Hazelcast Component

Hazelcast Component

Available as of Camel 2.7

The **hazelcast:** component allows you to work with the Hazelcast distributed data grid / cache. Hazelcast is a in memory data grid, entirely written in Java (single jar). It offers a great palette of different data stores like map, multi map (same key, n values), queue, list and atomic number. The main reason to use Hazelcast is its simple cluster support. If you have enabled multicast on your network you can run a cluster with hundred nodes with no extra configuration. Hazelcast can simply configured to add additional features like n copies between nodes (default is 1), cache persistence, network configuration (if needed), near cache, enviction and so on. For more information consult the Hazelcast documentation on http://www.hazelcast.com/docs.jsp.

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

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

URI format

```
hazelcast:[ map | multimap | queue | topic | seda | set | atomicvalue | instance | list | ringbuffer]:cachename
[?options]
```

Topic support is available as of Camel 2.15.
RingBuffer support is available as of Camel 2.16.

Options

Name	Required	Description
hazelcastInstance	No	**Camel 2.14:** The hazelcast instance **reference** which can be used for hazelcast endpoint. If you don't specify the instance reference, camel use the default hazelcast instance from the camel-hazelcast instance.
hazelcastInstanceName	No	**Camel 2.16:** The hazelcast instance **reference name** which can be used for hazelcast endpoint. If you don't specify the instance reference, camel use the default hazelcast instance from the camel-hazelcast instance.
operation	-1	To specify a default operation to use, if no operation header has been provided. **deprecated** use defaultOperation instead.
defaultOperation	-1	**Camel 2.15:** To specify a default operation to use, if no operation header has been provided.

You have to use the second prefix to define which type of data store you want to use.

Sections

1. Usage of #map
2. Usage of #multimap
3. Usage of #queue
4. Usage of #topic
5. Usage of #list
6. Usage of #seda
7. Usage of atomic number
8. Usage of #cluster support (instance)
9. Usage of #replicatedmap
10. Usage of #ringbuffer

Usage of Map

map cache producer - to("hazelcast:map:foo")

If you want to store a value in a map you can use the map cache producer. The map cache producer provides 5 operations (put, get, update, delete, query). For the first 4 you have to provide the operation inside the "hazelcast.operation.type" header variable. In Java DSL you can use the constants from `org.apache.camel.component.hazelcast.HazelcastConstants`.

Header Variables for the request message:

Name	Type	Description
`hazelcast.operation.type`	`String`	valid values are: put, delete, get, update, query
`hazelcast.objectId`	`String`	the object id to store / find your object inside the cache (not needed for the query operation)

Header variables have changed in Camel 2.8

Name	Type	Description
`CamelHazelcastOperationType`	`String`	valid values are: put, delete, get, update, query **Version 2.8** **From Camel 2.16:** getAll, putIfAbsent, clear.
`CamelHazelcastObjectId`	`String`	the object id to store / find your object inside the cache (not needed for the query operation) **Version 2.8**

You can call the samples with:

```
template.sendBodyAndHeader("direct:[put|get|update|delete|query]", "my-foo", HazelcastConstants.OBJECT_ID,
"4711");
```

Sample for **put**:

Java DSL:

```
from("direct:put")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.PUT_OPERATION))
.toF("hazelcast:%sfoo", HazelcastConstants.MAP_PREFIX);
```

Spring DSL:

```
<route>
        <from uri="direct:put" />
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
        <setHeader headerName="hazelcast.operation.type">
                <constant>put</constant>
        </setHeader>
        <to uri="hazelcast:map:foo" />
</route>
```

Sample for **get**:

Java DSL:

```
from("direct:get")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.GET_OPERATION))
.toF("hazelcast:%sfoo", HazelcastConstants.MAP_PREFIX)
.to("seda:out");
```

Spring DSL:

```
<route>
        <from uri="direct:get" />
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
        <setHeader headerName="hazelcast.operation.type">
                <constant>get</constant>
        </setHeader>
        <to uri="hazelcast:map:foo" />
        <to uri="seda:out" />
</route>
```

Sample for **update**:

