xTier™ Jobs Service

The xTier™ Jobs Service provides enterprise-grade job scheduling for Java/J2EE and .NET applications.

Key Benefits:

- Flexible Job scheduling
- Schedule Exceptions
- Seamless integration into existing applications

Introduction

Almost all enterprise software projects have a requirement for recurring task execution, be it reporting, system monitoring, automated data back-ups, sending regular informational messages, or simply the need to run a process repetitively at a given time. "Task Schedulers" are the type of program that address this type of functionality and provide the ability to execute tasks according to a predefined schedule. The following sections provide an overview of some common task schedulers and provide an in-depth analysis of the xTier™ Jobs Service.

Common Task Schedulers

There are many different task schedulers that are currently available. This section highlights some of the most widely used choices for job scheduling.

"Task Scheduler" on Windows

Everyone working with Microsoft Windows is familiar with the Task Scheduler utility that is included as a part of the standard set of Windows utilities. The Windows Task Scheduler uses Windows executables as the tasks to schedule and provides several scheduling choices: daily, weekly, monthly, at a specified time, upon PC startup and at Windows User login. For each scheduling type additional parameters such as execution time (for daily scheduling) are available. The Windows Task Scheduler provides password protection in addition to additional parameters such as maximum execution duration, additional execution options depending on PC idle time, and further options depending on the PC power supply.

For the majority of PC users this standard task scheduler provides more than enough functionality to perform simple everyday tasks such as hard drive de-fragmenting, setting reminders, etc. For the developer, simple programs can utilize the Windows Task Scheduler if the application target is only for the Windows operating systems.

cron daemon on UNIX Systems

The cron utility on UNIX systems is a daemon that executes scheduled commands. The cron utility uses UNIX commands and scripts as tasks and schedules periodical execution using minute, hour, day of month, month and day of week. The minimal execution schedule precision is 1 minute. The schedules are stored in property files and are edited with the crontab utility. Note, the UNIX utility "at" is used for single execution of commands or scripts.

Advanced users can use UNIX facilities such as containers, pipes and execution result redirection. The cron utility is well known and has been successfully used by UNIX administrators and users for a long time for everything from simple personal tasks to small projects.

Job on Oracle RDBMS

The Oracle job scheduling mechanism uses PLSQL code for its tasks. The schedule is defined by the first execution time or an execution interval. For use within the RDBMS Oracle Job is convenient and in most cases is sufficient. All settings, execution reports, etc are stored in system tables as with most Oracle utilities.

Conclusion

There are many more schedulers for Windows, both commercial and free, as well as many applications that extend the UNIX cron utility which are typically developed under the GNU license. Additionally many DBMS systems have utilities that perform similar functions. The list above is representational of the different types of schedulers and more importantly highlights the limitations with the existing scheduling utilities. The following lists the limitations and restrictions of the existing systems.

Platform Dependence: All of the existing scheduling applications and utilities are explicitly tied to a single operating system, environment, or application. For all but trivial scheduled tasks, migrating these tasks from one scheduler to another system is a difficult and time-consuming undertaking.

Restricted Job Scheduling: Typically task scheduling in enterprise application development projects often have wider needs in terms of flexible scheduling that cannot be accommodated by the majority of schedulers. Additionally, schedulers must be able to take into account exception cases or a set of non-standard execution variants.

Non-seamless Integration: This is a most critical limitation of using the schedulers discussed in this article and may not be apparent at first glance. For instance, consider a system in which the server side is implemented in Java while the tasks are implemented as UNIX scripts executed using cron. The results of this execution are typically written to the file system, the server side would then need to obtain and parse these results. Or, for example, in the case where all program logic is concentrated in classes loaded in an application server and the RDBMS is responsible for tasks execution.

In this case, the existing schedulers look more like a workaround rather than an enterprise grade job scheduling facility. The differences in technologies decrease system integrity and make the system harder to support and modify. The more systems that are involved in enterprise software projects the less appealing it is for the overall project.

Restricted Functionality: Most schedulers lack an API for scheduler management, do not provide the facility for execution of tasks simultaneously and lack the ability to define an event as a reaction to a task execution (i.e. task execution finish, a failed start, etc.).

xTier™ Jobs Service Overview

xTier™ Jobs Service is a powerful task scheduler with advanced functionality that provides a sophisticated way for developers to use scheduled jobs in their applications. Developers can extend the predefined functionality while still leveraging the built-in functionality which results in a job scheduling system that can quickly implement scheduled jobs while still deal with exceptional cases with the least amount of complexity.

