HL7Fuse

Configuration and Developer Guide, V1.0

Division by Zero

Table of Contents

Introduction

On my blog I get a lot of questions on how to set up a complete .Net system for HL7 message integration. In other words: all over the world developers create integration components from scratch to add HL7 integration to their applications. After working for a while with NHapi, the most complete and free component to support HL7 with .Net, I started to miss functionality. To make my life easier (and hopefully the life of other developers, I created the NHapiTools.

After that I build HL7Fuse. HL7Fuse isn't easy to describe in one word or sentence. It is based on SuperSocket and implements the MLLP/HL7 protocol, including SSL/TLS support. So using HL7Fuse you have a complete protocol implementation and you can focus on building the business logic for your application. Without any business logic implementation HL7Fuse can be used to act as a test HL7 server for your development environment.

Added to this HL7Fuse provides a standard implementation of such a business logic component, called the hub. The Hub allows you to receive HL7 messages and forward these messages based on routing rules. The message will be forwarded to an endpoint. A few standards endpoints, like a file end point or a MLLP/TCP end point, are also provided by the Hub. Of course, using the Endpoint interface you can always add your own endpoints.

I'd like to describe HL7Fuse as a Swiss pocket knife for your .Net/HL7 development needs. If you have any questions, please contact me through my blog at http://www.dib0.nl.

The sources and releases of NHapiTools and HL7Fuse can be found on Github:

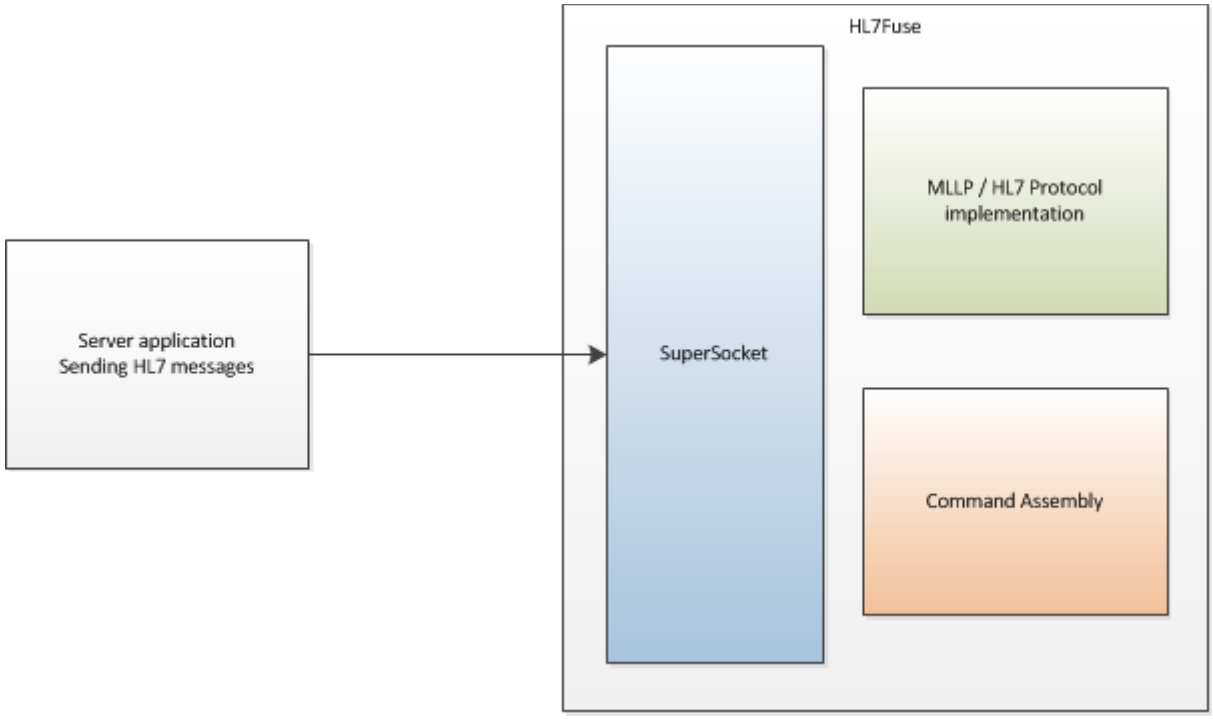
https://github.com/dib0/NHapiTools

https://github.com/dib0/HL7Fuse

Note: A basic understanding of HL7, NHapi, C#/.Net and .Net configuration is assumed in this document.

