Integration Community of Practice: Camel & CXF Message encoding

Posted by Michael Thirion May 10, 2016

For BankAudi we've been developing SOAP-based Camel & CXF components to integrate applications located in different countries.
Therefore those components had to support several character sets, like UTF-8, UTF-16, ISO-8859-9...
It's not always easy to have all that working on the first attempt as there are several places where character conversion can take place so here's some hints that should help you find where an error can occur.

Type conversion error

First of all, a conversion error will most probably raise an org.apache.camel.TypeConversionException (due to an org.xml.sax.SAXParseException if you're using XML) such as below:
Error during type conversion from type: java.lang.String to the required type: org.w3c.dom.Document with value [Body is instance of org.apache.camel.StreamCache] due org.xml.sax.SAXParseException: Content is not allowed in prolog.. Exchange[Message: [Body is instance of org.apache.camel.StreamCache]]. Caused by: [org.apache.camel.TypeConversionException

This has nothing to do with the type of the message body (here StreamCache but could also be String). The error is really from "Content is not allowed in prolog".

Camel header

There are a lot of variables in the Camel Message header that relate to encoding, particularly when using CXF, but most of them are only for information and filled directly from the HTTP header of the SOAP request/response.
For example:
- accept-encoding, connection, Content-Type, Host, User-Agent... are set from the HTTP header
- CamelHttpCharacterEncoding, CamelHttpMethod, CamelHttpPath, CamelHttpUri are also set from the HTTP header, but are CXF-specific
- CamelCxfMessage is the representation of the CXF message as it was processed by CXF, saved in the Camel header
- ResponseContext is a CamelCxfMessage that represent the response and which is only present after an interaction with a remote CXF endpoint.
...

Camel & CXF encoding

When the Camel route receives a request from a CXF endpoint, it actually receives an Input Stream that it places in the Camel Message body as a String. The reverse occurs when a message is to be sent to a remote CXF endpoint.
The conversion, from byte[] to String accepts a charset as a parameter, which is UTF-8 by default.
This charset can be set via the Exchange.CHARSET_NAME exchange property, mapped to the CamelCharsetName variable. This variable is first initialized based on the HTTP content-type that indicates the way the incoming stream is encoded.

```
<setProperty propertyName="CamelCharsetName">
   <constant>iso-8859-1</constant>
</setProperty>
```

The "convertBodyTo" method can be used to convert the message body to an object type, provided that there is a type converter for it.
It accepts a "charset" attribute and so can also be used to convert the body from a String of one encoding to a String of another encoding.
In such a case the principle is the same, and the charset attribute of the method will override the CamelCharsetName of the Exchange

NB: The conversion method actually replaces the CamelCharsetName temporarily and rely on the implicit conversion by creating a new message with the content of the body.
Be careful that the method REMOVES the CamelCharsetName variable from the header (instead of placing back the old value).

If you have some xpath expression in your route, then an implicit conversion will occur to turn the Message body from String to Document (org.w3c.dom.Document).
In such a case, the XML parser comes into play, and it will read the body content based on the XML encoding declaration (<?xml version="1.0" encoding="utf-8"?>).
When forced with the convertBodyTo method, the charset attribute of the method will be ignored.

Examples

Let's use the following route, which simply receives a soap message and print it before and after a conversion from String to String.

```
<route id="main" streamCache="true">
    <from uri="cxf:bean:{{myEndpoint}}" />

    <log message="Payload : ${in.body}" />
    <convertBodyTo type="String" />
    <log message="Converted Payload : ${in.body}" />
</route>
```

a) Let's first add charset="utf-8" in the convertBodyTo method and send over an XML message in UTF-16.

In the logs you'll see extra characters after the conversion due to the fact that an an UTF-16 message is being converted into UTF-8

Payload : <?xml version="1.0" encoding="utf-8"?>n | <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" ...
====================
Converted Payload : <?xml version="1.0" encoding="utf-8"?>n | <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" ...

This would lead to an error if this message is to be converted into an XML document.
This can also lead to errors later on if the CamelCharsetName property is not set to UTF-16 (or not set at all ! -> or removed by a previous convertBodyTo).
Changing the XML declaration within the message will here has no effect at all.

b) Let's now replace the <convertBodyTo type="String" /> line by <convertBodyTo type="org.w3c.dom.Document" />
and send an UTF-8 encoded message with also an UTF-8 encoding in its XML declaration.
In the logs you'll see that the XML declaration is processed by the parser and is not longer present in the converted message:
Payload : <?xml version="1.0" encoding="utf-8"?>n | <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" ...
====================
Converted Payload : <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" ...

c)
If we now send the same XML message in UTF-16 over the wire, we'll have a type conversion error.
What will happen is that the XML message, encoded in UTF-16, will be read in UTF-8 by the XML parser (due to the heading line) and the parser will detect the extra characters at the beginning of the file.

We'll have the same error if we send the message in UTF-8 but change the XML heading declaration to UTF-16.
The only way it can work is if both are set to UTF-16 (the response will be in UTF-16 also so be sure you can visualize this encoding).
Changing the charset attribute of the convertBodyTo method will here not be of any help.

Unit testing with different encoding

When performing unit tests, as the message is injected into the route without any HTTP Client, the content-type and content-encoding variables should also be set accordingly.
Also the file has to be read from its byte array with the proper encoding.
Here's an example for an UTF-16 encoded XML file. Of course this file should have a proper XML encoding declaration.

```
public class BlueprintUnitTest extends CamelBlueprintTestSupport {
  ...

  @Test
  public void doTest() throws InterruptedException {
    ...
    String encoding="UTF-16"; // example
    Exchange e = new DefaultExchange(this.context());
    Message m = e.getIn();
    m.setBody(readFromFile(BlueprintUnitTest.class.getResourceAsStream("my_filename"), encoding ) );
    e.setIn(m);
    e.setProperty(Exchange.CHARSET_NAME, encoding.toUpperCase());
    e.setProperty(Exchange.CONTENT_TYPE, "text/xml;charset="+encoding.toLowerCase());
    e.setProperty(Exchange.CONTENT_ENCODING, encoding.toLowerCase());

   template.send("from_endpoint",e);
   ...
  }

 // example reading the file char by char with the proper byte to char encoding
  public static String readFromFile(InputStream is, String cd) {
      ...
      StringBuilder sb = new StringBuilder(512);
      try {
          Reader r = new InputStreamReader(is, cd);
          int c = 0;
          while ((c = r.read()) != -1) {
              sb.append((char) c);
          }
      } catch (IOException e) {
          throw new RuntimeException(e);
      }
      return sb.toString();
   }
}
```

Thanks Mourad for the last piece of code !
15 Views Tags: cxf, encoding, utf-8, came

There are no comments on this post