appendix A

Simple, the expression language

Camel offers a powerful expression language, which was called *Simple* because back in the earlier days, it wasn't very powerful. It's evolved to become much more since then, but don't worry: it's still simple to use. The Simple language is provided out of the box in the camel-core JAR file, which means you don't have to add any JARs on the classpath to use it.

A.1 Introducing Simple

In a nutshell, the Simple expression language evaluates an expression on the current instance of `Exchange` that's under processing. The Simple language can be used for both expressions and predicates, which makes it perfect to use in your Camel routes.

For example, the Content-Based Router EIP can use the Simple language to define predicates in the `when` clauses, as shown here:

```
from("activemq:queue:quotes")
    .choice()
        .when(simple("${body} contains 'Camel'"))
            .to("activemq:camel")
        .when(simple("${header.amount} > 1000"))
            .to("activemq:bigspender")
        .otherwise()
            .to("activemq:queue:other");
```

The equivalent XML DSL example is as follows:

```
<route>
    <from uri="activemq:queue:quotes"/>
    <choice>
```

```
        <when>
            <simple>${body} contains 'Camel'</simple>
            <to uri="activemq:camel"/>
        </when>
        <when>
            <simple>${header.amount} &gt; 1000</simple>
            <to uri="activemq:bigspender"/>
        </when>
        <otherwise>
            <to uri="activemq:queue:other"/>
        </otherwise>
    </choice>
</route>
```

As you can see, the Simple expression is understandable and similar to other scripting languages. In these examples, Camel will evaluate the expression as a `Predicate`, which means the result is a `boolean`, which is either `true` or `false`. In the example, you use operators to determine whether the message body contains the word `Camel` or whether the message header `amount` is larger than 1000. Notice also how you had to escape the `>` in the XML DSL with `&gt;`. This applies to any other special characters in XML, like `<` (replaced with `$lt;`) or `&` (replaced with `&amp;`).

That gives you a taste of the Simple language. Let's look at its syntax.

A.2 Syntax

The Simple language uses `${` `}` placeholders for dynamic expressions, such as those in the previous examples. You can use multiple `${ }` placeholders in the same expression, or even nest placeholders.

For example, these are valid Simple expressions:

```
"Hello ${header.name} thanks for ordering ${body}"
"${header.${header.bar}}"
```

An alternative syntax is available to accommodate a clash with Spring's property placeholder feature. You can now also use `$simple{ }` placeholders with Simple, as shown here:

```
"Hello $simple{header.name} thanks for ordering $simple{body}"
```

These examples use variables such as `body` and `header`. The next section covers these variables.

A.3 Built-in variables

The Simple language provides variables that bind to information in the current `Exchange`. You've already seen `body` and `header`. Table A.1 lists all the variables available.

Table A.1 Variables in the Simple language

Variable	Type	Description
`body` `in.body`	`Object`	Contains the input message body. Note that `body` is preferred over `in.body`.

`header.` *`XXX`*
`headers.` *`XXX`*
`in.header.` *`XXX`*
`in.headers.` *`XXX`*
`header[` *`XXX`* `]`
`headers[` *`XXX`* `]`
`in.header[` *`XXX`* `]`
`in.headers[` *`XXX`* `]`
	`Object`	Contains the input message header *`XXX`*.

`exchangeProperty.` *`XXX`*
`exchangeProperty[` *`XXX`* `]`
	`Object`	Contains the exchange property *`XXX`*.

`headers` `in.headers`	`Map`	The input message headers.
`exchangeId`	`String`	Contains the unique ID of the `Exchange`.

`sys.` *`XXX`*
`sysenv.` *`XXX`*
	`String`	Contains the system environment variable *`XXX`*.

`exception`	`Object`	Contains the exception on the `Exchange`, if any exists.
`exception.stacktrace`	`String`	Contains the exception stacktrace on the `Exchange.` If not set, will fall back to the `Exchange.EXCEPTION_CAUGHT` property on the exchange.

