Job Scheduling in Java Applications

Job schedulers let developers focus on their applications and not on scheduling details.

Reading Time: 8 minutes

Scheduling tasks in J2EE applications is a common need. There is more to job scheduling than running backups at 3 o'clock in the morning. Customer relationship management (CRM) applications, for example, need to contact customers periodically during the sales cycle. File transfer applications need to transmit and receives files on a regular basis, except on holidays. Administrative applications need to send reminder emails to notify employees and customers about important events. All these enterprise applications can benefit from job scheduling.

This article first explores the need for job scheduling in J2EE applications. Section 2 explains why job scheduling can add critical functionality to enterprise applications. In Section 3, related work is reviewed. Section 4 presents a set of requirements for job scheduling software in J2EE applications. Finally, Section 5 walks through the design of an enterprise application that uses job scheduling.

The need for enterprise job scheduling

Many enterprise applications can benefit from job scheduling. Sales and customer relationship software, for example, needs to contact customers over the course of months and years. For example:

1. Followups are sent 10 and 30 days after a potential customer first contacts the sales organization.
2. Reminder notices are sent to existing customers 60 and 20 days before the customer's support contract expires.
3. If the customer does not contact the technical support department for 90 days, a followup is sent asking if the customer needs any help with the product or service.
4. Long-term customers are sent rewards every 18 months for their loyalty and patronage.

Other kinds of enterprise applications can also benefit from job scheduling. Credit card and financial institutions transfer thousands of files in hundreds of batches throughout the week. Different batches are scheduled for unattended transfer at different times. Often, the file transfers must occur during business hours. If the normal batch transfer would occur on a financial holiday, it must be rescheduled for the previous business day.

Data collection applications can also make use of a job scheduler. For example, some enterprise applications scour web sites for changes in content. The different web sites may be assigned scores depending on how often they change. Frequently changing web sites, those that change their content several times a day, are assessed the highest scores. Web sites that change only occasionally are assessed the lowest scores. Higher scoring web sites may be

scanned for new content every 37 minutes while the lowest scoring web sites are scanned once every 100 hours.

The previous example showed how data consuming applications benefit from job scheduling. Data producing applications can benefit as well. Web sites and other content providers publish stories and data at regular intervals. On work days, they may publish a story every 90 minutes during the work day while on weekends, stories may be published only twice a day.

Related work

Of course, job scheduling has been around since the early days of computing. The Unix cron command has always been a popular job scheduling solution. It has a powerful facility for determining when jobs, in this case Unix programs, should execute. This facility allows job to be run monthly, weekly, daily, and hourly. Each line of a cron configuration file describes when, and how often, a job runs. cron jobs that are entered in the configuration file persist until deleted from the file. In a Unix environment, it serves its purpose well. In a Java environment, however, cron has its drawbacks. When a job is fired, a new JVM must be started, which is costly. If the job were to communicate with a running Java application, it must use sockets or RMI to do so, which may be cumbersome. Furthermore, because cron is not written in Java, it is not guaranteed to exist on all platforms nor can different cron programs be expected to behave identically. For example, cron behaves differently on AT&T and BSD Unix. It is likely to behave differently on Windows as well. Finally, because cron is not implemented in Java, installing a pure Java application requires installing native software, making software distribution more difficult.

The WebLogic application server has a scheduling facility known as WebLogic Time Services. It is less sophisticated than cron. Jobs are scheduled on a much simpler time basis, based on milliseconds, and are not stored in any persistence mechanism. The advantage is that the Time Services are written in Java, so a new JVM is not created whenever a job fires. The primary disadvantage is that the programmer is required to make complex calculations in order to compute the number of milliseconds between job runs. These calculations belong with the job scheduler, not the programmer. The other disadvantage is that the Time Services run in a WebLogic environment only.

Lastly, programmers often develop customized scheduling software for their particular software applications. The upside, of course, is that this approach gets the job done. The downside is that these custom solutions may not be sophisticated enough to provide the application with all the functionality it could use. Furthermore, these solutions are frequently too specialized or have too narrow a feature set to be reused in additional applications. By using a pre-built job scheduling component, software teams can reduce development costs and bring their applications to market sooner.

Job Scheduling Requirements for Enterprise Applications

4.1: Describing Job Schedules. Job scheduling in enterprise applications have requirements that go beyond the solutions described above. Clearly, a job scheduler written in pure Java integrates best into any Java application, enterprise or not. Like cron, a job scheduler needs a way of expressing when and how often jobs should be scheduled. These expressions should support the need for rescheduling jobs periodically, including yearly, monthly, weekly, daily,

hourly, to-the-minute, to-the-second, and to-the-millisecond repetitions. While using a Java API to describe jobs schedules is traditional, cron shows the convenience of using simple strings to describe when jobs should run.

For example, to run a job every Monday morning at 5 o'clock in the morning, the cron expression is "0 5 * * 1".

At first glance, these expressions may not seem especially more convenient than an equivalent Java API. But consider the advantages of a simple string expression over a set of API calls. The string expression can be persisted to a database or file and can be transmitted over an RMI or socket connection. Furthermore, when an end user describes when to run a job, that description is most easily translated to a string and then fed to the job scheduling software, rather than taking the user's input and making a set of complex API calls to schedule the job.

