Expression Language | docs.camunda.org

Camunda 7 community

Camunda 7 supports the Unified Expression Language (EL), specified by the Jakarta Expression Language 4.0 standard. To do so, it maintains a custom version of the open source JUEL implementation.

Note, compared to EL 4.0 this JUEL implementation has the following limitations:

Initializing collections directly within expressions (e.g., `${[1,2,3]}`) is NOT supported.

Lambda expressions (e.g., inline functions `${((x,y)->x+y)(3,4)}`) are NOT supported.

Referencing static fields (e.g., `${Boolean.TRUE}`), static functions (e.g., `${Integer.parseInt("123")}`), and enums (e.g., `${Thread.State.TERMINATED}`) are NOT supported.

The Assignment Operator `(A=B)`, the String Concatenation Operator `(A+=B)`, and the Semicolon Operator `(A ; B)` are NOT supported.

To get more general information about the usage of Expression Language, please read the official documentation. It provides examples that give a good overview of the syntax of expressions.

Within Camunda 7, EL can be used in many circumstances to evaluate small script-like expressions. The following table provides an overview of the BPMN elements which support usage of EL.

BPMN element	EL support
Service Task, Business Rule Task, Send Task, Message Intermediate Throwing Event, Message End Event, Execution Listener and Task Listener	Expression language as delegation code
Sequence Flows, Conditional Events	Expression language as condition expression
All Tasks, All Events, Transaction, Subprocess and Connector	Expression language inside an inputOutput parameter mapping

Different Elements	Expression language as the value of an attribute or element
All Flow Nodes, Process Definition	Expression language to determine the priority of a job

Usage of Expression Language

Delegation Code

Besides Java code, Camunda 7 also supports the evaluation of expressions as delegation code. For general information about delegation code, see the corresponding section.

Two types of expressions are currently supported: `camunda:expression` and `camunda:delegateExpression`.

With `camunda:expression` it is possible to evaluate a value expression or to invoke a method expression. You can use special variables which are available inside an expression or Spring and CDI beans. For more information about variables and Spring, respectively CDI beans, please see the corresponding sections.

```
 <process id="process">
   <extensionElements>
     <!-- execution listener which uses an expression to set a process variable -->
     <camunda:executionListener event="start" expression="${execution.setVariable('test', 'foo')}"
/>
   </extensionElements>

   <!-- ... -->

   <userTask id="userTask">
     <extensionElements>
       <!-- task listener which calls a method of a bean with current task as parameter -->
```

```
      <camunda:taskListener event="complete" expression="${myBean.taskDone(task)}" />
    </extensionElements>
  </userTask>

  <!-- ... -->

  <!-- service task which evaluates an expression and saves it in a result variable -->
  <serviceTask id="serviceTask"
      camunda:expression="${myBean.ready}" camunda:resultVariable="myVar" />

  <!-- ... -->

</process>
```

The attribute `camunda:delegateExpression` is used for expressions which evaluate to a delegate object. This delegate object must implement either the `JavaDelegate` or `ActivityBehavior` interface.

```
<!-- service task which calls a bean implementing the JavaDelegate interface -->
<serviceTask id="task1" camunda:delegateExpression="${myBean}" />

<!-- service task which calls a method which returns delegate object -->
<serviceTask id="task2" camunda:delegateExpression="${myBean.createDelegate()}" />
```

Conditions

To use conditional sequence flows or conditional events, expression language is usually used. For conditional sequence flows, a `conditionExpression` element of a sequence flow has to be used. For conditional events, a `condition` element of a conditional event has to be used. Both are of the type `tFormalExpression`. The text content of the element is the expression to be evaluated.

Within the expression, some special variables are available which enable access of the current context. To find more information about the available variables, please see the corresponding section.

The following example shows usage of expression language as condition of a sequence flow:

```
<sequenceFlow>
  <conditionExpression xsi:type="tFormalExpression">
    ${test == 'foo'}
  </conditionExpression>
</sequenceFlow>
```

For usage of expression language on conditional events, see the following example:

```
<conditionalEventDefinition>
  <condition type="tFormalExpression">${var1 == 1}</condition>
</conditionalEventDefinition>
```

inputOutput Parameters

With the Camunda `inputOutput` extension element you can map an `inputParameter` or `outputParameter` with expression language.

Inside the expression some special variables are available which enable the access of the current context. To find more information about the available variables please see the corresponding section.

