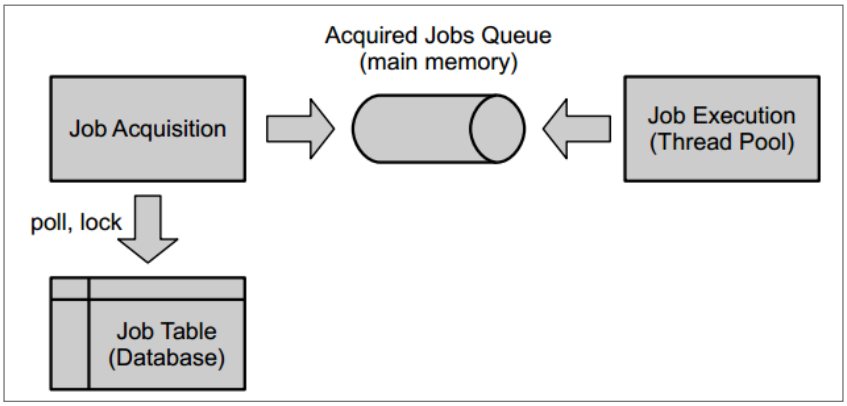
The Job Executor

A job is an explicit representation of a task to trigger process execution. A job is created when a timer event or a task marked for asynchronous execution (see transaction boundaries) is approached. Job processing can therefore be separated into three phases:

- Job Creation
- Job Acquisition
- Job Execution

While jobs are created during process execution, job acquisition and execution are the job executor's responsibility. The following diagram illustrates these two steps:



Job Executor Activation

When using an **embedded process engine**, by default, the Job Executor is not activated when the process engine boots. Specify

```
<property name="jobExecutorActivate" value="true" />
```

in the process engine configuration when you want the JobExecutor to be activated upon booting the process engine.

When using a **shared process engine**, the default is reversed: if you do not specify the `jobExecutorActivate` property on the process engine configuration, the job executor is automatically started. In order to turn it off, you have to explicitly set the property to false:

```
<property name="jobExecutorActivate" value="false" />
```

Job Executor in a Unit Test

For unit testing scenarios it is cumbersome to work with this background component. Therefore the Java API offers to query for (`ManagementService.createJobQuery`) and execute jobs (`ManagementService.executeJob`) *by hand*, which allows to control job execution from within a unit test.

Job Creation

Jobs are created for a range of purposes by the process engine. The following job types exist:

- Asynchronous continuations to set transaction boundaries in the process
- Timer jobs for BPMN timer events
- Asynchronous handling of BPMN events

During creation, jobs can receive a priority for acquisition and execution.

Job Prioritization

In practice, the amount of jobs processed is seldomly spread evenly across the day. Instead, there are peaks of high load, for example when batch operations are run overnight. In such a situation, the job executor can be temporarily overloaded: the database contains many more jobs than the job executor can handle at a time. *Job Prioritization* can help cope with these situations in a well-defined matter by defining an order of importance and enabling execution by that order.

In general, there are two types of use cases that can be tackled with job prioritization:

- **Anticipating priorities at Design Time**: In many cases, a high-load scenario can be anticipated when designing a process model. In these scenarios, it is often important to prioritize job execution according to certain business objectives. Examples:
  - A retail store has casual and VIP customers. In case of high load, it is desired to handle orders of VIP customers with higher priority since their satisfaction is more important to the company's business goals.
  - A furniture store has human-centric processes for consulting customers in buying furniture as well as non-time-critical processes for delivery. Prioritization can be used to ensure fast response times in the consulting processes, improving user and customer satisfaction.
- **Prioritization as a Response to Runtime Conditions**: Some scenarios for high job executor load result from unforeseen conditions at runtime that cannot be dealt with during process design. Temporarily overriding priorities can help deal with these kind of situations gracefully. Example:
  - A service task accesses a web service to process a payment. The payment service encounters an overload and responds very slowly. To avoid occupying all the job executor's resources with waiting for the service to respond, the respective jobs' priorities can be temporarily reduced. This way, unrelated process instances and jobs are not slowed down. After the service recovers, the overriding priority can be cleared again.

The Job Priority

A priority is a natural number in the range of a Java `long` value. A higher number represents a higher priority. Once assigned, the priority is static, meaning that the process engine will not go through the process of assigning a priority for that job again at any point in the future.

Job priorities affect two phases during process execution: job creation and job acquisition. During job creation, a job is assigned a priority. During job acquisition, the process engine can evaluate the given job priorities to order their execution accordingly. That means, jobs are strictly acquired by the order of priorities.

> A note on Job Starvation
>
> In scheduling scenarios, starvation is a typical concern. When high priority jobs are continuously created, it may happen that low priority jobs are never acquired.
>
> Performance-wise, acquiring jobs strictly by priority enables the job executor to use indexes for ordering. Solutions like aging that dynamically boost priorities of starving jobs cannot be easily supplemented with an index.
>
> In addition, in an environment where the job executor can never catch up to execute all jobs in the job table such that low priority jobs are not executed in a reasonable amount of time, there may be a general issue with overloaded resources. In this case, a solution could be to distribute the work load based on Job Executor priority ranges (see Job Executor priority range) or increase the job executor resources by adding a new node to a cluster.

Configure the Process Engine for Job Priorities

This section explains how to enable and disable job priorities in the configuration. There are two relevant configuration properties which can be set on the process engine configuration:

`producePrioritizedJobs`: Controls whether the process engine assigns priorities to jobs. The default value is `true`. If priorities are not needed, the process engine configuration property `producePrioritizedJobs` can be set to `false`. In this case, all jobs receive a priority of 0. For details on how to specify job priorities and how the process engine assigns them, see the following section on Specifying Job Priorities.

