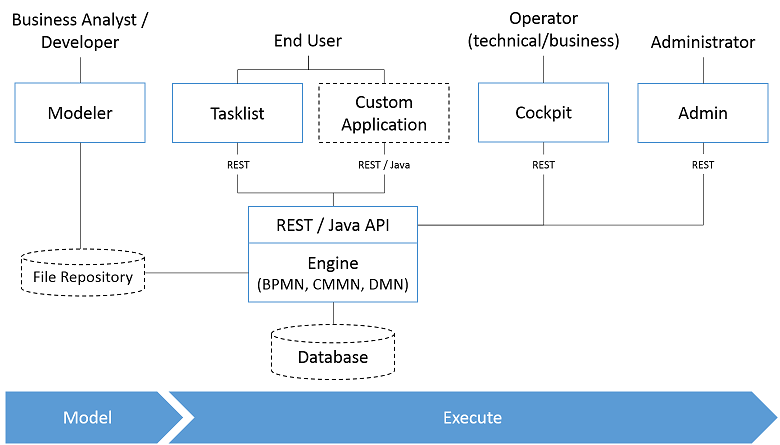
**Camunda framework**

Camunda is a Java-based framework supporting BPMN for workflow and process automation, CMMN for Case Management and DMN for Business Decision Management.

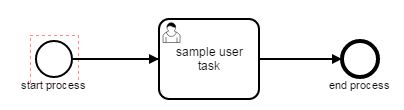


Camunda implements three different standards in the Business Process Management scope: BPMN 2.0, CMMN 1.1 and DMN 1.1. These three standards are defined by the Object Management Group with active collaboration of Camunda.

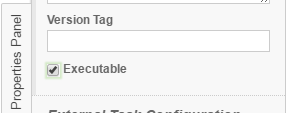
* Process Engine The process engine is a Java library responsible for executing BPMN 2.0 processes, CMMN 1.1 cases and DMN 1.1 decisions. It has a lightweight POJO core and uses a relational database for persistence. ORM mapping is provided by the MyBatis mapping framework.
* Camunda Modeler: Modeling tool for BPMN 2.0 and CMMN 1.1 diagrams as well as DMN 1.1 decision tables.
  + - Camunda Modeler is a desktop application for modeling BPMN, DMN and CMMN. It allows you to model files located directly on your local file system.
* REST API REST API allows you to use the process engine from a remote application or a JavaScript application. (Note: The documentation of the REST API is factored out into own documents.)
* Camunda Tasklist A web application for human workflow management and user tasks that allows process participants to inspect their workflow tasks and navigate to task forms in order to work on the tasks and provide data input.
* Camunda Cockpit A web application for process monitoring and operations that allows you to search for process instances, inspect their state and repair broken instances.
* Camunda Admin A web application that allows you to manage users, groups and authorizations.
* Camunda Cycle: A web application for synchronizing BPMN 2.0 process models between different modeling tools and modelers.

**BPMN (Business Process Model and Notation)**

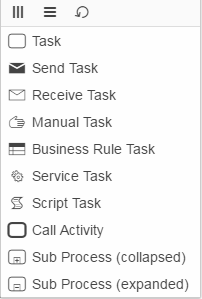
* Camunda BPM provides open source implementations of execution and modeling tools (Camunda modeler, eclipse plugin (deprecated)) to implement BPMN process.



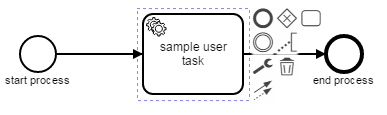
* Make sure to check executable in properties panel, else process can’t be deployed.



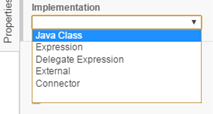
* There are different types of tasks available in BPMN to implement a BPM process



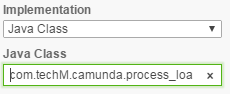
**Service Task and Send Task**

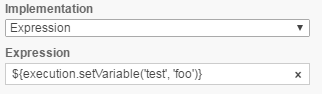
Service task and Send Task will have the common functionality to execute business requirement. We need to specify the type of implementation to execute.