Variable	Type	Description
`exception.message`	`String`	Contains the exception message on the `Exchange`, if any exists.
`threadName`	`String`	Contains the name of the current thread; can be used for logging purposes.
`camelId`	`String`	The `CamelContext` name.
`exchange`	`Exchange`	The current `Exchange` object.
`routeId`	`String`	The ID of the route where this `Exchange` is currently being routed.
`null`		Represents `null`.
`messageHistory`	`String`	The message history of the current exchange. This shows how the message has been routed.
`messageHistory(false)`	`String`	The message history of the current exchange, but without showing any of the `Exchange` content. Helpful if you don’t want sensitive details showing up in the logs.
`\n`	`String`	A newline character.
`\t`	`String`	A tab character.

Variable	Type	Description
`\r`	`String`	A carriage return character.
`\}`	`String`	The `}` character, which is special for a simple expression, of course.

Notice that all the `in.*` variables from table A.1 are being considered for removal in Camel 3.0. Instead, use the non-`in.*` variables. The variables can easily be used in a Simple expression, as you've already seen. Logging the message body can be done by using `${body}`, as shown in the following route snippet:

```
from("activemq:queue:quotes")
    .log("We received ${body}")
    .to("activemq:queue:process");
```

The Simple language also has a set of built-in functions.

A.4 Built-in functions

The Simple language has many functions at your disposal, as listed in table A.2.

Table A.2 Functions provided in the Simple language

Function	Type	Description
`bodyAs(type)`	`type`	Converts the `body` to the given type `bodyAs(String)` or `bodyAs(com.fo` Returns `null` if the body can't be co
`bodyAs(type).OGNL`	`Object`	Converts the `body` to the given type invokes a method on the resulting o Object-Graph Navigation Language (notation. May return `null` if the bo converted or if the method returns
`mandatoryBodyAs(type)`	`type`	Converts the body to the given type. `NoTypeConversionAvailableExcept` body can't be converted.
`mandatoryBodyAs(type).OGNL`	`Object`	Converts the body to the given type invokes a method on the resulting o OGNL notation. Throws a `NoTypeConversionAvailableExcept` body can't be converted.
`headerAs(key, type)`	`type`	Converts the header with the given k given type. Returns `null` if the head converted.
`bean:beanId[?method]`	`Object`	Invokes a method on a bean. Camel bean with the given ID from the `Reg` invokes the appropriate method. Yo optionally explicitly specify the nam method to invoke.
`date:command:pattern`	`String`	Formats a date. The command must or `header.`*`XXX`*: `now` represents th timestamp, whereas `header.`*`XXX`* u with the key *`XXX`*.

Function	Type	Description
		The pattern is based on the `java.text.SimpleDataFormat` form
`camelContext.OGNL`	`Object`	Invokes a method on the `CamelCont` OGNL notation. You can see more ab later in this appendix.
`collate(sub_list_size)`	`Iterator`	Splits a message body into sublists o `sub_list_size`. The result is an iter points to the sublists.
`exchange.OGNL`	`Object`	Invokes a method on the `Exchange` OGNL notation. You can see more ab later in this appendix.
`exchangeProperty.XXX.OGNL`	`Object`	Gets the exchange property *`XXX`* an a method on the resulting object by notation.
`properties-location:[locations:]key`	`String`	Resolves a property with the given the Camel Properties component.
`properties:key[:default]`	`String`	Resolves a property with the given k the Camel Properties component. If doesn't exist or has no value, a speci can be used.
`random(max)`	`Integer`	Returns a random number between and the specified `max` (excluded).
`random(min, max)`	`Integer`	Returns a random number between `min` (included) and the specified `ma`
`ref:XXX`	`Object`	Looks up and returns a bean with II

Function	Type	Description
`skip(number_of_items)`	`Iterator`	Skips the specified number of items message body and returns the rema
`type:name.field`	`Object`	Refers to a type or a field by its FQN `${type:org.apache.camel.Exchang` refers to the constant `Exchange.FIL` which resolves to `CamelFileName`.