Java DSL:

```
from("direct:update")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.UPDATE_OPERATION))
.toF("hazelcast:%sfoo", HazelcastConstants.MAP_PREFIX);
```

Spring DSL:

```
<route>
        <from uri="direct:update" />
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
        <setHeader headerName="hazelcast.operation.type">
                <constant>update</constant>
        </setHeader>
        <to uri="hazelcast:map:foo" />
</route>
```

Sample for **delete**:

Java DSL:

```
from("direct:delete")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.DELETE_OPERATION))
.toF("hazelcast:%sfoo", HazelcastConstants.MAP_PREFIX);
```

Spring DSL:

```
<route>
        <from uri="direct:delete" />
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
        <setHeader headerName="hazelcast.operation.type">
                <constant>delete</constant>
        </setHeader>
        <to uri="hazelcast:map:foo" />
</route>
```

Sample for **query**

Java DSL:

```
from("direct:query")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.QUERY_OPERATION))
.toF("hazelcast:%sfoo", HazelcastConstants.MAP_PREFIX)
.to("seda:out");
```

Spring DSL:

```
<route>
        <from uri="direct:query" />
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
        <setHeader headerName="hazelcast.operation.type">
                <constant>query</constant>
        </setHeader>
        <to uri="hazelcast:map:foo" />
        <to uri="seda:out" />
</route>
```

For the query operation Hazelcast offers a SQL like syntax to query your distributed map.

```
String q1 = "bar > 1000";
template.sendBodyAndHeader("direct:query", null, HazelcastConstants.QUERY, q1);
```

map cache consumer - from("hazelcast:map:foo")

Hazelcast provides event listeners on their data grid. If you want to be notified if a cache will be manipulated, you can use the map consumer. There're 4 events: **put**, **update**, **delete** and **envict**. The event type will be stored in the "**hazelcast.listener.action**" header variable. The map consumer provides some additional information inside these variables:

Header Variables inside the response message:

Name	Type	Description
`hazelcast.listener.time`	`Long`	time of the event in millis
`hazelcast.listener.type`	`String`	the map consumer sets here "cachelistener"
`hazelcast.listener.action`	`String`	type of event - here **added**, **updated**, **envicted** and **removed**
`hazelcast.objectId`	`String`	the oid of the object
`hazelcast.cache.name`	`String`	the name of the cache - e.g. "foo"
`hazelcast.cache.type`	`String`	the type of the cache - here map

Header variables have changed in Camel 2.8

Name	Type	Description
`CamelHazelcastListenerTime`	`Long`	time of the event in millis **Version 2.8**
`CamelHazelcastListenerType`	`String`	the map consumer sets here "cachelistener" **Version 2.8**
`CamelHazelcastListenerAction`	`String`	type of event - here **added**, **updated**, **envicted** and **removed**. **Version 2.8**
`CamelHazelcastObjectId`	`String`	the oid of the object **Version 2.8**
`CamelHazelcastCacheName`	`String`	the name of the cache - e.g. "foo" **Version 2.8**
`CamelHazelcastCacheType`	`String`	the type of the cache - here map **Version 2.8**

The object value will be stored within **put** and **update** actions inside the message body.

Here's a sample:

```
fromF("hazelcast:%sfoo", HazelcastConstants.MAP_PREFIX)
.log("object...")
.choice()
    .when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ADDED))
         .log("...added")
         .to("mock:added")
    .when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ENVICTED))
         .log("...envicted")
         .to("mock:envicted")
    .when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.UPDATED))
         .log("...updated")
         .to("mock:updated")
    .when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.REMOVED))
         .log("...removed")
         .to("mock:removed")
    .otherwise()
         .log("fail!");
```

Usage of Multi Map

multimap cache producer - to("hazelcast:multimap:foo")

A multimap is a cache where you can store n values to one key. The multimap producer provides 4 operations (put, get, removevalue, delete).

