EncodingCharacters.Value = @"^~\&";
qry.MSH.VersionID.VersionID.Value = "2.3.1";
qry.MSH.DateTimeOfMessage.TimeOfAnEvent.SetLongDate(DateTime.Now);
```

```
qry.MSH.MessageControlID.Value = messageControlId;
qry.MSH.ProcessingID.ProcessingID.Value="P";

// Get the XCN field from the QRD segment
XCN st = qry.QRD.getWhoSubjectFilter(0);
st.AssigningAuthority.UniversalID.Value = facility;
st.IDNumber.Value = mrn;

// Fill the rest of the information in the QRD segment
qry.QRD.QueryDateTime.TimeOfAnEvent.SetLongDate(DateTime.Now);
qry.QRD.QueryFormatCode.Value = "R";
qry.QRD.QueryPriority.Value = "I";

// Get the CE field of the QRD segment
CE what = qry.QRD.getWhatSubjectFilter(0);
what.Identifier.Value = "RES";

// Create an instance of the PipeParser to parse the message
// to a piped HL7 message. The Encode method will return a string
// containing the message.
PipeParser parser = new PipeParser();
string result = parser.encode(qry);
```

The code here is fairly simple. After instantiating a specific message class the different properties are filled (segments and fields). In the end the encode method of the *PipeParser* is used to generate the HL7 piped message. When you instantiate a message class the constructor builds up the complete structure (except for repeatable fields). It is quite handy that you can instantly use most of the properties (that's probably the reason that it works like this, since parsing HL7 messages will be easier), but on the other hand it is impossible, after parsing a message to an object model, to know which segments weren't included in the message. Of course, if a segment lacks important information you can deduce it.

The CE and XCN classes are data types (complex fields). In this example the objects are retrieved by calling a method: *getWhoSubjectFilter(0)* and *getWhatSubjectFilter(0)*. Both are repeatable fields, you can add more than one CE and XCN fields to the message. The parameter that is given to these methods is the index that has to be returned. If there is an object at that instance it will be returned. If there is no object present NHapi will add a new object and return that new instance. In other words: by calling the method with zero as an argument an CE (or XCN) object is added to the QRY segment. If you want to add a second CE or XCN object to the QRY segment, you can call these methods with a number one as argument. This is the way all the repeatable fields work. Repeatable field always provide a property to determine how many repetitions (objects) exist in the field. The method is called *[Field name]RepetitionsUsed*. So, to determine how many XPN repetitions are in that field, get the property

WhoSubjectFilterRepetitionsUsed.

2.4 Using the terser: Creating an Ack message

HAPI provides a method in the IMessage interface to implement the generation of an ACK message based on the current message. NHapi lacks this functionality. There are a few ways to create an ACK message. One way is to create an object of the ACK message class and populate the message like the example in paragraph 2.3. Another way is to use the terser provided by NHapi. An ACK message is basically a copy of the MSH (with a few changed fields) and an added MSA segment (with an optional ERR segment is case of an error). The next example shows how to use the Terser and the *DeepCopy* classes to create an ACK message based on the current message.

```
using NHapi.Base.Model;
using NHapi.Base.Util;


/// <summary>
/// Create an Ack message based on a received message
/// </summary>
/// <param name="inboundMessage">received message</param>
/// <param name="ackCode">Acknowlegde code</param>
/// <param name="ackMessage">Message to be created</param>
/// <param name="errorMessage">Error message to send</param>
public static void MakeACK(IMessage inboundMessage, string ackCode, IMessage ackMessage,
string errorMessage)
{
        Terser t = new Terser(inboundMessage);
        ISegment inboundHeader = null;
        try
        {
                inboundHeader = t.getSegment("MSH");
        }
        catch (NHapi.Base.HL7Exception)
        {
                throw new NHapi.Base.HL7Exception("Need an MSH segment to create a
                response ACK");
        }
        MakeACK(inboundHeader, ackCode, ackMessage);
}


