Tasks scheduler

scheduler, final, lrt, long, running, task, description, triggers, cron, scheduled

The task can be scheduled in three different ways - types of `triggers`:

- Settled time - will enter a settled launching time If the time was entered in the past, then the task is launched right away.
- Cron - Quartz expression
- Dependent task - trigger task, when other task successfully ended. Dependent tasks are executed by LongRunningTaskExecuteDependentProcessor.

To run, schedule and review tasks, interfaces have been created:

- `LongRunningTaskExecutor` - any executable task. This task can be run from the code.
- `SchedulableTaskExecutor` - scheduled task. The task can be scheduled through UI schedulers.
- `AbstractSchedulableStatefulExecutor` - stateful scheduled task. Result is logged for every processed item (see next section).

The descendants of these interfaces can be transmitted to the service `LongRunningTaskManager` to be run. This is how the records related to them needed for their future management in the database `IdmLongRunningTask` will be created. If the task fails or it is interrupted, again a record in the database is made. On the records of the running / finished tasks, an agenda on front-end has been created where it is possible to see the course of the task and cancel the tasks as needed.

To schedule the task, the Quartz Scheduler has been integrated through the service `SchedulerManager`. The service will automatically find all the tasks implementing `SchedulableTaskExecutor` and offers them for scheduling in UI. The task parameters can be specified in the `getParameterNames()` method, these parameters being passed before running through the `init(Map<String, Object> properties)` method - there are string values falling through the UI so it depends on the task how it will accept the parameters (the prepared `ParameterConverter` can be used).

Every task has a defined instance (server) where it should be run. From one instance, it is possible to schedule tasks on more instances (one front-end for more back-ends over one database). The task running on the selected instance is ensured by the service `LongRunningTaskManager` which controls actively the tasks to be run prepared by the scheduler (through the database). The instance identifier is defined in application configuration. To run the instance, the tasks which are marked as running in the previous launching of the instance in question are checked, and they are marked as finished (they couldn´t finish).

When task type is removed (e.g. between module versions), then all scheduled task from this obsolete type are removed automatically - task type doesn't exist anymore, so cannot be scheduled.

Configured scheduled task is persisted with selected `SchedulableTaskExecutor` implementation class ⇒ when executor implementation is renamed or moved into different package, then scheduled task with modified executor has to be reconfigured too (tasks are stored in quartz storage).

When tasks has to run exclusively, annotation `@DisallowConcurrentExecution` can be used. Task with one type (class) and with this annotation will be not be executed, if the same task with type (class) is running. Task is not canceled - he will be executed after previous task will end (respectively, when prepared task will be executed from queue again - see above). When task type is not sufficient for choosing, when to not run concurrently, then `AbstractSchedulableStatefulExecutor#validate`method can be overriden.

Cron in profile DEV

Tasks started by CRON in maven profile DEV was started only once for 3600000ms. This configuration is set by property: `scheduler.task.queue.process` in application-dev.properties. **Server restart is needed!** This property can not be overloaded by FE configuration agenda.

Configuration QUARTZ (datasource, etc.)

Configuration of QUARTZ (scheduler) it is using application properties by individual profiles. In every profile application properties is necessary set path for quartz's settings (his own application properties) - `scheduler.properties.location`. Change property `scheduler.task.queue.process` **needed server restart!** This property can not be overloaded by FE configuration agenda.

Example

Example of an implementing task - you will find everything that has been mentioned here.

Stateful task executors

Stateful task executors are an extension of standard long running task executor, which hold a state and processed items log. Such functionality is suitable for tasks, which has to keep historical records of processed items.

The key concepts of stateful executors are queue and log. The queue is a set of records that have been processed, but the information about them is required for next run of the task. Take contract enable task as an example. Once a contract is valid, the task should process it. But it must be processed only once, not every time the task is run and the contract is valid. Therefore until the contract is valid, it is kept in the queue and will not be processed again. The log is a mechanism to keep track of what kind of items were processed by the task and what was the processing result.

Stateful tasks are tightly coupled with Quartz' scheduled tasks, that is the tasks that are run multiple times. Running a stateful task from code as a simple LongRunningExecutor is pointless, because there is no reason to keep neither history nor processed items queue. The relation to Quartz is realized through `IdmScheduledTask` entity, which holds Quartz job's unique job key. The task queue then references the `IdmScheduledTask`. Original `IdmLongRunningTask` entity is the owner of the log items. Log items are related to each task run, which is done by the mentioned entity.

