HL7

HL7 Component

The **HL7** component is used for working with the HL7 MLLP protocol and HL7 v2 messages using the HAPI library.

This component supports the following:

- HL7 MLLP codec for Mina
- HL7 MLLP codec for Netty4 from **Camel 2.15** onwards
- Type Converter from/to HAPI and String
- HL7 DataFormat using the HAPI library
- Even more ease-of-use as it's integrated well with the camel-mina2 component.

Maven users will need to add the following dependency to their `pom.xml` for this component:

xml<dependency> <groupId>org.apache.camel</groupId> <artifactId>camel-hl7</artifactId> <version>x.x.x</version> <!-- use the same version as your Camel core version --> </dependency>

HL7 MLLP protocol

HL7 is often used with the HL7 MLLP protocol, which is a text based TCP socket based protocol. This component ships with a Mina and Netty4 Codec that conforms to the MLLP protocol so you can easily expose an HL7 listener accepting HL7 requests over the TCP transport layer. To expose a HL7 listener service, the camel-mina2 or camel-netty4 component is used with the `HL7MLLPCodec` (mina2) or `HL7MLLPNettyDecoder/HL7MLLPNettyEncoder` (Netty4).

HL7 MLLP codec can be configured as follows:

confluenceTableSmall

Name	Default Value	Description
`startByte`	`0x0b`	The start byte spanning the HL7 payload.
`endByte1`	`0x1c`	The first end byte spanning the HL7 payload.
`endByte2`	`0x0d`	The 2nd end byte spanning the HL7 payload.
`charset`	JVM Default	The encoding (a charset name) to use for the codec. If not provided, Camel will use the JVM default Charset.
`produceString`	`true`	**(as of Camel 2.14.1)** If true, the codec creates a string using the defined charset. If false, the codec sends a plain byte array into the route, so that the HL7 Data Format can determine the actual charset from the HL7 message content.
`convertLFtoCR`	`false`	Will convert `\n` to `\r` (`0x0d`, 13 decimal) as HL7 stipulates `\r` as segment terminators. The HAPI library requires the use of `\r`.

Exposing an HL7 listener using Mina

In the Spring XML file, we configure a mina2 endpoint to listen for HL7 requests using TCP on port `8888`:

xml <endpoint id="hl7MinaListener" uri="mina2:tcp://localhost:8888?sync=true&codec=#hl7codec"/>

sync=true indicates that this listener is synchronous and therefore will return a HL7 response to the caller. The HL7 codec is setup with **codec=#hl7codec** . Note that `hl7codec` is just a Spring bean ID, so it could be named `mygreatcodecforhl7` or whatever. The codec is also set up in the Spring XML file:

xml <bean id="hl7codec" class="org.apache.camel.component.hl7.HL7MLLPCodec"> <property name="charset" value="iso-8859-1"/> </bean>

The endpoint **hl7MinaLlistener** can then be used in a route as a consumer, as this Java DSL example illustrates:

java from("hl7MinaListener").beanRef("patientLookupService");

This is a very simple route that will listen for HL7 and route it to a service named **patientLookupService**. This is also Spring bean ID, configured in the Spring XML as:

xml <bean id="patientLookupService" class="com.mycompany.healthcare.service.PatientLookupService"/>

The business logic can be implemented in POJO classes that do not depend on Camel, as shown here:

javaimport ca.uhn.hl7v2.HL7Exception; import ca.uhn.hl7v2.model.Message; import ca.uhn.hl7v2.model.v24.segment.QRD; public class PatientLookupService { public Message lookupPatient(Message input) throws HL7Exception { QRD qrd = (QRD)input.get("QRD"); String patientId = qrd.getWhoSubjectFilter(0).getIDNumber().getValue(); // find patient data based on the patient id and create a HL7 model object with the response Message response = ... create and set response data return response }

Exposing an HL7 listener using Netty (available from Camel 2.15 onwards)

In the Spring XML file, we configure a netty4 endpoint to listen for HL7 requests using TCP on port `8888`:

xml <endpoint id="hl7NettyListener" uri="netty4:tcp://localhost:8888?sync=true&encoder=#hl7encoder&decoder=#hl7decoder"/>

sync=true indicates that this listener is synchronous and therefore will return a HL7 response to the caller. The HL7 codec is setup with **encoder=#hl7encoder** and **decoder=#hl7decoder**. Note that `hl7encoder` and `hl7decoder` are just bean IDs, so they could be named differently. The beans can be set in the Spring XML file:

xml <bean id="hl7decoder" class="org.apache.camel.component.hl7.HL7MLLPNettyDecoderFactory"/> <bean id="hl7encoder" class="org.apache.camel.component.hl7.HL7MLLPNettyEncoderFactory"/>

The endpoint **hl7NettyListener** can then be used in a route as a consumer, as this Java DSL example illustrates:

java from("hl7NettyListener").beanRef("patientLookupService");

HL7 Model using java.lang.String or byte[]

The HL7 MLLP codec uses plain String as its data format. Camel uses its Type Converter to convert to/from strings to the HAPI HL7 model objects, but you can use the plain String objects if you prefer, for instance if you wish to parse the data yourself.

