Netty4

Netty Component

Available as of Camel 2.14

The **netty4** component in Camel is a socket communication component, based on the Netty project version 4.
Netty is a NIO client server framework which enables quick and easy development of network applications such as protocol servers and clients.
Netty greatly simplifies and streamlines network programming such as TCP and UDP socket server.

This camel component supports both producer and consumer endpoints.

The Netty component has several options and allows fine-grained control of a number of TCP/UDP communication parameters (buffer sizes, keepAlives, tcpNoDelay etc) and facilitates both In-Only and In-Out communication on a Camel route.

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

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-netty4</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

```
netty4:tcp://localhost:99999[?options]
netty4:udp://remotehost:99999/[?options]
```

This component supports producer and consumer endpoints for both TCP and UDP.

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

Options

Name	Default Value	Description
`keepAlive`	`true`	Setting to ensure socket is not closed due to inactivity
`tcpNoDelay`	`true`	Setting to improve TCP protocol performance
`backlog`		Allows to configure a backlog for netty consumer (server). Note the backlog is just a best effort depending on the OS. Setting this option to a value such as `200`, `500` or `1000`, tells the TCP stack how long the "accept" queue can be. If this option is not configured, then the backlog depends on OS setting.
`broadcast`	`false`	Setting to choose Multicast over UDP
`connectTimeout`	`10000`	Time to wait for a socket connection to be available. Value is in millis.
`reuseAddress`	`true`	Setting to facilitate socket multiplexing
`sync`	`true`	Setting to set endpoint as one-way or request-response
`synchronous`	`false`	By default, the Asynchronous Routing Engine is used. Set to `true` to force processing synchronously.
`ssl`	`false`	Setting to specify whether SSL encryption is applied to this endpoint
`sslClientCertHeaders`	`false`	When enabled and in SSL mode, then the Netty consumer will enrich the Camel Message with headers having information about the client certificate such as subject name, issuer name, serial number, and the valid date range.
`sendBufferSize`	`65536 bytes`	The TCP/UDP buffer sizes to be used during outbound communication. Size is bytes.
`receiveBufferSize`	`65536 bytes`	The TCP/UDP buffer sizes to be used during inbound communication. Size is bytes.

`option.XXX`	`null`	Allows to configure additional netty options using "option." as prefix. For example "option.child.keepAlive=false" to set the netty option "child.keepAlive=false". See the Netty documentation for possible options that can be used.
`corePoolSize`	`10`	The number of allocated threads at component startup. Defaults to 10. **Note:** This option is removed from Camel 2.9.2 onwards. As we rely on Nettys default settings.
`maxPoolSize`	`100`	The maximum number of threads that may be allocated to this endpoint. Defaults to 100. **Note:** This option is removed from Camel 2.9.2 onwards. As we rely on Nettys default settings.
`disconnect`	`false`	Whether or not to disconnect(close) from Netty Channel right after use. Can be used for both consumer and producer.
`lazyChannelCreation`	`true`	Channels can be lazily created to avoid exceptions, if the remote server is not up and running when the Camel producer is started.
`transferExchange`	`false`	Only used for TCP. You can transfer the exchange over the wire instead of just the body. The following fields are transferred: In body, Out body, fault body, In headers, Out headers, fault headers, exchange properties, exchange exception. This requires that the objects are serializable. Camel will exclude any non-serializable objects and log it at WARN level.
`allowSerializedHeaders`	`false`	**Camel 2.18** Only used for TCP when transferExchange is true. Serializable objects in In/Out headers and exchange properties are transfered.
`disconnectOnNoReply`	`true`	If sync is enabled then this option dictates NettyConsumer if it should disconnect where there is no reply to send back.
`noReplyLogLevel`	`WARN`	If sync is enabled this option dictates NettyConsumer which logging level to use when logging a there is no reply to send back. Values are: `FATAL, ERROR, INFO, DEBUG, OFF`.
`serverExceptionCaughtLogLevel`	`WARN`	If the server (NettyConsumer) catches an exception then its logged using this logging level.
`serverClosedChannelExceptionCaughtLogLevel`	`DEBUG`	If the server (NettyConsumer) catches an `java.nio.channels.ClosedChannelException` then its logged using this logging level. This is used to avoid logging the closed channel exceptions, as clients can disconnect abruptly and then cause a flod of closed exceptions in the Netty server.
`allowDefaultCodec`	`true`	The netty component installs a default codec if both, encoder/deocder is null and textline is false. Setting allowDefaultCodec to false prevents the netty component from installing a default codec as the first element in the filter chain.
`textline`	`false`	Only used for TCP. If no codec is specified, you can use this flag to indicate a text line based codec; if not specified or the value is false, then Object Serialization is assumed over TCP.
`delimiter`	`LINE`	The delimiter to use for the textline codec. Possible values are `LINE` and `NULL`.
`decoderMaxLineLength`	`1024`	The max line length to use for the textline codec.
`autoAppendDelimiter`	`true`	Whether or not to auto append missing end delimiter when sending using the textline codec.
`encoding`	`null`	The encoding (a charset name) to use for the textline codec. If not provided, Camel will use the JVM default Charset.
`workerCount`	`null`	When netty works on nio mode, it uses default workerCount parameter from Netty, which is cpu_core_threads*2. User can use this operation to override the default workerCount from Netty.
`sslContextParameters`	`null`	SSL configuration using an `org.apache.camel.util.jsse.SSLContextParameters` instance. See Using the JSSE Configuration Utility.
`receiveBufferSizePredictor`	`null`	Configures the buffer size predictor. See details at Jetty documentation and this mail thread.
`requestTimeout`	`0`	Allows to use a timeout for the Netty producer when calling a remote server. By default no timeout is in use. The value is in milli seconds, so eg `30000` is 30 seconds. The requestTimeout is using Netty's ReadTimeoutHandler to trigger the timeout. **Camel 2.16, 2.15.3** you can also override this setting by setting the CamelNettyRequestTimeout header.
`needClientAuth`	`false`	Configures whether the server needs client authentication when using SSL.
`usingExecutorService`	`true`	Whether to use executorService to handle the message inside the camel route, the executorService can be set from NettyComponent.
`maximumPoolSize`	`16`	The core pool size for the ordered thread pool, if its in use. NOTE: you can just setup this on the NettyComponent level since **Camel 2.15, 2.14.1.**
`producerPoolEnabled`	`true`	Producer only. Whether producer pool is enabled or not. **Important:** Do not turn this off, as the pooling is needed for handling concurrency and reliable request/reply.