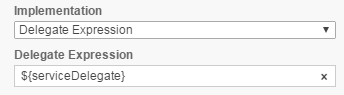
**Java Class:** path of the java class to fire (class should be implemented by JavaDelegate)



**Expression:** Besides Java code, Camunda BPM also supports the evaluation of expressions as delegation code.

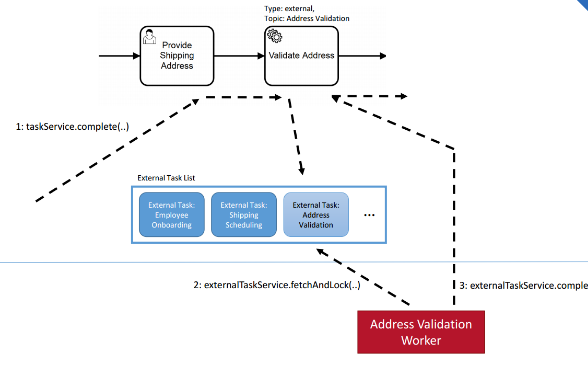


**Delegate Expression:** Here we can specify bean id which intern will call the corresponding class file.

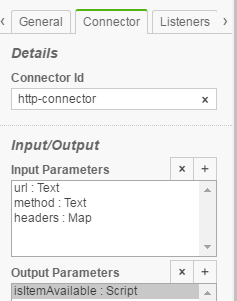
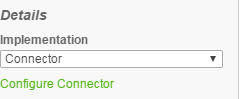


**External:** An external task is created and then added to a topic. An external application then queries the topic and locks the task. After the task is locked, the application can work on it and complete it.

The entry point to the Java API for external tasks is the ExternalTaskService. It can be accessed via processEngine.getExternalTaskService().

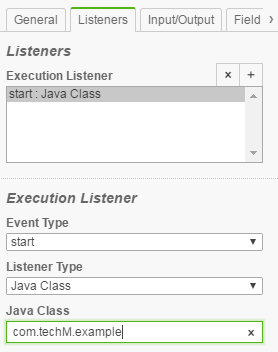


**Connector:** Here we have to configure connector (rest/ soap service connectors).



**Listeners:**

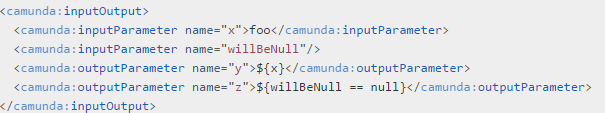
Listeners will execute the corresponding implementation of class or expression, at the start or end of the event.



**Input/output:**

To improve the reusability of source code and business logic, Camunda BPM offers input/output mapping of process variables. This can be used for tasks, events and sub processes.

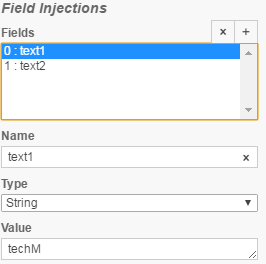
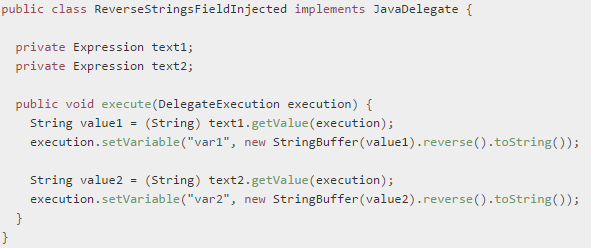
In order to use the variable mapping, the Camunda extension element inputOutput has to be added to the element. It can contain multiple inputParameter and outputParameter elements that specify which variables should be mapped. The name attribute of an inputParameter denotes the variable name inside the activity (a local variable to be created), whereas the name attribute of an outputParameter denotes the variable name outside of the activity.