The following example shows an `inputParameter` which uses expression language to call a method of a bean.

```
<serviceTask id="task" camunda:class="org.camunda.bpm.example.SumDelegate">
  <extensionElements>
    <camunda:inputOutput>
      <camunda:inputParameter name="x">
        ${myBean.calculateX()}
      </camunda:inputParameter>
```

```
      </camunda:inputOutput>
    </extensionElements>
  </serviceTask>
```

External Task Error Handling

For External Tasks it is possible to define camunda:errorEventDefinition elements which can be provided with a JUEL expression. The expression is evaluated on `ExternalTaskService#complete` and `ExternalTaskService#handleFailure`. If the expression evaluates to `true`, a BPMN error is thrown which can be caught by an Error Boundary Event.

In the scope of an External Task, expressions have access to the ExternalTaskEntity object via the key `externalTask` which provides getter methods for `errorMessage`, `errorDetails`, `workerId`, `retries` and more.

Examples:

How to access the External Task object:

```
<bpmn:serviceTask id="myExternalTaskId" name="myExternalTask" camunda:type="external"
camunda:topic="myTopic">
  <bpmn:extensionElements>
    <camunda:errorEventDefinition id="myErrorEventDefinition" errorRef="myError"
expression="${externalTask.getWorkerId() == 'myWorkerId'}" />
  </bpmn:extensionElements>
</bpmn:serviceTask>
```

How to match an error message:

```
<bpmn:serviceTask id="myExternalTaskId" name="myExternalTask" camunda:type="external"
camunda:topic="myTopic">
  <bpmn:extensionElements>
    <camunda:errorEventDefinition id="myErrorEventDefinition" errorRef="myError"
```

```
expression="${externalTask.getErrorDetails().contains('myErrorMessage')}" />
  </bpmn:extensionElements>
</bpmn:serviceTask>
```

For further details on the functionality of error event definitions in the context of external tasks, consult the External Tasks Guide.

Value

Different BPMN and CMMN elements allow to specify their content or an attribute value by an expression. Please see the corresponding sections for BPMN and CMMN in the references for more detailed examples.

Availability of Variables and Functions Inside Expression Language

Process Variables

All process variables of the current scope are directly available inside an expression. So a conditional sequence flow can directly check a variable value:

```
<sequenceFlow>
  <conditionExpression xsi:type="tFormalExpression">
    ${test == 'start'}
  </conditionExpression>
</sequenceFlow>
```

Internal Context Variables

Depending on the current execution context, special built-in context variables are available while evaluating expressions:

Variable	Java Type	Context
`execution`	`DelegateExecution`	Available in a BPMN execution context like a service task, execution listener or sequence flow.

Variable	Java Type	Context
`task`	`DelegateTask`	Available in a task context like a task listener.
`externalTask`	`ExternalTask`	Available during an external task context activity (e.g. in camunda:errorEventDefinition expressions).
`caseExecution`	`DelegateCaseExecution`	Available in a CMMN execution context.
`authenticatedUserId`	`String`	The id of the currently authenticated user. Only returns a value if the id of the currently authenticated user has been set through the corresponding methods of the `IdentityService`. Otherwise it returns `null`.

The following example shows an expression which sets the variable `test` to the current event name of an execution listener.

```
<camunda:executionListener event="start"
  expression="${execution.setVariable('test', execution.eventName)}" />
```

External Context Variables With Spring and CDI

If the process engine is integrated with Spring or CDI, it is possible to access Spring and CDI beans inside of expressions. Please see the corresponding sections for Spring and CDI for more information. The following example shows the usage of a bean which implements the `JavaDelegate` interface as delegateExecution.

```
<serviceTask id="task1" camunda:delegateExpression="${myBean}" />
```

With the expression attribute any method of a bean can be called.

```
<serviceTask id="task2" camunda:expression="${myBean.myMethod(execution)}" />
```

Internal Context Functions

Special built-in context functions are available while evaluating expressions:

Function	Return Type	Description

Function	Return Type	Description
`currentUser()`	`String`	Returns the user id of the currently authenticated user or `null` no user is authenticated at the moment.
`currentUserGroups()`	`List of Strings`	Returns a list of the group ids of the currently authenticated user or `null` if no user is authorized at the moment.
`now()`	`Date`	Returns the current date as a Java Date object.
`dateTime()`	`DateTime`	Returns a Joda-Time DateTime object of the current date. Please see the Joda-Time documentation for all available functions.

The following example sets the due date of a user task to the date 3 days after the creation of the task.

```
<userTask id="theTask" name="Important task" camunda:dueDate="${dateTime().plusDays(3).toDate()}"/>
```

Built-In Camunda Spin Functions

If the Camunda Spin process engine plugin is activated, the Spin functions `S`, `XML` and `JSON` are also available inside of an expression. See the Data Formats section for a detailed explanation.

```
 <serviceTask id="task" camunda:expression="${XML(xml).attr('test').value()}" resultVariable="test"
/>
```