`producerPoolMaxActive`	`-1`	Producer only. Sets the cap on the number of objects that can be allocated by the pool (checked out to clients, or idle awaiting checkout) at a given time. Use a negative value for no limit.
`producerPoolMinIdle`	`0`	Producer only. Sets the minimum number of instances allowed in the producer pool before the evictor thread (if active) spawns new objects.
`producerPoolMaxIdle`	`100`	Producer only. Sets the cap on the number of "idle" instances in the pool.
`producerPoolMinEvictableIdle`	`300000`	Producer only. Sets the minimum amount of time (value in millis) an object may sit idle in the pool before it is eligible for eviction by the idle object evictor.
`bootstrapConfiguration`	`null`	Consumer only. Allows to configure the Netty ServerBootstrap options using a `org.apache.camel.component.netty4.NettyServerBootstrapConfiguration` instance. This can be used to reuse the same configuration for multiple consumers, to align their configuration more easily.
`bossGroup`	`null`	To use a explicit `io.netty.channel.EventLoopGroup` as the boss thread pool. For example to share a thread pool with multiple consumers. By default each consumer has their own boss pool with 1 core thread.
`workerGroup`	`null`	To use a explicit `io.netty.channel.EventLoopGroup` as the worker thread pool. For example to share a thread pool with multiple consumers or producers. By default each consumer or producer has their own worker pool with 2 x cpu count core threads.
channelGroup	null	**Camel 2.17** To use a explicit `io.netty.channel.group.ChannelGroup` for example to broadact a message to multiple channels.
`networkInterface`	`null`	Consumer only. When using UDP then this option can be used to specify a network interface by its name, such as `eth0` to join a multicast group.
`clientInitializerFactory`	`null`	**Camel 2.15:** To use a custom client initializer factory to control the pipelines in the channel. See further below for more details.
`serverInitializerFactory`	`null`	**Camel 2.15:** To use a custom server initializer factory to control the pipelines in the channel. See further below for more details.
`clientPipelineFactory`	`null`	**Deprecated**: Use clientInitializerFactory instead.
`serverPipelineFactory`	`null`	**Deprecated**: Use serverInitializerFactory instead.
`udpConnectionlessSending`	`false`	**Camel 2.15:** Producer only. This option supports connection less udp sending which is a real fire and forget. A connected udp send receive the PortUnreachableException if no one is listen on the receiving port.
`clientMode`	`false`	**Camel 2.15:** Consumer only. If the `clientMode` is true, netty consumer will connect the address as a TCP client.
`reconnect`	`true`	**Camel 2.16:** Consumer only. Used only in clientMode in consumer, the consumer will attempt to reconnect on disconnection automatically.
`reconnectInterval`	`10000`	**Camel 2.16:** Consumer only. Used if reconnect and clientMode is enabled. The interval in milli seconds to attempt reconnection.
`useByteBuf`	`false`	**Camel 2.16:** Producer only. If the `useByteBuf` is true, netty producer will turn the message body into ByteBuf before sending it out.
`udpByteArrayCodec`	`false`	**Camel 2.16:** When using UDP protocol then turning this option to true sends the data as a byte array instead of the default object serialization codec.
`reuseChannel`	`false`	**Camel 2.17:** Producer only. This option allows producers to reuse the same Netty `Channel` for the lifecycle of processing the Exchange. This is useable if you need to call a server multiple times in a Camel route and want to use the same network connection. When using this the channel is not returned to the connection pool until the Exchange is done; or disconnected if the disconnect option is set to true. The reused `Channel` is stored on the Exchange as an exchange property with the key `NettyConstants#NETTY_CHANNEL` which allows you to obtain the channel during routing and use it as well.
`nativeTransport`	`false`	**Camel 2.18:** Whether to use native transport instead of NIO. Native transport takes advantage of the host operating system and is only supported on some platforms. You need to add the netty JAR for the host operating system you are using. See more details at: http://netty.io/wiki/native-transports.html