Basic Elements of the xTier™ Jobs Service

The xTier™ Jobs Service works in the same way as other schedulers, i.e. it uses tasks and schedules with the added benefit of working in both Java and C#. The xTier™ Jobs Service contains the following basic elements: Execution Module, Schedule, Exclusions, Task, Control Module, Job Execution Context, and Job Listener.

Execution Module (task) - to set up a task (execution module) the developer must implement the **JobBody** interface containing the single method **invoke()**. All task logic should be implemented in this method. The **JobBody** is the only service interface that must be implemented by the developer.

Schedule - to create a custom scheduler the user has to implement the **JobScheduler** interface. This interface has the main method **getNextExecTime()**, which returns the next task execution time. The xTier™ Jobs Service includes 6 predefined schedulers which cover the general cases in which a scheduler would be used. The list of predefined schedulers includes:

- **JobOneTimeScheduler** - This type of scheduler executes the job only once and the execution time is determined by either the initial delay or the initial time.
- **JobFixedDelayScheduler** - In fixed-delay execution, each execution is scheduled relative to the actual execution time of the previous execution. If an execution is delayed for any reason (such as garbage collection or other background activity), subsequent executions will be delayed as well. In the long run, the frequency of execution will generally be slightly lower than the reciprocal of the specified period.
- **JobFixedRateScheduler** - In fixed-rate execution, each execution is scheduled relative to the scheduled execution time of the initial execution. If an execution is delayed for any reason (such as garbage collection or other background activity), two or more executions will occur in rapid succession to "catch up." In the long run, the frequency of execution will be exactly the reciprocal of the specified period.
- **JobDailyScheduler** - A daily job used to schedule repetitive jobs during a day (24 hours).
- **JobWeeklyScheduler** - A weekly job scheduler used to schedule repetitive jobs during a week.
- **JobMonthlyScheduler** - A monthly job scheduler used to schedule repetitive jobs during a month.

When using the xTier™ Jobs Service the developer must either use one of these predefined schedules, or implement their own custom scheduler by implementing the **JobScheduler** interface.

Exclusions – Tasks can be executed according to a schedule while taking into account exclusions from this schedule. For example, every Monday, except in the case that it is a holiday, a task is to be executed to generate a report. With other

systems, such exclusions are difficult to implement, but in the case of the xTier™ Jobs Service all the developer is required to do to take exclusions into account is to implement the **JobCalendar** interface. The **JobCalendar** interface contains a method **isSchedulableOn(long timestamp)** which returns a Boolean to denote if the value in **timestamp** is an exclusion to the schedule. This is an optional interface and is implemented only if required.

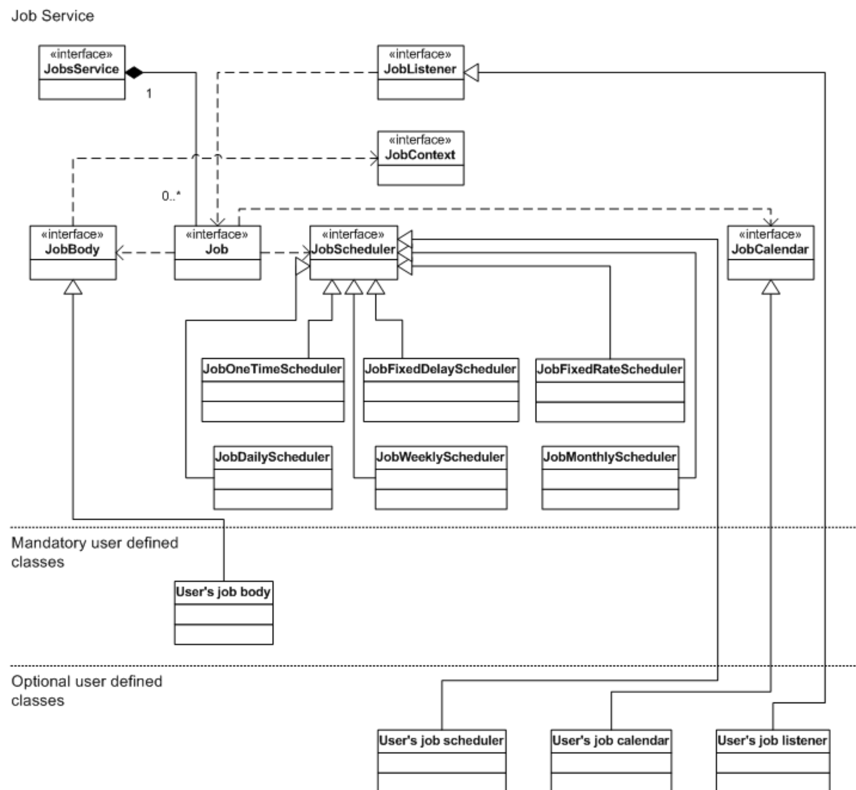
Task – A task is defined by the Job interface. Developers do not implement this interface directly. Instead the developer works with the other elements that have been described in this section: Execution Module, Schedule and Exception.

