Using PropertyPlaceholder

Using `PropertyPlaceholder`

Available as of Camel 2.3

Camel now provides a new **`PropertiesComponent`** in **`camel-core`** which allows you to use property placeholders when defining Camel Endpoint URIs. This works much like you would do if using Spring's **`<property-placeholder>`** tag. However Spring has a limitation that prevents third-party frameworks from fully leveraging Spring property placeholders.

For more details see: How do I use Spring Property Placeholder with Camel XML.

Bridging Spring and Camel Property Placeholders
From **Camel 2.10**: Spring's property placeholder can be bridged with Camel's. See below for more details.

The property placeholder is typically used when trying to do any of the following:

- Lookup or creating endpoints.
- Lookup of beans in the Registry.
- Additional supported in Spring XML (see below in examples).
- Using Blueprint **`PropertyPlaceholder`** with Camel Properties component.
- Using **`@PropertyInject`** to inject a property in a POJO.
- **Camel 2.14.1** Using default value if a property does not exists.
- **Camel 2.14.1** Include out of the box functions, to lookup property values from OS environment variables, JVM system properties, or the service idiom.
- **Camel 2.14.1** Using custom functions, which can be plugged into the property component.

Format

The value of a Camel property can be obtained by specifying its key name within a property placeholder, using the following format: **`{{key}}`**.

For example:

{{file.uri}}

where **`file.uri`** is the property key.

Property placeholders can be used to specify parts, or all, of an endpoint's URI by embedding one or more placeholders in the URI's string definition.

From **Camel 2.14.1**: you can specify a default value to use if a property with the key does not exists, e.g., **`file.url:/some/path`** where the default value is the text after the colon, e.g., **`/some/path`**.

From **Camel 2.14.1**: do *not* use a colon in the property key. The colon character is used as a token separator when providing a default value.

Using `PropertyResolver`

Camel provides a pluggable mechanism that allows third-parties to specify their own resolver to use for the lookup of properties.

Camel provides a default implementation **`org.apache.camel.component.properties.DefaultPropertiesResolver`** which is capable of loading properties from the file system, classpath or Registry. To indicate which source to use the location must contain the appropriate prefix.

The list of prefixes is:

Prefix	Description
`ref:`	Lookup in the Registry.
`file:`	Load the from file system.
`classpath:`	Load from the classpath (this is also the default if no prefix is provided).
`blueprint:`	Use a specific OSGi blueprint placeholder service.

Defining Location

The **`PropertiesResolver`** must be configured with the location(s) to use when resolving properties. One or more locations can be given. Specifying multiple locations can be done a couple of ways: using either a single comma separated string, or an array of strings.

javapc.setLocation("com/mycompany/myprop.properties,com/mycompany/other.properties"); pc.setLocation(new String[] {"com/mycompany/myprop.properties", "com/mycompany/other.properties"});
From **Camel 2.19.0**: you can set which location can be discarded if missing by setting **`optional=true`**, (**`false`** by default).

Example:

javapc.setLocations("com/mycompany/override.properties;optional=true,com/mycompany/defaults.properties");

Using System and Environment Variables in Locations

Available as of Camel 2.7

The location now supports using placeholders for JVM system properties and OS environments variables.

Example:

location=file:${karaf.home}/etc/foo.properties

In the location above we defined a location using the file scheme using the JVM system property with key **`karaf.home`**.

To use an OS environment variable instead you would have to prefix with **`env`**:

location=file:${env:APP_HOME}/etc/foo.properties

Where **`APP_HOME`** is an OS environment variable.

You can have multiple placeholders in the same location, such as:

location=file:${env:APP_HOME}/etc/${prop.name}.properties

Using System or Environment Variables to Configure Property Prefixes and Suffixes

From **Camel 2.12.5, 2.13.3, 2.14.0**: **`propertyPrefix`**, **`propertySuffix`** configuration properties support the use of placeholders for de-referencing JVM system properties and OS environments variables.

