Appendix A. Spring Expression Language (SpEL)

A.1 Introduction

Many Spring Integration components can be configured using expressions. These expressions are written in the Spring Expression Language.

In most cases, the *#root* object is the `Message` which, of course, has two properties - `headers` and `payload` - allowing such expressions as `payload`, `payload.foo`, `headers['my.header']` etc.

In some cases, additional variables are provided, for example the `<int-http:inbound-gateway/>` provides `#requestParams` (parameters from the HTTP request) and `#pathVariables` (values from path placeholders in the URI).

For all SpEL expressions, a `BeanResolver` is available, enabling references to any bean in the application context. For example `@myBean.foo(payload)`. In addition, two `PropertyAccessors` are available; a `MapAccessor` enables accessing values in a `Map` using a key, and a `ReflectivePropertyAccessor` which allows access to fields and or JavaBean compliant properties (using getters and setters). This is how the `Message` headers and payload properties are accessible.

A.2 SpEL Evaluation Context Customization

Starting with Spring Integration 3.0, it is possible to add additional `PropertyAccessor` s to the SpEL evaluation contexts used by the framework. The framework provides the `JsonPropertyAccessor` which can be used (read-only) to access fields from a `JsonNode`, or JSON in a `String`. Or you can create your own `PropertyAccessor` if you have specific needs.

In addition, custom functions can be added. Custom functions are `static` methods declared on a class. Functions and property accessors are available in any SpEL expression used throughout the framework.

The following configuration shows how to directly configure the `IntegrationEvaluationContextFactoryBean` with custom property accessors and functions. However, for convenience, namespace support is provided for both, as described in the following sections, and the framework will automatically configure the factory bean on your behalf.

```
<bean id="integrationEvaluationContext"
                        class="org.springframework.integration.config.IntegrationEvaluationContextFactoryBean">
        <property name="propertyAccessors">
                <util:map>
                        <entry key="foo">
                                <bean class="foo.MyCustomPropertyAccessor"/>
                        </entry>
                </util:map>
        </property>
        <property name="functions">
                <map>
                        <entry key="barcalc" value="#{T(foo.MyFunctions).getMethod('calc', T(foo.MyBar))}"/>
                </map>
        </property>
</bean>
```

This factory bean definition will override the default `integrationEvaluationContext` bean definition, adding the custom accessor to the list (which also includes the standard accessors mentioned above), and one custom function.

Note that custom functions are static methods. In the above example, the custom function is a static method `calc` on class `MyFunctions` and takes a single parameter of type `MyBar`.

Say you have a `Message` with a payload that has a type `MyFoo` on which you need to perform some action to create a `MyBar` object from it, and you then want to invoke a custom function `calc` on that object.

The standard property accessors wouldn't know how to get a `MyBar` from a `MyFoo` so you could write and configure a custom property accessor to do so. So, your final expression might be `"#barcalc(payload.myBar)"`.

The factory bean has another property `typeLocator` which allows you to customize the `TypeLocator` used during SpEL evaluation. This might be necessary when running in some environments that use a non-standard `ClassLoader`. In the following example, SpEL expressions will always use the bean factory's class loader:

```
<bean id="integrationEvaluationContext"
                class="org.springframework.integration.config.IntegrationEvaluationContextFactoryBean">
        <property name="typeLocator">
                <bean class="org.springframework.expression.spel.support.StandardTypeLocator">
                        <constructor-arg value="#{beanFactory.beanClassLoader}"/>
                </bean>
        </property>
</bean>
```

A.3 SpEL Functions

Namespace support is provided for easy addition of SpEL custom functions. You can specify `<spel-function/>` components to provide custom SpEL functions to the `EvaluationContext` used throughout the framework. Instead of configuring the factory bean above, simply add one or more of these components and the framework will automatically add them to the default *integrationEvaluationContext* factory bean.

For example, assuming we have a useful static method to evaluate XPath:

```
<int:spel-function id="xpath"
        class="com.foo.test.XPathUtils" method="evaluate(java.lang.String, java.lang.Object)"/>

<int:transformer input-channel="in" output-channel="out"
                 expression="#xpath('//foo/@bar', payload)" />
```

With this sample:

- The default `IntegrationEvaluationContextFactoryBean` bean with id *integrationEvaluationContext* is registered with the application context.
- The `<spel-function/>` is parsed and added to the `functions` Map of *integrationEvaluationContext* as map entry with `id` as the key and the static `Method` as the value.
- The *integrationEvaluationContext* factory bean creates a new `StandardEvaluationContext` instance, and it is configured with the default `PropertyAccessor` s, `BeanResolver` and the custom functions.
- That `EvaluationContext` instance is injected into the `ExpressionEvaluatingTransformer` bean.

Note

SpEL functions declared in a parent context are also made available in any child context(s). Each context has its own instance of the *integrationEvaluationContext* factory bean because each needs a different `BeanResolver`, but the function declarations are inherited and can be overridden if needed by declaring a SpEL function with the same name.

Built-in SpEL Functions

Spring Integration provides some standard functions, which are registered with the application context automatically on start up:

#jsonPath - to evaluate a *jsonPath* on some provided object. This function invokes `JsonPathUtils.evaluate(...)`. This static method delegates to the Jayway JsonPath library. The following shows some usage examples:

```
<transformer expression="#jsonPath(payload, '$.store.book[0].author')"/>

<filter expression="#jsonPath(payload,'$..book[2].isbn') matches '\d-\d{3}-\d{5}-\d'"/>

<splitter expression="#jsonPath(payload, '$.store.book')"/>

<router expression="#jsonPath(payload, headers.jsonPath)">
        <mapping channel="output1" value="reference"/>
        <mapping channel="output2" value="fiction"/>
</router>
```

#jsonPath also supports the third optional parameter - an array of `com.jayway.jsonpath.Filter`, which could be provided by a reference to a bean or bean method, for example.

Note

Using this function requires the Jayway JsonPath library (json-path.jar) to be on the classpath; otherwise the *#jsonPath* SpEL function won't be registered.

For more information regarding JSON see *JSON Transformers* in Section 7.1, "Transformer".

#xpath - to evaluate an *xpath* on some provided object. For more information regarding xml and xpath see Chapter 35, *XML Support - Dealing with XML Payloads*.

A.4 PropertyAccessors

Namespace support is provided for the easy addition of SpEL custom `PropertyAccessor` implementations. You can specify the `<spel-property-accessors/>` component to provide a list of custom `PropertyAccessor` s to the `EvaluationContext` used throughout the framework. Instead of configuring the factory bean above, simply add one or more of these components, and the framework will automatically add the accessors to the default_integrationEvaluationContext_ factory bean:

```
<int:spel-property-accessors>
        <bean id="jsonPA" class="org.springframework.integration.json.JsonPropertyAccessor"/>
        <ref bean="fooPropertyAccessor"/>
</int:spel-property-accessors>
```

With this sample, two custom `PropertyAccessor` s will be injected to the `EvaluationContext` in the order that they are declared.

Note

Custom `PropertyAccessor` s declared in a parent context are also made available in any child context(s). They are placed at the end of result list (but before the default `org.springframework.context.expression.MapAccessor` and `o.s.expression.spel.support.ReflectivePropertyAccessor`). If a `PropertyAccessor` with the same bean id is declared in a child context(s), it will override the parent accessor. Beans declared within a `<spel-property-accessors/>` must have an *id* attribute. The final order of usage is: the accessors in the current context, in the order in which they are declared, followed by any from parent contexts, in order, followed by the `MapAccessor` and finally the `ReflectivePropertyAccessor`.