Simple

Simple Expression Language

The Simple Expression Language was a really simple language when it was created, but has since grown more powerful. It is primarily intended for being a really small and simple language for evaluating Expressions and Predicates without requiring any new dependencies or knowledge of XPath; so it is ideal for testing in **`camel-core`**. The idea was to cover 95% of the common use cases when you need a little bit of expression based script in your Camel routes.

However for much more complex use cases you are generally recommended to choose a more expressive and powerful language such as:

- SpEL
- Mvel
- Groovy
- JavaScript
- EL
- OGNL
- one of the supported Scripting Languages

The simple language uses **`${body}`** placeholders for complex expressions where the expression contains constant literals.

Deprecated: The **`${}`** placeholders can be omitted if the expression starts with the token, or if the token is only itself.

Alternative syntax
From Camel 2.5 you can also use the alternative syntax which uses **`$simple{}`** as placeholders. This can be used in situations to avoid clashes when using for example Spring property placeholder together with Camel.
Configuring result type
From Camel 2.8 you can configure the result type of the Simple expression. For example to set the type as a **`java.lang.Boolean`** or a **`java.lang.Integer`** etc.
File language is now merged with Simple language
From Camel 2.2, the File Language is now merged with Simple language which means you can use all the file syntax directly within the simple language.
Simple Language Changes in Camel 2.9 onwards
The Simple language have been improved from Camel 2.9 to use a better syntax parser, which can do index precise error messages, so you know exactly what is wrong and where the problem is. For example if you have made a typo in one of the operators, then previously the parser would not be able to detect this, and cause the evaluation to be true. There are a few changes in the syntax which are no longer backwards compatible. When using Simple language as a Predicate then the literal text **must** be enclosed in either single or double quotes. For example: **`"${body} == 'Camel'"`**. Notice how we have single quotes around the literal. The old style of using **`"body"`** and **`"header.foo"`** to refer to the message body and header is **`@deprecated`**, and it is encouraged to always use **`${}`** tokens for the built-in functions.
The range operator now requires the range to be in single quote as well as shown: **`"${header.zip} between '30000..39999'"`**.

To get the body of the in message: **`body`**, or **`in.body`** or **`${body}`**.

A complex expression must use **`${}`** placeholders, such as: **`Hello ${in.header.name} how are you?`**.

You can have multiple functions in the same expression: **`"Hello ${in.header.name} this is ${in.header.me} speaking"`**. However you can ***not*** nest functions in Camel 2.8.x or older e.g., having another **`${}`** placeholder in an existing, is not allowed. From **Camel 2.9** you can nest functions.

Variables

confluenceTableSmall

Variable	Type	Description
`camelId`	`String`	**Camel 2.10:** the CamelContext name.
`camelContext.OGNL`	`Object`	**Camel 2.11:** the **`CamelContext`** invoked using a Camel OGNL expression.
`collate(group)`	`List`	**Camel 2.17:** The collate function iterates the message body and groups the data into sub lists of specified size. This can be used with the Splitter EIP to split a message body and group/batch the split sub messages into a group of **`N`** sub lists. This method works similar to the collate method in Groovy.
`exchange`	`Exchange`	**Camel 2.16:** the Exchange.
`exchange.OGNL`	`Object`	**Camel 2.16:** the Exchange invoked using a Camel OGNL expression.
`exchangeId`	`String`	**Camel 2.3:** the exchange Id.
`id`	`String`	The input message Id.
`body`	`Object`	The input body.
`in.body`	`Object`	The input body.
`body.OGNL`	`Object`	**Camel 2.3:** the input body invoked using a Camel OGNL expression.
`in.body.OGNL`	`Object`	**Camel 2.3:** the input body invoked using a Camel OGNL expression.
`bodyAs(type)`	`Type`	**Camel 2.3:** Converts the body to the given type determined by its classname. The converted body can be **`null`**.