Example:

Assume the **`PropertiesComponent`** is configured with the following properties file:

textdev.endpoint = result1 test.endpoint = result2

The same properties file is then referenced from a route definition:

javaPropertiesComponent pc = context.getComponent("properties", PropertiesComponent.class); pc.setPropertyPrefix("${stage}."); // ... context.addRoutes (new RouteBuilder() { @Override public void configure() throws Exception { from("direct:start") .to("properties:mock:{{endpoint}}"); } });

By using the configuration options **`propertyPrefix`** it's possible to change the target endpoint simply by changing the value of the system property **`stage`** either to **`dev`** (the message will be routed to **`mock:result1`**) or **`test`** (the message will be routed to **`mock:result2`**).

Configuring in Java DSL

You have to create and register the **`PropertiesComponent`** under the name **`properties`** such as:

javaPropertiesComponent pc = new PropertiesComponent(); pc.setLocation("classpath:com/mycompany/myprop.properties"); context.addComponent ("properties", pc);

Configuring in Spring XML

Spring XML offers two variations to configure. You can define a spring bean as a **`PropertiesComponent`** which resembles the way done in Java DSL. Or you can use the **`<propertyPlaceholder>`** tag.

xml<bean id="properties" class="org.apache.camel.component.properties.PropertiesComponent"> <property name="location" value="classpath:com /mycompany/myprop.properties"/> </bean>

Using the **`<propertyPlaceholder>`** tag makes the configuration a bit more fresh such as:

xml<camelContext ...> <propertyPlaceholder id="properties" location="com/mycompany/myprop.properties"/> </camelContext>

Setting the properties location through the location tag works just fine but sometime you have a number of resources to take into account and starting from **Camel 2.19.0** you can set the properties location with a dedicated **`propertiesLocation`**:

xml<camelContext ...> <propertyPlaceholder id="myPropertyPlaceholder"> <propertiesLocation resolver = "classpath" path = "com/my/company/something /my-properties-1.properties" optional = "false"/> <propertiesLocation resolver = "classpath" path = "com/my/company/something/my-properties-2. properties" optional = "false"/> <propertiesLocation resolver = "file" path = "${karaf.home}/etc/my-override.properties" optional = "true"/> < /propertyPlaceholder> </camelContext>Specifying the cache option in XML
From **Camel 2.10**: Camel supports specifying a value for the **`cache`** option both inside the Spring as well as the Blueprint XML.

Using a Properties from the Registry

Available as of Camel 2.4
For example in OSGi you may want to expose a service which returns the properties as a **`java.util.Properties`** object.

Then you could setup the Properties component as follows:

xml<propertyPlaceholder id="properties" location="ref:myProperties"/>

Where **`myProperties`** is the id to use for lookup in the OSGi registry. Notice we use the **`ref:`** prefix to tell Camel that it should lookup the properties for the Registry.

Examples Using Properties Component

When using property placeholders in the endpoint URIs you can either use the **`properties:`** component or define the placeholders directly in the URI. We will show example of both cases, starting with the former.

java// properties cool.end=mock:result // route from("direct:start") .to("properties:{{cool.end}}");

You can also use placeholders as a part of the endpoint URI:

java// properties cool.foo=result // route from("direct:start") .to("properties:mock:{{cool.foo}}");

In the example above the to endpoint will be resolved to **`mock:result`**.

You can also have properties with refer to each other such as:

java// properties cool.foo=result cool.concat=mock:{{cool.foo}} // route from("direct:start") .to("properties:mock:{{cool.concat}}");

Notice how **`cool.concat`** refer to another property.