Registry based Options

Codec Handlers and SSL Keystores can be enlisted in the Registry, such as in the Spring XML file.
The values that could be passed in, are the following:

Name	Description

`passphrase`	password setting to use in order to encrypt/decrypt payloads sent using SSH
`keyStoreFormat`	keystore format to be used for payload encryption. Defaults to "JKS" if not set
`securityProvider`	Security provider to be used for payload encryption. Defaults to "SunX509" if not set.
`keyStoreFile`	**deprecated:** Client side certificate keystore to be used for encryption
`trustStoreFile`	**deprecated:** Server side certificate keystore to be used for encryption
`keyStoreResource`	**Camel 2.11.1:** Client side certificate keystore to be used for encryption. Is loaded by default from classpath, but you can prefix with `"classpath:"`, `"file:"`, or `"http:"` to load the resource from different systems.
`trustStoreResource`	**Camel 2.11.1:** Server side certificate keystore to be used for encryption. Is loaded by default from classpath, but you can prefix with `"classpath:"`, `"file:"`, or `"http:"` to load the resource from different systems.
`sslHandler`	Reference to a class that could be used to return an SSL Handler
`encoder`	A custom `ChannelHandler` class that can be used to perform special marshalling of outbound payloads. Must override `io.netty.channel.ChannelInboundHandlerAdapter`.
`encoders`	A list of encoders to be used. You can use a String which have values separated by comma, and have the values be looked up in the Registry. Just remember to prefix the value with # so Camel knows it should lookup.
`decoder`	A custom `ChannelHandler` class that can be used to perform special marshalling of inbound payloads. Must override `io.netty.channel.ChannelOutboundHandlerAdapter`.
`decoders`	A list of decoders to be used. You can use a String which have values separated by comma, and have the values be looked up in the Registry. Just remember to prefix the value with # so Camel knows it should lookup.

Important: Read below about using non shareable encoders/decoders.

Using non shareable encoders or decoders

If your encoders or decoders is not shareable (eg they have the @Shareable class annotation), then your encoder/decoder must implement the `org.apache.camel.component.netty.ChannelHandlerFactory` interface, and return a new instance in the `newChannelHandler` method. This is to ensure the encoder/decoder can safely be used. If this is not the case, then the Netty component will log a WARN when
an endpoint is created.

The Netty component offers a `org.apache.camel.component.netty.ChannelHandlerFactories` factory class, that has a number of commonly used methods.

Sending Messages to/from a Netty endpoint

Netty Producer

In Producer mode, the component provides the ability to send payloads to a socket endpoint
using either TCP or UDP protocols (with optional SSL support).

The producer mode supports both one-way and request-response based operations.

Netty Consumer

In Consumer mode, the component provides the ability to:

- listen on a specified socket using either TCP or UDP protocols (with optional SSL support),
- receive requests on the socket using text/xml, binary and serialized object based payloads and
- send them along on a route as message exchanges.

The consumer mode supports both one-way and request-response based operations.