`bodyAs(type).OGNL`	`Object`	**Camel 2.18:** Converts the body to the given type determined by its classname and then invoke methods using a Camel OGNL expression. The converted body can be **`null`**.
`mandatoryBodyAs(type)`	`Type`	**Camel 2.5:** Converts the body to the given type determined by its classname, and expects the body to be not **`null`**.
`mandatoryBodyAs(type).OGNL`	`Object`	**Camel 2.18:** Converts the body to the given type determined by its classname and then invoke methods using a Camel OGNL expression.
`out.body`	`Object`	The output body.
`header.foo`	`Object`	Refer to the input **`foo`** header.
`header[foo]`	`Object`	**Camel 2.9.2:** refer to the input **`foo`** header.
`headers.foo`	`Object`	Refer to the input **`foo`** header.
`headers[foo]`	`Object`	**Camel 2.9.2:** refer to the input **`foo`** header.
`in.header.foo`	`Object`	Refer to the input **`foo`** header.
`in.header[foo]`	`Object`	**Camel 2.9.2:** refer to the input **`foo`** header.
`in.headers.foo`	`Object`	Refer to the input **`foo`** header.
`in.headers[foo]`	`Object`	**Camel 2.9.2:** refer to the input **`foo`** header.
`header.foo[bar]`	`Object`	**Camel 2.3:** regard input **`foo`** header as a map and perform lookup on the map with **`bar`** as key.
`in.header.foo[bar]`	`Object`	**Camel 2.3:** regard input **`foo`** header as a map and perform lookup on the map with **`bar`** as key.
`in.headers.foo[bar]`	`Object`	**Camel 2.3:** regard input **`foo`** header as a map and perform lookup on the map with **`bar`** as key.
`header.foo.OGNL`	`Object`	**Camel 2.3:** refer to the input **`foo`** header and invoke its value using a Camel OGNL expression.
`in.header.foo.OGNL`	`Object`	**Camel 2.3:** refer to the input **`foo`** header and invoke its value using a Camel OGNL expression.
`in.headers.foo.OGNL`	`Object`	**Camel 2.3:** refer to the input **`foo`** header and invoke its value using a Camel OGNL expression.
`out.header.foo`	`Object`	Refer to the out header **`foo`**.
`out.header[foo]`	`Object`	**Camel 2.9.2:** refer to the out header **`foo`**.
`out.headers.foo`	`Object`	Refer to the out header **`foo`**.
`out.headers[foo]`	`Object`	**Camel 2.9.2:** refer to the out header **`foo`**.
`headerAs(key,type)`	`Type`	**Camel 2.5:** Converts the header to the given type determined by its classname.
`headers`	`Map`	**Camel 2.9:** refer to the input headers.
`in.headers`	`Map`	**Camel 2.9:** refer to the input headers.
`property.foo`	`Object`	**Deprecated:** refer to the **`foo`** property on the exchange.
`exchangeProperty.foo`	`Object`	**Camel 2.15:** refer to the **`foo`** property on the exchange.
`property[foo]`	`Object`	**Deprecated:** refer to the **`foo`** property on the exchange.
`exchangeProperty[foo]`	`Object`	**Camel 2.15:** refer to the **`foo`** property on the exchange.
`property.foo.OGNL`	`Object`	**Deprecated:** refer to the **`foo`** property on the exchange and invoke its value using a Camel OGNL expression.
`exchangeProperty.foo.OGNL`	`Object`	**Camel 2.15:** refer to the **`foo`** property on the exchange and invoke its value using a Camel OGNL expression.
`sys.foo`	`String`	Refer to the system property **`foo`**.
`sysenv.foo`	`String`	**Camel 2.3:** refer to the system environment property **`foo`**.
`exception`	`Object`	**Camel 2.4:** Refer to the exception object on the exchange, is **`null`** if no exception set on exchange. Will fallback and grab caught exceptions (**`Exchange.EXCEPTION_CAUGHT`**) if the Exchange has any.
`exception.OGNL`	`Object`	**Camel 2.4:** Refer to the exchange exception invoked using a Camel OGNL expression object
`exception.message`	`String`	Refer to the exception.message on the exchange, is **null** if no exception set on exchange. Will fallback and grab caught exceptions (**`Exchange.EXCEPTION_CAUGHT`**) if the Exchange has any.
`exception.stacktrace`	`String`	**Camel 2.6.** Refer to the **`exception.stacktrace`** on the exchange. Result is **`null`** if no exception set on exchange. Will fallback and grab caught exceptions (**`Exchange.EXCEPTION_CAUGHT`**) if the Exchange has any.