The **`properties:`** component also offers you to override and provide a location in the given URI using the **`locations`** option:

javafrom("direct:start") .to("properties:bar.end?locations=com/mycompany/bar.properties");

Examples

You can also use property placeholders directly in the endpoint URIs without having to use **`properties:`**.

java// properties cool.foo=result // route from("direct:start") .to("mock:{{cool.foo}}");

And you can use them in multiple wherever you want them:

java// properties cool.start=direct:start cool.showid=true cool.result=result // route from("{{cool.start}}") .to("log:{{cool.start}}?showBodyType=false&showExchangeId={{cool.showid}}") .to("mock:{{cool.result}}");

You can also your property placeholders when using ProducerTemplate for example:

javatemplate.sendBody("{{cool.start}}", "Hello World");

Example with Simple language

The Simple language now also support using property placeholders, for example in the route below:

java// properties cheese.quote=Camel rocks // route from("direct:start") .transform().simple("Hi ${body} do you think ${properties:cheese.quote}?");

You can also specify the location in the Simple language for example:

java// bar.properties bar.quote=Beer tastes good // route from("direct:start") .transform().simple("Hi ${body}. ${properties:com/mycompany/bar.properties:bar.quote}.");

Additional Property Placeholder Support in Spring XML

The property placeholders is also supported in many of the Camel Spring XML tags such as **`<package>`**, **`<packageScan>`**, **`<contextScan>`**, **`<jmxAgent>`**, **`<endpoint>`**, **`<routeBuilder>`**, **`<proxy>`** and the others.

Example:

xmlUsing property placeholders in the <jmxAgent> tag<camelContext xmlns="http://camel.apache.org/schema/spring"> <propertyPlaceholder id="properties" location="org/apache/camel/spring/jmx.properties"/> <!-- we can use propery placeholders when we define the JMX agent --> <jmxAgent id="agent" registryPort="{{myjmx.port}}" disabled="{{myjmx.disabled}}" usePlatformMBeanServer="{{myjmx.usePlatform}}" createConnector="true" statisticsLevel="RoutesOnly" useHostIPAddress="true"/> <route id="foo" autoStartup="false"> <from uri="seda:start"/> <to uri="mock:result"/> </route> </camelContext>

Example:

xmlUsing property placeholders in the attributes of <camelContext><camelContext trace="{{foo.trace}}" xmlns="http://camel.apache.org/schema/spring"> <propertyPlaceholder id="properties" location="org/apache/camel/spring/processor/myprop.properties"/> <template id="camelTemplate" defaultEndpoint="{{foo.cool}}"/> <route> <from uri="direct:start"/> <setHeader headerName="{{foo.header}}"> <simple>${in.body} World!</simple> </setHeader> <to uri="mock:result"/> </route> </camelContext>

Overriding a Property Setting Using a JVM System Property

Available as of Camel 2.5
It is possible to override a property value at runtime using a JVM System property without the need to restart the application to pick up the change. This may also be accomplished from the command line by creating a JVM System property of the same name as the property it replaces with a new value.

Example:

javaPropertiesComponent pc = context.getComponent("properties", PropertiesComponent.class); pc.setCache(false); System.setProperty("cool.end", "mock:override"); System.setProperty("cool.result", "override"); context.addRoutes(new RouteBuilder() { @Override public void configure() throws Exception { from("direct:start").to("properties:cool.end"); from("direct:foo").to("properties:mock:{{cool.result}}"); } }); context.start(); getMockEndpoint("mock:override").expectedMessageCount(2); template.sendBody("direct:start", "Hello World"); template.sendBody("direct:foo", "Hello Foo"); System.clearProperty("cool.end"); System.clearProperty("cool.result"); assertMockEndpointsSatisfied();

Using Property Placeholders for Any Kind of Attribute in the XML DSL

Available as of Camel 2.7

If you use OSGi Blueprint then this only works from **2.11.1** or **2.10.5** on.

Previously it was only the **`xs:string`** type attributes in the XML DSL that support placeholders. For example often a timeout attribute would be a **`xs:int`** type and thus you cannot set a string value as the placeholder key. This is now possible from Camel 2.7 on using a special placeholder namespace.

