# Task Manager

Task Manager is controlling tasks. It is kind of smart "cluster-aware" scheduler combined with a thread pool. It will scan for tasks to execute, coordinate with other hosts in the cluster, allocate a thread for execution, execute the task and monitor it. It will also make sure that the task entry in the repository is updated accordingly.

Currently, Task Manager is based on Quartz scheduler, which provides a lot of this functionality.

## Task

A **task** is a logical unit of work or a "thread of execution" that takes non-negligible amount of time. A typical example is a modification of user object with subsequent provisioning, optionally including approvals. Generally speaking, tasks in midPoint are of two kinds: synchronously executing ones and asynchronously executing ones. The former are executing quite shortly, are usually stored in memory only, and therefore invisible to the system administrator. The latter ones generally take longer to execute, are stored primarily in the repository, and the system administrator can see and manage them through the administrative user interface (see Persistence status below).

An example of an asynchronous task is an approval that waits for interaction of (several) users and therefore cannot be executed synchronously. Another example is a data import which usually takes a lot of time. Yet another examples are synchronization and reconciliation tasks that need to be scheduled and run in background.

As for asynchronous tasks, their state must be completely serializable, i.e. able to be stored in the repository. Tasks move between IDM nodes, may be passed to a non-IDM node, may be suspended, resumed and must survive even a sudden restart of all nodes. Therefore a task must be able to store its complete state to the repository and also to recover the state back to memory. Tasks should also survive system upgrades. Therefore an approval process that started in version X must be able to finish after upgrade to version X+1. These requirements place significant constraints on the design of task mechanism and task flexibility and also impact performance. Therefore asynchronous tasks are **not** suitable for every operation. We definitely do not want every "read" operation to be wrapped in such a heavyweight task. Generally, only operations that require asynchronous execution should be implemented as asynchronous tasks.

Task is a data structure that can be stored in memory or in the repository. As said above, in-memory tasks are usually used for synchronous operations. In such a case storing task state in the repository would present unacceptable overhead. The repository tasks are stored in cases of asynchronous, background or scheduled tasks. In such cases storing the task state in the repository is necessary.

### Task State

From the general point of view, task state is a superposition of persistence status and execution status. Additionally, each task can have a business-specific state, e.g. the live synchronization token, described below.

#### Persistence Status

Persistence status tells whether the task is in-memory or persisted in the repository.

- **Transient**: The task is in-memory only, it is not stored in the repository. Only synchronous foreground tasks may use this approach. As the task data only exists while the task is being executed, the user or the client application needs to (synchronously) wait for a task to complete.
- **Persistent**: The task is stored in the repository. Both synchronous (foreground) and asynchronous (background, scheduled, etc.) tasks may use this approach, however it is used almost exclusively for asynchronous tasks.

#### Execution Status

Execution status provides information about the task overall high-level execution state. It tells whether the task is running, waits for something or is done.

- **Running:** The task is actually executing at a midPoint node. It means that there is a thread on one of the nodes that executes the task.

- **Runnable**: The task is ready to be executed. This state implies that the task is prepared to be started when a defined time arrives.
- **Waiting**: The IDM system is waiting while the task is being executed on an external node (e.g. external workflow engine) or is waiting for some kind of external signal (e.g. approval in internal workflow). The task may be running on external node or be blocked on IDM node. One way or another, there is no point in allocating a thread to run this task. Other task properties provide more information about the actual "business" state of the task.
- **Suspended**: The task is suspended, e.g. by the system administrator. The task will not be executed until it is explicitly resumed, usually by system administrator.
- **Closed**: The task is done. No other changes or progress will happen. The task in this state is considered immutable and the only things that can happen to it is a delete by a cleanup code.

The distinction between Running and Runnable states is currently not done at the level of task attributes; the `executionState` attribute for both cases is `runnable`. These states can be distinguished by calling `TaskManager.isTaskThreadActiveClusterwide` method.

### Business Status

TODO: ....

## Single-run and Recurring Tasks

Some tasks in midPoint are **single-run**. A usual example is the initial import of accounts from a resource. This import is started by the system administrator, executes on background, and after its work is done (or a unsolvable error is encountered), it finishes. Another example is execution of an operation requiring an approval, e.g. assignment of a role to a user. After the approval is got and the operation is executed, the task finishes and is closed. Single-run tasks may start either immediately after creation (this is the majority of cases) or according to a defined schedule (this is not implemented yet); in the latter case we call them scheduled single-run tasks.

However, there are tasks that have to be repeated at defined time instants. We call such tasks **recurring**. Examples are live synchronization (scheduled to run e.g. every 5 seconds) or reconciliation (scheduled to run e.g. every day at 3:00am). MidPoint currently supports two styles of schedule description: interval-based and cron-like. The former says that a task should repeat every N seconds, while the latter provides the ability of specifying starting time intervals using cron expression. For more information on these expressions, please see the Quartz documentation.

