Batch 2

Overview

KFS's batch framework is implemented using the Quartz scheduler. For most batch-related setup and configuration, the developer/administrator does not need to know about Quartz internals. Most of the configuration is done via Spring dependency injection.

There is no requirement to use the KFS batch framework. The "Step"s described below are simple Java classes that may be scheduled/executed in a number of ways.

Terminology

- **Step**: conceptually, a stage in a process (i.e. job). Technically, a step is a class that implements the `org.kuali.kfs.batch.Step` interface.
- **Job**: a series of steps that runs sequentially (and not in parallel). However, depending on how jobs are set up, jobs may run in parallel.
- **Dependency**: a job may rely on another job's execution before running. For example, there may be a job "A" that creates database tables, and another job "B" that writes data into those tables. Job "B" is said to be have a dependency (or be dependent upon) job "A". Therefore, job B (the dependent) should be run after job A (the dependee). A job may be dependent upon multiple jobs; in which case, all dependee jobs must run before the dependent job may run.
- **Hard dependency**: a dependency that specifies that the dependent job may run only after the dependee job has run successfully.
- **Soft dependency**: a dependency that specifies that the dependent job may run only after the dependee job has run (successfully or not).

Implementing a step

A step is a stage in a job. Although a step can be a part of many jobs, we speak of a step being part of a single job for clarity.

Java Step implementation

A step class implements the `org.kuali.kfs.batch.Step` interface. Steps are normally constructed as Spring beans, which means that they will require the appropriate setter methods for any properties that need to be injected. The `org.kuali.kfs.batch.AbstractStep` class provides a default implementation for many of the methods in the `Step` interface. For most steps, the developer only needs to implement the `execute(String)` method and any setter methods required for Spring property injection. Steps should not have a substantial amount of logic in them, as core business logic should be delegated to services.

The execution of the `execute(String)` may have 3 possible outcomes:

1. returns `true`, meaning that the step has succeeded, that the job should continue running, and that the job is succeeding so far.
2. returns `false`, meaning that the step has succeeded, but the rest of the steps in the job should **not** be run. Since no further steps in the job will be run, the job will succeed.
3. throws an exception, meaning that the step has failed, that the rest of the steps in a job should **not** be run, and that the job has failed.

Step interruption
The batch schedule screen provides functionality to interrupt (i.e. cancel) a job during execution. When a job is canceled, it will notify the currently running step that the job has been interrupted/canceled. A step can be implemented to check whether it has been interrupted by calling `Step.isInterrupted()` and implement logic to gracefully terminate execution early. If the running step ignores the interruption status, then the job will be canceled after the running step completes execution.

Step transactionality
In accordance with KFS design principles, step classes should not be annotated with `@Transactional`. Transactions should start at the service layer. If transactionality is desired, then the step should call a `@Transactional` service.

Sample step

```
/**
 * This step executes the enterprise feeder
 */
public class EnterpriseFeedStep extends AbstractStep {

    private EnterpriseFeederService enterpriseFeederService;

    /**
     * Runs the enterprise feeder process
     *
     * @jobName the name of the job this step is being execute as part of
     * @return true if the job completed successfully, false if otherwise
     * @see org.kuali.kfs.batch.Step#execute(String)
     */
    public boolean execute(String jobName) throws InterruptedException {
        enterpriseFeederService.feed(jobName, true);
        return true;
    }

    /**
     * Gets the enterpriseFeederService attribute.
     *
     * @return Returns the enterpriseFeederService.
     * @see org.kuali.module.gl.service.EnterpriseFeederService
     */
    public EnterpriseFeederService getEnterpriseFeederService() {
        return enterpriseFeederService;
    }

    /**
     * Sets the enterpriseFeederService attribute value.
     *
     * @param enterpriseFeederService The enterpriseFeederService to set.
     * @see org.kuali.module.gl.service.EnterpriseFeederService
     */
    public void setEnterpriseFeederService(EnterpriseFeederService
enterpriseFeederService) {
        this.enterpriseFeederService = enterpriseFeederService;
    }
}
```