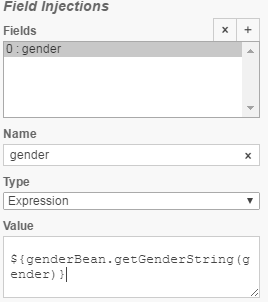
**Field Injection:**

It is possible to inject values into the fields of the delegated classes. The following types of injection are supported:

* Fixed string values

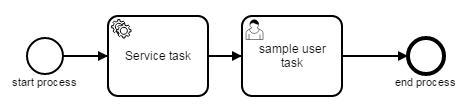
 

* Expressions

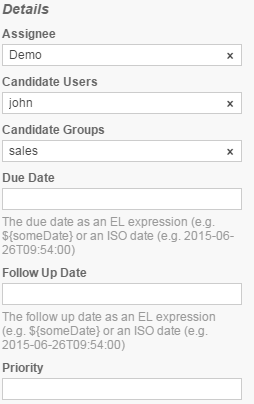


**User Task:**

A User Task is used to model work that needs to be done by a human actor. When the process execution arrives at such a User Task, a new task is created in the task list of the user(s) or group(s) assigned to that task.



We can assign this task to a particular assignee, group and candidate user

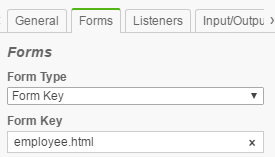
**Due date:** Each task has a field indicating the due date of that task. The Query API can be used to query for tasks that are due on, before or after a certain date.

**Follow up Date:** Each task has a field indicating the follow up date of that task. The Query API can be used to query for tasks that need to be followed up on, before or after a certain date.

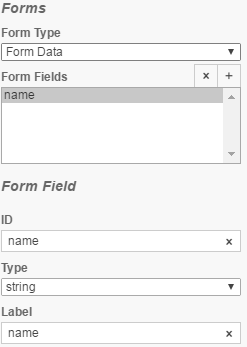
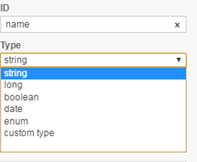
**Forms:**

Forms will provide user interface for the user. There two ways in designing forms.

**Form Key:** Here we can use external forms by specifying the form path.



Form Data:

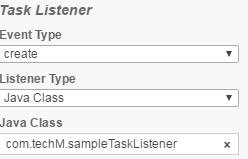
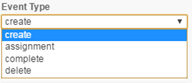
Using form data we can create the user interface using camunda modeler, by specifying the fields that we need and, the type of the field. We can add constraints like required, readonly etc.

**Listeners:**

User task is having two types of listener whereas service task is having only one (execution listener)

**Execution Listener:** Same as Service task refer above.

**Task Listener:** A task listener is used to execute custom Java logic or an expression upon the occurrence of a certain task-related event. It can only be added in the process definition as a child element of a user task.  This task listener class should be implemented by **TaskListener** interface

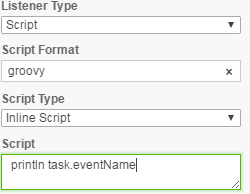
 

**Event (required)**: the type of task event on which the task listener will be invoked. Possible events are:

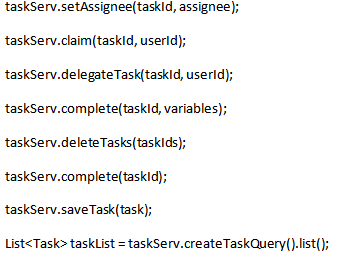
* **create**: occurs when the task has been created and all task properties are set.
* **assignment**: occurs when the task is assigned to somebody. Note: when process execution arrives in a userTask, an assignment event will be fired first, before the create event is fired. This might seem like an unnatural order but the reason is pragmatic: when receiving the create event, we usually want to inspect all properties of the task, including the assignee.
* **complete**: occurs when the task is completed and just before the task is deleted from the runtime data.
* **Delete**: occurs just before the task is deleted from the runtime data.

Listener type: Listener type can be java class, expression, delegate expression and script. And the functionality is common as specified in service task.