Usage Samples

A UDP Netty endpoint using Request-Reply and serialized object payload

```
RouteBuilder builder = new RouteBuilder() {
  public void configure() {
    from("netty4:udp://localhost:5155?sync=true")
      .process(new Processor() {
         public void process(Exchange exchange) throws Exception {
           Poetry poetry = (Poetry) exchange.getIn().getBody();
           poetry.setPoet("Dr. Sarojini Naidu");
           exchange.getOut().setBody(poetry);
         }
       }
    }
};
```

A TCP based Netty consumer endpoint using One-way communication

```
RouteBuilder builder = new RouteBuilder() {
  public void configure() {
       from("netty4:tcp://localhost:5150")
           .to("mock:result");
  }
};
```

An SSL/TCP based Netty consumer endpoint using Request-Reply communication

Using the JSSE Configuration Utility

As of Camel 2.9, the Netty component supports SSL/TLS configuration through the Camel JSSE Configuration Utility. This utility greatly decreases the amount of component specific code you need to write and is configurable at the endpoint and component levels. The following examples demonstrate how to use the utility with the Netty component.

Programmatic configuration of the component

```
KeyStoreParameters ksp = new KeyStoreParameters();
ksp.setResource("/users/home/server/keystore.jks");
ksp.setPassword("keystorePassword");

KeyManagersParameters kmp = new KeyManagersParameters();
kmp.setKeyStore(ksp);
kmp.setKeyPassword("keyPassword");

SSLContextParameters scp = new SSLContextParameters();
scp.setKeyManagers(kmp);

NettyComponent nettyComponent = getContext().getComponent("netty4", NettyComponent.class);
nettyComponent.setSslContextParameters(scp);
```

Spring DSL based configuration of endpoint

```
...
  <camel:sslContextParameters
      id="sslContextParameters">
    <camel:keyManagers
        keyPassword="keyPassword">
      <camel:keyStore
          resource="/users/home/server/keystore.jks"
          password="keystorePassword"/>
    </camel:keyManagers>
  </camel:sslContextParameters>...
...
  <to uri="netty4:tcp://localhost:5150?sync=true&ssl=true&sslContextParameters=#sslContextParameters"/>
...
```

Using Basic SSL/TLS configuration on the Jetty Component

```
JndiRegistry registry = new JndiRegistry(createJndiContext());
registry.bind("password", "changeit");
registry.bind("ksf", new File("src/test/resources/keystore.jks"));
registry.bind("tsf", new File("src/test/resources/keystore.jks"));

context.createRegistry(registry);
context.addRoutes(new RouteBuilder() {
  public void configure() {
      String netty_ssl_endpoint =
         "netty4:tcp://localhost:5150?sync=true&ssl=true&passphrase=#password"
         + "&keyStoreFile=#ksf&trustStoreFile=#tsf";
      String return_string =
         "When You Go Home, Tell Them Of Us And Say,"
         + "For Your Tomorrow, We Gave Our Today.";

      from(netty_ssl_endpoint)
       .process(new Processor() {
          public void process(Exchange exchange) throws Exception {
            exchange.getOut().setBody(return_string);
          }
       }
  }
});
```

Getting access to SSLSession and the client certificate

Available as of Camel 2.12

You can get access to the `javax.net.ssl.SSLSession` if you eg need to get details about the client certificate. When `ssl=true` then the Netty4 component will store the `SSLSession` as a header on the Camel Message as shown below:

```
SSLSession session = exchange.getIn().getHeader(NettyConstants.NETTY_SSL_SESSION, SSLSession.class);
// get the first certificate which is client certificate
javax.security.cert.X509Certificate cert = session.getPeerCertificateChain()[0];
Principal principal = cert.getSubjectDN();
```

Remember to set `needClientAuth=true` to authenticate the client, otherwise `SSLSession` cannot access information about the client certificate, and you may get an exception `javax.net.ssl.SSLPeerUnverifiedException: peer not authenticated`. You may also get this exception if the client certificate is expired or not valid etc.

The option `sslClientCertHeaders` can be set to `true` which then enriches the Camel Message with headers having details about the client certificate. For example the subject name is readily available in the header `CamelNettySSLClientCertSubjectName`.

Using Multiple Codecs

In certain cases it may be necessary to add chains of encoders and decoders to the netty pipeline. To add multpile codecs to a camel netty endpoint the 'encoders' and 'decoders' uri parameters should be used. Like the 'encoder' and 'decoder' parameters they are used to supply references (to lists of ChannelUpstreamHandlers and ChannelDownstreamHandlers) that should be added to the pipeline. Note that if encoders is specified then the encoder param will be ignored, similarly for decoders and the decoder param.