### Tightly and Loosely Bound Tasks

A recurring task can be bound to an IDM node either tightly or loosely.

A **tightly-bound task** is given a thread, in which it executes. Even between executions, the thread is allocated to the task. (Technically speaking, it just sleeps using `Thread.sleep` method.) A direct consequence is that each execution of this task occurs on the same node. Main positive aspects are that the execution is a bit more efficient (scheduling via Quartz is avoided) and that the troubleshooting is a simpler, as all executions are recorded in the same midPoint log file. A negative aspect is that such a task consumes permanently one thread. A general rule is that a task should be tightly bound only when its scheduling interval is quite small, e.g. under 30 seconds. (In current Quartz-based implementation of Task Manager, it is even not possible to use a cron expression for a tightly-bound task.)

On the other hand, a **loosely-bound task** has no thread permanently allocated to it. It waits in the repository until its start time arrives. After that, it is started at (any) available IDM node. When its execution finishes, the thread is released and the task waits for the next start time. Individual task executions can be on the same IDM node or on different nodes (therefore the name 'loosely-bound'), as determined by the Quartz scheduler algorithm. (Quartz doc states that "The load balancing mechanism is near-random for busy schedulers (lots of triggers) but favors the same node for non-busy (e.g. few triggers) schedulers.")

### Task Execution - a Bit of Terminology

These are only preliminary terms, open to discussion.

**Task run** (or sometimes "task cycle run") denotes one execution of task's logic, provided by task handler or handlers, see below. **Task thread run** denotes one execution of task's thread.

For *single-run tasks*, task run is the same as task thread run - and there is only one such run (or thread run) during the task lifetime.

For *loosely-bound recurring tasks*, task run is the same as task thread run as well. However, in this case, there are potentially many runs (or thread runs) during the task lifetime.

For *tightly-bound recurring tasks*, there is only one task thread run, because the task thread is allocated to the task permanently. Within this task thread run there are many task runs, occurring at defined points in time.

(For this discussion, we are not thinking about task failovers and node restarts.)

Start and end of task thread run are logged to the console as debug messages. Start and end of task run are recorded as `lastRunStartTimestamp` and `lastRunFinishTimestamp` attributes.

### Task Scheduling

Task scheduling is governed by the `schedule` attribute, having the following parts:

1. `interval`: Denotes interval in seconds between task runs. Used only for recurring tasks.
2. `cronLikePattern`: Cron-like pattern specifying time(s) when the task is to be run. Currently only loosely-bound recurring tasks can use this feature. (In the future, scheduled single-run tasks could use this feature to specify their first - and only - run start time.)
3. `earliestStartTime`: Earliest time when the task is allowed to start. Usable for any kind of task.
4. `latestStartTime`: Latest time when the task is allowed to start. Usable for any kind of task.
5. `latestFinishTime`: Latest time when the task is allowed to run (a reason that one could have to specify this time is perhaps because another task that conflicts with this one is scheduled to start at this time, so the first one must NOT run at that moment). It is the responsibility of the task handler to finish working when this time arrives - it will not be enforced by the task manager.

Besides these parameters, there is also the last one, called `misfireAction`, which controls what is to be done when the planned start time arrives without the task actually starting (e.g. because no node or thread is available to execute the task at that time). There are the following possibilities:

1. `executeImmediately`: The task will be executed immediately.
2. `reschedule`: The task will be rescheduled according to its schedule. (This can be used only for loosely-bound recurring tasks.)
3. `forget`: The task will not be executed at all. (This can be used only for scheduled single-run tasks. And is not implemented yet.)

## Resilient and Non-resilient Tasks (ThreadStopAction Attribute)

By default, all persistent tasks are resilient. It means that after a node is stopped (either regularly, e.g. by shutting down the application server, or not regularly, e.g. caused by hardware malfunction), the task continues to execute on another node in a cluster, or (if no other nodes are available), after a node becoming ready.

However, there are situations in which such a resilience is not suitable. In these cases, a task can be declared non-resilient; which means that after a node shutdown or failure the task will **not** be started on another node, and will be simply suspended or closed. The use case for such a feature would be perhaps the "manual" synchronization of resources - something that will be started by the system administrator, with the expectation that it will execute only until the node is down.

More precisely, this task behavior is controlled by `threadStopAction` attribute, which can have the following values:

1. `restart`: The task will be restarted on first node available (i.e. either immediately, if there is a suitable node in the cluster, or later, when a suitable node becomes available).
2. `reschedule`: The task will be rescheduled according to its schedule (for single-run and tightly-bound recurring tasks this is the same as 'restart').
3. `suspend`: The task will be suspended.
4. `close`: The task will be closed.