`date:`*`command`*`:`*`pattern`*	`String`	Date formatting using the **`java.text.SimpleDateFormat`** patterns. Supported commands are: • **`now`** for current timestamp. • **`in.header.xxx`** or **`header.xxx`** to use the **`Date`** object in the **`IN`** header with the key **`xxx`**. • **`out.header.xxx`** to use the **`Date`** object in the **`OUT`** header with the key **`xxx`**.
`bean:`*`bean expression`*	`Object`	Invoking a bean expression using the Bean language. Specifying a method name you must use dot as separator. We also support the **`?method=methodname`** syntax that is used by the Bean component.
`properties:`*`locations`*`:key`	`String`	**Deprecated: (use properties-location instead) Camel 2.3:** Lookup a property with the given key. The **`locations`** option is optional. See more at Using PropertyPlaceholder.
`properties-location:`*`locations:key`*	`String`	**Camel 2.14.1:** Lookup a property with the given key. The `locations` option is optional. See more at Using PropertyPlaceholder.
`properties:key:default`	`String`	**Camel 2.14.1**: Lookup a property with the given key. If the key does not exists or has no value, then an optional default value can be specified.
`routeId`	`String`	**Camel 2.11:** Returns the Id of the current route the Exchange is being routed.
`threadName`	`String`	**Camel 2.3:** Returns the name of the current thread. Can be used for logging purpose.
`ref:`*`xxx`*	`Object`	**Camel 2.6:** To lookup a bean from the Registry with the given Id.
`type:name.field`	`Object`	**Camel 2.11:** To refer to a type or field by its FQN name. To refer to a field you can append **`.FIELD_NAME`**. For example you can refer to the constant field from Exchange as: **`org.apache.camel.Exchange.FILE_NAME`**
`null`	`null`	**Camel 2.12.3:** represents a **`null`.**
`random(`*`value`*`)`	`Integer`	**Camel 2.16.0:** returns a random Integer between ***`0`*** (included) and ***`value`*** (excluded)
`random(`*`min`*`,`*`max`*`)`	`Integer`	**Camel 2.16.0:** returns a random Integer between ***`min`*** (included) and ***`max`*** (excluded)
`skip(number)`	`Iterator`	**Camel 2.19:** The skip function iterates the message body and skips the first number of items. This can be used with the Splitter EIP to split a message body and skip the first N number of items.
`messageHistory`	`String`	**Camel 2.17:** The message history of the current exchange how it has been routed. This is similar to the route stack-trace message history the error handler logs in case of an unhandled exception.
`messageHistory(false)`	`String`	**Camel 2.17:** As **`messageHistory`** but without the exchange details (only includes the route strack-trace). This can be used if you do not want to log sensitive data from the message itself.

OGNL expression support

Available as of Camel 2.3

Camel's OGNL support is for invoking methods only. You cannot access fields.
From **Camel 2.11.1**: we added special support for accessing the length field of Java arrays.

The Simple and Bean language now supports a Camel OGNL notation for invoking beans in a chain like fashion. Suppose the Message **`IN`** body contains a POJO which has a **`getAddress()`** method.