Read further above about using non shareable encoders/decoders.

The lists of codecs need to be added to the Camel's registry so they can be resolved when the endpoint is created.

```
ChannelHandlerFactory lengthDecoder = ChannelHandlerFactories.newLengthFieldBasedFrameDecoder(1048576, 0, 4, 0,
4);

StringDecoder stringDecoder = new StringDecoder();
registry.bind("length-decoder", lengthDecoder);
registry.bind("string-decoder", stringDecoder);

LengthFieldPrepender lengthEncoder = new LengthFieldPrepender(4);
StringEncoder stringEncoder = new StringEncoder();
registry.bind("length-encoder", lengthEncoder);
registry.bind("string-encoder", stringEncoder);

List<ChannelHandler> decoders = new ArrayList<ChannelHandler>();
decoders.add(lengthDecoder);
decoders.add(stringDecoder);

List<ChannelHandler> encoders = new ArrayList<ChannelHandler>();
encoders.add(lengthEncoder);
encoders.add(stringEncoder);

registry.bind("encoders", encoders);
registry.bind("decoders", decoders);
```

Spring's native collections support can be used to specify the codec lists in an application context

```
<util:list id="decoders" list-class="java.util.LinkedList">
        <bean class="org.apache.camel.component.netty4.ChannelHandlerFactories" factory-method="
newLengthFieldBasedFrameDecoder">
            <constructor-arg value="1048576"/>
            <constructor-arg value="0"/>
            <constructor-arg value="4"/>
            <constructor-arg value="0"/>
            <constructor-arg value="4"/>
        </bean>
        <bean class="io.netty.handler.codec.string.StringDecoder"/>
    </util:list>

    <util:list id="encoders" list-class="java.util.LinkedList">
        <bean class="io.netty.handler.codec.LengthFieldPrepender">
            <constructor-arg value="4"/>
        </bean>
        <bean class="io.netty.handler.codec.string.StringEncoder"/>
    </util:list>

    <bean id="length-encoder" class="io.netty.handler.codec.LengthFieldPrepender">
        <constructor-arg value="4"/>
    </bean>
    <bean id="string-encoder" class="io.netty.handler.codec.string.StringEncoder"/>

    <bean id="length-decoder" class="org.apache.camel.component.netty4.ChannelHandlerFactories" factory-method="
newLengthFieldBasedFrameDecoder">
        <constructor-arg value="1048576"/>
        <constructor-arg value="0"/>
        <constructor-arg value="4"/>
        <constructor-arg value="0"/>
        <constructor-arg value="4"/>
    </bean>
    <bean id="string-decoder" class="io.netty.handler.codec.string.StringDecoder"/>
```

The bean names can then be used in netty endpoint definitions either as a comma separated list or contained in a List e.g.

```
 from("direct:multiple-codec").to("netty4:tcp://localhost:{{port}}?encoders=#encoders&sync=false");

 from("netty4:tcp://localhost:{{port}}?decoders=#length-decoder,#string-decoder&sync=false").to("mock:multiple-
codec");
```

or via spring.

```
  <camelContext id="multiple-netty-codecs-context" xmlns="http://camel.apache.org/schema/spring">
       <route>
           <from uri="direct:multiple-codec"/>
           <to uri="netty4:tcp://localhost:5150?encoders=#encoders&amp;sync=false"/>
       </route>
       <route>
           <from uri="netty4:tcp://localhost:5150?decoders=#length-decoder,#string-decoder&amp;sync=false"/>
           <to uri="mock:multiple-codec"/>
       </route>
   </camelContext>
```

Closing Channel When Complete

When acting as a server you sometimes want to close the channel when, for example, a client conversion is finished.
You can do this by simply setting the endpoint option `disconnect=true`.

However you can also instruct Camel on a per message basis as follows.
To instruct Camel to close the channel, you should add a header with the key `CamelNettyCloseChannelWhenComplete` set to a boolean `true` value.
For instance, the example below will close the channel after it has written the bye message back to the client:

```
       from("netty4:tcp://localhost:8080").process(new Processor() {
           public void process(Exchange exchange) throws Exception {
               String body = exchange.getIn().getBody(String.class);
               exchange.getOut().setBody("Bye " + body);
               // some condition which determines if we should close
               if (close) {
                   exchange.getOut().setHeader(NettyConstants.NETTY_CLOSE_CHANNEL_WHEN_COMPLETE, true);
               }
           }
       });
```

Adding custom channel pipeline factories to gain complete control over a created pipeline

Custom channel pipelines provide complete control to the user over the handler/interceptor chain by inserting custom handler(s), encoder(s) & decoders without having to specify them in the Netty Endpoint URL in a very simple way.