All stateful tasks shall implement the `SchedulableStatefulExecutor`. Its abstract implementation `AbstractSchedulableStatefulExecutor`, which can be viewed as a template method for task execution, hold most of the common logic. The intent is to always extend the `AbstractSchedulableStatefulExecutor` and define the `getItemsToProcess` and `processItem` method by concrete task. Therefore the developer should mostly focus only on retrieving items and the business logic, which does the processing.

The item processing result is represented by `OperationResult`. Successful operations are inserted both into the queue and log, all results go to the log.

After the item processing finishes correctly (task is executed without exception or not canceled manually), the queue is refreshed. The algorithm is simple, as it only removes all items from queue, which were not processed correctly in current run. This mechanism ensures the removed items from queue will be processed in next task run, if will be given in `getItemsToProcess`.

`AbstractSchedulableStatefulExecutor` methods (to implement / overridable):

- **`getItemsToProcess`** - required. Returns item to be processed by this LRT.
- **`processItem`** - required. Business logic - processing single item.
- `supportsDryRun` - `true` by default. If task could be executed in Dry run mode = ~test run. Task executed in this mode only iterates through items to process and log them ⇒ **`processItem`** is not executed.
- `continueOnException` - `false` by default. LRT will continue, after some item fails.
- `requireNewTransaction` `false` by default. Each item will be processed in new transaction - ensures successfully processed item will be commited after other item or LRT task fails. Use this property everywhere, when LRT can be executed synchronously (e.g. for synchronization dependent task, which are executed synchronously and wrapped which one parent transaction).
- `supportsQueue` - `true` by default. If is `false`, then each item (candidate) will be processed without checking already processed items (queue is ignored). Usable, when processed items should be logged, but queue has to be ignored.
- `isRecoverable` - `false` by default. See recoverable task introduction below.

Look out: when scheduled task is removed, then all item will be processed again! Processed items are linked to scheduled task instance. Prevent to remove tasks, which sends notifications etc. (e.g. password expire warning) - notifications will be sent again.

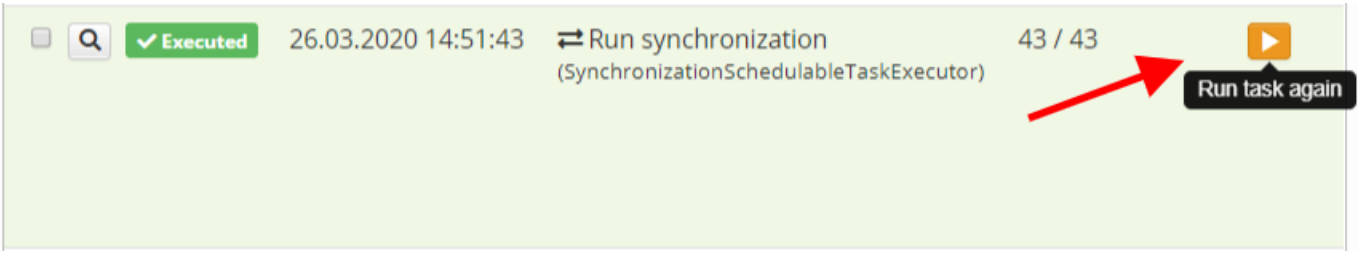
Recoverable tasks

@since 10.2.0

Task can be executed repetitively without reschedule is needed. When task is canceled (e.g. by server is restarted), then task can be executed again (~recovered) directly from long running task agenda. New task will be created and executed with the same configuration as original task. **When task** is stateful and **supports queue**, then **already processed items will not be processed again**.

To enable this feature, task method `isRecoverable` has to be overriden and return `true`.

Only tasks created and **running** in the version **10.2.0** and higher can be run again!



Implemented task types

Implemented long running task are listed below. Non schedulable long running tasks are included.

PasswordExpirationWarningTaskExecutor

Sends warning notification before password expires. Notification is **not send to disabled identities**. Days before has to be given as task's parameter (number greater than zero). More task could be configured e.g. for sending warning notification 14,7,3 days before password expires. Default notification topic is configured to email sender.

