Netty HTTP

Netty HTTP Component

Available as of Camel 2.12

The **netty-http** component is an extension to Netty component to facilitiate HTTP transport with Netty.

This camel component supports both producer and consumer endpoints.

This component is deprecated. You should use Netty4 HTTP.
Stream

Netty is stream based, which means the input it receives is submitted to Camel as a stream. That means you will only be able to read the content of the stream **once**.
If you find a situation where the message body appears to be empty or you need to access the data multiple times (eg: doing multicasting, or redelivery error handling)
you should use Stream caching or convert the message body to a `String` which is safe to be re-read multiple times.

Notice Netty4 HTTP reads the entire stream into memory using `io.netty.handler.codec.http.HttpObjectAggregator` to build the entire full http message. But the resulting message is still a stream based message which is readable once.

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

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

URI format

The URI scheme for a netty component is as follows

```
netty-http:http://localhost:8080[?options]
```

You can append query options to the URI in the following format, `?option=value&option=value&...`

Query parameters vs endpoint options

You may be wondering how Camel recognizes URI query parameters and endpoint options. For example you might create endpoint URI as follows - `netty-http:http//example.com?myParam=myValue&compression=true` . In this example `myParam` is the HTTP parameter, while `compression` is the Camel endpoint option. The strategy used by Camel in such situations is to resolve available endpoint options and remove them from the URI. It means that for the discussed example, the HTTP request sent by Netty HTTP producer to the endpoint will look as follows - `http//example.com?myParam=myValue` , because `compression` endpoint option will be resolved and removed from the target URL.

Keep also in mind that you cannot specify endpoint options using dynamic headers (like `CamelHttpQuery`). Endpoint options can be specified only at the endpoint URI definition level (like `to` or `from` DSL elements).

HTTP Options

A lot more options

Important: This component inherits all the options from Netty. So make sure to look at the Netty documentation as well.
Notice that some options from Netty is not applicable when using this Netty HTTP component, such as options related to UDP transport.

Name	Default Value	Description
`chunkedMaxContentLength`	`1mb`	Value in bytes the max content length per chunked frame received on the Netty HTTP server.
`compression`	`false`	Allow using gzip/deflate for compression on the Netty HTTP server if the client supports it from the HTTP headers.
`headerFilterStrategy`		To use a custom `org.apache.camel.spi.HeaderFilterStrategy` to filter headers.