In order to add a custom pipeline, a custom channel pipeline factory must be created and registered with the context via the context registry (JNDIRegistry, or the camel-spring ApplicationContextRegistry etc).

A custom pipeline factory must be constructed as follows

- A Producer linked channel pipeline factory must extend the abstract class `ClientPipelineFactory`.
- A Consumer linked channel pipeline factory must extend the abstract class `ServerInitializerFactory`.
- The classes should override the initChannel() method in order to insert custom handler(s), encoder(s) and decoder(s). Not overriding the initChannel() method creates a pipeline with no handlers, encoders or decoders wired to the pipeline.

The example below shows how ServerInitializerFactory factory may be created

Using custom pipeline factory

```
public class SampleServerInitializerFactory extends ServerInitializerFactory {
    private int maxLineSize = 1024;

     protected void initChannel(Channel ch) throws Exception {
        ChannelPipeline channelPipeline = ch.pipeline();

        channelPipeline.addLast("encoder-SD", new StringEncoder(CharsetUtil.UTF_8));
        channelPipeline.addLast("decoder-DELIM", new DelimiterBasedFrameDecoder(maxLineSize, true, Delimiters.
lineDelimiter()));
        channelPipeline.addLast("decoder-SD", new StringDecoder(CharsetUtil.UTF_8));
        // here we add the default Camel ServerChannelHandler for the consumer, to allow Camel to route the
message etc.
        channelPipeline.addLast("handler", new ServerChannelHandler(consumer));
    }
}
```

The custom channel pipeline factory can then be added to the registry and instantiated/utilized on a camel route in the following way

```
Registry registry = camelContext.getRegistry();
ServerInitializerFactory factory = new TestServerInitializerFactory();
registry.bind("spf", factory);
context.addRoutes(new RouteBuilder() {
  public void configure() {
      String netty_ssl_endpoint =
         "netty4:tcp://localhost:5150?serverInitializerFactory=#spf"
      String return_string =
         "When You Go Home, Tell Them Of Us And Say,"
         + "For Your Tomorrow, We Gave Our Today.";

      from(netty_ssl_endpoint)
       .process(new Processor() {
          public void process(Exchange exchange) throws Exception {
            exchange.getOut().setBody(return_string);
          }
       }
  }
});
```

Reusing Netty boss and worker thread pools

Available as of Camel 2.12

Netty has two kind of thread pools: boss and worker. By default each Netty consumer and producer has their private thread pools. If you want to reuse these thread pools among multiple consumers or producers then the thread pools must be created and enlisted in the Registry.

For example using Spring XML we can create a shared worker thread pool using the `NettyWorkerPoolBuilder` with 2 worker threads as shown below:

```
  <!-- use the worker pool builder to create to help create the shared thread pool -->
  <bean id="poolBuilder" class="org.apache.camel.component.netty.NettyWorkerPoolBuilder">
    <property name="workerCount" value="2"/>
  </bean>

  <!-- the shared worker thread pool -->
  <bean id="sharedPool" class="org.jboss.netty.channel.socket.nio.WorkerPool"
        factory-bean="poolBuilder" factory-method="build" destroy-method="shutdown">
  </bean>
```

For boss thread pool there is a `org.apache.camel.component.netty4.NettyServerBossPoolBuilder` builder for Netty consumers, and a `org.apache.camel.component.netty4.NettyClientBossPoolBuilder` for the Netty produces.

Then in the Camel routes we can refer to this worker pools by configuring the `workerPool` option in the URI as shown below:

```
    <route>
      <from uri="netty4:tcp://localhost:5021?textline=true&amp;sync=true&amp;workerPool=#sharedPool&amp;
usingExecutorService=false"/>
      <to uri="log:result"/>
      ...
    </route>
```

And if we have another route we can refer to the shared worker pool:

```
    <route>
      <from uri="netty4:tcp://localhost:5022?textline=true&amp;sync=true&amp;workerPool=#sharedPool&amp;
usingExecutorService=false"/>
      <to uri="log:result"/>
      ...
    </route>
```

... and so forth.

See Also

- Configuring Camel
- Component
- Endpoint
- Getting Started

- Netty HTTP
- MINA