Control Module – The control module is defined by the **JobService** interface. The control module provides access to the Jobs Service and is designed for task management providing the functionality to create, activate, delete, etc. tasks.

Job Execution Context – Defined by the **JobContext** interface, it is created by the service (control module) and is used for getting lists of active jobs. It is used as an input parameter for the **JobBody** (executing module).

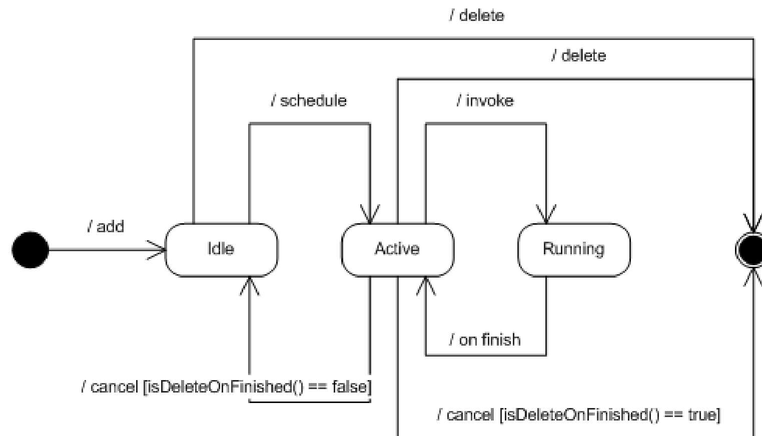
Job Listener - Defined by the **JobListener** interface, this interface is an optional interface but must be implemented by the developer if there is a requirement to execute some code as a reaction to a task's lifecycle events such as task execution finish, task failure, etc.

The following diagram illustrates the interaction and relationship between these elements.



Task State Diagram

The Control Module is responsible for the creation of the task (based on schedule, calendar and execution module), its activation, de-activation and its removal from the system. The following is a state diagram depicting the task states.



- A task is created by the xTier™ Jobs Service and has the initial state "Idle" (i.e. the task is registered in the system, but it is not active (not executing according to schedule)
- From the "idle" state a task can be deleted from the system (by the **delete()** method) or it can transit into the "active" state and become activated for execution.
- From the "active" state a task can:
  - transit into the "running" state (task execution, **invoke()** method)
  - be deleted from the system (**delete()** method)
  - after **cancel()** method depending on **isDeleteOnFinished** attribute:
    - return to the "Idle" state
    - be deleted from the system
- After execution a task returns from the "running" state to the "active" state

Using the xTier™ Configuration Service

Similar to other xTier™ services, the Jobs Service provides the ability for the developer to define Job Service objects in both XML at service start, or programmatically at run-time. The xTier™ Jobs Service has both mandatory and optional objects which are to be implemented by the developer (refer to the class diagram that documents the xTier™ Jobs Service). The following examples will demonstrate how to use the xTier™ Jobs Service.

Simple Example

The minimal implementation that is required for a functioning job to use with the xTier™ Jobs Service is to 1) implement the **JobBody** interface which makes up the execution module and 2) choose one of the 6 predefined schedulers and then

make the appropriate entries in the xTier™ Jobs Service configuration file. Once the job has been defined in the configuration file, the job will be activated upon xTier™ Kernel start.

The following two steps provide a simple demonstration of utilizing the xTier™ Jobs Service. (Please note: the example provided is written in Java. Algorithmically there is no significant difference between Java and C# and as such, a C# implementation is left to the reader.)

Step 1: Implement the **JobBody** interface

```
public class SimpleJobBody implements JobBody {
   /**
    * @see JobBody#invoke(com.fitechlabs.xtier.services.jobs.JobContext)
    */
   public void invoke(JobContext ctx) throws JobException {
      System.out.println("It's example");
   }
}
```

This **JobBody** implementation simply prints a message to the console and in the case of actual work would contain the code to be executed.