Header Variables for the request message:

Name	Type	Description
`hazelcast.operation.type`	`String`	valid values are: put, get, removevalue, delete
`hazelcast.objectId`	`String`	the object id to store / find your object inside the cache

Header variables have changed in Camel 2.8

Name	Type	Description
`CamelHazelcastOperationType`	`String`	valid values are: put, get, removevalue, delete **Version 2.8** **From Camel 2.16:** clear.
`CamelHazelcastObjectId`	`String`	the object id to store / find your object inside the cache **Version 2.8**

Sample for **put**:

Java DSL:

```
from("direct:put")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.PUT_OPERATION))
.to(String.format("hazelcast:%sbar", HazelcastConstants.MULTIMAP_PREFIX));
```

Spring DSL:

```
<route>
        <from uri="direct:put" />
        <log message="put.."/>
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
        <setHeader headerName="hazelcast.operation.type">
                <constant>put</constant>
        </setHeader>
        <to uri="hazelcast:multimap:foo" />
</route>
```

Sample for **removevalue**:

Java DSL:

```
from("direct:removevalue")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.REMOVEVALUE_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.MULTIMAP_PREFIX);
```

Spring DSL:

```
<route>
        <from uri="direct:removevalue" />
        <log message="removevalue..."/>
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
        <setHeader headerName="hazelcast.operation.type">
                <constant>removevalue</constant>
        </setHeader>
        <to uri="hazelcast:multimap:foo" />
</route>
```

To remove a value you have to provide the value you want to remove inside the message body. If you have a multimap object {`key: "4711" values: { "my-foo", "my-bar"`}} you have to put "my-foo" inside the message body to remove the "my-foo" value.

Sample for **get**:

Java DSL:

```
from("direct:get")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.GET_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.MULTIMAP_PREFIX)
.to("seda:out");
```

Spring DSL:

```
<route>
        <from uri="direct:get" />
        <log message="get.."/>
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
        <setHeader headerName="hazelcast.operation.type">
                <constant>get</constant>
        </setHeader>
        <to uri="hazelcast:multimap:foo" />
        <to uri="seda:out" />
</route>
```

Sample for **delete**:

Java DSL:

```
from("direct:delete")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.DELETE_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.MULTIMAP_PREFIX);
```

Spring DSL:

```
<route>
        <from uri="direct:delete" />
        <log message="delete.."/>
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
        <setHeader headerName="hazelcast.operation.type">
                <constant>delete</constant>
        </setHeader>
        <to uri="hazelcast:multimap:foo" />
</route>
```

you can call them in your test class with:

```
template.sendBodyAndHeader("direct:[put|get|removevalue|delete]", "my-foo", HazelcastConstants.OBJECT_ID,
"4711");
```

multimap cache consumer - from("hazelcast:multimap:foo")

For the multimap cache this component provides the same listeners / variables as for the map cache consumer (except the update and enviction listener). The only difference is the **multimap** prefix inside the URI. Here is a sample:

```
fromF("hazelcast:%sbar", HazelcastConstants.MULTIMAP_PREFIX)
.log("object...")
.choice()
        .when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ADDED))
                .log("...added")
                .to("mock:added")
        //.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ENVICTED))
        //        .log("...envicted")
        //        .to("mock:envicted")
        .when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.REMOVED))
                .log("...removed")
                .to("mock:removed")
        .otherwise()
                .log("fail!");
```

Header Variables inside the response message:

Name	Type	Description
`hazelcast.listener.time`	`Long`	time of the event in millis
`hazelcast.listener.type`	`String`	the map consumer sets here "cachelistener"
`hazelcast.listener.action`	`String`	type of event - here **added** and **removed** (and soon **envicted**)
`hazelcast.objectId`	`String`	the oid of the object
`hazelcast.cache.name`	`String`	the name of the cache - e.g. "foo"
`hazelcast.cache.type`	`String`	the type of the cache - here multimap

Eviction will be added as feature, soon (this is a Hazelcast issue).

Header variables have changed in Camel 2.8

Name	Type	Description
`CamelHazelcastListenerTime`	`Long`	time of the event in millis **Version 2.8**
`CamelHazelcastListenerType`	`String`	the map consumer sets here "cachelistener" **Version 2.8**
`CamelHazelcastListenerAction`	`String`	type of event - here **added** and **removed** (and soon **envicted**) **Version 2.8**
`CamelHazelcastObjectId`	`String`	the oid of the object **Version 2.8**
`CamelHazelcastCacheName`	`String`	the name of the cache - e.g. "foo" **Version 2.8**
`CamelHazelcastCacheType`	`String`	the type of the cache - here multimap **Version 2.8**

Usage of Queue

Queue producer – to(“hazelcast:queue:foo”)

The queue producer provides 6 operations (add, put, poll, peek, offer, removevalue).