Lets try a few of the functions from table A.2. We'll start with the date function. To log a formatted date from the message header, you can do as follows:

```
<route>
    <from uri="activemq:queue:quote"/>
    <log message="Quote date ${date:header.myDate:yyyy-MM-dd HH:mm:ss}"/
    <to uri="activemq:queue:process"/>
</route>
```

In this example, the input message is expected to contain a header with the key `myDate`, which should be of type `java.util.Date` (or a `long`, which will be converted to a `java.util.Date` by Camel automatically).

Suppose you need to organize received messages into a directory structure containing the current day's date as a parent folder. The file producer has direct support for specifying the target filename by using the Simple language as shown in bold:

```
from("activemq:queue:quote")
    .to("file:backup/?fileName=${date:now:yyyy-MM-dd}/${exchangeId}.txt"
    .to("activemq:queue:process");
```

Now suppose the file must use a filename generated from a bean. You can use the `bean` function to achieve this:

```
from("activemq:queue:quote")
    .to("file:backup/?fileName=${bean:uuidBean?method=generate}")
```

```
        .to("activemq:queue:process");
```

In this example, Camel looks up the bean with the ID `uuidBean` from the `Registry` and invokes the `generate` method. The output of this method invocation is returned and used as the filename.

The Camel Properties component is used for property placeholders. For example, you can store a property in a file containing a configuration for a big-spender threshold:

```
big=5000
```

Then you can refer to the `big` properties key from the Simple language:

```
from("activemq:queue:quotes")
    .choice()
        .when(simple("${header.amount} > ${properties:big}")
            .to("activemq:bigspender")
        .otherwise()
            .to("activemq:queue:other");
```

The Simple language also has built-in variables when working with the Camel File and FTP components.

A.5 Built-in file variables

Files consumed using the File or FTP components have file-related variables available to the Simple language. Table A.3 lists those variables.

Table A.3 File-related variables available when consuming files

Variable	Type	Description
`file:name`	`String`	Contains the filename (relative to the starting directory).
`file:name.ext`	`String`	Contains the file extension.
`file:name.ext.single`	`String`	Contains the file extension whereby only the extension after the last dot is returned. A tar.gz file would have an extension of gz.
`file:name.noext`	`String`	Contains the filename without extension (relative to the starting directory).
`file:name.noext.single`	`String`	Contains the filename without extension (relative to the starting directory), whereby only the extension after the last dot is stripped. A file named my-dir/backup.tar.gz would mean my-dir/backup.tar is returned.

Variable	Type	Description
`file:onlyname`	`String`	Contains the filename without any leading paths.
`file:onlyname.noext`	`String`	Contains the filename without extension and leading paths.
`file:onlyname.noext.single`	`String`	Contains the filename without extension and leading paths, whereby only the extension after the last dot is stripped. A file named backup.tar.gz would mean backup.tar is returned.
`file:parent`	`String`	Contains the file parent (the paths leading to the file).
`file:path`	`String`	Contains the file path (including leading paths).
`file:absolute`	`Boolean`	Indicates whether the filename is an absolute or relative file path.
`file:absolute.path`	`String`	Contains the absolute file path.

Variable	Type	Description
`file:length` `file:size`	`long`	Contains the file length.
`file:modified`	`Date`	Contains the modification date of the file as a `java.util.Date` type.

Among other things, the file variables can be used to log which file has been consumed:

```
<route>
    <from uri="file://inbox"/>
    <log message="Picked up ${file:name}"/>
    ...
</route>
```

The File and FTP endpoints have options that accept Simple language expressions. For example, the File consumer can be configured to move processed files into a folder you specify. Suppose you must move files into a directory structure organized by dates. You can do that by specifying the expression in the `move` option, as follows:

```
<from uri="file://inbox?move=backup/${date:now:yyyyMMdd}/${file:name}"/>
```

TIP The FTP endpoint supports the same move option as shown here.

Another example where the file variables come in handy is if you have to process files differently based on the file extension. For example, suppose you have CSV and XML files:

```
from("file://inbox")
    .choice()
```

```
          .when(simple("${file:ext} == 'txt'")).to("direct:txt")
          .when(simple("${file.ext} == 'xml'")).to("direct:xml")
          .otherwise().to("direct:unknown");
```

NOTE You can read more about the file variables at the Camel website: http://camel.apache.org/file-language.html.