`jobExecutorAcquireByPriority`: Controls whether jobs are acquired according to their priority. The default value is `false` which means that it needs to be explicitly enabled. Hint: when enabling this, additional database indexes should be created as well: See the section The Order of Job Acquisition for details.

Specify Job Priorities

Job priorities can be specified in the BPMN model as well as overridden at runtime via API.

Priorities in BPMN XML

Job Priorities can be assigned at the process or the activity level. To achieve this the Camunda extension attribute `camunda:jobPriority` can be used.

For specifying the priority, both constant values and expressions are supported. When using a constant value, the same priority is assigned to all instances of the process or activity.

Expressions, on the other hand, allow assigning a different priority to each instance of the process or activity. Expression must evaluate to a number in the Java `long` range. The concrete value can be the result of a complex calculation and be based on user-provided data (resulting from a task form or other sources).

Priorities at the Process Level

When configuring job priorities at the process instance level, the `camunda:jobPriority` attribute needs to be applied to the bpmn `<process ...>` element:

```
<bpmn:process id="Process_1" isExecutable="true" camunda:jobPriority="8">
  ...
</bpmn:process>
```

The effect is that all activities inside the process inherit the same priority (unless it is overridden locally). See also: Job Priority Precedence Schema.

The above example shows how a constant value can be used for setting the priority. This way the same priority is applied to all instances of the process. If different process instances need to be executed with different priorities, an expression can be used:

```
<bpmn:process id="Process_1" isExecutable="true" camunda:jobPriority="${order.priority}
  ...
</bpmn:process>
```

In the above example the priority is determined based on the property `priority` of the variable `order`.

Priorities at the Activity Level

When configuring job priorities at the activity level, the `camunda:jobPriority` attribute needs to be applied to the corresponding bpmn element:

```
<bpmn:serviceTask id="ServiceTask_1"
  name="Prepare Payment"
  camunda:asyncBefore="true"
  camunda:jobPriority="100" />
```

The effect is that the priority is applied to all instances of the given service task. The priority overrides a process level priority. See also: Job Priority Precedence Schema.

When using a constant value, as shown in the above example, the same priority is applied to all instances of the service task. It is also possible to use an expression:

```
<bpmn:serviceTask id="ServiceTask_1"
  name="Schedule Delivery"
  camunda:asyncBefore="true"
  camunda:jobPriority="${customer.status == 'VIP' ? 10 : 0}" />
```

In the above example the priority is determined based on the property `status` of the current customer.

Resolution Context of Priority Expressions

This section explains which context variables and functions are available when evaluating priority expressions. For some general documentation on this, see the corresponding documentation section.

All priority expressions are evaluated in the context of an existing execution. This means that variable `execution` is implicitly defined as well as all of the execution's variables by their name.

The only exceptions are the priority of jobs which lead to the instantiation of a new process instance. Examples:

- Timer Start Event
- Asynchronous Signal Start Event

Priority Propagation to Called Process Instances

When starting a process instance via a call activity, you sometimes want the process instance to "inherit" the priority of the calling process instance. The easiest way to achieve this is by passing the priority using a variable and referencing it using an expression in the called process. See also Call Activity Parameters for details on how to pass variables using call activities.

Set Job Definition Priorities via ManagementService API

Sometimes job priorities need to be changed at runtime to deal with unforeseen circumstances. For example: consider the service task *Process Payment* of an order process: the service task invokes some external payment service which may be overloaded and thus respond slowly. The job executor is therefore blocked waiting for responses. Other concurrent jobs with the same or lower priorities cannot proceed, although this is desirable in this exceptional situation.

Override Priority by Job Definition

While expressions may help in these cases to a certain extent, it is cumbersome to change process data for all involved process instances and to make sure to restore it when the exceptional condition is over. Thus the ManagementService API allows to temporarily set an overriding priority for a job definition. The following operation can be performed to downgrade the priority for all future jobs of a given job definition:

```
// find the job definition
JobDefinition jobDefinition = managementService
  .createJobDefinitionQuery()
  .activityIdIn("ServiceTask_1")
  .singleResult();

// set an overriding priority
managementService.setOverridingJobPriorityForJobDefinition(jobDefinition.getId(), 0L);
```

Setting an overriding priority makes sure that every new job that is created based on this definition receives the given priority. This setting overrides any priority specified in the BPMN XML.

Optionally, the overriding priority can be applied to all the existing jobs of that definition by using the `cascade` parameter:

```
managementService.setOverridingJobPriorityForJobDefinition(jobDefinition.getId(), 0L,
```

Note that this will not lead to preemption of jobs that are currently executed.