**Script type:** Here we can specify the script format and script, which need to execute when this listener is fired.

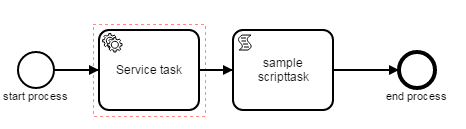
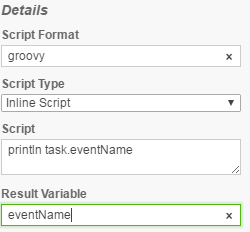


We have set of methods which are used to perform certain actions on user task. They are



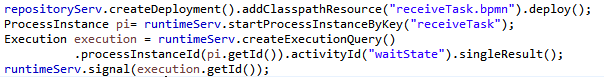
**Script Task**

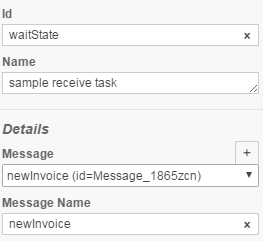
A Script Task is an automated activity. When a process execution arrives at the Script Task, the corresponding script is executed.

**Receiver Task:**

A Receive Task is a simple task that waits for the arrival of a certain message. When the process execution arrives at a Receive Task, the process state is committed to the persistence storage. This means that the process will stay in this wait state until a specific message is received by the engine, which triggers continuation of the process beyond the Receive Task.





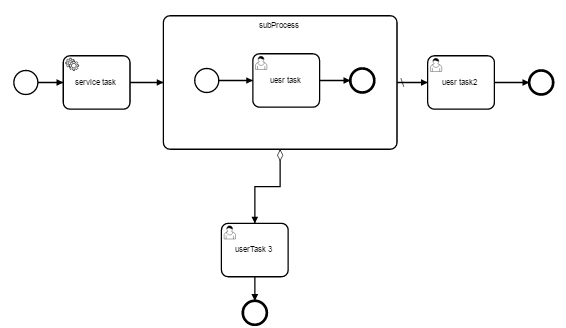
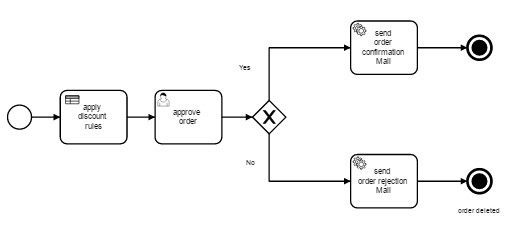
**Manual Task:**

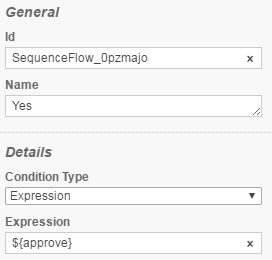
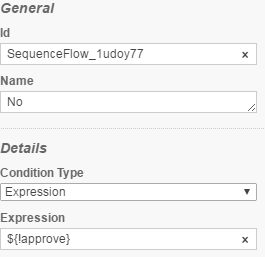
A Manual Task defines a task that is external to the BPM engine. It is used to model work that is done by somebody who the engine does not need to know of and is there no known system or UI interface. For the engine, a manual task is handled as a pass-through activity, automatically continuing the process at the moment the process execution arrives at it.



**Data-based Exclusive Gateway (XOR):**

An exclusive gateway (also called the XOR gateway or, in more technical terms, the exclusive data-based gateway), is used to model a decision in the process. When the execution arrives at this gateway, all outgoing sequence flows are evaluated in the order in which they have been defined. The sequence flow which condition evaluates to ‘true’ (or which doesn’t have a condition set, conceptually having a ‘true’ value defined on the sequence flow) is selected for continuing the process.



**Sub Process:**

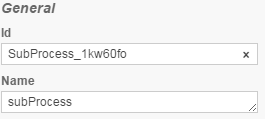
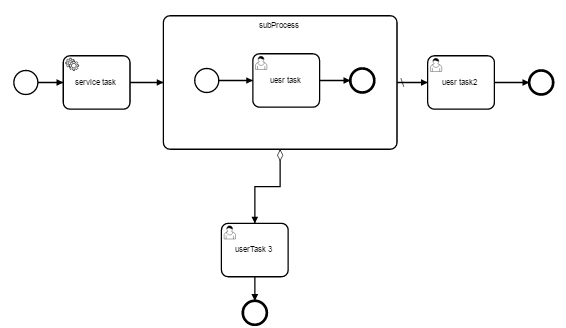
A Subprocess is an activity that contains other activities, gateways, events, etc. which itself forms a process that is part of the bigger process. A Subprocess is completely defined inside a parent process (that’s why it’s often called an embedded Subprocess).