4.2: Scheduling Jobs. A job scheduler for enterprise applications needs the ability to:

- programmatically schedule and cancel jobs
- schedule jobs to run once
- schedule jobs to run a fixed or variable number of times
- schedule recurring jobs that repeat indefinitely or until some ending date
- schedule jobs that are sensitive to local holidays, such as banking holidays

Different businesses and different countries have many different holiday schedules. A job scheduler for J2EE applications needs to take holidays into account. For instance, some jobs should not run if the bank is closed for a national holiday.

4.3: Job Implementations. In any Java application, including J2EE applications, the actions that a job take are most naturally described using Java code. Typically, a job would be implemented by a Java object that either implements a job interface or subclasses a job class.

4.4: Persistence. A job scheduler that is ready for enterprise use needs an optional persistence mechanism. Scheduled jobs need the ability to be saved in a persistent store such as a relational, object, or XML database. Saved jobs avoid the need to reschedule jobs if the application is stopped and later restarted.

4.5: Looking Up a Job Scheduler Resource. In enterprise applications, resources are often discovered using JNDI, the Java Naming and Directory Interface. JNDI lookups make it possible to use the same mechanism to lookup both local and remote resources. They can also be used to start servers. Finally, they can be used as a factory to provide different implementations of the same interface.

For example, to lookup a remote job scheduler, the following JNDI code can be used.

```
Properties props = new Properties();
props.setProperty(Context.INITIAL_CONTEXT_FACTORY,
                  "com.foo.InitialContextFactory");
props.setProperty(Context.PROVIDER_URL, "js://remotehost");

Context ctx = new InitialContext(props);
Scheduler scheduler = (Scheduler) ctx.lookup("Scheduler");
```

The Properties object is initialized with an appropriate initial context factory and a URL to the remote computer where the job scheduler resides (as shown above in red). Once the InitialContext is instantiated, it is used to lookup a job scheduler resource (as shown above in blue).

The use of JNDI and context factories allows different job scheduler implementations to be swapped in, simply by changing the class name of the initial context factory.

By making use of alternate context provider URLs and context lookup strings, the above JNDI code can be used to start job scheduler servers as well as create job scheduler objects that are purely local to the JVM.

For example, to start a job scheduler server that can be looked up by J2EE applications, the following JNDI code can be used.

```
Properties props = new Properties();
props.setProperty(Context.INITIAL_CONTEXT_FACTORY,
                  "com.foo.InitialContextFactory");
Context ctx = new InitialContext(props);
Scheduler scheduler = (Scheduler) ctx.lookup("SchedulerServer");
```

Note how the context provider URL is omitted and the context lookup string (shown in blue) is different from the previous example. These simple differences permit different object implementations to be lookup and created.

As a final example, to start a purely local job scheduler that will not be exported outside the JVM, the following JNDI code can be used.

```
Properties props = new Properties();
props.setProperty(Context.INITIAL_CONTEXT_FACTORY,
                  "com.foo.InitialContextFactory");
Context ctx = new InitialContext(props);
Scheduler scheduler = (Scheduler) ctx.lookup("Scheduler");
```

Again, note how the use of different context provider URLs and context lookup strings can be used to create different job scheduler objects for different situations.

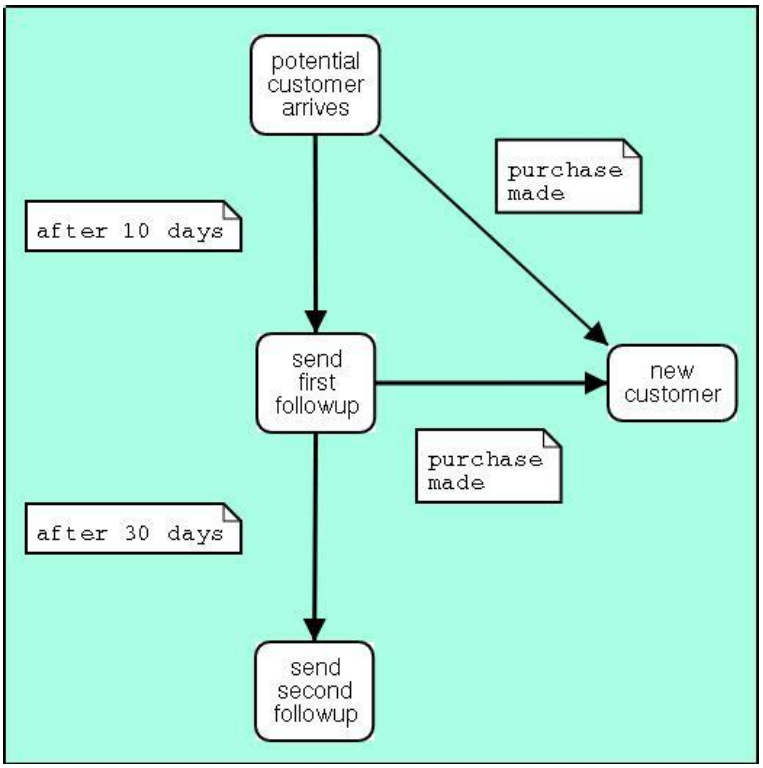
Finally, additional context lookup strings can be used to create helper objects that may be needed by a job scheduler.