In the example below we use the **`prop`** prefix for the namespace `http://camel.apache.org/schema/placeholder` by which we can use the **`prop`** prefix in the attributes in the XML DSLs. Notice how we use that in the Multicast to indicate that the option **`stopOnException`** should be the value of the placeholder with the key **`stop`**.

xml<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:prop="http://camel.apache.org/schema/placeholder" xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd"> <!-- Notice in the declaration above, we have defined the prop prefix as the Camel placeholder namespace --> <bean id="damn" class="java.lang.IllegalArgumentException"> <constructor-arg index="0" value="Damn"/> </bean> <camelContext xmlns="http://camel.apache.org/schema/spring"> <propertyPlaceholder id="properties" location="classpath:org/apache/camel/component/properties/myprop.properties" xmlns="http://camel.apache.org/schema/spring"/> <route> <from uri="direct:start"/> <!-- use prop namespace, to define a property placeholder, which maps to option stopOnException={{stop}} --> <multicast prop:stopOnException="stop"> <to uri="mock:a"/> <throwException ref="damn"/> <to uri="mock:b"/> </multicast> </route> </camelContext> </beans>

In our properties file we have the value defined as

stop=true

Using Property Placeholder in the Java DSL

Available as of Camel 2.7

Likewise we have added support for defining placeholders in the Java DSL using the new **`placeholder`** DSL as shown in the following equivalent example:

javafrom("direct:start") // use a property placeholder for the option stopOnException on the Multicast EIP // which should have the value of {{stop}} key being looked up in the properties file .multicast() .placeholder("stopOnException", "stop") .to("mock:a") .throwException(new IllegalAccessException("Damn")) .to("mock:b");

Using Blueprint Property Placeholder with Camel Routes

Available as of Camel 2.7

Camel supports Blueprint which also offers a property placeholder service. Camel supports convention over configuration, so all you have to do is to define the OSGi Blueprint property placeholder in the XML file as shown below:

xml<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.0.0" xsi:schemaLocation="http://www.osgi.org/xmlns/blueprint/v1.0.0 https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd"> <!-- OSGI blueprint property placeholder --> <cm:property-placeholder id="myblueprint.placeholder" persistent-id="camel.blueprint"> <!-- list some properties as needed --> <cm:default-properties> <cm:property name="result" value="mock:result"/> </cm:default-properties> </cm:property-placeholder> <camelContext xmlns="http://camel.apache.org/schema/blueprint"> <!-- in the route we can use {{ }} placeholders which will lookup in blueprint as Camel will auto detect the OSGi blueprint property placeholder and use it --> <route> <from uri="direct:start"/> <to uri="mock:foo"/> <to uri="{{result}}"/> </route> </camelContext> </blueprint>

By default Camel detects and uses OSGi blueprint property placeholder service. You can disable this by setting the attribute **`useBlueprintPropertyResolver`** to false on the **`<camelContext>`** definition.

About placeholder syntaxes

Notice how we can use the Camel syntax for placeholders **{{ }}** in the Camel route, which will lookup the value from OSGi blueprint.
The blueprint syntax for placeholders is **${}**. So outside the **<camelContext>** you must use the **${}** syntax. Where as inside **<camelContext>** you must use **{{ }}** syntax. OSGi blueprint allows you to configure the syntax, so you can actually align those if you want.

You can also explicit refer to a specific OSGi blueprint property placeholder by its id. For that you need to use the Camel's **<propertyPlaceholder>** as shown in the example below:

xml<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.0.0" xsi:schemaLocation="http://www.osgi.org/xmlns/blueprint/v1.0.0 https://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd"> <!-- OSGI blueprint property placeholder --> <cm:property-placeholder id="myblueprint.placeholder" persistent-id="camel.blueprint"> <!-- list some properties as needed --> <cm:default-properties> <cm:property name="prefix.result" value="mock:result"/> </cm:default-properties> </cm:property-placeholder> <camelContext xmlns="http://camel.apache.org/schema/blueprint"> <!-- using Camel properties component and refer to the blueprint property placeholder by its id --> <propertyPlaceholder id="properties" location="blueprint:myblueprint.placeholder" prefixToken="[[" suffixToken="]]" propertyPrefix="prefix."/> <!-- in the route we can use {{ }} placeholders which will lookup in blueprint --> <route> <from uri="direct:start"/> <to uri="mock:foo"/> <to uri="[[result]]"/> </route> </camelContext> </blueprint>

Notice how we use the **blueprint** scheme to refer to the OSGi blueprint placeholder by its id. This allows you to mix and match, for example you can also have additional schemes in the location. For example to load a file from the classpath you can do:

location="blueprint:myblueprint.placeholder,classpath:myproperties.properties"

Each location is separated by comma.