First two options are used to implement resilient task behavior, while the last two ones for non-resilient tasks.

Please note that for tasks, which have no threads allocated at the moment of node down (e.g. loosely-bound recurring tasks, but also scheduled single-run tasks), this setting has no effect. These tasks will be simply started when their start time arrives. For these cases it is advisable to use suspend/close option only when there is a strong reason to do it, e.g. when an administrator wants to manually review task state after such an interruption.

## Handler URI and Task Category

Handler URI indirectly specifies which class (called handler, implementing TaskHandler interface) is responsible to handle the task. The handler will execute reaction to a task lifecycle events such as executing the task, task heartbeat, etc.

Handler URI can be also understood as a specification of task *subtype*.

The task handlers will register themselves with appropriate URI at midPoint initialization. The URI is used instead of a direct class name to provide additional robustness during system upgrades.

A single-run task can have a list of handler URIs. After first handler finishes its execution, it is removed from the list of handlers and second handler starts. The process continues until the list of handlers is empty. At this moment the task is automatically closed.

Task category denotes a user-recognizable type of task. Some examples: LiveSynchronization, Reconciliation, ImportingAccounts, ImportFromFile, UserRecomputation, Workflow, Demo.

## Associated Object

Tasks may be associated with a particular objects. For example a "import from resource" task is associated with the resource definition object that it imports from. Similarly for synchronization and reconciliation tasks. This is an optional property.

The object could be also specified using usual extension mechanism. But it would be difficult to search for all the tasks that work on a particular resource or other object.

## Task Owner

Task owner is (usually) an IDM user that initiated the task. This attribute is used e.g. for auditing reasons.

## Clustering and High Availability

As mentioned above, there can be more nodes working in a **cluster**. These nodes share the workload: when a task becomes ready to be executed, one of nodes takes and executes it. (This process is governed by Quartz.)

When a node becomes unavailable (either because of shutdown, or due to sudden crash), the task manager:

1. takes tasks running on that node and restarts them on remaining available nodes - subject to the threadStopAction described above,
2. executes other (scheduled) tasks on remaining available nodes.

In this way, high availability of the solution with regards to task execution is ensured.

## Task State in midPoint Repository and Quartz Job Store

Generally speaking, midPoint repository contains general task information, including its execution and business state, while Quartz job store is responsible for maintaining information necessary for task scheduling (e.g. next planned start time). With minor limitations, the information in Quartz job store can be erased at any time, and recreated from midPoint repository (at node startup) - the only damage that could occur is that some tasks could be executed one more or one less time.

Because of this, the most simple installations (e.g. demonstration ones) can be run with **in-memory Quartz job store** - a store that will be re-created at node startup. The limitations of this approach are:

1. clustering (failover) feature is not available,
2. tasks do not "know" when they run last time, so e.g. interval-based loosely-coupled tasks will be started immediately, even if their expected start time has not come yet; or misfired cron-based tasks would not start (even if configured to do so), because the information on the misfire event would not be present. This may cause for example reconciliation task(s) to be started immediately after midPoint is started.

More advanced installations could use **JDBC-based Quartz job store** - a store that will remember task scheduling information.

## Task Manager Configuration and Administration

This topic is discussed in detail in Administration Interface#ServerTasks.

## Authorizing specific operations

### Task-related operations

In order to authorize task-related operations, the following action URIs are defined. These are evaluated with respect to task objects, i.e. you can define a filter that selects tasks that can be acted upon.

| Operation | Action URI |
|---|---|
| suspend a task | http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#**suspendTask** |
| suspend and delete a task | http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#**delete** |
| resume a task | http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#**resumeTask** |
| schedule a task to run instantly | http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#**runTaskImmediately** |

Note that "suspend and delete a task" operation uses standard "delete" action URI. I.e. for simply deleting a task and deleting a task after suspending it you would use the same authorizations.

### Node-related operations

For node-related operations, the following action URIs are defined. These are evaluated with respect to node objects, i.e. you can define a filter that selects nodes that can be acted upon (although we do not expect such a selection would be frequently used in practice).

| Operation | Action URI |
|---|---|
| start task scheduler | http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#**startTaskScheduler** |
| stop task scheduler (optionally with stopping tasks that are executing on it) | http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#**stopTaskScheduler** |

### Other operations

Finally, the following actions URIs are defined for operations that are not bound to specific task nor node:

| Operation | Action URI |
|---|---|
| stop all service threads | http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#**stopServiceThreads** |

| | |
|---|---|
| start all service threads | http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#**startServiceThreads** |
| synchronize tasks between midPoint repository and Quartz scheduler | http://midpoint.evolveum.com/xml/ns/public/security/authorization-model-3#**synchronizeTasks** |