Reset Priority by Job Definition

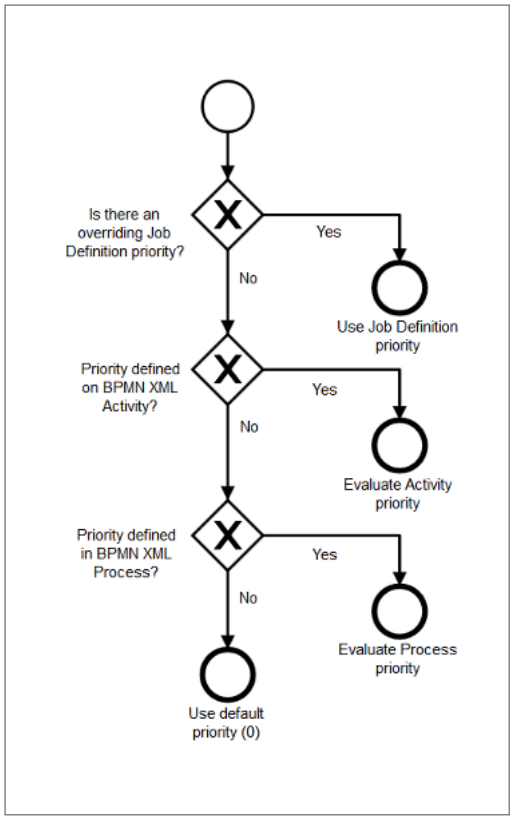
When the service has recovered from the overload situation, overriding priority can be cleared again as follows:

```
managementService.clearOverridingJobPriorityForJobDefinition(jobDefinition.getId());
```

From now on, all new jobs receive the priorities specified in the BPMN XML again.

Job Priority Precedence Schema

The following diagram sums up the precedence of priority sources when a job's priority is determined:



Set Job Priorities via ManagementService API

The ManagementService also offers a method to change a single job's priority via `ManagementService#setJobPriority(String jobId, long priority)`.

Job Acquisition

Job acquisition is the process of retrieving jobs from the database that are to be executed next. Therefore jobs must be persisted to the database together with properties determining whether a job can be executed. For example, a job created for a timer event may not be executed before the defined time span has passed.

Persistence

Jobs are persisted to the database, in the `ACT_RU_JOB` table. This database table has the following columns (among others):

```
ID_ | REV_ | LOCK_EXP_TIME_ | LOCK_OWNER_ | RETRIES_ | DUEDATE_
```

Job acquisition is concerned with polling this database table and locking jobs.

Acquirable Jobs

A job is acquirable, i.e., a candidate for execution, if it fulfills all following conditions:

- it is due, meaning that the value in the `DUEDATE_` column is in the past
- it is not locked, meaning that the value in the `LOCK_EXP_TIME_` column is in the past
- its retries have not been depleted, meaning that the value in the `RETRIES_` column is greater than zero.

In addition, the process engine has a concept of job suspension. For example, a job gets suspended when the process instance it belongs gets suspended. A job is only acquirable if it is not suspended.

Job Acquisition performance optimization

To optimize the acquisition of jobs that need to be executed immediately, the `DUEDATE_` column is not set (`null`) and a (positive) null check is added as a condition for acquisition.

In case each job must have a `DUEDATE_` set, the optimization can be disabled. This can be done by setting the `ensureJobDueDateNotNull` process engine configuration flag to `true`.

However, any jobs created with a `null` value for `DUEDATE_` before disabling the optimization will not be picked up by the Job Acquisition phase, unless the jobs are explicitly updated with a due date through the **Set Due Date** Java /REST or **Set Retries** Java /REST APIs.

The Two Phases of Job Acquisition

Job acquisition has two phases. In the first phase the job executor queries for a configurable amount of acquirable jobs. If at least one job can be found, it enters the second phase, locking the jobs. Locking is necessary in order to ensure that jobs are executed exactly once. In a clustered scenario, it is customary to operate multiple job executor instances (one for each node) that all poll the same ACT_RU_JOB table. Locking a job ensures that it is only acquired by a single job executor instance. Locking a job means updating its values in the LOCK_EXP_TIME_ and LOCK_OWNER_ columns. The LOCK_EXP_TIME_ column is updated with a timestamp signifying a date that lies in the future. The intuition behind this is that we want to lock the job until that date is reached. The LOCK_OWNER_ column is updated with a value uniquely identifying the current job executor instance. In a clustered scenario this could be a node name uniquely identifying the current cluster node.

The situation where multiple job executor instances attempt to lock the same job concurrently is accounted for by using optimistic locking (see REV_ column).

After having locked a job, the job executor instance has effectively reserved a time slot for executing the job: once the date written to the LOCK_EXP_TIME_ column is reached it will be visible to job acquisition again. In order to execute the acquired jobs, they are passed to the acquired jobs queue.

The Job Order of Job Acquisition

By default the job executor does not impose an order in which acquirable jobs are acquired. This means that the job acquisition order depends on the database and its configuration. That's why job acquisition is assumed to be non-deterministic. The intention for this is to keep the job acquisition query simple and fast.

This method of acquiring jobs is not sufficient in all cases, such as:

- **Job Prioritization**: When creating prioritized jobs, the job executor must acquire jobs according to the given priorities
- **Job Starvation**: In a high load scenario, job starvation is theoretically possible when new jobs are repeatedly created in a rate higher than the job executor can handle.
- **Preferred Handling of Timers**: In a high load scenario, timer execution can be delayed to a significantly later point in time than the actual due date. While a due date is not a real-time boundary by which the job is guaranteed to be executed, in some scenarios it may be preferable to acquire timer jobs as soon as they become available to execute.

To address the previously described issues, the job acquisition query can be controlled by the process engine configuration properties. Currently, three options are supported:

- `jobExecutorAcquireByPriority`. If set to `true`, the job executor will acquire the jobs with the highest priorities.
- `jobExecutorPreferTimerJobs`. If set to `true`, the job executor will acquire all acquirable timer jobs before other job types. This doesn't specify an order within types of jobs which are acquired.
- `jobExecutorAcquireByDueDate`. If set to `true`, the job executor will acquire jobs by ascending due date. An asynchronous continuation receives its creation date as due date, so it is immediately executable.