Spring Step configuration

Configuring a step in Spring is relatively easy. Just define a Spring bean definition corresponding to step class implementation and inject any of the dependencies required by the step. If the step implementation extends `AbstractStep` (and it should), the Spring bean definition should include the `parent="step"` attribute in the XML file like in the example below.

```
<bean id="enterpriseFeedStep" class="org.kuali.module.gl.batch.EnterpriseFeedStep"
parent="step">
    <property name="enterpriseFeederService">
        <ref bean="glOriginEntryEnterpriseFeederService" />
    </property>
</bean>
```

Jobs

Execution of jobs

Steps within a job are run sequentially. If parallel execution of steps is desired, then multiple jobs should be defined.

The execution of a job may have one of three outcomes:

- Succeeded
- Failed
- Cancelled, which, for all practical purposes, is equivalent to failed

Java implementation of jobs

Under most circumstances, the base Job implementation should suffice, and developers do not need to write any job-related code.

Spring job configuration

There are two steps in configuring a job:

1. Creating the job's Spring bean
2. Registering the job within the appropriate module

Creating the job's Spring bean

The following is an example of a Spring job bean. Explanation of important elements of the job bean will be explained below.

```
<bean id="scrubberJob" parent="scheduledJobDescriptor">
    <property name="steps">
        <list>
            <ref bean="createBackupGroupStep" />
            <ref bean="scrubberStep" />
        </list>
    </property>
    <property name="dependencies">
        <map>
            <entry key="enterpriseFeedJob" value="hardDependency"/>
            <entry key="collectorJob" value="softDependency" />
            <entry key="nightlyOutJob" value="hardDependency" />
        </map>
    </property>
</bean>
```

Defining a job bean consists of three primary steps:

Step 1: Defining whether a job is scheduled or unscheduled

A scheduled job is a job that will be executed by the SchedulerService once all of its dependencies have been satisfied, and the `parent="scheduledJobDescriptor"` attribute is used on the `<bean>` tag to define the job as such. (See below to see how scheduled jobs are triggered.) An unscheduled job must be manually invoked using the batch schedule screen, and the `parent="unscheduledJobDescriptor"` attribute is used on the `<bean>` tag to define the job as such.

In the example job Spring bean definition above, the scrubberJob is defined as a scheduledJob.

Step 2: Defining the steps of a job

Steps in a job are defined in a list for the "steps" property of the job bean. The order in which the steps are defined is the order in which the steps will be executed. In the example above, the `createBackupGroupStep` and `scrubberStep` steps comprise the scrubber job, and they will be run in that order.

Note that when steps are specified in a job, the `<ref bean="..."/>` notation is used.

Step 3: Defining the dependencies of a job

A job may be need to wait for other jobs to complete before its execution begins. This relationship is called a dependency.

There are two types of dependencies:

- Hard dependency: a hard dependency is satisfied only when the dependee job has run successfully.
- Soft dependency: a soft dependency is satisfied when the dependee job has succeeded, failed, or been canceled.

The dependencies of a job are defined in the "dependencies" property of the job bean. Because the property is a map, the dependee jobs may be defined in any order. The key of each mapping is the dependee job name. The value of the mapping is either the string `hardDependency` or `softDependency`, for a hard or soft dependency, respectively.

Registering the job within the appropriate module

A job bean needs to be supplied into the appropriate module so that the framework is aware of its existence. For the scrubber job, it's a job in the GL component, so we need to register the job in the GL module Spring bean.

From KualiSpringBeansGl.xml

```
<bean id="glModule" class="org.kuali.core.KualiModule">
    <!-- other module attributes -->
    <property name="jobNames">
        <list>
            <!-- other GL jobs listed here -->
            <value>scrubberJob</value>
        </list>
    </property>
</bean>
```

Scheduler job

Arguably, the most important job within KFS is the scheduler job, which consists of a single step, `org.kuali.kfs.batch.SchedulerStep`. This step is responsible for running all job Spring beans defined with `scheduledJobDescriptor` as its parent, assuming that a job does not have a trigger of its own (discussed later in this section), its dependencies has been satisfied, and the schedule job has been configured correctly.