Overriding Blueprint Property Placeholders Outside CamelContext

Available as of Camel 2.10.4

When using Blueprint property placeholder in the Blueprint XML file, you can declare the properties directly in the XML file as shown below:{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-test-blueprint/src/test/resources/org/apache/camel/test/blueprint/configadmin-outside.xml}Notice that we have a **<bean>** which refers to one of the properties. And in the Camel route we refer to the other using the **{{ }}** notation.

Now if you want to override these Blueprint properties from an unit test, you can do this as shown below:{snippet:id=e1|lang=java|url=camel/trunk/components/camel-test-blueprint/src/test/java/org/apache/camel/test/blueprint/ConfigAdminOverridePropertiesOutsideCamelContextTest.java}To do this we override and implement the **useOverridePropertiesWithConfigAdmin** method. We can then put the properties we want to override on the given props parameter. And the return value *must* be the persistence-id of the **<cm:property-placeholder>** tag, which you define in the blueprint XML file.

Using a `.cfg` or `.properties` File For Blueprint Property Placeholders

Available as of Camel 2.10.4

When using Blueprint property placeholder in the Blueprint XML file, you can declare the properties in a .properties or **.cfg** file. If you use Apache ServiceMix/Karaf then this container has a convention that it loads the properties from a file in the etc directory with the naming **etc/pid.cfg**, where **pid** is the persistence-id.

For example in the blueprint XML file we have the **persistence-id="stuff"**, which mean it will load the configuration file as **etc/stuff.cfg**.{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-test-blueprint/src/test/resources/org/apache/camel/test/blueprint/configadmin-loadfile.xml}Now if you want to unit test this blueprint XML file, then you can override the **loadConfigAdminConfigurationFile** and tell Camel which file to load as shown below:{snippet:id=e1|lang=java|url=camel/trunk/components/camel-test-blueprint/src/test/java/org/apache/camel/test/blueprint/ConfigAdminLoadConfigurationFileTest.java}Notice that this method requires to return a **String[]** with 2 values. The 1st value is the path for the configuration file to load. The second value is the persistence-id of the **<cm:property-placeholder>** tag.

The **stuff.cfg** file is just a plain properties file with the property placeholders such as:

this is a comment greeting=Bye

Using a `.cfg` file and Overriding Properties for Blueprint Property Placeholders

You can do both as well. Here is a complete example. First we have the Blueprint XML file:{snippet:id=e1|lang=xml|url=camel/trunk/components/camel-test-blueprint/src/test/resources/org/apache/camel/test/blueprint/configadmin-loadfileoverride.xml}And in the unit test class we do as follows:{snippet:id=e1|lang=java|url=camel/trunk/components/camel-test-blueprint/src/test/java/org/apache/camel/test/blueprint/ConfigAdminLoadConfigurationFileAndOverrideTest.java}And the **etc/stuff.cfg** configuration file contains:

greeting=Bye echo=Yay destination=mock:result

Bridging Spring and Camel Property Placeholders

Available as of Camel 2.10

The Spring Framework does not allow third-party frameworks such as Apache Camel to seamless hook into the Spring property placeholder mechanism. However you can easily bridge Spring and Camel by declaring a Spring bean with the type **org.apache.camel.spring.spi.BridgePropertyPlaceholderConfigurer**, which is a Spring **org.springframework.beans.factory.config.PropertyPlaceholderConfigurer** type.