Using a combination of these options results in a multi-level ordering. The precedence hierarchy of options is as in the order above: If all three options are active, priorities are the primary, job types the secondary, and due dates the tertiary ordering. This also shows that activating all options is not the optimal solution to the problems of prioritization, starvation, and timer handling. For example, in this case timer jobs are only preferred within one level of priority. Timers with lower priority are acquired after all jobs of higher priorities have been acquired. It is recommended to decide based on the concrete use case which options to activate.

For example:

- For prioritized job execution, only `jobExecutorAcquireByPriority` should be set to true
- To execute timer jobs as soon as possible, the two options `jobExecutorPreferTimerJobs` and `jobExecutorAcquireByDueDate` should be activated. The job executor will first acquire timer jobs and after that asynchronous continuation jobs. And also sort these jobs within the type ascending by due date.

> All of these options are set to `false` by default and should only be activated if required by the use case. The options alter the used job acquisition query and may affect its performance. That's why we also advise to add an index on the corresponding column(s) of the `ACT_RU_JOB` table.

jobExecutorAcquireByPriority	jobExecutorPreferTimerJobs	jobExecutorAcquireByDueDate	Recommended Index
`true`	`false`	`false`	`ACT_RU_JOB(PRIORITY_ DESC)`
`false`	`true`	`false`	`ACT_RU_JOB(TYPE_ DESC)`
`false`	`false`	`true`	`ACT_RU_JOB(DUEDATE_ ASC)`
`false`	`true`	`true`	`ACT_RU_JOB(TYPE_ DESC, DUEDATE_ ASC)`

Job Executor priority range

By default, the Job Executor executes all jobs regardless of their priorities. Some jobs might be more important to finish quicker than others, so we assign them priorities and set `jobExecutorAcquireByPriority` to `true` as described above. Depending on the workload, the Job Executor might be able to execute all jobs eventually. But if the load is high enough, we might face starvation where a Job Executor is always busy working on high-priority jobs and never manages to execute the lower priority jobs.

To prevent this, you can specify a priority range for the job executor by setting values for `jobExecutorPriorityRangeMin` or `jobExecutorPriorityRangeMax` (or both). The Job Executor will only acquire jobs that are inside its priority range (inclusive). Both properties are optional, so it is fine only to set one of them.

To avoid job starvation, make sure to have no gaps between Job Executor priority ranges. If, for example, Job Executor A has a priority range of 0 to 100 and Job Executor B executes jobs from priority 200 to `Long.MAX_VALUE` any job that receives a priority of 101 to 199 will never be executed. Job starvation can also occur with `batch jobs` and `history cleanup jobs` as both types of jobs also receive priorities (default: `0`). You can configure them via their respective properties: `batchJobPriority` and `historyCleanupJobPriority`.

This feature is particularly useful if you want to separate multiple types of jobs from each other. For example, short-running, urgent jobs with high priority and long-running jobs that are not urgent but should finish eventually. Setting up a Job Executor priority range for both types will ensure that long-running jobs can not block urgent ones.

Backoff Strategy

The Job Executor uses a backoff strategy to avoid acquisition conflicts in clusters and to reduce the database load when no jobs are due. The second point may result in a delay between job creation and job execution as the job acquisition by default doubles the delay to the next acquisition run. The default maximum wait time is 60 seconds. You can decrease the delay by setting the configuration parameter `maxWait` to a value lower than 60000 milliseconds.

Job Execution

Thread Pool

Acquired jobs are executed by a thread pool. The thread pool consumes jobs from the acquired jobs queue. The acquired jobs queue is an in-memory queue with a fixed capacity. When an executor starts executing a job, it is first removed from the queue.

In the scenario of an embedded process engine, the default implementation for this thread pool is a `java.util.concurrent.ThreadPoolExecutor`. However, this is not allowed in Java EE environments. There we hook into the application server capabilities of thread management. See the platform-specific information in the Runtime Container Integration section on how this achieved.

Failed Jobs

Upon failure of job execution, e.g., if a service task invocation throws an exception, a job will be retried a number of times (by default 2 so that the job is tried three times in total). It is not immediately retried and added back to the acquisition queue, but the value of the RETRIES_ column is decreased and the executor unlocks the job. The process engine thus performs bookkeeping for failed jobs. The unlocking also includes erasing the time LOCK_EXP_TIME_ and the owner of the lock LOCK_OWNER_ by setting both entries to `null`. Subsequently, the failed job will automatically be retried once the job is acquired for execution. Once the number of retries is exhausted (the value of the RETRIES_ column equals 0), the job is not executed any more and the engine stops at this job, signaling that it cannot proceed.

> While all failed jobs are retried, there is one case in which a job's retries are not decremented. This is, if a job fails due to an optimistic locking exception. Optimistic Locking is the process engine's mechanism to resolve conflicting resource updates, for example when two jobs of a process instance are executed in parallel (see the following sections on concurrent job execution). As an optimistic locking exception is no exceptional situation from an operator's point of view and resolves eventually, it does not cause a retry decrement.