`httpMethodRestrict`		To disable HTTP methods on the Netty HTTP consumer. You can specify multiple separated by comma.
`mapHeaders`	`true`	If this option is enabled, then during binding from Netty to Camel Message then the headers will be mapped as well (eg added as header to the Camel Message as well). You can turn off this option to disable this. The headers can still be accessed from the `org.apache.camel.component.netty.http.NettyHttpMessage` message with the method `getHttpRequest()` that returns the Netty HTTP request `org.jboss.netty.handler.codec.http.HttpRequest` instance.
`matchOnUriPrefix`	`false`	Whether or not Camel should try to find a target consumer by matching the URI prefix if no exact match is found. See further below for more details.
`nettyHttpBinding`		To use a custom `org.apache.camel.component.netty.http.NettyHttpBinding` for binding to/from Netty and Camel Message API.
`bridgeEndpoint`	`false`	If the option is `true`, the producer will ignore the `Exchange.HTTP_URI` header, and use the endpoint's URI for request. You may also set the `throwExceptionOnFailure` to be `false` to let the producer send all the fault response back. The consumer working in the bridge mode will skip the gzip compression and WWW URL form encoding (by adding the `Exchange.SKIP_GZIP_ENCODING` and `Exchange.SKIP_WWW_FORM_URLENCODED` headers to the consumed exchange).
`throwExceptionOnFailure`	`true`	Option to disable throwing the `HttpOperationFailedException` in case of failed responses from the remote server. This allows you to get all responses regardles of the HTTP status code.
`traceEnabled`	`false`	Specifies whether to enable HTTP TRACE for this Netty HTTP consumer. By default TRACE is turned off.
`transferException`	`false`	If enabled and an Exchange failed processing on the consumer side, and if the caused Exception was send back serialized in the response as a `application/x-java-serialized-object` content type. On the producer side the exception will be deserialized and thrown as is, instead of the `HttpOperationFailedException`. The caused exception is required to be serialized.
`urlDecodeHeaders`		If this option is enabled, then during binding from Netty to Camel Message then the header values will be URL decoded (eg %20 will be a space character. Notice this option is used by the default `org.apache.camel.component.netty.http.NettyHttpBinding` and therefore if you implement a custom `org.apache.camel.component.netty.http.NettyHttpBinding` then you would need to decode the headers accordingly to this option. **Notice:** This option is default `true` for Camel 2.12.x, and default `false` from Camel 2.13 onwards.
`nettySharedHttpServer`	`null`	To use a shared Netty HTTP server. See Netty HTTP Server Example for more details.
`disableStreamCache`	`false`	Determines whether or not the raw input stream from Netty `HttpRequest#getContent()` is cached or not (Camel will read the stream into a in light-weight memory based Stream caching) cache. By default Camel will cache the Netty input stream to support reading it multiple times to ensure it Camel can retrieve all data from the stream. However you can set this option to `true` when you for example need to access the raw stream, such as streaming it directly to a file or other persistent store. Mind that if you enable this option, then you cannot read the Netty stream multiple times out of the box, and you would need manually to reset the reader index on the Netty raw stream. Notice Netty4 HTTP reads the entire stream into memory using `io.netty.handler.codec.http.HttpObjectAggregator` to build the entire full http message. But the resulting message is still a stream based message which is readable once.
`securityConfiguration`	`null`	**Consumer only**. Refers to a `org.apache.camel.component.netty.http.NettyHttpSecurityConfiguration` for configuring secure web resources.
`send503whenSuspended`	`true`	**Consumer only**. Whether to send back HTTP status code 503 when the consumer has been suspended. If the option is `false` then the Netty Acceptor is unbound when the consumer is suspended, so clients cannot connect anymore.
`maxHeaderSize`	`8192`	**Camel 2.15.3: Consumer only.** The maximum length of all headers. If the sum of the length of each header exceeds this value, a TooLongFrameException will be raised.
`okStatusCodeRange`	`200-299`	**Camel 2.16:** The status codes which is considered a success response. The values are inclusive. The range must be defined as from-to with the dash included.
useRelativePath	false	**Camel 2.16: Producer only:** Whether to use a path (/myapp) in the request line or an absolute URI (http://0.0.0.0:8080/myapp), which is default.

The `NettyHttpSecurityConfiguration` has the following options:

Name	Default Value	Description
`authenticate`	`true`	Whether authentication is enabled. Can be used to quickly turn this off.
`constraint`	`Basic`	The constraint supported. Currently only `Basic` is implemented and supported.
`realm`	`null`	The name of the JAAS security realm. This option is mandatory.
`securityConstraint`	`null`	Allows to plugin a security constraint mapper where you can define ACL to web resources.
`securityAuthenticator`	`null`	Allows to plugin a authenticator that performs the authentication. If none has been configured then the `org.apache.camel.component.netty.http.JAASSecurityAuthenticator` is used by default.
`loginDeniedLoggingLevel`	`DEBUG`	Logging level used when a login attempt failed, which allows to see more details why the login failed.
`roleClassName`	`null`	To specify FQN class names of `Principal` implementations that contains user roles. If none has been specified, then the Netty HTTP component will by default assume a `Principal` is role based if its FQN classname has the lower-case word `role` in its classname. You can specify multiple class names separated by comma.

Message Headers

The following headers can be used on the producer to control the HTTP request.

Name	Type	Description
`CamelHttpMethod`	`String`	Allow to control what HTTP method to use such as GET, POST, TRACE etc. The type can also be a `org.jboss.netty.handler.codec.http.HttpMethod` instance.
`CamelHttpQuery`	`String`	Allows to provide URI query parameters as a `String` value that overrides the endpoint configuration. Separate multiple parameters using the & sign. For example: `foo=bar&beer=yes`.
`CamelHttpPath`	`String`	**Camel 2.13.1/2.12.4:** Allows to provide URI context-path and query parameters as a `String` value that overrides the endpoint configuration. This allows to reuse the same producer for calling same remote http server, but using a dynamic context-path and query parameters.
`Content-Type`	`String`	To set the content-type of the HTTP body. For example: `text/plain; charset="UTF-8"`.
`CamelHttpResponseCode`	`int`	Allows to set the HTTP Status code to use. By default 200 is used for success, and 500 for failure.