Then you can use Camel OGNL notation to access the address object:

javasimple("${body.address}") simple("${body.address.street}") simple("${body.address.zip}")

Camel understands the shorthand names for accessors, but you can invoke any method or use the real name such as:

javasimple("${body.address}") simple("${body.getAddress.getStreet}") simple("${body.address.getZip}") simple("${body.doSomething}")

You can also use the null safe operator (**`?.`**) to avoid a NPE if for example the body does *not* have an address

javasimple("${body?.address?.street}")

It is also possible to index in **`Map`** or **`List`** types, so you can do:

javasimple("${body[foo].name}")

To assume the body is **`Map`** based and lookup the value with **`foo`** as key, and invoke the **`getName`** method on that value.

key with spaces
If the key has space, then you ***`must`*** enclose the key with quotes, for example:

javasimple("${body['foo bar'].name}")

You can access the **`Map`** or **`List`** objects directly using their key name (with or without dots) :

javasimple("${body[foo]}") simple("${body[this.is.foo]}")

Suppose there was no value with the key **`foo`** then you can use the null safe operator to avoid a NPE as shown:

javasimple("${body[foo]?.name}")

You can also access **`List`** types, for example to get lines from the address you can do:

javasimple("${body.address.lines[0]}") simple("${body.address.lines[1]}") simple("${body.address.lines[2]}")

There is a special **`last`** keyword which can be used to get the last value from a list.

javasimple("${body.address.lines[last]}")

And to get the penultimate line use subtraction. In this case use **`last-1`** for this:

javasimple("${body.address.lines[last-1]}")

And the third last is of course:

javasimple("${body.address.lines[last-2]}")

And you can call the **`size`** method on the list with

javasimple("${body.address.lines.size}")

From **Camel 2.11.1** we added support for the length field for Java arrays as well. Example:

javaString[] lines = new String[]{"foo", "bar", "cat"}; exchange.getIn().setBody(lines); simple("There are ${body.length} lines")

And yes you can combine this with the operator support as shown below:

javasimple("${body.address.zip} > 1000")

Operator Support

The parser is limited to only support a single operator. To enable it the left value must be enclosed in **`${}`**.

The syntax is:

java${leftValue} OP rightValue

Where the **`rightValue`** can be a **`String`** literal enclosed in **`' '`**, **`null`**, a constant value or another expression enclosed in **`${}`**.

Important
There ***must*** be spaces around the operator.

Camel will automatically type convert the **`rightValue`** type to the **`leftValue`** type, so it is possible to for example, convert a string into a numeric so you can use **>** comparison for numeric values.

The following operators are supported:

Operator	Description
`==`	Equals.
`=~`	**Camel 2.16:** equals ignore case (will ignore case when comparing **`String`** values).
`>`	Greater than.
`>=`	Greater than or equals.
`<`	Less than.
`<=`	Less than or equals.
`!=`	Not equals.
`contains`	For testing if contains in a string based value.
`not contains`	For testing if not contains in a string based value.
`regex`	For matching against a given regular expression pattern defined as a **`String`** value.
`not regex`	For not matching against a given regular expression pattern defined as a **`String`** value.
`in`	For matching if in a set of values, each element must be separated by comma. If you want to include an empty value, then it must be defined using double comma, eg ',,bronze,silver,gold', which is a set of four values with an empty value and then the three medals.

`not in`	For matching if not in a set of values, each element must be separated by comma. If you want to include an empty value, then it must be defined using double comma. Example: **`',,bronze,silver,gold'`**, which is a set of four values with an empty value and then the three medals.
`is`	For matching if the left hand side type is an **`instanceof`** the value.
`not is`	For matching if the left hand side type is not an **`instanceof`** the value.
`range`	For matching if the left hand side is within a range of values defined as numbers: **`from..to`**. From **Camel 2.9**: the range values must be enclosed in single quotes.
`not range`	For matching if the left hand side is not within a range of values defined as numbers: **`from..to`**. From **Camel 2.9**: the range values must be enclosed in single quotes.
`starts with`	**Camel 2.17.1, 2.18**: For testing if the left hand side string starts with the right hand string.
`ends with`	**Camel 2.17.1, 2.18**: For testing if the left hand side string ends with the right hand string.