1 How HL7Fuse works

HL7Fuse is based on SuperSocket (http://docs.supersocket.net/). There is quite a lot of documentation on this product, so I won't be going in to detail on SuperSocket and stick to HL7Fuse.



HL7Fuse in it's basic form is a configured SuperSocket application with a HL7 (using MLLP) over TCP/IP. SuperSockets provides the possibility to add SSL or TLS as encyption layer to the TCP/IP connection.

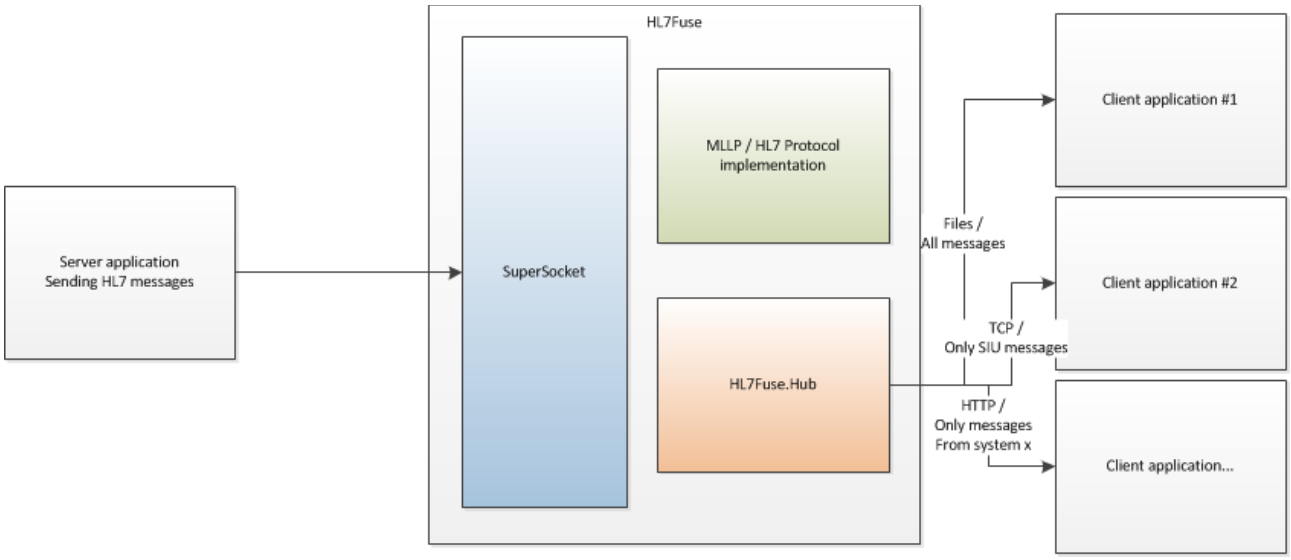
So the server application will send a HL7 message, which is received by HL7Fuse. HL7Fuse will cover all the protocol details for you. After receiving the message HL7Fuse will parse the message using NHapi. After successfully parsing the message to a NHapi Imessage object, SuperSocket will call the, so called, command assembly for further processing. After the processing is done, HL7Fuse will automatically return a ACK message to the server application. This can be a AA or AE message, based on configuration, implemented HL7 events and error states from the command assembly.

Without a command assembly HL7Fuse is just an implementation of the HL7 protocol over TCP/IP (using MLLP). And can be used as a development server that consumes/parses the HL7 message and returns an ACK.
You can implement your own command assembly to implement the logic needed for HL7 integration on your application.

1.1 HL7Fuse.Hub

The HL7Fuse.Hub assembly is an implementation of a command assembly. This implementation provides several endpoints and rule based routing of HL7 messages to these endpoints.