If incident creation is enabled for jobs, then once job retries are depleted, an incident is created (see (De-)Activate Incidents). Incidents and historic incidents related to the job can be requested via Java API like this:

```
List<Incident> incidents = engineRule.getRuntimeService()
        .createIncidentQuery().configuration(jobId).list();

List<HistoricIncident> historicIncidents = engineRule.getHistoryService()
        .createHistoricIncidentQuery().configuration(jobId).list();
```

Retry Time Cycle Configuration

By default, a failed job will be retried three times and the retries are performed immediately after the failure. In daily business it might be useful to configure a retry strategy, i.e., by setting how often a job is retried and how long the engine should wait until it tries to execute a job again. This configuration can be specified globally in the process engine configuration.

```
<process-engine name="default">
  ...
  <properties>
    ...
    <property name="failedJobRetryTimeCycle">R5/PT5M</property>
  </properties>
</process-engine>
```

The configuration follows the ISO_8601 standard for repeating time intervals. In the example, `R5/PT5M` means that the maximum number of retries is 5 (`R5`) and the delay of retry is 5 minutes (`PT5M`).

The Camunda engine allows you to configure this setting for the following specific elements:

- Activities (tasks, call activities, subprocesses)
- Events
- Multi-Instance Activities

Use a Custom Job Retry Configuration for Activities

As soon as the retry configuration is enabled, it can be applied to tasks, call activities, embedded subprocesses and transactions subprocesses. For instance, the job retry in a task can be configured in the Camunda engine in the BPMN 2.0 XML as follows:

```
<definitions xmlns:camunda="http://camunda.org/schema/1.0/bpmn">
  ...
  <serviceTask id="failingServiceTask" camunda:asyncBefore="true" camunda:class="org.my
    <extensionElements>
      <camunda:failedJobRetryTimeCycle>R5/PT5M</camunda:failedJobRetryTimeCycle>
    </extensionElements>
  </serviceTask>
  ...
</definitions>
```

You can also set an expression as in the retry configuration. For example:

```
  <camunda:failedJobRetryTimeCycle>${retryCycle}</camunda:failedJobRetryTimeCycle>
```

The LOCK_EXP_TIME_ is used to define when the job can be executed again, meaning the failed job will automatically be retried once the LOCK_EXP_TIME_ date is expired.

Use a Custom Job Retry Configuration for Events

The job retries can also be configured for the following events:

- Timer Start Event
- Boundary Timer Event
- Intermediate Timer Catch Event
- Intermediate Throw Event

Similar to tasks, the retries can be configured as an extension element of the event. The following example defines three retries after 5 seconds each for a boundary timer event:

```
<definitions xmlns:camunda="http://camunda.org/schema/1.0/bpmn">
  ...
  <boundaryEvent id="BoundaryEvent" name="BoundaryName" attachedToRef="MyActivity">
    <extensionElements>
      <camunda:failedJobRetryTimeCycle>R3/PT5S</camunda:failedJobRetryTimeCycle>
    </extensionElements>
    <outgoing>SequenceFlow_3</outgoing>
    <timerEventDefinition>
      <timeDuration>PT10S</timeDuration>
    </timerEventDefinition>
  </boundaryEvent>
  ...
</definitions>
```

Reminder: a retry may be required if there are any failures during the transaction which follows the timer.

Use a Custom Job Retry Configuration for Multi-Instance Activities

If the retry configuration is set for a multi-instance activity then the configuration is applied to the multi-instance body. Additionally, the retries of the inner activities can also be configured using the extension element as child of the `multiInstanceLoopCharacteristics` element.

The following example defines the retries of a multi-instance service task with asynchronous continuation of the multi-instance body and the inner activity. If a failure occur during one of the five parallel instances then the job of the failed instance will be retried up to 3 times with a delay of 5 seconds. In case all instances ended successful and a failure occur during the transaction which follows the task, the job will be retried up to 5 times with a delay of 5 minutes.

```
<definitions xmlns:camunda="http://camunda.org/schema/1.0/bpmn">
  ...
  <serviceTask id="failingServiceTask" camunda:asyncAfter="true" camunda:class="org.my
    <extensionElements>
      <!-- configuration for multi-instance body, e.g. after task ended -->
      <camunda:failedJobRetryTimeCycle>R5/PT5M</camunda:failedJobRetryTimeCycle>
    </extensionElements>
    <multiInstanceLoopCharacteristics isSequential="false" camunda:asyncBefore="true">
      <extensionElements>
        <!-- configuration for inner activities, e.g. before each instance started -->
        <camunda:failedJobRetryTimeCycle>R3/PT5S</camunda:failedJobRetryTimeCycle>
      </extensionElements>
      <loopCardinality>5</loopCardinality>
    </multiInstanceLoopCharacteristics>
  </serviceTask>
  ...
</definitions>
```

Retry Intervals

The retry time cycle (e.g. R5/PT5M) allows to define the number of retries and an interval when the failed job should be retried. Regardless of the values, the interval is always (at least) 5

minutes. You can configure the list of retry intervals (separated by comma) on a global level or for a specific job configuration. The local configuration takes precedence. Here is an example of a global process engine configuration:

```
<process-engine name="default">
  ...
  <properties>
    ...
    <property name="failedJobRetryTimeCycle">PT10M,PT17M,PT20M</property>
  </properties>
</process-engine>
```

The retry times would be three and the behaviour for this example would be the following:

- A job fails for the first time: the job will be retried in 10 minutes (PT10M is applied).
- A job fails for the second time: the job will be retried in 17 minutes (PT17M is applied).
- A job fails for the third time: the job will be retried in 20 minutes (PT20M is applied).
- A job fails for the fourth time: the job will **NOT** be retried again and the next due date is in 20 minutes (PT20M is applied again).

If the user decides to increase the retry number during retries, the last interval of the list would be applied within the difference between the new value and the size of the list. After that, it would continue with the normal flow as above. The API to increase the number of retries also supports passing a new due date for the job which will determine when the job will be picked up again. This is useful in cases where you want to control when a failed job should be tried again. For example, an incident has been resolved and the job should run as soon as possible instead of waiting for the next retry cycle. In this case, setting the job due date to the current date or a date in the past would make the job executor pick up the job immediately.