It's good to schedule `PasswordExpiredTaskExecutor` and `PasswordExpirationWarningTaskExecutor` on the start of day - e.g. **0 5 0 ? * *** ⇒ passwords which expired yesterday will be processed and proper notifications will be send.

PasswordExpiredTaskExecutor

Sends warning notification after password expires. Notification is **not send to disabled identities**. Default notification topic is configured to email sender.

Task is scheduled by default on the start of day: **0 5 0 ? * *** - passwords which expired yesterday will be processed and proper notification will be send.

AccountProtectionExpirationTaskExecutor

Removes accounts with expired protection interval. Account has to have `inProtection` flag set to `true`.

RetryProvisioningTaskExecutor

Retry failed provisioning operation periodically. When provisioning operation fails, then is logged into provisioning queue with result (state, exception, result code etc.) and new attempt time. Number of attempts is **6** and time between attempts grows (2 mins, 5 mins, 20 mins, 2 hours, 12 hours - configuration coming soon).

ExecuteScriptTaskExecutor

Long running task for execute script by code. Script can increment and initialize counter.

ProvisioningQueueTaskExecutor

Process provisioning operations in queue periodically. This LRT executes prepared (``created``) requests for provisioning, when target system is switched do use asynchronous provisioning.

Parameters

- ``virtualSystem``, values:
  - ``true`` - processes ``created`` operations from queue for virtual system only,
  - ``false`` - processes ``created`` operations from queue for non virtual system only,
  - ``null`` - if ``virtualSystem`` parameters is not configured, then processes ``created`` operations from queue for all systems.

SynchronizationSchedulableTaskExecutor

Schedule synchronization.

Parameters

- ``Synchronization uuid`` - synchronization configuration to schedule.

~~AddNewAutomaticRoleTaskExecutor~~

@deprecated since 10.4.0 - `ProcessAutomaticRoleByTreeTaskExecutor` is used from version 10.4.0.

Long running task for add newly added automatic role to users. Can be executed repetitively to assign role to unprocessed identities, after process was stopped or interrupted (e.g. by server restart).

Parameters

- ``roleTreeNode`` - automatic role identifier.

RemoveAutomaticRoleTaskExecutor

Long running task for remove automatic roles from identities.

Parameters

- ``roleTreeNode`` - automatic role identifier.

ProcessAutomaticRoleByTreeTaskExecutor

@since 10.4.0

Recalculate automatic role by tree structure for contracts and positions. Recalculate roles for identities with contract or other position match selected automatic role by tree structure. Can be executed repetitively to assign role to unprocessed identities, after process was stopped or interrupted (e.g. by server restart).

Parameters

- ``roletreenode`` - automatic roles by tree structure to recount.

ProcessAllAutomaticRoleByTreeTaskExecutor

@since 10.4.0

Recalculate all automatic roles by tree structure for contracts and positions. Recalculate all roles for identities with contract or other position assigned by automatic role by tree structure. Can be executed repetitively to assign role to unprocessed identities, after process was stopped or interrupted (e.g. by server restart).

ProcessSkippedAutomaticRoleByTreeTaskExecutor

@since 10.4.0

Recalculate skipped automatic role by tree structure. Recalculate automatic roles by tree structure, which were not processed (skipped) after tree structure was changed (e.g. from synchronization).

ProcessSkippedAutomaticRoleByTreeForContractTaskExecutor

@since 10.4.0

Recalculate skipped automatic role by tree structure for contracts and other positions. Recalculate automatic roles by tree structure for contracts and other positions, which were not processed (skipped) after work position was changed (e.g. from synchronization).

AddNewRoleCompositionTaskExecutor

@since 9.0.0

Long running task for add newly added role composition to users. Sub roles defined by this composition will be assigned to identities having superior role. Can be executed repetitively to assign unprocessed roles to identities, after process was stopped or interrupted (e.g. by server restart).

Parameters

- ``role-composition-id`` - role composition identifier.

RemoveRoleCompositionTaskExecutor

@since 9.0.0

Long running task for remove assigned roles by given composition from identities.

Parameters

- ``role-composition-id`` - role composition identifier.

IdentityContractExpirationTaskExecutor

Remove roles by expired identity contracts (⇒ removes assigned roles).