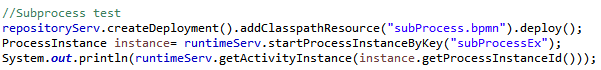
Subprocesses have two major use cases:

* Subprocesses allow hierarchical modeling. Many modeling tools allow that subprocesses can be collapsed, hiding all the details of the subprocess and displaying a high-level end-to-end overview of the business process.
* A subprocess creates a new scope for events. Events that are thrown during execution of the subprocess can be caught by a boundary event on the boundary of the subprocess, thus creating a scope for that event, limited to the subprocess.

Using a subprocess does impose some constraints:

* A subprocess can only have one none start event, no other start event types are allowed. A subprocess must have at least one end event. Note that the BPMN 2.0 specification allows to omit the start and end events in a subprocess, but the current engine implementation does not support this.
* Sequence flows can’t cross subprocess boundaries.



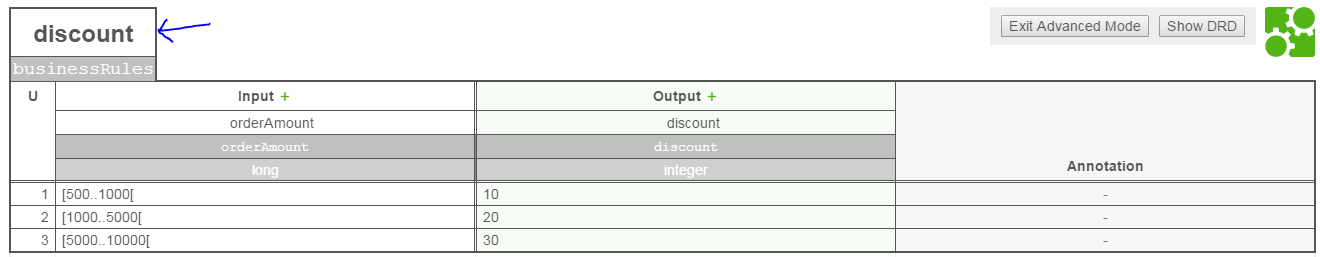


**DMN (Decision Model and Notation)**

Decision Model and Notation (DMN) is a standard for Business Decision Management.

A decision table represents decision logic which can be depicted as a table in DMN 1.1. It consists of inputs, outputs and rules.

A decision table is represented by a decisionTable element inside a decision XML element.

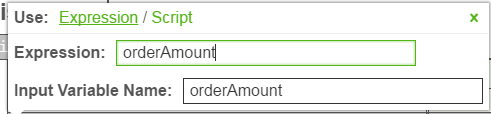
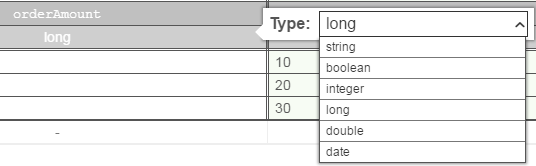


The name (discount) describes the decision for which the decision table provides the decision logic. It is set as the name attribute on the decision element.

The id (businessRules) is the technical identifier of the decision. It is set in the id attribute on the decisionelement.

Each decision should have a unique id when it is deployed to the Camunda BPM platform. The engine uses the id as the decision key of the deployed DecisionDefinition.

Click on enter Advance mode to edit the field names and data type

An **input expression** specifies how the value of the input clause is generated. It is an expression which will be evaluated by the DMN engine.

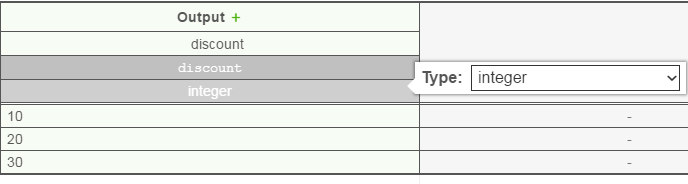
The **type** of the input clause can be specified by the typeRef attribute on the inputExpressionXML element. After the input expression is evaluated by the DMN engine it converts the result to the specified type.