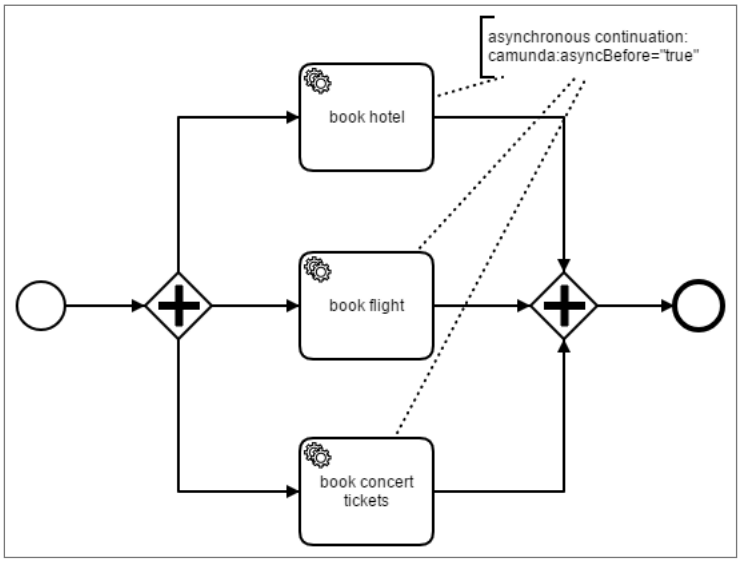
Custom Retry Configuration

You can configure an custom retry configuration by adding the `customPostBPMNParseListeners` property and specify your custom `FailedJobParseListener` to the process engine configuration:

```
<bean id="processEngineConfiguration" class="org.camunda.bpm.engine.impl.cfg.Standalone
  <!-- Your defined properties! -->
  ...
  <property name="customPostBPMNParseListeners">
    <list>
      <bean class="com.company.impl.bpmn.parser.CustomFailedJobParseListener" />
    </list>
  </property>
  ...
</bean>
```

Concurrent Job Execution

The Job Executor makes sure that **jobs from a single process instance are never executed concurrently**. Why is this? Consider the following process definition:



We have a parallel gateway followed by three service tasks which all perform an asynchronous continuation. As a result of this, three jobs are added to the database. Once such a job is present in the database it can be processed by the job executor. It acquires the jobs and delegates them to a thread pool of worker threads which actually process the jobs. This means that using an

asynchronous continuation, you can distribute the work to this thread pool (and in a clustered scenario even across multiple thread pools in the cluster).

This is usually a good thing. However it also bears an inherent problem: consistency. Consider the parallel join after the service tasks. When the execution of a service task is completed, we arrive at the parallel join and need to decide whether to wait for the other executions or whether we can move forward. That means, for each branch arriving at the parallel join, we need to take a decision whether we can continue or whether we need to wait for one or more other executions from the other branches.

This requires synchronization between the branches of execution. The engine addresses this problem with optimistic locking. Whenever we take a decision based on data that might not be current (because another transaction might modify it before we commit), we make sure to increment the revision of the same database row in both transactions. This way, whichever transaction commits first wins and the other ones fail with an optimistic locking exception. This solves the problem in the case of the process discussed above: if multiple executions arrive at the parallel join concurrently, they all assume that they have to wait, increment the revision of their parent execution (the process instance) and then try to commit. Whichever execution is first will be able to commit and the other ones will fail with an optimistic locking exception. Since the executions are triggered by a job, the job executor will retry to perform the same job after waiting for a certain amount of time and hopefully this time pass the synchronizing gateway.

However, while this is a perfectly fine solution from the point of view of persistence and consistency, this might not always be desirable behavior at a higher level, especially if the execution has non-transactional side effects, which will not be rolled back by the failing transaction. For instance, if the *book concert tickets* service does not share the same transaction as the process engine, we might book multiple tickets if we retry the job. That is why jobs of the same process instance are processed *exclusively* by default.

Exclusive Jobs

An exclusive job cannot be performed at the same time as another exclusive job from the same process instance. Consider the process shown in the section above: if the jobs corresponding to the service tasks are treated as exclusive, the job executor will try to avoid that they are executed in parallel. Instead, it will ensure that whenever it acquires an exclusive job from a certain process instance, it also acquires all other exclusive jobs from the same process instance and delegates them to the same worker thread. This enforces sequential execution of these jobs and in most cases avoids optimistic locking exceptions. However, this behavior is a heuristic, meaning that the job executor can only enforce sequential execution of the jobs that are available during **lookup time**. If a potentially conflicting job is created after that, is currently running or is already scheduled for execution, the job may be processed by another job execution thread in parallel.