IdentityRoleExpirationTaskExecutor

Long running task for expired identity roles removal. Expected usage is in cooperation with ``CronTaskTrigger``, running once a day after midnight.

IdentityRoleValidRequestTaskExecutor

When role was created with validity in future, then this executor publish even, when role starts to be

valid. This is needed specially for account management, when new account for roles that was newly valid has to be created.

HrEnableContractProcess

@since 7.5.1

HR process - enable identity's contract process. The processes is started for contracts that are both valid (meaning `validFrom` and `validTill`) and enabled. This LRT should be configured even event processors for HR processes are enabled - this process depends on current date ⇒ event processors cannot work this way. Event processors checks active changes on given contract.

HrContractExclusionProcess

@since 7.5.1

HR process - identity's contract exclusion. The processes is started for contracts that are both valid (meaning validFrom and validTill) and excluded. This LRT is optional with event processors for HR processes. One of LRT or processors should be enabled / scheduled. If LRT is scheduled and processors are enabled, then this LRT just checks, if everything is done (contract is already excluded by processor).

HrEndContractProcess

@since 7.5.1

HR process - end of identity's contract process. The processes is started for contracts that are not valid (meaning validFrom and validTill). This LRT is optional with event processors for HR processes and IdentityContractExpirationTaskExecutor. One of this LRT or processors with IdentityContractExpirationTaskExecutor should be enabled / scheduled. If this LRT is scheduled and processors with IdentityContractExpirationTaskExecutor are enabled / scheduled, then this LRT just checks, if everything is done (contract is already expired by processor or IdentityContractExpirationTaskExecutor).

SelectCurrentContractSliceTaskExecutor

@since 8.1.0

Recalculate current using slices as contract. Find all slices which should be for actual date using as contract and copy their values to parent contracts.

ChangeConfidentialStorageKey

@since 8.2.0

Change key for crypt confidential storage. The task can be started after you change key in application properties/file to newer. As parameter will be given old key. See the admin guide.

GenerateConfidentialStorageInitializationVectorsTaskExecutor

@since 10.8.0

The task processes every value in the confidential storage, generates a new initialization vector and encrypts the value using this new initialization vector. See the admin guide.

RemoveOldLogsTaskExecutor

Long running task for remove old logging event (persisted in database by logback appender configuration), logging event exception and logging event property. Parameter remove old logs than given number of days.

Parameters

- ``Number of days`` - removes logs older than given days count.

DeleteProvisioningArchiveTaskExecutor

@since 9.7.0

Delete provisioning operations in archive older than given number of days.

Parameters

- ``Number of days`` - Delete operations older than given number of days.
- ``Operation state`` - Delete operations in given state only. Available options [CREATED, RUNNING, EXECUTED, NOT_EXECUTED, BLOCKED, EXCEPTION, CANCELED].
- ``System`` - Delete operations with given system only.
- ``Empty provisioning`` - Delete provisioning operations only without attributes. Operation with DELETE type is not considered as empty (even haven't attributes).

```
-- PostgreSql
-- Delete archives
DELETE FROM sys_provisioning_archive WHERE created < now() - INTERVAL '90
day';
-- Delete attributes
DELETE FROM sys_provisioning_attribute attr WHERE NOT EXISTS (SELECT FROM
sys_provisioning_archive arch WHERE attr.provisioning_id = arch.id) AND NOT
EXISTS (SELECT FROM sys_provisioning_operation oper WHERE
attr.provisioning_id = oper.id);
```

DeleteExecutedEventTaskExecutor

@since 9.7.0

Delete successfully executed entity events (in the state 'EXECUTED') older than given number of days.

Parameters

- ``Number of days`` - Delete events older than given number of days.

```
-- PostgreSql
-- Delete events older then 3 days
DELETE FROM idm_entity_event WHERE result_state='EXECUTED' AND created <=
NOW() - INTERVAL '72 HOURS';
vacuum FULL idm_entity_event;
```

DeleteLongRunningTaskExecutor

@since 9.7.12

Delete long running tasks.