As of Camel 2.14.1 you can also let both the Mina and Netty codecs use a plain `byte[]` as its data format by setting the `produceString` property to false. The Type Converter is also capable of converting the `byte[]` to/from HAPI HL7 model objects.

HL7v2 Model using HAPI

The HL7v2 model uses Java objects from the HAPI library. Using this library, you can encode and decode from the EDI format (ER7) that is mostly used with HL7v2.

The sample below is a request to lookup a patient with the patient ID `0101701234`.

MSH|^~\\&|MYSENDER|MYRECEIVER|MYAPPLICATION||200612211200||QRY^A19|1234|P|2.4
QRD|200612211200|R|I|GetPatient|||1^RD|0101701234|DEM||

Using the HL7 model you can work with a `ca.uhn.hl7v2.model.Message` object, e.g. to retrieve a patient ID:

javaMessage msg = exchange.getIn().getBody(Message.class); QRD qrd = (QRD)msg.get("QRD"); String patientId = qrd.getWhoSubjectFilter(0).getIDNumber().getValue(); // 0101701234

This is powerful when combined with the HL7 listener, because you don't have to work with `byte[]`, `String` or any other simple object formats. You can just use the HAPI HL7v2 model objects. If you know the message type in advance, you can be more type-safe:

javaQRY_A19 msg = exchange.getIn().getBody(QRY_A19.class); String patientId = msg.getQRD().getWhoSubjectFilter(0).getIDNumber().getValue();

HL7 DataFormat

Message Headers

The unmarshal operation adds these fields from the MSH segment as headers on the Camel message:

confluenceTableSmall

Key	MSH field	Example
`CamelHL7SendingApplication`	`MSH-3`	`MYSERVER`
`CamelHL7SendingFacility`	`MSH-4`	`MYSERVERAPP`
`CamelHL7ReceivingApplication`	`MSH-5`	`MYCLIENT`
`CamelHL7ReceivingFacility`	`MSH-6`	`MYCLIENTAPP`
`CamelHL7Timestamp`	`MSH-7`	`20071231235900`
`CamelHL7Security`	`MSH-8`	`null`
`CamelHL7MessageType`	`MSH-9-1`	`ADT`

`CamelHL7TriggerEvent`	`MSH-9-2`	`A01`
`CamelHL7MessageControl`	`MSH-10`	`1234`
`CamelHL7ProcessingId`	`MSH-11`	`P`
`CamelHL7VersionId`	`MSH-12`	`2.4`
`CamelHL7Context`	`-`	**(Camel 2.14)** contains the HapiContext that was used to parse the message
`CamelHL7Charset`	`MSH-18`	**(Camel 2.14.1)** `UNICODE UTF-8`

All headers except `CamelHL7Context` are `String` types. If a header value is missing, its value is `null`.

Options

The HL7 Data Format supports the following options:

confluenceTableSmall

Option	Default	Description
`validate`	true	Whether the HAPI Parser should validate the message using the default validation rules. It is recommended to use the `parser` or `hapiContext` option and initialize it with the desired HAPI `ValidationContext`
`parser`	`ca.uhn.hl7v2.parser.GenericParser`	Custom parser to be used. Must be of type `ca.uhn.hl7v2.parser.Parser`. Note that `GenericParser` also allows to parse XML-encoded HL7v2 messages
`hapiContext`	`ca.uhn.hl7v2.DefaultHapiContext`	**Camel 2.14:** Custom HAPI context that can define a custom parser, custom ValidationContext etc. This gives you full control over the HL7 parsing and rendering process.

Dependencies

To use HL7 in your Camel routes you'll need to add a dependency on **camel-hl7** listed above, which implements this data format.

The HAPI library is split into a base library and several structure libraries, one for each HL7v2 message version:

- v2.1 structures library
- v2.2 structures library
- v2.3 structures library
- v2.3.1 structures library
- v2.4 structures library
- v2.5 structures library
- v2.5.1 structures library
- v2.6 structures library

By default `camel-hl7` only references the HAPI base library. Applications are responsible for including structure libraries themselves. For example, if an application works with HL7v2 message versions 2.4 and 2.5 then the following dependencies must be added:

xml<dependency> <groupId>ca.uhn.hapi</groupId> <artifactId>hapi-structures-v24</artifactId> <version>2.2</version> <!-- use the same version as your hapi-base version --> </dependency> <dependency> <groupId>ca.uhn.hapi</groupId> <artifactId>hapi-structures-v25</artifactId> <version>2.2</version> <!-- use the same version as your hapi-base version --> </dependency>

Alternatively, an OSGi bundle containing the base library, all structures libraries and required dependencies (on the bundle classpath) can be downloaded from the central Maven repository.

xml<dependency> <groupId>ca.uhn.hapi</groupId> <artifactId>hapi-osgi-base</artifactId> <version>2.2</version> </dependency>

Terser language

HAPI provides a Terser class that provides access to fields using a commonly used terse location specification syntax. The Terser language allows to use this syntax to extract values from messages and to use them as expressions and predicates for filtering, content-based routing etc.