Exclusive Jobs are the default configuration. All asynchronous continuations and timer events are thus exclusive by default. In addition, if you want a job to be non-exclusive, you can configure it as such using `camunda:exclusive="false"`. For example, the following service task would be asynchronous but non-exclusive.

```
<serviceTask id="service" camunda:expression="${myService.performBooking(hotel, dates)}
```

Is this a good solution? We had some people asking whether it was. Their concern was that it would prevent you from *doing things in parallel* and would thus be a performance problem. Again, two things have to be taken into consideration:

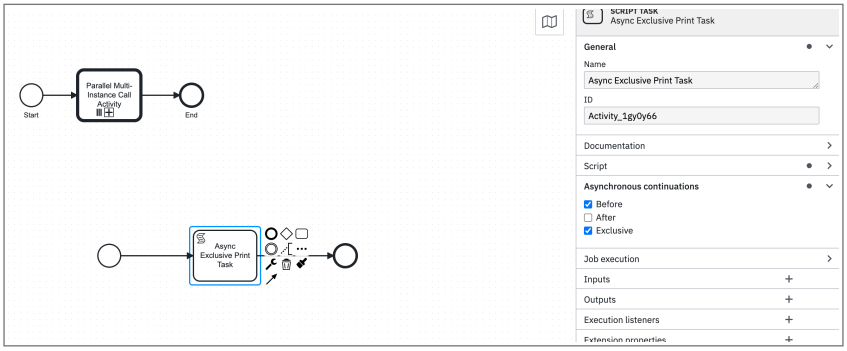
- It can be turned off if you are an expert and know what you are doing (and have understood this section). Other than that, it is more intuitive for most users if things like asynchronous continuations and timers just work. Note: one strategy to deal with OptimisticLockingExceptions at a parallel gateway is to configure the gateway to use asynchronous continuations. This way the job executor can be used to retry the gateway until the exception resolves.
- It is actually not a performance issue. Performance is an issue under heavy load. Heavy load means that all worker threads of the job executor are busy all the time. With exclusive jobs the engine will simply distribute the load differently. Exclusive jobs means that jobs from a single process instance are performed by the same thread sequentially. But consider: you have more than one single process instance. Jobs from other process instances are delegated to other threads and executed concurrently. This means that with exclusive jobs the engine will not execute jobs from the same process instance concurrently but it will still execute multiple instances concurrently. From an overall throughput perspective this is desirable in most scenarios as it usually leads to individual instances being done more quickly.

Exclusive Jobs of Process Hierarchies

As explained above, the `exclusive` asynchronous continuation will instruct the job executor to acquire and execute the jobs of a given process instance by one thread.

How does `exclusive` behave when a process contains hierarchies e.g. when multiple parallel subprocesses can be spawned by a root process?

By default, the exclusive *acquisition & execution* is only guaranteed for the jobs that originate from the root process instance. In a multi-instance call activity setting, the subprocess instances that will be spawned can run in parallel despite selecting `exclusive` asynchronous continuation as depicted in the image below.



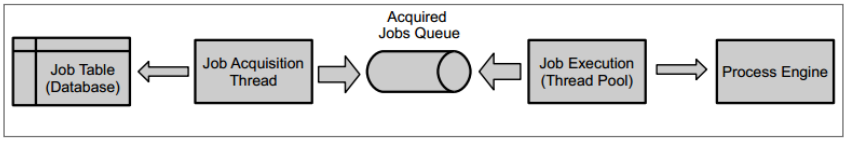
If there is a use case where the subprocess-jobs **should not be performed in parallel across each single process instance**, the following configuration can be used:

```
<process-engine name="default">
  ...
  <properties>
    <property name="jobExecutorAcquireExclusiveOverProcessHierarchies">true</property>
        ...
  </properties>
</process-engine>
```

> The property `jobExecutorAcquireExclusiveOverProcessHierarchies` is by default set to `false`. See the property under the `Configuration Properties` section.
>
> Keep in mind that enabling the feature to guarantee exclusive jobs across all subprocesses originating from a root process might have performance implications, especially for process definitions that involve complex and numerous hierarchies.
>
> Use the feature in combination with awareness of your process model.

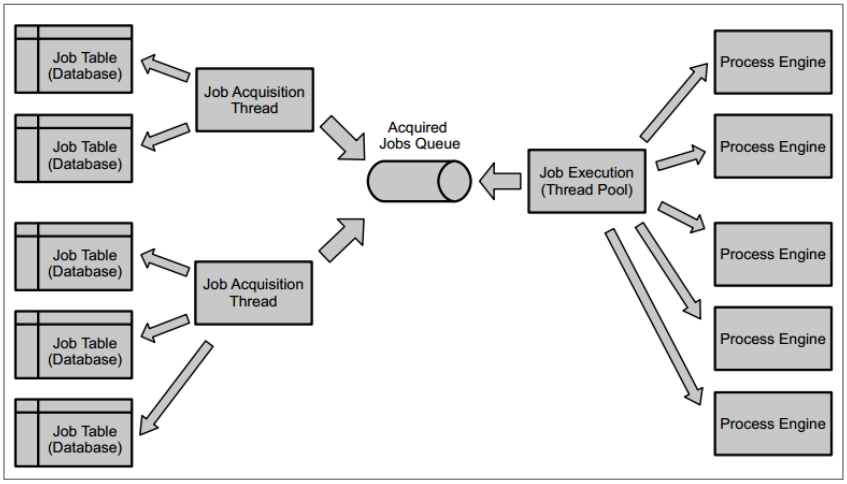
The Job Executor and Multiple Process Engines

In the case of a single, application-embedded process engine, the job executor setup is the following:



There is a single job table that the engine adds jobs to and the acquisition consumes from. Creating a second embedded engine would therefore create another acquisition thread and execution thread-pool.

In larger deployments however, this quickly leads to a poorly manageable situation. When running Camunda 7 on Tomcat or an application server, the platform allows to declare multiple process engines shared by multiple process applications. With respect to job execution, one job acquisition may serve multiple job tables (and thus process engines) and a single thread-pool for execution may be used.



This setup enables centralized monitoring of job acquisition and execution. See the platform-specific information in the Runtime Container Integration section on how the thread pooling is implemented on the different platforms.