When the **input** expression is evaluated then the return value is stored in a **variable**. The name of the variable can be specified by the camunda:inputVariable extension attribute on the input element. By default, the name is cellInput.

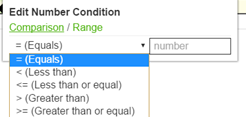
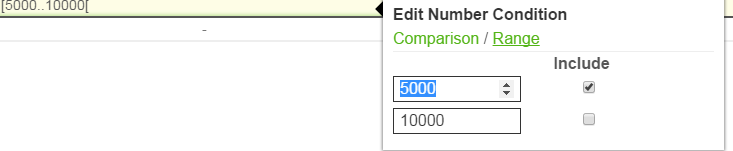
**Output:**

A decision table can have one or more output, also called output clauses. An output clause defines the id, label, name and type of a decision table output.





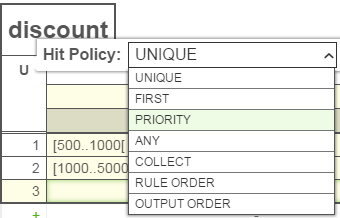
Note that the type is not required but recommended since it provides a type safety of the output values.



A decision table can have one or more rules. Each rule contains input and output entries. The input entries are the condition and the output entries the conclusion of the rule. If each input entry (condition) is satisfied then the rule is satisfied and the decision result contains the output entries (conclusion) of this rule.

**Hit policy:**

A decision table has a hit policy that specifies what the results of the evaluation of a decision table consist of.



**Unique**: Only a single rule can be satisfied. The decision table result contains the output entries of the satisfied rule.

**Any:** Multiple rules can be satisfied. However, all satisfied rules must generate the same output. The decision table result contains only the output of one of the satisfied rules.

If multiple rules are satisfied which generate different outputs, then hit policy is violated.

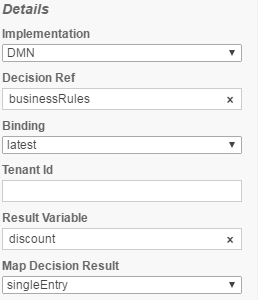
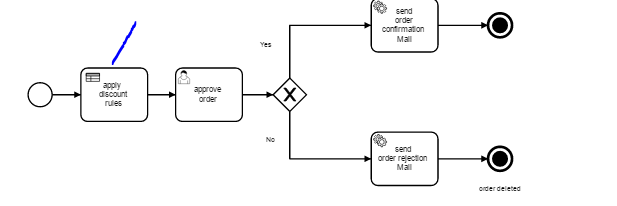
**First:** Multiple rules can be satisfied. The decision table result contains only the output of the first satisfied rule.

**Rule Order:** Multiple rules can be satisfied. The decision table result contains the output of all satisfied rules in the order of the rules in the decision table.

**Collect:** Multiple rules can be satisfied. The decision table result contains the output of all satisfied rules in an arbitrary order as a list.

**Integrating DMN with BPMN:**

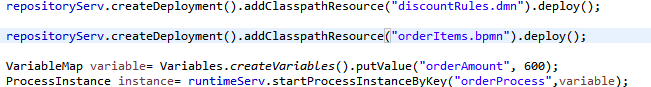
Through businessRuleTask we can include DMN into BPMN



Here we have to specify **Decision ref** as **DMN decision Id**



And we can input the values through BPMN process and these can passed to include DMN and result will be returned back to in the **Result variable** specified (defined in business rule task).

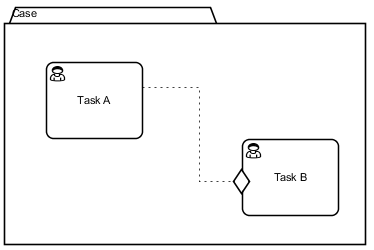


Here the order amount 600 will serve as input to DMN **orderAmount** variable

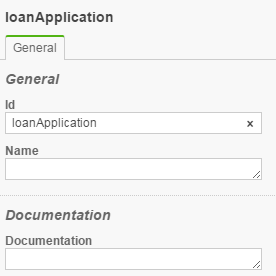
 And in Bpmn will get the value from DMN 

**CMMN (Case Management Model and Notation)**

CMMN is mainly used to record human activities which can’t be automated.