In other words, using the HL7Fuse.Hub command assembly, you have a HL7 message broker that allows you to receive messages and forward them to various endpoints using various protocols. Also you are able to manipulate the messages using your own implementation of the IMessageHandler interface.



1.2 Supported protocols

The current release of HL7Fuse supports these protocols:

- HL7 piped messages for all connections;
- TCP/IP using MLLP or UDP using MLLP for incoming connections (SLL/TLS is possible)[1];
- TCP/IP using MLLP (SSL/TLS is possible), HTTP(S), Files for outgoing connections.

1 *UDP is provided by SuperSocket, but highly unusual for HL7 connections*

2 The basic configuration options

The configuration of HL7Fuse consist of four parts. One for the SuperSocket settings, one for the general application settings and two for the configuration of the HL7Fuse.Hub command assembly.

2.1 General application settings (AppSettings)

The ServiceName is used as a name for the service when installed as a Windows service. Using the commandline HL7Fuse can be installed as a services, so it can automatically be started on boot and managed as a Windows service (so restart when stopped, etc.).

```
<add key="ServiceName" value="HL7Fuse"/>
```

The CommunicationName is used to to fill the sending application name within the HL7 messages. For example if a message is received and an acknowledgment is returned, the sending application of the latter will be filled using this configuration key. The receiving application will be filled with the name of the sending application of the original message.

```
<add key="CommunicationName" value="HL7Fuse"/>
```

The EnvironmentIdentifier is used to fill the HL7 messages with an environment parameter. This you can use to show if you are working in a development, test, production, environment etc.

```
<add key="EnvironmentIdentifier" value="Development"/>
```

The configuration key HandleEachMessageAsEvent is to determing in which way the command assembly will handle HL7 messages. The command assembly must implement one or more classes to handle for different types of events. So HL7Fuse can look for a specific class for a specific HL7 message or send all HL7 messages to one class. If this key is set to "true" HL7Fuse will look for a specific handler class for each separate HL7 message.

For example, if a SIU_S12 (V2.3) message is received:

If the setting is false: HL7Fuse wil look for a class named V23.MessageFactory in the command assembly.

If the setting is true: HL7Fuse wil look for a class named V23.SIU_S12 in the command assembly.

You can find more detail on this in chapter 3.

```
<add key="HandleEachMessageAsEvent" value="false"/>
```

The AcceptEventIfNotImplemented determines how HL7Fuse will reply to a sending system if no handler class could be found in the command assembly (see the HandleEachMessageAsEvent configuration key or chapter 3). After receiving a message HL7Fuse will search for a class that handles that type of message. If this class is not found HL7Fuse will return an ACK with an error status ("AE") if this setting is set to "false". Otherwise HL7Fuse always returns an ACK with the status "AA". "AE" status ACK messages

are handled in different ways, most of the time the sending system will react by trying to resend the message. If the sending system receives an ACK message with the status "AA", the message was accepted and handled correctly.

Note: If you are using HL7Fuse as a test server for your development environment and you set this configuration key to "true", all messages are accepted if parsed correctly. So there is no real need for a command assembly, unless you need specific behaviour for you testing needs.

```
<add key="AcceptEventIfNotImplemented" value="true"/>
```

2.2 SuperSocket settings

As explained in chapter 1, HL7Fuse is based on SuperSocket. Actually, HL7Fuse *is* SuperSocket with a HL7/MLLP protocol implementation. Naturally you'll have to configure SuperSocket correctly for HL7Fuse to run.

For starters, the SuperSocket section needs to be added:

```
<configSections>
       <section name="superSocket"
type="SuperSocket.SocketEngine.Configuration.SocketServiceConfig,
SuperSocket.SocketEngine"/>
</configSections>
```

After that you can add "servers". Adding a server does nothing more than making a server class implementation known to SuperSocket with a specific key. This key is then used to configure a listener for incoming connections.