Sample:

javaimport static org.apache.camel.component.hl7.HL7.terser; ... // extract patient ID from field QRD-8 in the QRY_A19 message above and put into message header from("direct:test1") .setHeader("PATIENT_ID",terser("QRD-8(0)-1")) .to("mock:test1"); // continue processing if extracted field equals a message header from("direct:test2") .filter(terser("QRD-8(0)-1").isEqualTo(header("PATIENT_ID")) .to("mock:test2");

HL7 Validation predicate

Often it is preferable to first parse a HL7v2 message and in a separate step validate it against a HAPI ValidationContext.

Sample:

javaimport static org.apache.camel.component.hl7.HL7.messageConformsTo; import ca.uhn.hl7v2.validation.impl.DefaultValidation; ... // Use standard or define your own validation rules ValidationContext defaultContext = new DefaultValidation(); // Throws PredicateValidationException if message does not validate from("direct:test1") .validate(messageConformsTo(defaultContext)) .to("mock:test1");

HL7 Validation predicate using the HapiContext (Camel 2.14)

The HAPI Context is always configured with a ValidationContext (or a ValidationRuleBuilder), so you can access the validation rules indirectly. Furthermore, when unmarshalling the HL7DataFormat forwards the configured HAPI context in the `CamelHL7Context` header, and the validation rules of this context can be easily reused:

javaimport static org.apache.camel.component.hl7.HL7.messageConformsTo; import static org.apache.camel.component.hl7.HL7.messageConforms ... HapiContext hapiContext = new DefaultHapiContext(); hapiContext.getParserConfiguration().setValidating(false); // don't validate during parsing // customize HapiContext some more ... e.g. enforce that PID-8 in ADT_A01 messages of version 2.4 is not empty ValidationRuleBuilder builder = new ValidationRuleBuilder() { @Override protected void configure() { forVersion(Version.V24) .message("ADT", "A01") .terser("PID-8", not(empty())); } }; hapiContext.setValidationRuleBuilder(builder); HL7DataFormat hl7 = new HL7DataFormat(); hl7.setHapiContext(hapiContext); from("direct:test1") . unmarshal(hl7) // uses the GenericParser returned from the HapiContext .validate(messageConforms()) // uses the validation rules returned from the HapiContext // equivalent with .validate(messageConformsTo(hapiContext)) // route continues from here

HL7 Acknowledgement expression

A common task in HL7v2 processing is to generate an acknowledgement message as response to an incoming HL7v2 message, e.g. based on a validation result. The `ack` expression lets us accomplish this very elegantly:

javaimport static org.apache.camel.component.hl7.HL7.messageConformsTo; import static org.apache.camel.component.hl7.HL7.ack; import ca.uhn.hl7v2. validation.impl.DefaultValidation; ... // Use standard or define your own validation rules ValidationContext defaultContext = new DefaultValidation(); from ("direct:test1") .onException(Exception.class) .handled(true) .transform(ack()) // auto-generates negative ack because of exception in Exchange .end() . validate(messageConformsTo(defaultContext)) // do something meaningful here ... // acknowledgement .transform(ack())

More Samples

In the following example, a plain `String` HL7 request is sent to an HL7 listener that sends back a response:

{snippet:id=e2|lang=java|url=camel/trunk/components/camel-hl7/src/test/java/org/apache/camel/component/hl7/HL7MLLPCodecTest.java}

In the next sample, HL7 requests from the HL7 listener are routed to the business logic, which is implemented as plain POJO registered in the registry as h `l7service`.

{snippet:id=e2|lang=java|url=camel/trunk/components/camel-hl7/src/test/java/org/apache/camel/component/hl7/HL7RouteTest.java}

Then the Camel routes using the `RouteBuilder` may look as follows:

{snippet:id=e1|lang=java|url=camel/trunk/components/camel-hl7/src/test/java/org/apache/camel/component/hl7/HL7RouteTest.java}

Note that by using the HL7 DataFormat the Camel message headers are populated with the fields from the MSH segment. The headers are particularly useful for filtering or content-based routing as shown in the example above.

Endpoint See Also