When the scheduler job is triggered, it will immediately schedule all jobs that do not have unsatisfied dependencies and do not have their own triggers defined. As dependee jobs complete execution, dependent jobs are scheduled to be run. Note that this means that whenever the scheduler job starts, it starts up other scheduled jobs as well.

The scheduler job is used to start up nightly processing jobs.

When and how long the scheduler job runs is defined using configuration properties

Job Triggers

A trigger is an object that causes a job to be run when a certain event occurs.

This section will describe how to define a trigger on a job so that it will be run at a different time than the main scheduler job. This is useful for situations where jobs need to be run on a monthly or yearly basis.

Defining a trigger

Defining the Spring bean

This is a sample trigger Spring bean for the scheduleJob. Note that it inherits from the "cronTrigger" parent, and the format of the cron expression.

```
<bean id="cfdaJobTrigger" parent="cronTrigger">
    <property name="jobName" value="cfdaJob" />
    <property name="cronExpression" value="00 00 00 1 1,4,7,10  ?" />
</bean>
```

For this particular example, the cfdaJob will be triggered on midnight of Jan. 1, Apr. 1, Jul. 1, and Oct. 1 of every year.

Be careful about defining jobs that depend on jobs with a trigger. If the scheduler job is not running when the dependee job completes execution, then the dependent job will not be scheduled to run. Note that since starting the schedule job causes the nightly jobs to begin execution, it is usually not prudent to start up the scheduler job merely to handle dependencies for custom-triggered jobs.

If there are jobs that depend on jobs with triggers, then the configuration properties should be set to ensure that the scheduler job is still running when the dependee jobs with a trigger complete execution.

Registering the trigger with the module

After the trigger bean has been defined, the trigger needs to be registered to the module to which the triggered job belongs.

In the `KualiModule` definition, the new trigger needs to be specified in the "triggerNames" property.

From KualiSpringBeansCg.xml

```
<bean id="cgModule" class="org.kuali.core.KualiModule">
    <!-- various other module properties -->

    <property name="triggerNames">
        <list>
            <value>cfdaJobTrigger</value>
        </list>
    </property>
</bean>
```

Batch configuration

To configure the batch job, modifying properties and parameters may be needed.

Properties-based batch configuration

There are several properties that are used to control how the batch schedule system behaves. Prior to building KFS, these properties can be overridden by modifying the kuali-build.properites file in the home directory of the user building the application (for other overriding mechanisms, consult the build overview page). Changing these properties will require a rebuild and restart of the application server.

Some important properties are:

- use.quartz.scheduling: true or false, used to indicate whether KFS's batch scheduling mechanism should be turned on
- batch.mailing.list: the email address to notify regarding the batch execution status. This address may be used by other components as well
- batch.schedule.cron.expression: a cron expression that indicates when all scheduled jobs will be eligible to run, subject to any dependencies. Note that if a trigger is defined for a job, then it will not be affected by the value of this parameter.
- use.quartz.jdbc.jobstore: true or false. True indicates that the database should be used as a job store, and if false, memory is used. The advantage of using the database as the job store is that the scheduling state of jobs is retained when the application server shuts down or crashes.

Parameter-based batch configuration

System parameters are used primarily for 3 purposes: to control schedule service parameters, to control whether a step is executable, and to control the user used to run the step.

For this section, parameters are named using the slash notation (i.e. namespace / component / parameter name).

Scheduler job configuration

The following parameters are used to control the scheduler.

- KFS-SY / ScheduleStep / CUTOFF_TIME: Defines the time of which after which the scheduler step will quit after it's been triggered by the default cron trigger (see the batch.schedule.cron.expression) expression above.
- KFS-SY / ScheduleStep / CUTOFF_TIME_NEXT_DAY_IND: Y or N, indicates whether the CUTOFF_TIME listed above represents the cutoff time of the next day. If N, the cutoff time is assumed to be the time on the current day. For example, if the schedule job is triggered at 11PM and the cutoff time is "02:00:00:AM" (i.e. 2 AM), defined in the CUTOFF_TIME parameter above, then this flag will need to be set to Y because the cutoff time is 2 AM *of the next day*.
- KFS-SY / ScheduleStep / STATUS_CHECK_INTERVAL: this parameter represents the number of milliseconds that the schedule step sleeps as it waits for dependencies to be satisfied. This reduces the load on the server (and, if the DB job store is used, the database).