To bridge Spring and Camel you must define a single bean as shown below:{snippet:id=e1|lang=xml|title=Bridging Spring and Camel property placeholders|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/component/properties/CamelSpringPropertyPlaceholderConfigurerTest.xml}You **must not** use the spring **<context:property-placeholder>** namespace at the same time; this is not possible.

After declaring this bean, you can define property placeholders using both the Spring style, and the Camel style within the **<camelContext>** tag as shown below:{snippet:id=e2|lang=xml|title=Using bridge property placeholders|url=camel/trunk/components/camel-spring/src/test/resources/org/apache/camel/component/properties/CamelSpringPropertyPlaceholderConfigurerTest.xml}Notice how the hello bean is using pure Spring property placeholders using the **${}** notation. And in the Camel routes we use the Camel placeholder notation with **{{ }}**.

Clashing Spring Property Placeholders with Camels Simple Language

Take notice when using Spring bridging placeholder then the spring **${}** syntax clashes with the Simple in Camel, and therefore take care.

Example:

xml<setHeader headerName="Exchange.FILE_NAME"> <simple>{{file.rootdir}}/${in.header.CamelFileName}</simple> </setHeader>

clashes with Spring property placeholders, and you should use **$simple{}** to indicate using the Simple language in Camel.

xml<setHeader headerName="Exchange.FILE_NAME"> <simple>{{file.rootdir}}/$simple{in.header.CamelFileName}</simple> </setHeader>

An alternative is to configure the **PropertyPlaceholderConfigurer** with **ignoreUnresolvablePlaceholders** option to **true**.

Overriding Properties from Camel Test Kit

Available as of Camel 2.10

When Testing with Camel and using the Properties component, you may want to be able to provide the properties to be used from directly within the unit test source code. This is now possible from Camel 2.10, as the Camel test kits, e.g., **CamelTestSupport** class offers the following methods

- **useOverridePropertiesWithPropertiesComponent**
- **ignoreMissingLocationWithPropertiesComponent**

So for example in your unit test classes, you can override the **useOverridePropertiesWithPropertiesComponent** method and return a **java.util.Properties** that contains the properties which should be preferred to be used.{snippet:id=e1|lang=java|title=Providing properties from within unit test source|url=camel/trunk/components/camel-test-blueprint/src/test/java/org/apache/camel/test/blueprint/ConfigAdminOverridePropertiesTest.java}This can be done from any of the Camel Test kits, such as **camel-test**, **camel-test-spring** and **camel-test-blueprint**.

The **ignoreMissingLocationWithPropertiesComponent** can be used to instruct Camel to ignore any locations which was not discoverable. For example if you run the unit test, in an environment that does not have access to the location of the properties.

Using @PropertyInject

Available as of Camel 2.12

Camel allows to inject property placeholders in POJOs using the **@PropertyInject** annotation which can be set on fields and setter methods. For example you can use that with **RouteBuilder** classes, such as shown below:

javapublic class MyRouteBuilder extends RouteBuilder { @PropertyInject("hello") private String greeting; @Override public void configure() throws Exception { from("direct:start") .transform().constant(greeting) .to("{{result}}"); } }

Notice we have annotated the greeting field with **@PropertyInject** and define it to use the key **hello**. Camel will then lookup the property with this key and inject its value, converted to a String type.

You can also use multiple placeholders and text in the key, for example we can do:

java@PropertyInject("Hello {{name}} how are you?") private String greeting;

This will lookup the placeholder with they key **name**.

You can also add a default value if the key does not exists, such as:

java@PropertyInject(value = "myTimeout", defaultValue = "5000") private int timeout;

Using Out of the Box Functions

Available as of Camel 2.14.1

The Properties component includes the following functions out of the box

- **env** - A function to lookup the property from OS environment variables.
- **sys** - A function to lookup the property from Java JVM system properties.
- **service** - A function to lookup the property from OS environment variables using the service naming idiom.
- **service.host** - **Camel 2.16.1:** A function to lookup the property from OS environment variables using the service naming idiom returning the hostname part only.
- **service.port** - **Camel 2.16.1:** A function to lookup the property from OS environment variables using the service naming idiom returning the port part only.