The following headers is provided as meta-data when a route starts from an Netty HTTP endpoint:

The description in the table takes offset in a route having: `from("netty-http:http:0.0.0.0:8080/myapp")...`

Name	Type	Description
`CamelHttpMethod`	`String`	The HTTP method used, such as GET, POST, TRACE etc.
`CamelHttpUrl`	`String`	The URL including protocol, host and port, etc: `http://0.0.0.0:8080/myapp`
`CamelHttpUri`	`String`	The URI without protocol, host and port, etc: `/myapp`
`CamelHttpQuery`	`String`	Any query parameters, such as `foo=bar&beer=yes`

`CamelHttpRawQuery`	`String`	**Camel 2.13.0**: Any query parameters, such as `foo=bar&beer=yes`. Stored in the raw form, as they arrived to the consumer (i.e. before URL decoding).
`CamelHttpPath`	`String`	Additional context-path. This value is empty if the client called the context-path `/myapp`. If the client calls `/myapp /mystuff`, then this header value is `/mystuff`. In other words its the value after the context-path configured on the route endpoint.
`CamelHttpCharacterEncoding`	`String`	The charset from the content-type header.
`CamelHttpAuthentication`	`String`	If the user was authenticated using HTTP Basic then this header is added with the value `Basic`.
`Content-Type`	`String`	The content type if provided. For example: `text/plain; charset="UTF-8"`.

Access to Netty types

This component uses the `org.apache.camel.component.netty.http.NettyHttpMessage` as the message implementation on the Exchange. This allows end users to get access to the original Netty request/response instances if needed, as shown below. Mind that the original response may not be accessible at all times.

```
org.jboss.netty.handler.codec.http.HttpRequest request = exchange.getIn(NettyHttpMessage.class).
getHttpRequest();
```

Examples

In the route below we use Netty HTTP as a HTTP server, which returns back a hardcoded "Bye World" message.

```
    from("netty-http:http://0.0.0.0:8080/foo")
      .transform().constant("Bye World");
```

And we can call this HTTP server using Camel also, with the ProducerTemplate as shown below:

```
    String out = template.requestBody("netty-http:http://localhost:8080/foo", "Hello World", String.class);
    System.out.println(out);
```

And we get back "Bye World" as the output.

How do I let Netty match wildcards

By default Netty HTTP will only match on exact uri's. But you can instruct Netty to match prefixes. For example

```
from("netty-http:http://0.0.0.0:8123/foo").to("mock:foo");
```

In the route above Netty HTTP will only match if the uri is an exact match, so it will match if you enter `http://0.0.0.0:8123/foo` but not match if you do `http://0.0.0.0:8123/foo/bar`.

So if you want to enable wildcard matching you do as follows:

```
from("netty-http:http://0.0.0.0:8123/foo?matchOnUriPrefix=true").to("mock:foo");
```

So now Netty matches any endpoints with starts with `foo`.

To match **any** endpoint you can do:

```
from("netty-http:http://0.0.0.0:8123?matchOnUriPrefix=true").to("mock:foo");
```

Using multiple routes with same port

In the same CamelContext you can have multiple routes from Netty HTTP that shares the same port (eg a `org.jboss.netty.bootstrap.ServerBootstrap` instance). Doing this requires a number of bootstrap options to be identical in the routes, as the routes will share the same `org.jboss.netty.bootstrap.ServerBootstrap` instance. The instance will be configured with the options from the first route created.

The options the routes must be identical configured is all the options defined in the `org.apache.camel.component.netty.NettyServerBootstrapConfiguration` configuration class. If you have configured another route with different options, Camel will throw an exception on startup, indicating the options is not identical. To mitigate this ensure all options is identical.

Here is an example with two routes that share the same port.

Two routes sharing the same port

```
from("netty-http:http://0.0.0.0:{{port}}/foo")
  .to("mock:foo")
  .transform().constant("Bye World");

from("netty-http:http://0.0.0.0:{{port}}/bar")
  .to("mock:bar")
  .transform().constant("Bye Camel");
```

And here is an example of a mis configured 2nd route that do not have identical `org.apache.camel.component.netty.NettyServerBootstrapConfiguration` option as the 1st route. This will cause Camel to fail on startup.