To configure HL7Fuse, you'll need to add the MLLP server:

```
<serverTypes>
       <add name="MLLPServer" type="HL7Fuse.MLLPServer, HL7Fuse"/>
</serverTypes>
```

Now the servertype can be used to configure a listener (for incoming TCP/IP, MLLP based HL7 connections). The "serverTypeName" property links the servertype to this listener.

```
<server name="HL7Fuse" serverTypeName="MLLPServer" ip="Any" port="2020"
maxRequestLength="2048" maxConnectionNumber="100">
</server>
```

The configuration options (properties) for these servers are plenty. The following table contains all the properties, if they are mandatory and their default value.

Property	Description	Mandatory	Default value
ServerTypeName	This links the server types to the listener.	Yes	-
ServerType	Can be used to set a servertype. Used instead of the serverTypeName	No	-

Property	Description	Mandatory	Default value
	combined with registered server types.		
ReceiveFilterFactory	Allows the use of an alternative ReceiveFilterFactory.	No	-
Ip	Ip-adress the listener uses to listen to. Use the value "Any" to listen to all available ip-adresses.	Yes	-
Port	TCP port the listener uses to listen.	Yes	-
Disabled	Option to (temporarily) disable this listener.	No	False
Name	Unique name for the listener. Is used to identify different listeners in the logging etc.	Yes	-
Mode	Determine the socket mode (TCP or Udp)	No	TCP
SendTimeOut	Timeout for the session (0=not passed to the socket)	No	5000ms
MaxConnectionNumber	Maximum number of connections allowed per listener	No	100
ReceiveBufferSize	Size of the buffer for receiving messages. (0=not passed to the socket)	No	4096
SendBufferSize	Size of the send buffer. (0=not passed to the socket)	No	0
SyncSend	If sending messages need to be synchronously or not.	No	False
LogCommand	Indicator if messages must be logged in the log file or not.	No	False
ClearIdleSession	Indication if idle session must be cleared or kept.	No	False
ClearIdleSessionInterval	Interval in which idle session have to be cleared.	No	120s
IdleSessionTimeOut	Timeout after which the	No	300s

Property	Description	Mandatory	Default value
	session is determined to be idle.		
Certificate	See paragraph 2.2.1	No	-
Security	Indicates is encryption has to be applied on the connection. Values are SSL or TLS.	No	-
MaxRequestLength	Maximum allowed length of incoming messages. (Note that HL7 messages can get quite large).	No	1024
DisableSessionSnapshot	Disables taking snapshot of the listener states. Snapshots are used for recovery.	No	False
SessionSnapshotInterval	Interval timer for taking snapshots.	No	1 second
ConnectionFilter	Names (comma seperated) of the connection filters that apply to this listener	No	-
CommandLoader	Names of the custom command loaders that must be used for this listener.	No	-
KeepAliveTime	Time to keep the connection (socket) alive.	No	10 minutes
KeepAliveInterval	Interval in which the keepalive time is set.	No	60 seconds
ListenBacklog	Sets the size of the backlog for this listener.	No	100
StartupOrder	Defines in which order the listeners must be started.	No	-
LogFactory	Creates the possibility to apply a custom LogFactory.	No	-
SendingQueueSize	Determines the size of the sending queue.	No	5
LogBasicSessionActivity	Indicates whether basic session activity (like connected or disconnected) must be logged.	No	False

Property	Description	Mandatory	Default value
LogAllSocketException	Indicator if the socket exceptions must be logged or not.	No	False
TextEncoding	Name of the default encoding that must be used. The name must be formatted so it can be understood by the Encoding.GetEncoding() method.	No	ASCII encoding

2.2.1 Certificates

If the option Security (see paragraph 2.2) is set to either SSL or TLS, a certificate must be provided. You can do this by adding the following certificate tag:

```
<server name="HL7Fuse" serverTypeName="MLLPServer" ip="Any" port="2020"
maxRequestLength="2048" maxConnectionNumber="100">
	<certificate filePath="localhost.pfx" password="supersocket"></certificate>
</server>
```

The filePath is the absolute or relative path to the PFX file (other certificate methods can be used, but this seems to be the easiest). In the password property the password used for the PFX file must be given.

Of course, the certificate must be a valid one (with a valid certificate authority) for the other party of this connection to accept it without any problems.