In this appendix, we've used the Simple language for predicates. In fact, the previous example determines whether the file is a text file. Doing this requires operators.

A.6 Built-in operators

The first example in this appendix implemented the Content-Based Router EIP with the Simple expression language. It used predicates to determine where to route a message, and these predicates use operators. Table A.4 lists all the operators supported in Simple.

Table A.4 Operators provided in the Simple language

Operator	Description
`==`	Tests whether the left side is equal to the right side
`=~`	Tests whether the left side is equal to the right side, ignoring case
`>`	Tests whether the left side is greater than the right side
`>=`	Tests whether the left side is greater than or equal to the right side
`<`	Tests whether the left side is less than the right side
`<=`	Tests whether the left side is less than or equal to the right side
`!=`	Tests whether the left side isn't equal to the right side
`contains`	Tests whether the left side contains the `String` value on the right side
`not contains`	Tests whether the left side doesn't contain the `String` value on the right side
`starts with`	Tests whether the left side starts with the `String` value on the right side
`ends with`	Tests whether the left side ends with the `String` value on the right side
`in`	Tests whether the left side is in a set of values specified on the right side; the values must be

Operator	Description
	separated by commas
`not in`	Tests whether the left side isn't in a set of values specified on the right side; the values must be separated by commas
`range`	Tests whether the left side is within a range of values defined with the following syntax: `'from..to'`
`not range`	Tests whether the left side isn't within a range of values defined with the following syntax: `'from..to'`
`regex`	Tests whether the left side matches a regular expression pattern defined as a `String` value on the right side
`not regex`	Tests whether the left side doesn't match a regular expression pattern defined as a `String` value on the right side
`is`	Tests whether the left-side type is an instance of the value on the right side
`not is`	Tests whether the left-side type isn't an instance of the value on the right side
`++`	Unary operator that increments a left-side function and returns that value. So if you have a counter header with `value = 1`, the expression `${header.counter}++` would return 2.
`--`	Unary operator that decrements a left-side function and returns that value. If you have a counter

Operator	Description
	header with `value = 2`, the expression `${header.counter}--` would return 1.

The operators require the following syntax:

```
${leftValue} <OP> rightValue
```

The value on the left side must be enclosed in a `${ }` placeholder. The operator must be separated with a single space on the left and right. The right value can either be a fixed value or another dynamic value enclosed using `${ }`.

Let’s look at an example:

```
simple("${in.header.foo} == Camel")
```

Here you test whether the `foo` header is equal to the `String` value `"Camel"`. If you want to test for `"Camel rocks"`, you must enclose the `String` in quotes (because the value contains a space):

```
simple("${in.header.foo} == 'Camel rocks'")
```

Camel automatically type coerces, so you can compare apples to oranges. Camel will regard both as fruit:

```
simple("${in.header.bar} < 200")
```

Suppose the `bar` header is a `String` with the value `"100"`. Camel will convert this value to the same type as the value on the right side, which is numeric. It will therefore compute `100 < 200`, which renders `true`.

You can use the range operator to test whether a value is in a numeric range.

```
simple("${in.header.bar} range '100..199'")
```

Both the *from* and *to* range values are inclusive. You must define the range exactly as shown.

A regular expression can be used to test a variety of things, such as whether a value is a four-digit value:

```
simple("${in.header.bar} regex '\d{4}'")
```

You can also use the built-in functions with the operators. For example, to test whether a given header has today’s date, you can use the `date` function:

```
simple("${in.header.myDate} == ${date:now:yyyyMMdd}")
```

TIP You can see more examples in the Camel Simple online documentation: http://camel.apache.org/simple.html.

The Simple language also allows you to combine two expressions.

A.6.1 Combining expressions

The Simple language can combine expressions via the `&&` (and) or `||` (or) operators. The syntax for combining two expressions is as follows:

```
${leftValue} <OP> rightValue <&& or ||> ${leftValue} <OP> rightValue
```

Here’s an example using `&&` to group two expressions:

```
simple("${in.header.bar} < 200 && ${body} contains 'Camel'")
```

The Simple language also supports an OGNL feature.