And the following unary operators can be used:

Operator	Description
`++`	**Camel 2.9:** To increment a number by one. The left hand side must be a function, otherwise parsed as literal.
`--`	**Camel 2.9:** To decrement a number by one. The left hand side must be a function, otherwise parsed as literal.
`\`	**Camel 2.9.3 to 2.10.x** To escape a value, e.g., **`\$`**, to indicate a **$** sign. Special: Use **`\n`** for new line, **`\t`** for tab, and **`\r`** for carriage return. **Note:** Escaping is **not** supported using the File Language. **Note:** from Camel 2.11, *the escape character is no longer supported*. It has been replaced with the following three escape sequences.
`\n`	**Camel 2.11:** To use newline character.
`\t`	**Camel 2.11:** To use tab character.
`\r`	**Camel 2.11:** To use carriage return character.
`\}`	**Camel 2.18:** To use the **`}`** character as text.

And the following logical operators can be used to group expressions:

Operator	Description
`and`	**Deprecated**: use **`&&`** instead. The logical and operator is used to group two expressions.
`or`	**Deprecated**: use **`||`** instead. The logical or operator is used to group two expressions.
`&&`	**Camel 2.9:** The logical and operator is used to group two expressions.
`||`	**Camel 2.9:** The logical or operator is used to group two expressions.

Using and,or operators
In **Camel 2.4 and older** the **and** or **`or`** can only be used **once** in a simple language expression.

From **Camel 2.5**: you can use these operators multiple times.

The syntax for **`AND`** is:

java${leftValue} OP rightValue and ${leftValue} OP rightValue

And the syntax for **`OR`** is:

java${leftValue} OP rightValue or ${leftValue} OP rightValue

Some examples:

java// exact equals match simple("${in.header.foo} == 'foo'") // ignore case when comparing, so if the header has value FOO this will match simple("${in.header.foo} =~ 'foo'") // here Camel will type convert '100' into the type of in.header.bar and if it is an Integer '100' will also be converter to an Integer simple("${in.header.bar} == '100'") simple("${in.header.bar} == 100") // 100 will be converter to the type of in.header.bar so we can do > comparison simple("${in.header.bar} > 100") Comparing with different types
When you compare with different types such as **`String`** and **`int`**, then you have to take a bit care. Camel will use the type from the left hand side as first priority. And fallback to the right hand side type if both values couldn't be compared based on that type. This means you can flip the values to enforce a specific type. Suppose the bar value above is a **`String`**. Then you can flip the equation:

javasimple("100 < ${in.header.bar}")

which then ensures the **`int`** type is used as first priority.

This may change in the future if the Camel team improves the binary comparison operations to prefer numeric types over **`String`** based. It's most often the **`String`** type which causes problem when comparing with numbers.
java// testing for null simple("${in.header.baz} == null") // testing for not null simple("${in.header.baz} != null")

And a bit more advanced example where the right value is another expression,

javasimple("${in.header.date} == ${date:now:yyyyMMdd}") simple("${in.header.type} == ${bean:orderService?method=getOrderType}")

And an example with contains, testing if the title contains the word Camel:

javasimple("${in.header.title} contains 'Camel'")

And an example with regex, testing if the number header is a four digit value:

javasimple("${in.header.number} regex '\\d{4}'")

And finally an example if the header equals any of the values in the list. Each element must be separated by comma, and no space around. This also works for numbers etc, as Camel will convert each element into the type of the left hand side.

javasimple("${in.header.type} in 'gold,silver'")

And for all the last three we also support the negate test using **`not`**:

javasimple("${in.header.type} not in 'gold,silver'")

And you can test if the type is a certain instance, e.g., for instance a **`String`**:

javasimple("${in.header.type} is 'java.lang.String'")

We have added a shorthand for all **`java.lang`** types so you can write it as:

javasimple("${in.header.type} is 'String'")

Ranges are also supported. The range interval requires numbers and both from and end are inclusive. For instance to test whether a value is between **`100`** and **`199:`**

javasimple("${in.header.number} range 100..199")

Notice we use **`..`** in the range without spaces. It is based on the same syntax as Groovy.