/// <summary>
```

```
/// Create an Ack message based on a received message
/// </summary>
/// <param name="inboundMessage">received message</param>
/// <param name="ackCode">Acknowlegde code</param>
/// <param name="ackMessage">Message to be created</param>
/// <param name="errorMessage">Error message to send</param>
public static void MakeACK(ISegment inboundHeader, string ackCode, IMessage ackMessage,
string errorMessage)
{
        if (!inboundHeader.GetStructureName().Equals("MSH"))
                throw new NHapi.Base.HL7Exception("Need an MSH segment to create a
                response ACK (got " + inboundHeader.GetStructureName() + ")");


        // Find the HL7 version of the inbound message:
        string version = null;
        try
        {
                version = Terser.Get(inboundHeader, 12, 0, 1, 1);
        }
        catch (NHapi.Base.HL7Exception)
        {
                // I'm not happy to proceed if we can't identify the inbound
                // message version.
                throw new NHapi.Base.HL7Exception("Failed to get valid HL7 version from
                inbound MSH-12-1");
        }


        // Create a Terser instance for the outbound message (the ACK).
        Terser terser = new Terser(ackMessage);


        // Populate outbound MSH fields using data from inbound message
        ISegment outHeader = (ISegment)terser.getSegment("MSH");
        DeepCopy.copy(inboundHeader, outHeader);


        // Now set the message type, HL7 version number, acknowledgement code
        // and message control ID fields:
```

```
        string sendingApp = terser.Get("/MSH-3");
        string sendingEnv = terser.Get("/MSH-4");
        terser.Set("/MSH-3", “CommunicationName”);
        terser.Set("/MSH-4", “EnvironmentIdentifier”);
        terser.Set("/MSH-5", sendingApp);
        terser.Set("/MSH-6", sendingEnv);
        terser.Set("/MSH-7", DateTime.Now.ToString("yyyyMMddmmhh"));
        terser.Set("/MSH-9", "ACK");
        terser.Set("/MSH-12", version);
        terser.Set("/MSA-1", ackCode == null ? "AA" : ackCode);
        terser.Set("/MSA-2", Terser.Get(inboundHeader, 10, 0, 1, 1));

        // Set error message
        if (errorMessage != null)
            terser.Set("/ERR-7", errorMessage);
    }
```

This code takes the original message and copies the message header segment. Then with the terser the header is completed and the other segments are added. As you can see the terser interprets the field notation to set the values. By using this method *terser.Set("/MSH-9", "ACK");* you'll set the ninth field in the message header segment (as you know that is the message type field).

3. Building a HL7 Client

I'm sure you've heard the saying before: "The proof of the pudding is in the eating." After reading about the basics of HL7 and NHapi it is time to build an HL7 client in .Net. It is kind of useless to add a complete solution in a book, so I'm going to make a few assumptions. First, I'm assuming you have development experience in .Net. I'll be using C# in my examples as you've noticed. Second: this example will be an example of HL7 (V2.3 and V2.4) unsolicited ADT messages over TCP/IP using MLLP. To keep the example as simple as possible I will only accept one connection at a time (so no multi-threading). The next chapter will contain a link to the complete solution with this example.

Note: *This example still will cover only the basics. It is not perfect nor nearly production ready. There is no good structure and the code may not be the most efficient code: all this is done to make the example as clear as possible*.

To save you some time copying and pasting the code examples, you can download the code from this chapter with this link: http://tiny.cc/1y1b1w.

3.1 TCP/IP connection

The first step is to setup a basic TCP/IP listener and accept connections. So the basis of the application will look something like this.

```
class HL7App
{
    #region Defaults
    private const int DefaultPort = 1250;
    #endregion

    #region Private properties
    private bool continueProcessing = true;
    private TcpListener listener;
    #endregion

    #region Static method Main
    static void Main(string[] args)
    {
        int port = DefaultPort;
        if (args.Length == 1)
            int.TryParse(args[0], out port);

        Console.WriteLine("Starting HL7 client on port {0}.", port);
        Console.WriteLine("Press Ctrl-c to exit.");
```

```
        HL7App app = new HL7App();
        app.StartTCPListener(port);
}

private void Console_CancelKeyPress(object sender, ConsoleCancelEventArgs e)
{
        Console.WriteLine("Stopping...");

        continueProcessing = false;
        listener.Stop();
}
#endregion

#region Private methods
private void StartTCPListener(int port)
{
        // Catch the break key press
        Console.CancelKeyPress += Console_CancelKeyPress;