The scheduler service handles dependency resolution. If the cutoff time is set too closely to the time that the scheduler service starts, the scheduler service may not be active for long enough to ensure that after dependee jobs complete, that dependent jobs are executed.
Note that if the scheduler job quits while a dependee job is running, the dependee job will complete execution, but any dependent jobs will not execute because the scheduler will not start them.

Step execution configuration

Parameters are used to configure whether a step is runnable in a job and what user the step should run as. These parameters are optional.

The namespace and component of the parameter are configured dynamically based on the class name of the running step. The namespace generally corresponds to the package in which the step class is located. The detail type is the simple class name (i.e. without the package name) of the Step class. For more information, consult documentation about the parameter service.

There are two parameter names that apply for the namespace and component described above:

- RUN_IND: Y or N. This step will be run only if the RUN_IND parameter does not exist or has a value of "Y".
- USER: The username (network name in the universal user lookup) of the user that this step will run as. If not defined, the step will run as KULUSER, which is the KFS system user.

For example, to create parameters to control the step class `org.kuali.module.gl.batch.ScrubberStep`, parameters should have

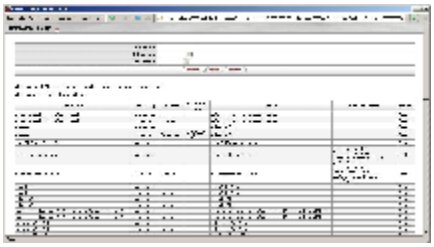
- namespace KFS-GL because the org.kuali.module.gl) package belongs to the GL module
- component name `ScrubberStep`, because it's the simple class name of the class.

Batch schedule screen

A UI has been implemented to help the user control the execution of scheduled jobs as well as manually run jobs. It is accessed through the "Administration" tab of the KFS portal screen.

Batch job lookup

When the "Batch Schedule" link is clicked, it brings up a screen similar to a standard BO lookup. Clicking search without entering criteria will return a list of all jobs defined in the system, similar to what's shown in this screenshot.



Job Groups
For each scheduled job (i.e. the spring job bean inherited from `scheduledJobDescriptor`), there are 2 results shown. One result is in the job group scheduled, and one result is in the job group unscheduled.

For each unscheduled job (i.e. the spring job bean inherited from `unscheduledJobDescriptor`), there is one result shown in the unscheduled group.

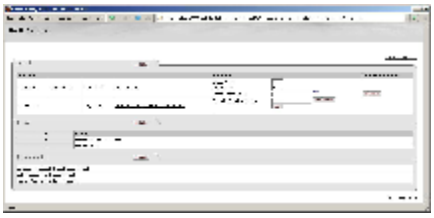
Unscheduling a scheduled job

The batch schedule screen can unschedule a scheduled job. Click "Modify" on the desired job in the "scheduled" group and click on the "unschedule" button. When the batch lookup is performed, the entry in the "scheduled" group will not exist.

Job Stores
If the value of the "use.quartz.jdbc.jobstore" property (look in configuration.properties) is true, any changes made to the schedule will be persisted in the database. If it's false, then changes won't be persisted when the server's restarted.

Manually running jobs and scheduling a job

The batch schedule screen is able to manually run jobs and schedule any jobs.



Under the "Running" section of the screen, the user is able to specify the step(s) that should be run, when the job should be run (if blank, it's run immediately), and to whom the email regarding the execution should be sent.

Depending on the job's state, the "Other commands" section allows the user to schedule or unschedule the job.

When manually running a job, job dependencies are not considered. Caution should be exercised that all necessary data that are normally added by the prerequisite jobs are set up to ensure a successful job run.

Unable to render {include} The included page could not be found.