From **Camel 2.9**: the range value must be in single quotes:

javasimple("${in.header.number} range '100..199'") Can be used in Spring XML
As the Spring XML does not have all the power as the Java DSL with all its various builder methods, you have to resort to use some other languages for testing with simple operators. Now you can do this with the simple language. In the sample below we want to test if the header is a widget order:

xml<from uri="seda:orders"> <filter> <simple>${in.header.type} == 'widget'</simple> <to uri="bean:orderService?method=handleWidget"/> </filter> < /from>

Using and / or

If you have two expressions you can combine them with the **`and`** or **`or`** operator.

Camel 2.9 onwards
Use **`&&`** or **`||`**

For instance:

javasimple("${in.header.title} contains 'Camel' and ${in.header.type'} == 'gold'")

And of course the **`or`** is also supported. The sample would be:

javasimple("${in.header.title} contains 'Camel' or ${in.header.type'} == 'gold'")

Note: currently **`and`** or **`or`** can only be used ***once*** in a simple language expression. This might change in the future. So you **cannot** do:

javasimple("${in.header.title} contains 'Camel' and ${in.header.type'} == 'gold' and ${in.header.number} range 100..200")

Samples

In the Spring XML sample below we filter based on a header value:

xml<from uri="seda:orders"> <filter> <simple>${in.header.foo}</simple> <to uri="mock:fooOrders"/> </filter> </from>

The Simple language can be used for the predicate test above in the Message Filter pattern, where we test if the in message has a **`foo`** header (a header with the key **`foo`** exists). If the expression evaluates to **`true`** then the message is routed to the **`mock:fooOrders`** endpoint, otherwise it is lost in the deep blue sea 😷.

The same example in Java DSL:

javafrom("seda:orders") .filter().simple("${in.header.foo}") .to("seda:fooOrders");

You can also use the simple language for simple text concatenations such as:

javafrom("direct:hello") .transform().simple("Hello ${in.header.user} how are you?") .to("mock:reply");

Notice that we must use **`${}`** placeholders in the expression now to allow Camel to parse it correctly.

And this sample uses the date command to output current date.

javafrom("direct:hello") .transform().simple("The today is ${date:now:yyyyMMdd} and it is a great day.") .to("mock:reply");

And in the sample below we invoke the bean language to invoke a method on a bean to be included in the returned string:

javafrom("direct:order") .transform().simple("OrderId: ${bean:orderIdGenerator}") .to("mock:reply");

Where **`orderIdGenerator`** is the id of the bean registered in the Registry. If using Spring then it is the Spring bean id.

If we want to declare which method to invoke on the order id generator bean we must prepend **`.method name`** such as below where we invoke the **`generateId`** method.

javafrom("direct:order") .transform().simple("OrderId: ${bean:orderIdGenerator.generateId}") .to("mock:reply");

We can use the **`?method=methodname`** option that we are familiar with the Bean component itself:

javafrom("direct:order") .transform().simple("OrderId: ${bean:orderIdGenerator?method=generateId}") .to("mock:reply");

From **Camel 2.3**: you can also convert the body to a given type, for example to ensure that it is a **`String`** you can do:

xml<transform> <simple>Hello ${bodyAs(String)} how are you?</simple> </transform>

There are a few types which have a shorthand notation, so we can use **`String`** instead of **`java.lang.String`**. These are: **`byte[]`**, **`String`**, **`Integer`**, **`Long`**. All other types must use their FQN name, e.g. **`org.w3c.dom.Document`**.