        IPAddress ip = IPAddress.Any;
        listener = new TcpListener(ip, port);

        listener.Start();
        while (continueProcessing)
        {
                Console.WriteLine("Waiting for connection.");
                TcpClient client = listener.AcceptTcpClient();
                Console.WriteLine("Connection received.");

                HandleClient(client);
        }
}

private void HandleClient(TcpClient client)
```

```
        {
            // Add code to handle client stream
        }
        #endregion
    }
```

Now this console application can accept a TCP/IP connection through any network adapter. After the connection is closed it will be ready to accept new connections. HL7 connections over TCP/IP will be one continuous connection. If, for whatever reason, the connection is lost, the server will build up a new connection.

3.2 Handling MLLP

The next step is to get the data from the connection and handle the MLLP protocol. A simple way of doing this is to read out every character (the *ReadByte()* will wait for the next byte to be available on the stream and then return the next available byte). By waiting for the start character and reading until the two end characters are read, the complete message is received and can be processed.

```
#region Private constants
private const int MLLP_START_CHARACTER = 11; // HEX 0B
private const int MLLP_FIRST_END_CHARACTER = 28; // HEX 1C
private const int MLLP_LAST_END_CHARACTER = 13; // HEX 0D
#endregion


private void HandleClient(TcpClient client)
{
    string message = string.Empty;
    while (client.Connected)
    {
        // Read the next byte from the stream
        int b = client.GetStream().ReadByte();
        if (b == -1)
        {
            // Client disconnected
            client.Close();
        }

        // Start adding characters to the message
        // if the MLLP start character is received.
        if ((b == MLLP_START_CHARACTER) || (message.Length > 0))
```

```
                message += (char) b;

            // Check if the message string ends with the two
            // MLLP end characters. If so, a complete HL7 message is
            // received.
            if ((message.Length > 3) && ((message[message.Length - 2] ==
            MLLP_FIRST_END_CHARACTER) && (message[message.Length - 1] ==
            MLLP_LAST_END_CHARACTER)))
            {
                // String away the MLLP characters to keep the pure HL7 message
                string hl7Message = message.Substring(1, message.Length - 3);

                // Parse the HL7 message and get a response
                string hl7Response = ParseHL7Message(hl7Message);
            }
        }
        Console.WriteLine("Connection closed.");
    }

    private string ParseHL7Message(string message)
    {
        return string.Empty;
    }
```

The result is a complete received HL7 message. The next step is to parse the message with NHapi.

3.3 Parsing the HL7 message

After receiving the complete message NHapi is used to parse the message. The parsed message is used to generate the response message. The ACK object created as a response message is from the specific namespace containing the HL7 version number. Adding and sending the ACK message will complete the HL7 communication sequence for this message.

```
using System.IO;
using NHapi.Base;
using NHapi.Base.Parser;
using NHapi.Base.Model;
using NHapi.Base.Util;

private string ParseHL7Message(string message)
```

```
{
        IMessage ackMessage = null;
        string result = string.Empty;
        PipeParser parser = new PipeParser();

        try
        {
                IMessage hl7Message = parser.Parse(message);

                // Create a response message
                ackMessage = CreateACK(hl7Message, "AA");
                result = parser.Encode(ackMessage);
        }
        catch (HL7Exception ex)
        {
                Console.WriteLine("Error while parsing: {0}", ex.Message);
        }

        return result;
}

private IMessage CreateACK(IMessage message, string returnCode)
{
        IMessage result = null;

        // Check the message version
        // to create an ACK object from the right
        // namespace
        switch (message.Version)
        {
                case "2.3":
                        result = new NHapi.Model.V23.Message.ACK();
                        break;
                case "2.4":
                        result = new NHapi.Model.V24.Message.ACK();
```

```
                    break;
                default:
                    throw new NotSupportedException("This HL7 version isn't supported.");
            }
            // An alternative to the switch statement here
            // is using reflection to load the right NHapi assembly
            // and create a new instance of the ACK class from there
            // This is better because it will work with each HL7 version
            //
            //string ackClassType = string.Format("NHapi.Model.V{0}.Message.ACK,
            NHapi.Model.V{0}", message.Version.Remove(message.Version.IndexOf('.'), 1));
            //Type x = Type.GetType(ackClassType);
            //result = (IMessage)Activator.CreateInstance(x);