Step 2: The next step is to provide the execution schedule for this task. In order to set the schedule modify the xTier™ Jobs Service configuration file (**xtier_jobs.xml**). The following XML schedules this job for execution at 9:15am and 10:30am daily.

```
<xtier-jobs>
   <region name="examples">
      <!--
         Daily job declaration.
      -->
      <job name="daily.job" start="true">
         <scheduler>
            <ioc policy="new">
               <java class="JobDailyScheduler">
                  <ctor>
                     <!-- Day times. -->
                     <arg type="string">09:15:00 am, 10:30:00 am</arg>
                  </ctor>
               </java>
            </ioc>
         </scheduler>

         <body>
            <ioc policy="new">
               <java class="SimpleJobBody"/>
            </ioc>
         </body>
```

```
    </job>
  </region>
</xtier-jobs>
```

- The tag "job", and the attribute "name" – represent the name of this job ("daily.job")
- The tag "job", and the attribute "start" – determine if the task will be active immediately after xTier™ Jobs Service start.
- The tag "scheduler", and the "java class" – represents the class name of the predefined daily job scheduler (**JobDailyScheduler**) in addition to the parameters for this scheduler ("09:15:00 am, 10:30:00 am" representing the schedule).
- The tag "body" and "java class" – represent the user class implementing the execution module.

Advanced Example

In this example a job will be implemented that meets the following conditions:

- The task should be executed daily, at a random time.
- Execution should not be performed on the 3rd and 5th days of each month.
- Execution should not start if at the potential start time the job "alternative.job" is active.
- After each job's invocation the system should perform some activity, such as, log the execution start time to a database.
- The job should not automatically start when the xTier™ Jobs Service starts, but should start by programmatic invocation.

Step 1: Implement the **JobBody** interface to define the execution module.

```
public class ExampleJobBody implements JobBody {
  /**
   * @see JobBody#invoke(com.fitechlabs.xtier.services.jobs.JobContext)
   */
  public void invoke(JobContext ctx) throws JobException {
    // Get all active jobs in system.
    Map activeJobs = ctx.getActiveJobs();

    Job alternativeJob = (Job)activeJobs.get("alternative.job");

    if (alternativeJob != null && alternativeJob.isActive() == true) {
      // Refuse task execution.
      return;
    }

    // Some work.
    ...
  }
}
```

The implementation checks the service's context and if it discovers that a job named "alternative.job" is registered and currently active, job execution does not start.

Step 2: Implement the scheduler which starts the job execution daily at a random time.

```
public class ExampleJobScheduler implements JobScheduler {
   private final static int DAY_MILLIS = 24 * 60 * 60 * 1000;

   private Calendar calendar = Calendar.getInstance();

   private Random random = new Random();

   /**
    * @see JobScheduler#getNextExecTime(Job)
    */
   public long getNextExecTime(Job job) {
      // Last execution time.
      long lastInvokeTime = job.getLastInvokeTime();

      calendar.setTimeInMillis(lastInvokeTime);

      // Add days to last execution time.
      calendar.add(Calendar.DAY_OF_YEAR, 1);

      // Drop hours, minutes and seconds.
      calendar.set(Calendar.HOUR, 0);
      calendar.set(Calendar.MINUTE, 0);
      calendar.set(Calendar.SECOND, 0);
      calendar.set(Calendar.MILLISECOND, 0);

      // Add random value.
      return calendar.getTimeInMillis() + random.nextInt(DAY_MILLIS);
   }

   ......
}
```

The preceding class implements the necessary schedule. Please note that this example demonstrates the implementation of only one basic **JobScheduler** method – **getNextExecTime()**. Also, for the sake of simplicity and clarity this method implementation does not perform some of the functions that would be part of a producting implementation (i.e. determining if this is the first execution, etc.).

Step 3: The third step is to implement the **JobCalendar** interface. This implementation should exclude the third and the fifth day of each month from the schedule.

```
public class ExampleJobCalendar implements JobCalendar {
   private Calendar calendar = Calendar.getInstance();
```

```
    /**
     * @see JobCalendar#isSchedulableOn(long)
     */
    public boolean isSchedulableOn(long timestamp) {
        calendar.setTimeInMillis(timestamp);

        int day = calendar.get(Calendar.DAY_OF_MONTH);

        if (day == 3 || day == 5) {
            return false;
        }

        return true;
    }
}
```

The preceding example provides the necessary exclusions. Please note that this implementation only implements one basic **JobCalendar** interface method – **isSchedulableOn()**.