4.6: J2EE Application Integration. Any Java job scheduler needs to notify listeners when scheduled jobs fire. A job scheduler for enterprise applications needs to take that a step further. First, REST and SOAP listeners need to be supported to allow the job scheduler to integrate with different applications and disparate environments. To integrate well with J2EE applications, a job scheduler must support listener notification using traditional J2EE mechanisms, including session beans and JMS messages. In other words, when a job fires, a job scheduler may call a method on a session bean or publish a JMS message containing the job details.

Design Walkthrough

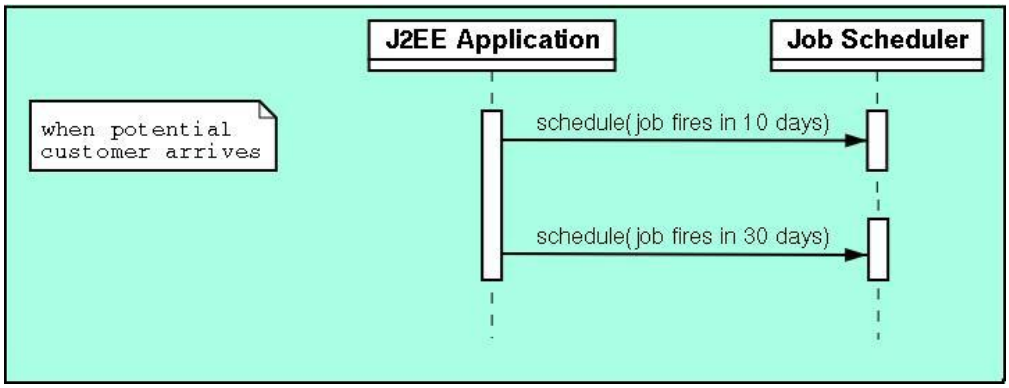
This section walks through the design of a simple customer relationship management (CRM) application use case. The use case is the situation described above. A potential customer contacts a sales organization. The organization will send followups to the potential customer 10 and 30 days after the initial contact. However, should the potential customer make the purchase, any remaining followups will be canceled.

The following state diagram depicts the different stages the customer can be in.

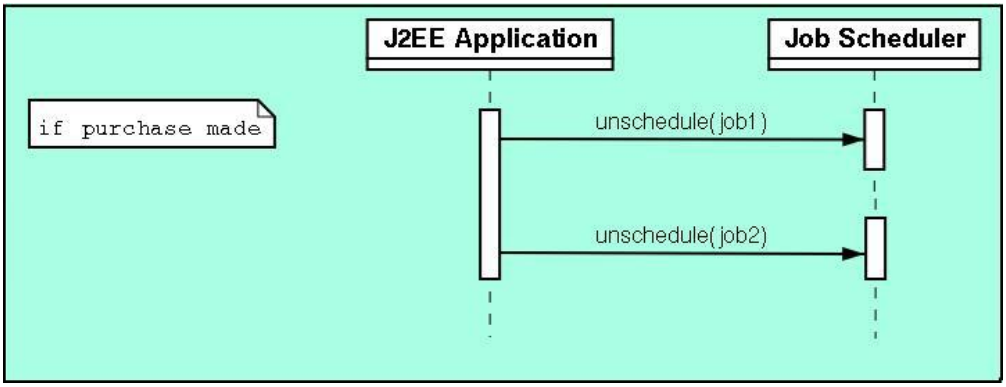


In the above state diagram, potential customers enter the *potential customer arrives* state. Ten days later, if no purchase is made, a followup is sent to the customer. If no purchase is made 30 days after the customer arrives, a second followup is sent. However, if a purchase is made, the followups are canceled.

When the potential customer enters the *potential customer arrives* state, two jobs are immediately scheduled as follows.



Finally, should a purchase be made, the followups should not be sent and the previous jobs are unscheduled as follows.



Conclusion

A wide variety of enterprise applications can take advantage of job schedulers. J2EE-ready job schedulers can enhance the functionality of enterprise applications as well as simplify their design. Furthermore, job scheduling components allow software development teams to focus on their applications and not on the intricate details of scheduling. By using server-side components, software teams can reduce development costs and bring their applications to market sooner.

Although job schedulers have been in the community for a long time, few job schedulers are suitable for use in J2EE applications. The requirements presented in this article provide needed functionality for any job scheduler that is to be used in a J2EE application. Moreover, the design walkthrough shows that integrating a job scheduler into an application can be straightforward.

About Flux

The Flux software platform orchestrates file transfers and batch processing workflows for banking and finance. First released in 2000, Flux has grown into a financial platform that the largest US, UK, and Canadian banks and financial services organizations rely on daily for their mission critical financial systems.

Contact Flux

+1 720-438-4304
sales@flux.ly flux.ly