Parameters

- ``Number of days`` - Delete long running tasks older than given number of days.
- ``Operation state`` - Delete long running tasks in given state only. Available options [CREATED, RUNNING, EXECUTED, NOT_EXECUTED, BLOCKED, EXCEPTION, CANCELED].

```
-- PostgreSql
-- Delete lrt by sql
DELETE FROM idm_long_running_task WHERE created < now() - INTERVAL '90 day'
AND result_state = 'EXECUTED';
-- Delete related processed items
DELETE FROM idm_processed_task_item WHERE long_running_task IS NOT NULL AND
long_running_task NOT IN (SELECT id FROM idm_long_running_task);
-- Clear lrt from related reports
UPDATE rpt_report SET long_running_task_id = NULL WHERE long_running_task_id
IS NOT NULL AND long_running_task_id NOT IN (SELECT id FROM
idm_long_running_task);
```

When long running task is deleted, then processed items are removed too. Scheduled task has another processed items - "queue" items ⇒ this items are preserved from delete.

DeleteNotificationTaskExecutor

@since 9.7.12

Delete notifications.

Parameters

- ``Number of days`` - Delete notifications older than given number of days.
- ``Sent only`` - Delete sent notifications only. All notifications older than given number of days will be deleted otherwise.

```
-- PostgreSql
-- Delete common notification
DELETE FROM idm_notification WHERE created < now() - INTERVAL '180 day';
-- Delete all related notification by type
DELETE FROM idm_notification_console WHERE id NOT IN (SELECT id FROM
idm_notification);
DELETE FROM idm_notification_email WHERE id NOT IN (SELECT id FROM
idm_notification);
DELETE FROM idm_notification_log WHERE id NOT IN (SELECT id FROM
idm_notification);
DELETE FROM idm_notification_sms WHERE id NOT IN (SELECT id FROM
idm_notification);
DELETE FROM idm_notification_websocket WHERE id NOT IN (SELECT id FROM
idm_notification);
-- Delete recipients
DELETE FROM idm_notification_recipient WHERE notification_id NOT IN (SELECT
id FROM idm_notification);
```

DeleteSynchronizationLogTaskExecutor

@since 9.7.12

Delete synchronization logs.

Parameters

- ``Number of days`` - Delete synchronization logsolder than given number of days.
- ``System`` - Delete logs on the given system only.

```
-- PostgreSql
-- Delete logs
DELETE FROM sys_sync_log WHERE created < now() - INTERVAL '180 day';
-- Delete actions
DELETE FROM sys_sync_action_log WHERE sync_log_id NOT IN (SELECT id FROM
sys_sync_log);
```

```
-- Delete items
DELETE FROM sys_sync_item_log WHERE sync_action_log_id NOT IN (SELECT id
FROM sys_sync_action_log);
```

DeleteWorkflowHistoricInstanceTaskExecutor

@since 9.7.12

Delete historic workflow process instances.

Parameters

- ``Number of days`` - Delete historic workflow processes older than given number of days.
- ``Workflow definition`` - Delete historic workflow processes with this definition only.

VsSystemGeneratorTaskExecutor

@since 10.4.0

Task generates given number of virtual systems, roles and identities. All generated entities are evenly distributed among themselves. I.e. Roles assigned to users and connected to generated systems. Task serves for generating required scenario and following performance test.

Parameters

- ``Item prefix`` - A name prefix of all generated entities. Serves for easier searching of entities in IdM.
- ``System count`` - Number of generated virtual systems.
- ``Role count`` - Number of generated roles.
- ``User count`` - Number of generated identities.

Testing tips

Integration tests are the preferred way of testing stateful executors in combination with stubbing the returned set of items to process by Mockito, using a 'spy' on the executor. This way the developer has full control over the task and can test both processing and items retrieval independently. List through the existing test cases for details.

Initializer

Module core has default implementation of **AbstractScheduledTaskInitializer** - **InitCoreScheduledTask**. This class is responsible for initializing default scheduled task. See

IdmCoreScheduledTasks.xml for xml structure.

For another module is necessary implement subclass of AbstractScheduledTaskInitializer.

Scheduled task is initialized by **type**. If exists task with same type as in xml, task will not be initialized.

Future development

- Support for the check of the competition of the running tasks. Now it is on the task implementation to check if it should be run or not. This could be extracted to the general.
- Display of the scheduled triggers in the detail of the scheduled job.
- Recover canceled task by server is restarted automatically.