Sample for **add**:

```
from("direct:add")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.ADD_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.QUEUE_PREFIX);
```

Sample for **put**:

```
from("direct:put")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.PUT_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.QUEUE_PREFIX);
```

Sample for **poll**:

```
from("direct:poll")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.POLL_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.QUEUE_PREFIX);
```

Sample for **peek**:

```
from("direct:peek")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.PEEK_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.QUEUE_PREFIX);
```

Sample for **offer**:

```
from("direct:offer")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.OFFER_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.QUEUE_PREFIX);
```

Sample for **removevalue**:

```
from("direct:removevalue")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.REMOVEVALUE_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.QUEUE_PREFIX);
```

Queue consumer – from(“hazelcast:queue:foo”)

The queue consumer provides 2 operations (add, remove).

```
fromF("hazelcast:%smm", HazelcastConstants.QUEUE_PREFIX)
   .log("object...")
   .choice()
        .when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ADDED))
                .log("...added")
                .to("mock:added")
        .when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.REMOVED))
                .log("...removed")
                .to("mock:removed")
        .otherwise()
                .log("fail!");
```

Usage of Topic

Topic producer – to(“hazelcast:topic:foo”)

The topic producer provides only one operation (publish).

Sample for **publish**:

```
from("direct:add")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.PUBLISH_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.PUBLISH_OPERATION);
```

Topic consumer – from(“hazelcast:topic:foo”)

The topic consumer provides only one operation (received). This component is supposed to support multiple consumption as it's expected when it comes to topics so you are free to have as much consumers as you need on the same hazelcast topic.

```
fromF("hazelcast:%sfoo", HazelcastConstants.TOPIC_PREFIX)
  .choice()
    .when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.RECEIVED))
      .log("...message received")
    .otherwise()
      .log("...this should never have happened")
```

Usage of List

List producer – to(“hazelcast:list:foo”)

The list producer provides 4 operations (add, addAll, set, get, removevalue, removeAll, clear).

Sample for **add**:

```
from("direct:add")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.ADD_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.LIST_PREFIX);
```

Sample for **get**:

```
from("direct:get")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.GET_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.LIST_PREFIX)
.to("seda:out");
```

Sample for **setvalue**:

```
from("direct:set")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.SETVALUE_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.LIST_PREFIX);
```

Sample for **removevalue**:

```
from("direct:removevalue")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.REMOVEVALUE_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.LIST_PREFIX);
```

Note that **CamelHazelcastObjectIndex** header is used for indexing purpose.

The list consumer provides 2 operations (add, remove).List consumer – from(“hazelcast:list:foo”)

```
fromF("hazelcast:%smm", HazelcastConstants.LIST_PREFIX)
        .log("object...")
        .choice()
                .when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ADDED))
                        .log("...added")
                        .to("mock:added")
                .when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.REMOVED))
                        .log("...removed")
                        .to("mock:removed")
                .otherwise()
                        .log("fail!");
```

Usage of SEDA

SEDA component differs from the rest components provided. It implements a work-queue in order to support asynchronous SEDA architectures, similar to the core "SEDA" component.

SEDA producer – to(“hazelcast:seda:foo”)

The SEDA producer provides no operations. You only send data to the specified queue.

Name	default value	Description
`transferExchange`	`false`	**Camel 2.8.0:** if set to true the whole Exchange will be transfered. If header or body contains not serializable objects, they will be skipped.

Java DSL :

```
from("direct:foo")
.to("hazelcast:seda:foo");
```

Spring DSL :

```
<route>
   <from uri="direct:start" />
   <to uri="hazelcast:seda:foo" />
</route>
```

SEDA consumer – from(“hazelcast:seda:foo”)

The SEDA consumer provides no operations. You only retrieve data from the specified queue.