CMMN doesn’t provide features to manipulate the process, it contains only General panel to provide id and name of the CMMN.



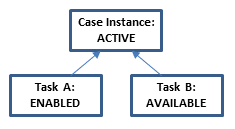
In CMMN two human activities can be connected by Sentry.

The sentry expresses that Task B can be enacted when Task A finishes. This is formally specified by lifecycles.

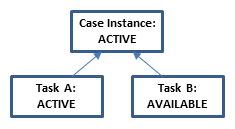
**Sentry** is having various standard types according to the type connected human will enact.

The following steps might take place:

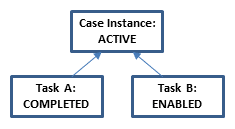
1. A user instantiates the case by caseService.createCaseInstanceByKey("case"). A new case instance is created in state ACTIVE.
2. Two instances for each human task are automatically created, both in state AVAILABLE.
3. Task A does not have a condition to start, so it immediately reaches state ENABLED. Note that the steps 1 to 3 all happens synchronously with the caseService invocation from step 1. The case is now in the following state:



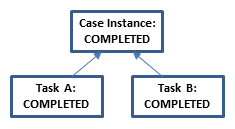
1. A user manually starts Task A by calling caseService.manuallyStartCaseExecution(taskAExecutionId);. As a consequence, Task A reaches state ACTIVE and a task instance is added to the assignee’s task list. Note that starting a task is only allowed if that task is in state ENABLED. Thus, trying to manually start Task B here by caseService.manuallyStartCaseExecution(taskBExecutionId);would fail. The state is now:



1. The assignee completes the task instance by calling taskService.complete(taskId); Task A reaches the state COMPLETED.
2. Task A’s state transition triggers Task B’s sentry. In consequence, Task B becomes ENABLED. This happens synchronously in the invocation from step 5. Accordingly, the case’s new state is:



1. Similar to Task A, a user may now use the CaseService and TaskService to start Task B, complete the corresponding task instance, and complete Task B. Ultimately, Task B reaches the state COMPLETED.
2. With both tasks in state COMPLETED, the case instance automatically reaches the state COMPLETED as well. The state has case has reached the following state:

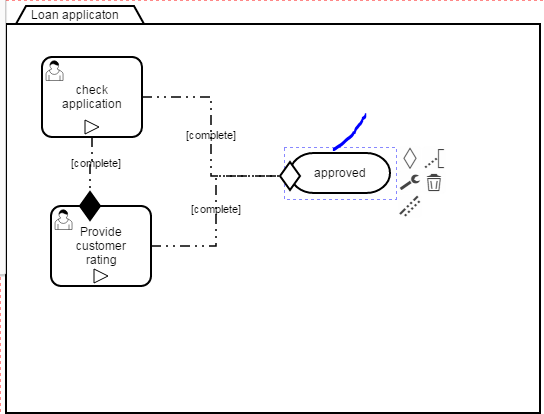
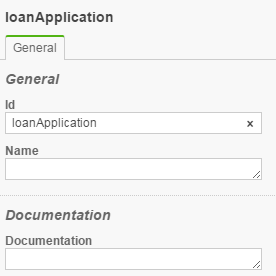


1. A user may close the case instance by invoking caseService.closeCaseInstance(caseInstanceId). The case instance reaches the state CLOSED.

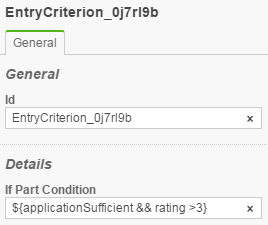
Notice how the lifecycle states define the overall state of the case and restrict the interactions that are possible. For example, the tasks A and B can only be worked on when in state ACTIVE. Before, they go through states AVAILABLE and ENABLED that represent that conditions for working on the task are not yet met, for example that the task was not manually started or that a sentry is not fulfilled yet.