As you can see these functions is intended to make it easy to lookup values from the environment. As they are provided out of the box, they can easily be used as shown below:

xml<camelContext xmlns="http://camel.apache.org/schema/blueprint"> <route> <from uri="direct:start"/> <to uri="{{env:SOMENAME}}"/> <to uri="{{sys:MyJvmPropertyName}}"/> </route> </camelContext>

You can use default values as well, so if the property does not exists, you can define a default value as shown below, where the default value is a **`log:foo`** and **`log:bar`** value.

xml<camelContext xmlns="http://camel.apache.org/schema/blueprint"> <route> <from uri="direct:start"/> <to uri="{{env:SOMENAME:log:foo}}"/> <to uri="{{sys:MyJvmPropertyName:log:bar}}"/> </route> </camelContext>

The service function is for looking up a service which is defined using OS environment variables using the service naming idiom, to refer to a service location using **`hostname : port`**

- *NAME*_**`SERVICE_HOST`**
- *NAME*_**`SERVICE_PORT`**

in other words the service uses _**`SERVICE_HOST`** and _**`SERVICE_PORT`** as prefix. So if the service is named **`FOO`**, then the OS environment variables should be set as

export $FOO_SERVICE_HOST=myserver export $FOO_SERVICE_PORT=8888

For example if the **`FOO`** service a remote HTTP service, then we can refer to the service in the Camel endpoint URI, and use the HTTP component to make the HTTP call:

xml<camelContext xmlns="http://camel.apache.org/schema/blueprint"> <route> <from uri="direct:start"/> <to uri="http://{{service:FOO}}/myapp"/> </route> </camelContext>

And we can use default values if the service has not been defined, for example to call a service on localhost, maybe for unit testing etc:

xml<camelContext xmlns="http://camel.apache.org/schema/blueprint"> <route> <from uri="direct:start"/> <to uri="http://{{service:FOO:localhost:8080}}/myapp"/> </route> </camelContext>

Using Custom Functions

Available as of Camel 2.14.1

The Properties component allow to plugin 3rd party functions which can be used during parsing of the property placeholders. These functions are then able to do custom logic to resolve the placeholders, such as looking up in databases, do custom computations, or whatnot. The name of the function becomes the prefix used in the placeholder. This is best illustrated in the example code below

xml<bean id="beerFunction" class="MyBeerFunction"/> <camelContext xmlns="http://camel.apache.org/schema/blueprint"> <propertyPlaceholder id="properties" location="none" ignoreMissingLocation="true"> <propertiesFunction ref="beerFunction"/> </propertyPlaceholder> <route> <from uri="direct:start"/> <to uri="{{beer:FOO}}"/> <to uri="{{beer:BAR}}"/> </route> </camelContext>

Here we have a Camel XML route where we have defined the **`<propertyPlaceholder>`** to use a custom function, which we refer to be the bean id - e.g., the **`beerFunction`**. As the beer function uses **`beer`** as its name, then the placeholder syntax can trigger the beer function by starting with **`beer:value`**.

The implementation of the function is only two methods as shown below:

javapublic static final class MyBeerFunction implements PropertiesFunction { @Override public String getName() { return "beer"; } @Override public String apply(String remainder) { return "mock:" + remainder.toLowerCase(); } }

The function must implement the **`org.apache.camel.component.properties.PropertiesFunction`** interface. The method **`getName`** is the name of the function, e.g., **`beer`**. And the **`apply`** method is where we implement the custom logic to do. As the sample code is from an unit test, it just returns a value to refer to a mock endpoint.

To register a custom function from Java code is as shown below:

javaPropertiesComponent pc = context.getComponent("properties", PropertiesComponent.class); pc.addFunction(new MyBeerFunction());

See Also

- Properties component