Name	default value	Description
`pollInterval`	`1000`	The timeout used when consuming from the SEDA queue. When a timeout occurs, the consumer can check whether it is allowed to continue running. Setting a lower value allows the consumer to react more quickly upon shutdown. (**deprecated** from Camel 2.15 onwards, use pollTimeout instead).
`pollTimeout`	1000	**Camel 2.15:** The timeout used when consuming from the SEDA queue. When a timeout occurs, the consumer can check whether it is allowed to continue running. Setting a lower value allows the consumer to react more quickly upon shutdown.
`concurrentConsumers`	`1`	To use concurrent consumers polling from the SEDA queue.
`transferExchange`	`false`	**Camel 2.8.0:** if set to true the whole Exchange will be transfered. If header or body contains not serializable objects, they will be skipped.
`transacted`	`false`	**Camel 2.10.4:** if set to true then the consumer runs in transaction mode, where the messages in the seda queue will only be removed if the transaction commits, which happens when the processing is complete.

Java DSL :

```
from("hazelcast:seda:foo")
.to("mock:result");
```

Spring DSL:

```
<route>
  <from uri="hazelcast:seda:foo" />
  <to uri="mock:result" />
</route>
```

Usage of Atomic Number

There is no consumer for this endpoint!

atomic number producer - to("hazelcast:atomicnumber:foo")

An atomic number is an object that simply provides a grid wide number (long). The operations for this producer are setvalue (set the number with a given value), get, increase (+1), decrease (-1) and destroy.

Header Variables for the request message:

Name	Type	Description
`hazelcast.operation.type`	`String`	valid values are: setvalue, get, increase, decrease, destroy

Header variables have changed in Camel 2.8

Name	Type	Description
`CamelHazelcastOperationType`	`String`	valid values are: setvalue, get, increase, decrease, destroy **Available as of Camel version 2.8**

Sample for **set**:

Java DSL:

```
from("direct:set")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.SETVALUE_OPERATION))
.toF("hazelcast:%sfoo", HazelcastConstants.ATOMICNUMBER_PREFIX);
```

Spring DSL:

```
<route>
        <from uri="direct:set" />
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
        <setHeader headerName="hazelcast.operation.type">
                <constant>setvalue</constant>
        </setHeader>
        <to uri="hazelcast:atomicvalue:foo" />
</route>
```

Provide the value to set inside the message body (here the value is 10): `template.sendBody("direct:set", 10);`

Sample for **get**:

Java DSL:

```
from("direct:get")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.GET_OPERATION))
.toF("hazelcast:%sfoo", HazelcastConstants.ATOMICNUMBER_PREFIX);
```

Spring DSL:

```
<route>
        <from uri="direct:get" />
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
        <setHeader headerName="hazelcast.operation.type">
                <constant>get</constant>
        </setHeader>
        <to uri="hazelcast:atomicvalue:foo" />
</route>
```

You can get the number with `long body = template.requestBody("direct:get", null, Long.class);`.

Sample for **increment**:

Java DSL:

```
from("direct:increment")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.INCREMENT_OPERATION))
.toF("hazelcast:%sfoo", HazelcastConstants.ATOMICNUMBER_PREFIX);
```

Spring DSL:

```
<route>
        <from uri="direct:increment" />
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
        <setHeader headerName="hazelcast.operation.type">
                <constant>increment</constant>
        </setHeader>
        <to uri="hazelcast:atomicvalue:foo" />
</route>
```

The actual value (after increment) will be provided inside the message body.

Sample for **decrement**:

Java DSL:

```
from("direct:decrement")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.DECREMENT_OPERATION))
.toF("hazelcast:%sfoo", HazelcastConstants.ATOMICNUMBER_PREFIX);
```

Spring DSL:

```
<route>
        <from uri="direct:decrement" />
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
        <setHeader headerName="hazelcast.operation.type">
                <constant>decrement</constant>
        </setHeader>
        <to uri="hazelcast:atomicvalue:foo" />
</route>
```

The actual value (after decrement) will be provided inside the message body.

Sample for **destroy**

There's a bug inside Hazelcast. So this feature may not work properly. Will be fixed in 1.9.3.