**Milestone**

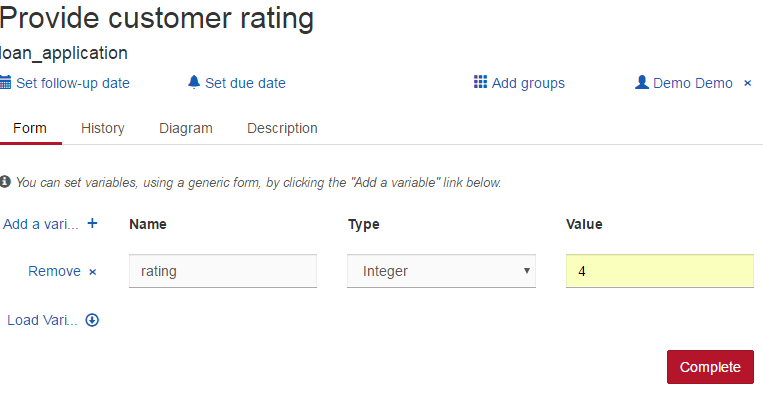
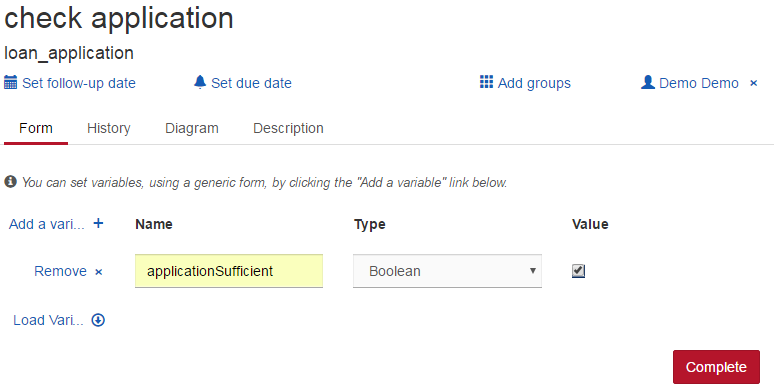
A milestone is used to represent achievable targets within the case. It is not associated with any work but rather marks that certain conditions have been reached within the case. As a milestone is a regular plan item definition, a milestone’s completion may be used as entry criteria for other tasks and stages. This way, a milestone can be used to bring logical stages within a case into order.

 A milestone gets completed as soon as its entry criteria (diamond shape attached to milestone) are fulfilled. We have to set manual activation rule to false.

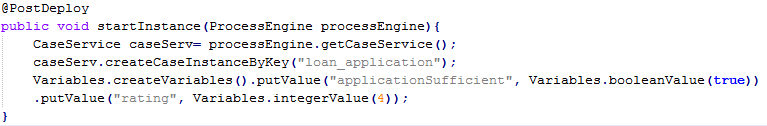
 

When ever CMMN deployed in the camunda server these two human tasks will appear in the task list and with complete button. There is an option to create two variable given in the criteria with values.

When ever the criteria met these task will complete after clicking on compete button. These tasks won’t get complete unless the criteria met.

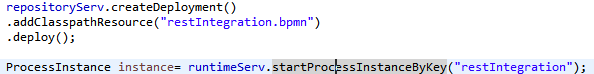
We can input the values to CMMN through java in the following way.



**Camunda Bpmn process deployment**

Bpmn process can be deployed in following ways

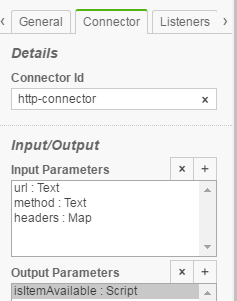
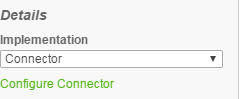
* Create .War file and deploy it in camunda server
* Through programming deploy .Bpmn by using repository service and, start it by using runtime service



* Camunda Bpmn with Rest services

There are two approaches to integrate rest services in BPMN

* By using Connector in Service task implementation by providing connector id (http-connector), rest service url, method (post/get) and headers (accepted media type).



* By Consuming rest services in delegate class