            // Fill the ACK message with the right values, the method from
            // paragraph 2.4 is used here
            MakeACK(message, returnCode, result);

            return result;
        }
```

With regard to creating the ACK object, using reflection to create the specific ACK object is better, since the code works in a more generic way and you don't have to make any decision based on the HL7 version provided. The version using a switch statement is added to be more explanatory. In the method *HandleClient* code to send the ACK message to the server is added.

```
// Parse the HL7 message and get a response
string hl7Response = ParseHL7Message(hl7Message);

// Add MLLP characters to the response message
string responseMessage = (char)MLLP_START_CHARACTER + hl7Response +
(char)MLLP_FIRST_END_CHARACTER + (char)MLLP_LAST_END_CHARACTER;
StreamWriter writer = new StreamWriter(client.GetStream());
writer.Write(responseMessage);
writer.Flush();
```

Now the program is ready to go to the business logic part of the HL7 parsing. You have a complete object structure of the HL7 message, now your program has to act on it. It is impossible to work this out in a book, so in the example I'll keep it simple by writing the PatientID to the console and/or returning an error. First there is a slight modification of the *ParseHL7Message* method as shown in the previous paragraph.

```
IMessage hl7Message = parser.Parse(message);


string errorCode = "AA";
string errorMessage = null;
if (!ProcessMessage(hl7Message, out errorMessage))
        errorCode = "AE";


// Create a response message
ackMessage = CreateACK(hl7Message, errorCode, errorMessage);
result = parser.Encode(ackMessage);
```

The *ProcessMessage* method is added to process the specific HL7 message versions. Depending on how much HL7 versions and message types you need to support this can range from a simple interpretation to an elaborate set of business logic.

```
private bool ProcessMessage(IMessage hl7Message, out string errorMessage)
{
        errorMessage = null;


        Console.WriteLine("A HL7 message of the type {0} and version {1} is received.",
        hl7Message.GetStructureName(), hl7Message.Version);
        if (!hl7Message.GetStructureName().StartsWith("ADT_"))
        {
                errorMessage = "This message structure is not supported.";
                return false;
        }


        switch (hl7Message.Version)
        {
                case "2.3":
                        // Add code to handle the V2.3 of these ADT messages
                        NHapi.Model.V23.Segment.PID pid1 = (NHapi.Model.V23.Segment.PID)
                        hl7Message.GetStructure("PID");
                        Console.WriteLine("PatientID {0}.", pid1.AlternatePatientID.ID.Value);
                        break;
                case "2.4":
                        // Add code to handle the V2.4 of these ADT messages
                        NHapi.Model.V24.Segment.PID pid2 = (NHapi.Model.V24.Segment.PID)
                        hl7Message.GetStructure("PID");
```

```
                Console.WriteLine("PatientID {0}.", pid2.PatientID.ID.Value);
                break;
            default:
                errorMessage = "This message version is not supported.";
                return false;
        }