Java DSL:

```
from("direct:destroy")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.DESTROY_OPERATION))
.toF("hazelcast:%sfoo", HazelcastConstants.ATOMICNUMBER_PREFIX);
```

Spring DSL:

```
<route>
        <from uri="direct:destroy" />
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
        <setHeader headerName="hazelcast.operation.type">
                <constant>destroy</constant>
        </setHeader>
        <to uri="hazelcast:atomicvalue:foo" />
</route>
```

cluster support

This endpoint provides no producer!

instance consumer - from("hazelcast:instance:foo")

Hazelcast makes sense in one single "server node", but it's extremly powerful in a clustered environment. The instance consumer fires if a new cache instance will join or leave the cluster.

Here's a sample:

```
fromF("hazelcast:%sfoo", HazelcastConstants.INSTANCE_PREFIX)
.log("instance...")
.choice()
        .when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ADDED))
                .log("...added")
                .to("mock:added")
        .otherwise()
                .log("...removed")
                .to("mock:removed");
```

Each event provides the following information inside the message header:

Header Variables inside the response message:

Name	Type	Description
`hazelcast.listener.time`	`Long`	time of the event in millis
`hazelcast.listener.type`	`String`	the map consumer sets here "instancelistener"
`hazelcast.listener.action`	`String`	type of event - here **added** or **removed**
`hazelcast.instance.host`	`String`	host name of the instance
`hazelcast.instance.port`	`Integer`	port number of the instance

Header variables have changed in Camel 2.8

Name	Type	Description
`CamelHazelcastListenerTime`	`Long`	time of the event in millis **Version 2.8**
`CamelHazelcastListenerType`	`String`	the map consumer sets here "instancelistener" **Version 2.8**
`CamelHazelcastListenerActionn`	`String`	type of event - here **added** or **removed**. **Version 2.8**
`CamelHazelcastInstanceHost`	`String`	host name of the instance **Version 2.8**
`CamelHazelcastInstancePort`	`Integer`	port number of the instance **Version 2.8**

Using hazelcast reference

By its name

```
<bean id="hazelcastLifecycle" class="com.hazelcast.core.LifecycleService"
      factory-bean="hazelcastInstance" factory-method="getLifecycleService"
      destroy-method="shutdown" />

<bean id="config" class="com.hazelcast.config.Config">
    <constructor-arg type="java.lang.String" value="HZ.INSTANCE" />
</bean>

<bean id="hazelcastInstance" class="com.hazelcast.core.Hazelcast" factory-method="newHazelcastInstance">
    <constructor-arg type="com.hazelcast.config.Config" ref="config"/>
</bean>
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route id="testHazelcastInstanceBeanRefPut">
        <from uri="direct:testHazelcastInstanceBeanRefPut"/>
        <setHeader headerName="CamelHazelcastOperationType">
            <constant>put</constant>
        </setHeader>
        <to uri="hazelcast:map:testmap?hazelcastInstanceName=HZ.INSTANCE"/>
    </route>

    <route id="testHazelcastInstanceBeanRefGet">
        <from uri="direct:testHazelcastInstanceBeanRefGet" />
        <setHeader headerName="CamelHazelcastOperationType">
            <constant>get</constant>
        </setHeader>
        <to uri="hazelcast:map:testmap?hazelcastInstanceName=HZ.INSTANCE"/>
        <to uri="seda:out" />
    </route>
</camelContext>
```

By instance

```
<bean id="hazelcastInstance" class="com.hazelcast.core.Hazelcast"
      factory-method="newHazelcastInstance" />
<bean id="hazelcastLifecycle" class="com.hazelcast.core.LifecycleService"
      factory-bean="hazelcastInstance" factory-method="getLifecycleService"
      destroy-method="shutdown" />

<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route id="testHazelcastInstanceBeanRefPut">
        <from uri="direct:testHazelcastInstanceBeanRefPut"/>
        <setHeader headerName="CamelHazelcastOperationType">
            <constant>put</constant>
        </setHeader>
        <to uri="hazelcast:map:testmap?hazelcastInstance=#hazelcastInstance"/>
    </route>

    <route id="testHazelcastInstanceBeanRefGet">
        <from uri="direct:testHazelcastInstanceBeanRefGet" />
        <setHeader headerName="CamelHazelcastOperationType">
            <constant>get</constant>
        </setHeader>
        <to uri="hazelcast:map:testmap?hazelcastInstance=#hazelcastInstance"/>
        <to uri="seda:out" />
    </route>
</camelContext>
```

Publishing hazelcast instance as an OSGI service

If operating in an OSGI container and you would want to use one instance of hazelcast across all bundles in the same container. You can publish the instance as an OSGI service and bundles using the cache al need is to reference the service in the hazelcast endpoint.