A.7 The OGNL feature

Both the Simple language and Bean component support an Object-Graph Navigation Language (OGNL) feature when specifying the method name to invoke. OGNL allows you to specify a chain of methods in the expression.

Suppose the message body contains a `Customer` object that has a `getAddress` method. To get the ZIP code of the address, you type the following:

```
simple("${body.getAddress().getZip()}")
```

You can use a shorter notation, omitting the `get` prefix and the parentheses:

```
simple("${body.address.zip}")
```

In this example, the ZIP code will be returned. But if the `getAddress` method returns `null`, the example would cause a `NoSuchMethodException` to be thrown by Camel. If you want to avoid this, you can use the null-safe operator `?.` as follows:

```
simple("${body?.address.zip}")
```

The methods in the OGNL expression can be any method name. For example, to invoke a `sayHello` method, you do this:

```
simple("${body.sayHello}")
```

Camel uses the bean parameter binding (covered in chapter 4). This means that the method signature of `sayHello` can have parameters that are bound to the current `Exchange` being routed:

```
public String sayHello(String body) {
    return "Hello " + body;
}
```

The OGNL feature has specialized support for accessing `Map` and `List` types. For example, suppose the `getAddress` method has a `getLines` method that returns a `List`. You could access the lines by their index values, as follows:

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

If you try to index an element that's out of bounds, an `IndexOutOfBoundsException` exception is thrown. You can use the null-safe operator to suppress this exception:

```
simple("${body.address?.lines[2]}")
```

If you want to access the last element, you can use `last` as the index value, as shown here:

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

The access support for `Map` s is similar, but you use a key instead of a numeric value as the index. Suppose the message body contains a `getType` method that returns a `Map` instance. You could access the `gold` entry as follows:

```
simple("${body.type[gold]}")
```

You could even invoke a method on the `gold` entry like this:

```
simple("${body.type[gold].sayHello}")
```

This concludes our tour of the various features supported by the Camel Simple language. We'll now take a quick look at how to use the Simple language from custom Java code.

A.8 Using Simple from custom Java code

The Simple language is most often used directly in your Camel routes, in either the Java DSL or XML DSL file. But it's also possible to use it from custom Java code.

Here's an example that uses the Simple language from a Camel `Processor`.

Listing A.1 Using the Simple language from custom Java code

```
package camelinaction;
```

```
import org.apache.camel.Exchange;
import org.apache.camel.Processor;
import org.apache.camel.builder.SimpleBuilder;

public class MyProcessor implements Processor {
    public void process(Exchange exchange) throws Exception {
        SimpleBuilder simple = new SimpleBuilder(
                                       "${body} contains 'Camel'");
        if (!simple.matches(exchange)) {
            throw new Exception("This is NOT a Camel message");
        }
    }
}
```

As you can see, all it takes is creating an instance of `SimpleBuilder,` which is capable of evaluating either a predicate or an expression. In the listing, you use the Simple language as a predicate.

To use an expression to say “Hello”, you could do the following:

```
SimpleBuilder simple = new SimpleBuilder("Hello ${header.name}");
String s = simple.evaluate(exchange, String.class);
System.out.println(s);
```

Notice how you specify that you want the response back as a `String` by passing in `String.class` to the `evaluate` method.

Listing A.1 uses the Simple language from within a Camel `Processor`, but you’re free to use it anywhere, such as from a custom bean. Just keep in mind that the `Exchange` must be passed into the `matches` method on the `SimpleBuilder`.

Summary

This appendix covered the Simple language, an expression language provided with Camel. You saw how well it blends with Camel routes, which makes it easy to define predicates in routes, such as those needed when using the Content-Based Router.

We also looked at how easy it is to access information from the `Exchange` message by using Simple’s built-in variables. You saw that Simple provides functions, such as a `date` function that formats dates and a `bean` function that invokes methods on beans.

Finally, we covered OGNL notation, which makes it even easier to access data from nested beans.

The Simple language is a great expression language that should help you with 95 percent of your use cases.