Two routes sharing the same port, but the 2nd route is misconfigured and will fail on starting

```
from("netty-http:http://0.0.0.0:{{port}}/foo")
  .to("mock:foo")
  .transform().constant("Bye World");

// we cannot have a 2nd route on same port with SSL enabled, when the 1st route is NOT
from("netty-http:http://0.0.0.0:{{port}}/bar?ssl=true")
  .to("mock:bar")
  .transform().constant("Bye Camel");
```

Reusing same server bootstrap configuration with multiple routes

By configuring the common server bootstrap option in an single instance of a `org.apache.camel.component.netty.NettyServerBootstrapConfiguration` type, we can use the `bootstrapConfiguration` option on the Netty HTTP consumers to refer and reuse the same options across all consumers.

```
<bean id="nettyHttpBootstrapOptions" class="org.apache.camel.component.netty.NettyServerBootstrapConfiguration">
  <property name="backlog" value="200"/>
  <property name="connectTimeout" value="20000"/>
  <property name="workerCount" value="16"/>
</bean>
```

And in the routes you refer to this option as shown below

```
<route>
  <from uri="netty-http:http://0.0.0.0:{{port}}/foo?bootstrapConfiguration=#nettyHttpBootstrapOptions"/>
  ...
</route>

<route>
  <from uri="netty-http:http://0.0.0.0:{{port}}/bar?bootstrapConfiguration=#nettyHttpBootstrapOptions"/>
  ...
</route>

<route>
  <from uri="netty-http:http://0.0.0.0:{{port}}/beer?bootstrapConfiguration=#nettyHttpBootstrapOptions"/>
  ...
</route>
```

Reusing same server bootstrap configuration with multiple routes across multiple bundles in OSGi container

See the Netty HTTP Server Example for more details and example how to do that.

Using HTTP Basic Authentication

The Netty HTTP consumer supports HTTP basic authentication by specifying the security realm name to use, as shown below

```
<route>
   <from uri="netty-http:http://0.0.0.0:{{port}}/foo?securityConfiguration.realm=karaf"/>
   ...
</route>
```

The realm name is mandatory to enable basic authentication. By default the JAAS based authenticator is used, which will use the realm name specified (karaf in the example above) and use the JAAS realm and the JAAS {{LoginModule}}s of this realm for authentication.

End user of Apache Karaf / ServiceMix has a karaf realm out of the box, and hence why the example above would work out of the box in these containers.

Specifying ACL on web resources

The `org.apache.camel.component.netty.http.SecurityConstraint` allows to define constrains on web resources. And the `org.apache.camel.component.netty.http.SecurityConstraintMapping` is provided out of the box, allowing to easily define inclusions and exclusions with roles.

For example as shown below in the XML DSL, we define the constraint bean:

```
  <bean id="constraint" class="org.apache.camel.component.netty.http.SecurityConstraintMapping">
    <!-- inclusions defines url -> roles restrictions -->
    <!-- a * should be used for any role accepted (or even no roles) -->
    <property name="inclusions">
      <map>
        <entry key="/*" value="*"/>
        <entry key="/admin/*" value="admin"/>
        <entry key="/guest/*" value="admin,guest"/>
      </map>
    </property>
    <!-- exclusions is used to define public urls, which requires no authentication -->
    <property name="exclusions">
      <set>
        <value>/public/*</value>
      </set>
    </property>
  </bean>
```

The constraint above is define so that

- access to /* is restricted and any roles is accepted (also if user has no roles)
- access to /admin/* requires the admin role
- access to /guest/* requires the admin or guest role
- access to /public/* is an exclusion which means no authentication is needed, and is therefore public for everyone without logging in

To use this constraint we just need to refer to the bean id as shown below:

```
<route>
   <from uri="netty-http:http://0.0.0.0:{{port}}/foo?matchOnUriPrefix=true&amp;securityConfiguration.
realm=karaf&amp;securityConfiguration.securityConstraint=#constraint"/>
   ...
</route>
```

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

- Netty
- Netty HTTP Server Example
- Jetty