Bundle A create an instance and publishes it as an OSGI service

```
<bean id="config" class="com.hazelcast.config.FileSystemXmlConfig">
    <argument type="java.lang.String" value="${hazelcast.config}"/>
</bean>

<bean id="hazelcastInstance" class="com.hazelcast.core.Hazelcast" factory-method="newHazelcastInstance">
    <argument type="com.hazelcast.config.Config" ref="config"/>
</bean>

<!-- publishing the hazelcastInstance as a service -->
<service ref="hazelcastInstance" interface="com.hazelcast.core.HazelcastInstance" />
```

Bundle B uses the instance

```
<!-- referencing the hazelcastInstance as a service -->
<reference ref="hazelcastInstance" interface="com.hazelcast.core.HazelcastInstance" />

<camelContext xmlns="http://camel.apache.org/schema/blueprint">
    <route id="testHazelcastInstanceBeanRefPut">
        <from uri="direct:testHazelcastInstanceBeanRefPut"/>
        <setHeader headerName="CamelHazelcastOperationType">
            <constant>put</constant>
        </setHeader>
        <to uri="hazelcast:map:testmap?hazelcastInstance=#hazelcastInstance"/>
    </route>

    <route id="testHazelcastInstanceBeanRefGet">
        <from uri="direct:testHazelcastInstanceBeanRefGet" />
        <setHeader headerName="CamelHazelcastOperationType">
            <constant>get</constant>
        </setHeader>
        <to uri="hazelcast:map:testmap?hazelcastInstance=#hazelcastInstance"/>
        <to uri="seda:out" />
    </route>
</camelContext>
```

Usage of Replicated map

Avalaible from Camel 2.16

replicatedmap cache producer - to("hazelcast:replicatedmap:foo")

A replicated map is a weakly consistent, distributed key-value data structure with no data partition. The replicatedmap producer provides 4 operations (put, get, delete, clear).

Header Variables for the request message:

Name	Type	Description
`hazelcast.operation.type`	`String`	valid values are: put, get, delete, clear
`hazelcast.objectId`	`String`	the object id to store / find your object inside the cache

Header variables have changed in Camel 2.8

Name	Type	Description
`CamelHazelcastOperationType`	`String`	valid values are: put, get, removevalue, delete **Version 2.8**
`CamelHazelcastObjectId`	`String`	the object id to store / find your object inside the cache **Version 2.8**

Sample for **put**:

Java DSL:

```
from("direct:put")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.PUT_OPERATION))
.to(String.format("hazelcast:%sbar", HazelcastConstants.REPLICATEDMAP_PREFIX));
```

Spring DSL:

```
<route>
        <from uri="direct:put" />
        <log message="put.."/>
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
        <setHeader headerName="hazelcast.operation.type">
                <constant>put</constant>
        </setHeader>
        <to uri="hazelcast:replicatedmap:foo" />
</route>
```

Sample for **get**:

Java DSL:

```
from("direct:get")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.GET_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.REPLICATEDMAP_PREFIX)
.to("seda:out");
```

Spring DSL:

```
<route>
        <from uri="direct:get" />
        <log message="get.."/>
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
        <setHeader headerName="hazelcast.operation.type">
                <constant>get</constant>
        </setHeader>
        <to uri="hazelcast:replicatedmap:foo" />
        <to uri="seda:out" />
</route>
```

Sample for **delete**:

Java DSL:

```
from("direct:delete")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.DELETE_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.REPLICATEDMAP_PREFIX);
```