2.2.2 Command assemblies

As explained in chapter 1 SuperSocket uses a so called command assembly to handle the specific handling of the received messages. The signature of such a command handler class depends on the type of session, protocol and RequestInfoParser object is used. HL7Fuse provides these implementations. More on that subject in the next chapter. To configure a command assembly for a specific listener, you'll need to add the following tag to the listener configuration:

```
<server name="HL7Fuse" serverTypeName="MLLPServer" ip="Any" port="2020"
maxRequestLength="2048" maxConnectionNumber="100">
	<commandAssemblies>
		<add assembly="HL7Fuse.Hub" />
	</commandAssemblies>
</server>
```

Since the HL7Fuse.Hub assembly is a command assembly for HL7Fuse it makes a great example. The assembly name should be like the file name, without the file extension (“.dll”).

2.3 HL7Fuse.Hub settings

As written a few times before this paragraph in the manual, the HL7Fuse.Hub is a command assembly that gives you standard functionality and some extensibility. To enable the HL7Fuse.Hub on one or more listeners, follow the example in paragraph 2.2.2. Since the HL7Fuse.Hub works with a MessageFactory, in stead of individual message handler classes, you'll need to set the application setting "HandleEachMessageAsEvent" to false, as described in paragraph 2.1.

After doing this, all the HL7 messages that are received go through the HL7Fuse.Hub.

2.3.1 Endpoints

Endpoints can send the received HL7 messages to other systems. This is the basic "hub" functionality. To determine which messages are send to which endpoint, routing rules are used (see paragraph 2.3.2). Each type of endpoint supports a different protocol. Of course it is possible to define more than one endpoint of one protocol type. To configure endpoints, first you need to add the configuration section.

```
<configSections>
       <section name="endpoints"
type="HL7Fuse.Hub.Configuration.EndPointConfigurationHandler, HL7Fuse.Hub"/>
</configSections>
```

In the endpoints configuration you can add endpoint of different types (different protocols). The name property is the unique identity of the endpoint to be used with the routing rules.

FileEndpoint

The FileEndPoint can be used for systems that integrate through the file system. Of course you can also use this endpoint to log all or certain HL7 messages. To add this endpoint add the following configuration:

```
<FileEndpoint name="TestFileEndPoint" targetDirectory="c:\data\t" />
```

The targetDirectory is where the files will be stored.

MLLPClientEndpoint

The MLLPClientEndpoint is your standard TCP/IP and MLLP connection. This is the most often used protocol combination for HL7 integration. Add the MLLPClientEndpoint as follows:

```
<MLLPClientEndPoint name="MLLPEndPoint" host="localhost" port="4050"
serverCommunicationName="TestServer2" serverEnvironment="Development" />
```

The host and port are where this endpoint needs to connect to. The serverCommunicationName and serverEnvironment are the values you want to use as the server you are connecting to and the environment you are connecting to.

SSLEndPoint

The SSLEndPoint is exactly the same as the MLLPClientEndpoint, with one difference. It will request a SSL connection. Besides that is also supports an optional client side vertificae authentication.

```
<SSLEndPoint name="TestSSLEndPoint" host="localhost" port="4050"
serverCommunicationName="TestServer2" serverEnvironment="Development"
```

```
clientSideCertificatePath="\path\to\cert.pfx"
clientSideCertificatePassword="certificatePW" />
```

As you can see the properties work the same as the MLLPClientEndpoint. The clientSideCertificatePath and clientSideCertificatePassword are optional. If both are filled with the right path and password, client side authentication will be added to the SSL stream.

HttpEndpoint
The HTTP endpoint is an experimental implementation of HL7 messages over HTTP. There is no standard on how to do this. However the people behind HAPI (the original Java implementation of HL7 from which NHapi was ported to .Net) wrote a proposal on this standard. This endpoint implements that proposal.

```
<HttpEndPoint name="TestHttpEndPoint" serverUri="http://yourserver/HL7Service"
serverCommunicationName="TestServer2" serverEnvironment="Development" />
```

The serverUri is the complete url. The serverCommunicationName and serverEnvironment are the values you want to use as the server you are connecting to and the environment you are connecting to.