Step 4: The next step is to implement the **JobListener** interface. The implementation should log the job execution in the database. Please note that "example.job" is the name of the job that is being implemented, and that this example only implements one basic **JobListener** method – **isSchedulableOn()**.

```
public class ExampleJobListener implements JobListener {
    /**
     * @see JobListener#onJobEvent(int, Job)
     */
    public void onJobEvent(int event, Job job) {
        String jobName = job.getName();

        if (jobName.equals("example.job") == true && event == JobListener.JOB_INVOKE_END) {
            // Get DB connection.
            Connection conn = ....;

            Statement stmt = conn.createStatement();

            stmt.execute("INSERT INTO some_table(job, time) VALUES(" + jobName + ", " +
                job.getLastInvokeTime() + ")");
            ....
        }
    }
}
```

Step 5: The final step is to register the job with the service and activate it depending on the necessary conditions.

```
…
// Get the instance of xTier kernel.
```

```
XtierKernel xtier = XtierKernel.getInstance();

// Get the instance of 'jobs' service.
JobService jobs = xtier.jobs();

// Register a job with specified body, scheduler и calendar.
jobs.addJob("example.job", new ExampleJobBody(), new ExampleJobScheduler(),
    new ExampleJobCalendar());

// Register listener for job.
exampleJob.addListener(new ExampleJobListener());

...

if (//someLogic) {
  // Job activation.
  jobs.schedule("example.job", true)
}
```

Although this example does not demonstrate all functionality available to the developer when using the xTier™ Jobs Service, it does illustrate, all basic steps necessary to implement a non-trivial job. This example demonstrates how easy it is to utilize the xTier™ Jobs Service to implement complex scheduled tasks.

It is important to note that when using the predefined schedulers, (for example, the one described previously in this article), when implementing a task the developer should implement the functionality that performs start condition checks, invocation time calculation, etc., in the job body itself. In the case where the developer implements all parts required for task execution, all aspects of the xTier™ Jobs Service are divided into the following parts:

- Job body
- Scheduler
- Exclusions
- Events and exceptions processing
- Common control

Additional Service Configuration Properties

One of additional service features – it provides the ability to determine the maximum number of jobs that are able to run simultaneously (see the "thread-pool-name" tag in **xtier_jobs.xml** configuration file).

Value of this tag provides the name of the thread pool dedicated for job processing. Note that this thread pool must be defined in 'objpool' xml configuration. On thread pool definition in objpool service configuration it is needed to specify size of the given pool. This value determines the maximum number of jobs that can run simultaneously. Of course, this value does not restrict possible number of jobs in this service; the user should only specify the maximum number of simultaneously executing jobs. Such a restriction from user's point of view can be stipulated by the definition of the maximum system loading or restricting of using other resources, such as threads.

Thus, if the service configuration defines, for example, a maximum of 20 jobs for simultaneous execution, and all those 20 jobs are executing when the 21st job should be invoked, the 21st job will not be invoked, but will switch to a waiting mode and wait until one of executing jobs is finished.

Therefore, the xTier™ Jobs Service can not be described as a real-time service since it only guarantees that a job will be invoked not earlier than scheduled. However, this should not be considered a drawback, since the user can define the maximum number of simultaneously executing jobs, and, furthermore, xTier™ is intended for use in operating systems that are not real-time.

Conclusions

This article has reviewed the most commonly used task schedulers, described their benefits and pitfalls in large software projects and reviewed the functionality that is provided by the xTier™ Jobs Service. For trivial projects, the simpler utility type job schedulers may be adequate, but to implement enterprise-class job scheduling, the xTier™ Jobs Service is the clear choice. When considering job scheduling, it is important to keep in mind the benefits of xTier™:

- Sophisticated functionality
- Ease of use
- Functionality is easily extensible by the developer
- Seamless integration with existing software systems.
- Portability
- Flexible configuration (setting maximum system workload, taking into account simultaneous tasks execution etc.)

The design and functionality of the xTier™ Jobs Service makes it the clear choice for enterprise application development projects.

Fitech Laboratories Inc.

Corporate Headquarters
300 Montgomery St., Suite 621
San Francisco, CA 94104
USA

Phone: 1-415-371-8234
Fax: 1-415-371-8237

East Coast Sales Office
330 Madison Ave., 9th floor
New York, NY 10017
USA

Phone: 1-646-495-5076

Fitech Laboratories Japan
Toranomon40 MT Bldg. 3F
5-13-1 Toranomon Minato-ku
Tokyo 105-0001, Japan

Phone: +81-3-5402-7711
Web: www.fitechlabs.co.jp

Copyright © 2003, Fitech Labs, Inc All rights reserved. All trademarks and registered trademarks of other products and services mentioned in this report are the property of their respective owners.