Spring DSL:

```
<route>
        <from uri="direct:delete" />
        <log message="delete.."/>
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
        <setHeader headerName="hazelcast.operation.type">
                <constant>delete</constant>
        </setHeader>
        <to uri="hazelcast:replicatedmap:foo" />
</route>
```

you can call them in your test class with:

```
template.sendBodyAndHeader("direct:[put|get|delete|clear]", "my-foo", HazelcastConstants.OBJECT_ID, "4711");
```

replicatedmap cache consumer - from("hazelcast:replicatedmap:foo")

For the multimap cache this component provides the same listeners / variables as for the map cache consumer (except the update and enviction listener). The only difference is the **multimap** prefix inside the URI. Here is a sample:

```
fromF("hazelcast:%sbar", HazelcastConstants.MULTIMAP_PREFIX)
.log("object...")
.choice()
        .when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ADDED))
                .log("...added")
                .to("mock:added")
        //.when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.ENVICTED))
        //        .log("...envicted")
        //        .to("mock:envicted")
        .when(header(HazelcastConstants.LISTENER_ACTION).isEqualTo(HazelcastConstants.REMOVED))
                .log("...removed")
                .to("mock:removed")
        .otherwise()
                .log("fail!");
```

Header Variables inside the response message:

Name	Type	Description
`hazelcast.listener.time`	`Long`	time of the event in millis
`hazelcast.listener.type`	`String`	the map consumer sets here "cachelistener"
`hazelcast.listener.action`	`String`	type of event - here **added** and **removed** (and soon **envicted**)
`hazelcast.objectId`	`String`	the oid of the object
`hazelcast.cache.name`	`String`	the name of the cache - e.g. "foo"
`hazelcast.cache.type`	`String`	the type of the cache - here replicatedmap

Eviction will be added as feature, soon (this is a Hazelcast issue).

Header variables have changed in Camel 2.8

Name	Type	Description
`CamelHazelcastListenerTime`	`Long`	time of the event in millis **Version 2.8**
`CamelHazelcastListenerType`	`String`	the map consumer sets here "cachelistener" **Version 2.8**
`CamelHazelcastListenerAction`	`String`	type of event - here **added** and **removed** (and soon **envicted**) **Version 2.8**
`CamelHazelcastObjectId`	`String`	the oid of the object **Version 2.8**
`CamelHazelcastCacheName`	`String`	the name of the cache - e.g. "foo" **Version 2.8**
`CamelHazelcastCacheType`	`String`	the type of the cache - here replicatedmap **Version 2.8**

Usage of Ringbuffer

Avalaible from Camel 2.16

ringbuffer cache producer - to("hazelcast:ringbuffer:foo")

Ringbuffer is a distributed data structure where the data is stored in a ring-like structure. You can think of it as a circular array with a certain capacity. The ringbuffer producer provides 5 operations (add, readonceHead, readonceTail, remainingCapacity, capacity).

Header Variables for the request message:

Name	Type	Description
`hazelcast.operation.type`	`String`	valid values are: add, readonceHead, readonceTail, remainingCapacity, capacity

Header variables have changed in Camel 2.8

Name	Type	Description
`CamelHazelcastOperationType`	`String`	valid values are: put, get, removevalue, delete **Version 2.8**
`CamelHazelcastObjectId`	`String`	the object id to store / find your object inside the cache **Version 2.8**

Sample for **put**:

Java DSL:

```
from("direct:put")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.ADD_OPERATION))
.to(String.format("hazelcast:%sbar", HazelcastConstants.RINGBUFFER_PREFIX));
```

Spring DSL:

```
<route>
        <from uri="direct:put" />
        <log message="put.."/>
        <!-- If using version 2.8 and above set headerName to "CamelHazelcastOperationType" -->
        <setHeader headerName="hazelcast.operation.type">
                <constant>add</constant>
        </setHeader>
        <to uri="hazelcast:ringbuffer:foo" />
</route>
```

Sample for **readonce from head**:

Java DSL:

```
from("direct:get")
.setHeader(HazelcastConstants.OPERATION, constant(HazelcastConstants.READ_ONCE_HEAD_OPERATION))
.toF("hazelcast:%sbar", HazelcastConstants.RINGBUFFER_PREFIX)
.to("seda:out");
```