Different job acquisitions can also be configured differently, e.g. to meet business requirements like SLAs. For example, the acquisition's timeout when no more executable jobs are present can be configured differently per acquisition.

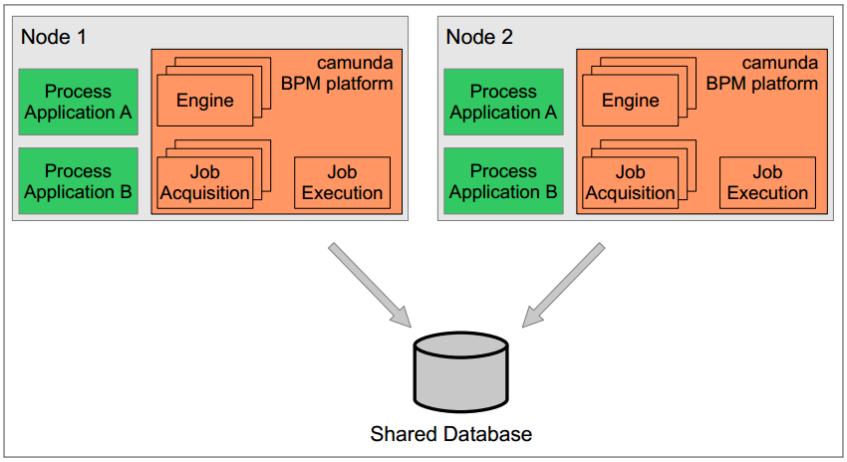
To which job acquisition a process engine is assigned can be specified in the declaration of the engine, so either in the `processes.xml` deployment descriptor of a process application or in the Camunda 7 descriptor. The following is an example configuration that declares a new engine and assigns it to the job acquisition named `default`, which is created when the platform is bootstrapped.

```
<process-engine name="newEngine">
  <job-acquisition>default</job-acquisition>
  ...
</process-engine>
```

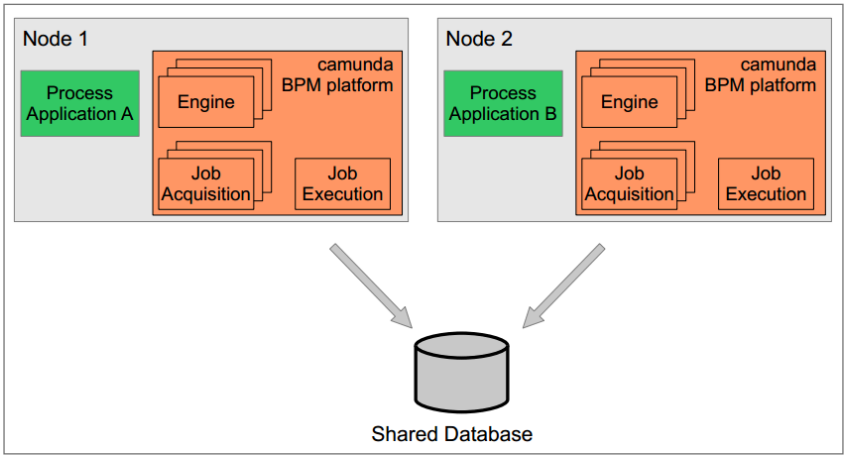
Job acquisitions have to be declared in Camunda 7's deployment descriptor, see the container-specific configuration options.

Cluster Setups

When running Camunda 7 in a cluster, there is a distinction between *homogeneous* and *heterogeneous* setups. We define a cluster as a set of network nodes that all run Camunda 7 against the same database (at least for one engine on each node). In the *homogeneous* case, the same process applications (and thus custom classes like JavaDelegates) are deployed to all of the nodes, as depicted below.



In the *heterogeneous* case, this is not given, meaning that some process applications are only deployed to a part of the nodes.



Job Execution in Heterogeneous Clusters

A heterogeneous cluster setup as described above poses additional challenges to the job executor. Both platforms declare the same engine, i.e. they run against the same database. This means that jobs will be inserted into the same table. However, in the default configuration the job acquisition thread of node 1 will lock any executable jobs of that table and submit them to the local job execution pool. This means that jobs created in the context of process application B (so on node 2) may be executed on node 1 and vice versa. As the job execution may involve classes that are part of B's deployment, you are likely going to see a `ClassNotFoundExeception` or any of the likes.

To prevent the job acquisition on node 1 from picking jobs that *belong* to node 2, the process engine can be configured as *deployment aware*, by the setting following property in the process engine configuration:

```
<process-engine name="default">
  ...
  <properties>
    <property name="jobExecutorDeploymentAware">true</property>
    ...
  </properties>
</process-engine>
```

Now, the job acquisition thread on node 1 will only pick up jobs that belong to deployments made on that node, which solves the problem. Digging a little deeper, the acquisition will only pick up those jobs that belong to deployments that were *registered* with the engines it serves. Every deployment gets automatically registered. Additionally, one can explicitly register and unregister single deployments with an engine by using the `ManagementService` methods `registerDeploymentForJobExecutor(deploymentId)` and `unregisterDeploymentForJobExecutor(deploymen` It also offers a method `getRegisteredDeployments()` to inspect the currently registered deployments.

As this is configurable on engine level, you can also work in a *mixed* setup, when some deployments are shared between all nodes and some are not. You can assign the globally shared process applications to an engine that is not deployment aware and the others to a deployment aware engine, probably both running against the same database. This way, jobs created in the context of the shared process applications will get executed on any cluster node, while the others only get executed on their respective nodes.