CustomEndpoint
Of course you are able to implement and add you own protocol as you see fit. There is more on how to do that in paragraph 3.3. The configuration is as follows:

```
<CustomEndpoint name="CustomEndPoint1" type="Yournamespace.Class, Assemblyname" />
```

The type must match the class that inherits the CustomEndpoint class. This class will then be instantiated and used as endpoint.

2.3.2 Routing rules

The routing rules determine if a message is send to an endpoint or not. The endpoints by themselves do nothing, unless a message is routed to them through the rules. The rules are quite plain, but very powerful. The consist of three filters, one for the HL7 message version, one for the HL7 message type and one for a specific field in the message. First you need to add the message routing configuration section.

```
<configSections>
       <section name="messageRouting"
type="HL7Fuse.Hub.Configuration.HL7RoutingRulesConfigurationHandler, HL7Fuse.Hub"/>
</configSections>
```

In the configuration section you can add rules. You can add multiple rules for one endpoint. So if one rule is not enough to comprehend the complete routing logic, other rules can be added. At least one of the rules must be true to route the message to that specific endpoint. Also you can define rules for multiple endpoints. One message can get routed to more than one endpoint.

A rule consist of one or more include parameters and zero or more exclude parameter. The include parameters can be configured as “All” or “Any”. If they are configured as “All”, all the include rules must be true for the message to be routed. In case of “Any” only one of the include rules has to be true.

The exclude rules always act as "Any". The include rules are validated first. If a message is validate as "true" through the include rules and one of the exclude rules filter the message out, the message won't be routed to the endpoint.

To add a rule, you do the following:

```
<rule endpoint="TestFileEndPoint" routeOnValidRules="Any"> <!-- routeOnValidRules is set to
All by default -->
       <include hl7Version="2.3" structurename="SIU_*" />
       <include hl7Version="*" structurename="ADT_1?" fieldFilter="MSH-3-1"
fieldFilterValue="U*" />
       <exclude hl7Version="2.3" structurename="SIU_S12" />
</rule>
```

The routeOnValidRules can be set to "Any" or "All". "All" is the default setting if the parameter is missing.
The hl7Version, structurename and fieldFilterValue parameters can contain the wildcards "?" and "*". "?" is used for one character in that specific spot. "*" allows any number of characters, as long as the characters before the wildcard and after the wildcard are a match.
The hl7Version checks the version of the HL7 message. The structurename validates the type of HL7 message. The fieldFilter and fieldFilterValue work together. The fieldFilter matches a specific field within the HL7 message and matches the content with the fieldFilterValue. The fieldFilter follows the common plain text annotation used to describe specific fields. A sample is shown in the rule configuration above.

2.4 IMessageHandler

There is one application setting that is used for the HL7Fuse.Hub. After a message is accepted and queued for rule validation and routing to endpoints, the Hub checks if a message handler is available. This way you can implement your own message handler that transforms the message or adds some business logic to the Hub if you need any.

```
<add key="HubMessageHandler" value="SomeApplication.Class, Assembly"/>
```

By adding the HubMessageHandler key to the application settings, the specific message handler will be loaded and the message is passed through the handler, before sending it to the relevant endpoint(s). Find more on how to implement a message handler in paragraph 3.2.

2.5 Command line parameters

The HL7Fuse executable has a few command line parameters. These are:

-r Run this application as a console application. Used for starting and stopping from the command line and or in development situations.

-i Will install HL7Fuse as a Windows service. The services can be managed from the Windows services configuration screen.

-u Uninstall the Windows service.

3 Developing your HL7 application

One of the ways you can use HL7Fuse is as a basis for your own HL7 application or integration component (paragraph 3.1). Other ways are to use the Hub and the extensibility of the Hub (paragraph 3.2 and 3.3). Last but not least NHapi and NHapiTools offer some customization.

3.1 Writing your own command assembly

Using HL7Fuse to handle all the TCP/IP and MLLP communication allows you to focus on the specific business logic you have to write. In that case you can write your own command assembly. There are two ways to do this: handle all HL7 messages you need through separate classes or handle all messages through one factory class.