It is also possible to lookup a value from a header **`Map`** in **Camel 2.3**:

xml<transform> <simple>The gold value is ${header.type[gold]}</simple> </transform>

In the code above we lookup the header with name `type` and regard it as a **`java.util.Map`** and we then lookup with the key **`gold`** and return the value. If the header is not convertible to Map an exception is thrown. If the header with name **`type`** does not exist **`null`** is returned.

From **Camel 2.9**: you can nest functions, such as shown below:

xml<setHeader headerName="myHeader"> <simple>${properties:${header.someKey}}</simple> </setHeader>

Referring to Constants or Enums

Available from Camel 2.11

Suppose you have an enum for customers:{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/Customer.java}And in a Content Based Router we can use the Simple language to refer to this enum, to check the message which enum it matches.{snippet:id=e1|lang=java|url=camel/trunk/camel-core/src/test/java/org/apache/camel/processor/CBRSimpleTypeTest.java}

Using New Lines or Tabs in XML DSLs

Available from Camel 2.9.3

From **Camel 2.9.3**: it is easier to specify new lines or tabs in XML DSLs as you can escape the value now

xml<transform> <simple>The following text\nis on a new line</simple> </transform>

Leading and Trailing Whitespace Handling

Available from Camel 2.10.0

From **Camel 2.10.0**: the **`trim`** attribute of the expression can be used to control whether the leading and trailing whitespace characters are removed or preserved. The default of **`trim=true`** removes all whitespace characters.

xml<setBody> <simple trim="false">You get some trailing whitespace characters. </simple> </setBody>

Setting the Result Type

Available from Camel 2.8

You can now provide a result type to the Simple expression, which means the result of the evaluation will be converted to the desired type. This is most usable to define types such as `boolean`'s, `integer`'s, etc.

For example to set a header as a **`boolean`** type you can do:

.setHeader("cool", simple("true", Boolean.class))

And in XML DSL

xml<setHeader headerName="cool"> <!-- use resultType to indicate that the type should be a java.lang.Boolean --> <simple resultType="java.lang. Boolean">true</simple> </setHeader>

Changing Function Start and End Tokens

Available from Camel 2.9.1

You can configure the function start and end tokens - **`${}`** using the setters **`changeFunctionStartToken`** and **`changeFunctionEndToken`** on **`Simple Language`**, using Java code. From Spring XML you can define a **`<bean>`** tag with the new changed tokens in the properties as shown below:

xml<!-- configure Simple to use custom prefix/suffix tokens --> <bean id="simple" class="org.apache.camel.language.simple.SimpleLanguage"> <property name="functionStartToken" value="["/> <property name="functionEndToken" value="]"/> </bean>

In the example above we use **`[]`** as the changed tokens. Notice by changing the start/end token you change those in all the Camel applications which share the same **`camel-core`** on their classpath. For example in an OSGi server this may affect many applications, where as a Web Application as a WAR file it only affects the Web Application.

Loading Script from External Resource

Available from Camel 2.11

You can externalize the script and have Camel load it from a resource such as: **`classpath:`**, **`file:`**, or **`http:`**. This is done using the following syntax: **`resource:scheme:location`**, e.g., to refer to a file on the classpath you can do:

java.setHeader("myHeader").simple("resource:classpath:mysimple.txt")

Setting Spring beans to Exchange properties

Available from Camel 2.6

You can set a spring bean into an exchange property as shown below:

xml<bean id="myBeanId" class="my.package.MyCustomClass"/> <route> <!-- ... --> <setProperty propertyName="monitoring.message"> <simple>ref: myBeanId</simple> </setProperty> <!-- ... --> </route>

Dependencies

The Simple language is part of **`camel-core`**.