        return true;
    }
```

3.4 Solution architecture

The example application in this chapter has its focus on a clear explanation on how to use NHapi. This isn't a very good example on how to integrate HL7 correctly and maintainable into you software, for example because it is all within one class. Remember that every piece of good software starts with a good solution architecture.

The network communication should be within a data layer of your application. Design this class well. Think on which communication pattern you need (is it only unsolicited messages or also solicited messages?). Does the communication have to be part of the software or is it better to create a separate component (like a Windows service). Do you need to set up the communication asynchronously or not (does the rest of the application need to wait for the reply message or will this happen 'under water' and how do you handle faults in that case).

For the integration you should really use the Message control ID (MSH-10) and Processing ID (MSH-11) to relate the different HL7 messages. Especially when implementing solicited messages. With these fields you can relate the ACK messages to the original messages.

You should separate the NHapi Model classes from your application. It can be tempting to use the Model namespaces of NHapi throughout you application. It is easy and the information is already there. If you ever download and use new versions of NHapi, this can break your code in a lot of places. Besides: if you are supporting more than one HL7 version the code will get complex quickly. It is better to implement a structure of translating the different HL7 versions to your own domain model. With this domain model you only have the information that is really interesting and can implement the business logic specific to your application. An ADT_A03 event (the discharge of a patient) can have a different effect in different systems: maybe this person is not of interest anymore and you can delete the records or maybe you have to start a specific process to handle the discharge.

So think hard on what you application needs. Write out the requirements and design and build your application accordingly. Of course this is applicable to every software implementation. If you are new to HL7 (and NHapi) make sure to keep all this in mind.

4. Tools and resources

There are a lot of useful resources on the internet regarding HL7. The following resources and tools will provide you with in depth insight and functionality to get your application development going. Besides these tools and resources there are multiple posts (and discussions) on my blog (http://www.dib0.nl/). Feel free to ask questions!

4.1 HL7 organization

The HL7 organization is the global authority on the HL7 standard. Besides providing you with the complete specifications on HL7 they provide many documents and guidelines for developing HL7 implementations, interoperability, security, etc. In many countries there are local HL7 organizations with additional standards. The international specifications are the U.S.A. standards. Other countries may have different standards (for example bank account numbers or insurance specifications). Your local HL7 organization will provide you with the implementation specifications with the local flavor.

http://www.hl7.org/

4.2 NHapi

Of course the project page of NHapi is a good starting point. There isn't much documentation, but there are some discussions on the forum and there is always the possibility to ask your questions there.

http://nhapi.sourceforge.net/home.php

4.3 HL7Inspector

The open source project HL7Inspector provides a tool for reading, checking and altering HL7 messages. What's more important: it provides functionality for sending and receiving HL7 messages. The example application in chapter 3 has been tested with HL7Inspector. A really good tool for the development and maintenance of you HL7 implementation.

http://sourceforge.net/projects/hl7inspector/

4.4 Enterprise Library

You'll probably know the Enterprise Library, since it's kind of a default library for .Net applications. The reason for mentioning it is that the logging within NHapi is using the Enterprise Library (EntLib). EntLib isn't a requirement for using NHapi, but to enable the logging that is built in to NHapi (which can be handy for tracing production issues) you'll have to use EntLib.

http://entlib.codeplex.com/

4.5 Mirth Connect

The Swiss army knife of HL7 integration is Mirth Connect. Whether you are working in a large scale health care organization or in a development environment, Mirth Connect will probably be useful. Mirth Connect is an open source solution (with optional commercial support) that can help you with development, (large scale) testing and monitoring the production environment. Besides that you can use it as a HL7 communications hub by receiving HL7 messages and forward them to one or more systems. You can also filter messages by type, so applications will only receive the messages that are of interest to them. Mirth Connect supports a variety of protocols including TCP/IP (with MLLP), HTTP, files and SOAP.

http://www.mirthcorp.com/products/mirth-connect/

4.6 Medical Free/Libre and Open Source Software

This library contains a list of software related to healthcare. Ranging from a hospital information system (HIS) to specific tools used within the healthcare branch. If you need any tooling regarding to healthcare, check this library out. The following link contains the projects specifically related to HL7.

http://www.medfloss.org/project-wizard?f[0]=taxonomy_vocabulary_6%3A131

###

About the author:

Bas van den Berg lives in The Netherlands. He studied Information Technology (bachelor degree) and Theology (master degree, specialized in pastoral counseling). He's working as a domain architect (enterprise architecture specifically for one of the divisions of the company) with the largest insurance company in the Netherlands. Besides being a coach and pastoral counselor in his spare time he loves to enable people to work together as efficiently as possible. Creating solutions for business problems and developing business strategy is a part of that.

Bas has experience with quite a lot of different technologies, like Linux, Windows, AS/400 and Web, and programming languages, like Java, C#, C++, Cool:plex, classic ASP, Coldfusion and Delphi (among others). He's a big fan of open source and worldwide standards.

Connect with Me Online:

Twitter: http://twitter.com/Division_by_Zer
LinkedIn: http://nl.linkedin.com/in/basvdb
Smashwords: https://www.smashwords.com/profile/view/BvdBerg
My blog: http://www.dib0.nl