The way HL7Fuse works is that, based on the HandleEachMessageAsEvent configuration key (see paragraph 2.1), it will try to find the right class to handle the message. In case each message is handled as an event HL7Fuse will look for a class named Your.Namespace.V{HL7 version without dots}.{HL7Message name}. If the messages are all send to one factory class, HL7fuse will look for a class named Your.Namespace.V{HL7 version without dots}.MessageFactory. The classes that handle the message(s) have to implement/inherit the `ICommand<MLLPSession, HL7RequestInfo>`. For example:

```
namespace TestApplication.V24
{
    public class MessageFactory : ICommand<MLLPSession, HL7RequestInfo>
    {
        #region Public properties
        public virtual string Name
        {
            get { return "My implementation name"; }
        }
        #endregion

        #region Public methods
        public void ExecuteCommand(MLLPSession session, HL7RequestInfo requestInfo)
        {
            // Do something with the HL7 message
            // requestInfo.Message
        }
        #endregion
    }
}
```

The ExecuteCommand will be executed to handle the message. The HL7RequestInfo object is important here. This will contain the message (as parsed by NHapi). Also you have to use it to let HL7Fuse know if the message was processed correctly. To return an error set the requestInfo.HasError to true and put an error message in requestInfo.ErrorMessage.

3.2 Writing your own IMessageHandler

When you are using the Hub, you have full functionality to route messages from an incoming connection to one or more other systems and/or save the messages as files. If there are relevant endpoints found, using the rules, for a message the message will first be passed through an available Message handler. This handler allows you to manipulate the message or add any handling or logic before the message is send to the relevant endpoints. For example:

```
using HL7Fuse.Hub.Handling;
using NHapi.Base.Model;

namespace MyApplication.MessageHandler
{
    public class MyMessageHandler : IMessageHandler
    {
        public IMessage HandleMessage(IMessage message)
        {
            // add logic for message
            // then message object that is returned
            // will be the one send to the endpoints
            return message;
        }
    }
}
```

After the logic that you want you have to return a message object. That is the message that is send to the relevant endpoints.

3.3 Adding your own EndPoint

Of course there is always a possibility that one of the standard endpoint doesn't do exactly what you want or you need another protocol. By inheriting the CustomEndPoint class you can implement your own endpoint and use this through the Hub's configuration.

```
using System.Xml;
using HL7Fuse.Hub.EndPoints;
using NHapi.Base.Model;

namespace MyApplication.EndPoints
{
    public class MyEndpoint : CustomEndPoint
    {
        public override void Setup(XmlNodeList config)
        {
            // This will allow you to add your
            // own configuration options
            base.Setup(config);
        }

        public override bool Send(IMessage msg)
        {
            // Add your own sending logic here
            return true;
        }
    }
}
```

The Send method returns false if sending failed. True means that everything went well.

3.4 NHapi and NHapiTools

HL7Fuse uses NHapi to parse, terse and encode the HL7 messages. It also includes the NHapiTools, so that you can use this to your advantage. Even if you don't add any message handling, NHapiTools can help you by allowing you to use the default validation rule it provides. HL7Fuse uses the parser including the ConfigurableContext by default. If there isn't any parsing configuration, the default parsing context will be used. If you want to add validation rules, provided by NHapiTools or your own, you can add them by adding the ConfigurableContext configuration.

4 Working with the HL7Fuse solution

The solution is completely available from GitHub:

HL7Fuse: https://github.com/dib0/HL7Fuse

NHapiTools: https://github.com/dib0/NHapiTools

NHapi and SuperSocket are available through Codeplex:

NHapi: https://nhapi.codeplex.com

SuperSocket: https://supersocket.codeplex.com

After opening the solution and building the projects, HL7Fuse should run perfectly. Keep in mind that you need to adapt the configuration to your own needs and system. The project HL7Fuse should be set as start up project. This project has some post-build events, since not all assemblies are copied to the output directory. If a NHapi message isn't parsed correctly or you get some unexpected errors, please check if the copy-action went ok.

The project NHapi also uses "-r" as a commandline parameter to run the executable as console application.

###

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