

6

Task Scheduling with Quartz

In this chapter, we will explore the Quartz task scheduler and its integration with OSWorkflow. We will also give a tutorial with Quartz sending events and actions to OSWorkflow. This gives OSWorkflow temporal capabilities found in some business domains, such as call centers or customer care services.

Both people-oriented and system-oriented BPM systems need a mechanism to execute tasks within an event or temporal constraint, for example, every time a state change occurs or every two weeks. BPM suites address these requirements with a job-scheduling component responsible for executing tasks at a given time.

OSWorkflow, the core of our open-source BPM solution, doesn't include these temporal capabilities by default. Thus, we can enhance OSWorkflow by adding the features present in the Quartz open-source project.

What is Quartz?

Quartz is a Java job-scheduling system capable of scheduling and executing jobs in a very flexible manner. The latest stable Quartz version is 1.6. You can download Quartz from <http://www.opensymphony.com/quartz/download.action>.

Installing

The only file you need in order to use Quartz out of the box is `quartz.jar`. It contains everything you need for basic usage. Quartz configuration is in the `quartz.properties` file, which you must put in your application's classpath.

Basic Concepts

The Quartz API is very simple and easy to use. The first concept that you need to be familiar with is the scheduler. The scheduler is the most important part of Quartz, managing as the word implies the scheduling and unscheduling of jobs and the firing of triggers.

A job is a Java class containing the task to be executed and the trigger is the temporal specification of when to execute the job. A job is associated with one or more triggers and when a trigger fires, it executes all its related jobs. That's all you need to know to execute our `HelloWorld` job.

Integration with OSWorkflow

By complementing the features of OSWorkflow with the temporal capabilities of Quartz, our open-source BPM solution greatly enhances its usefulness. The Quartz-OSWorkflow integration can be done in two ways—Quartz calling OSWorkflow `workflow` instances and OSWorkflow scheduling and unscheduling Quartz jobs. We will cover the former first, by using `trigger-functions`, and the latter with the `ScheduleJob` function provider.

Creating a Custom Job

Jobs are built by implementing the `org.quartz.Job` interface as follows:

```
public void execute(JobExecutionContext context) throws  
                JobExecutionException;
```

The interface is very simple and concise, with just one method to be implemented. The Scheduler will invoke the `execute` method when the trigger associated with the job fires. The `JobExecutionContext` object passed as an argument has all the context and environment data for the job, such as the `JobDataMap`.

The `JobDataMap` is very similar to a Java map but provides strongly typed `put` and `get` methods. This `JobDataMap` is set in the `JobDetail` file before scheduling the job and can be retrieved later during the execution of the job via the `JobExecutionContext's getJobDetail().getJobDataMap()` method.

Trigger Functions

`trigger-functions` are a special type of OSWorkflow function designed specifically for job scheduling and external triggering. These functions are executed when the Quartz trigger fires, thus the name. `trigger-functions` are not associated with an action and they have a unique ID. You shouldn't execute a `trigger-function` in your code.

To define a `trigger-function` in the definition, put the `trigger-functions` declaration before the `initial-actions` element.

```
...  
<trigger-functions>  
  <trigger-function id="10">
```

```

    <function type="beanshell">
      <arg name="script">
        propertySet.setString("triggered", "true");
      </arg>
    </function>
  </trigger-function>
</trigger-functions>
<initial-actions>
...

```

This XML definition fragment declares a `trigger-function` (having an ID of 10), which executes a beanshell script. This script will put a named property inside the `PropertySet` of the instance but you can define a `trigger-function` just like any other Java- or BeanShell-based function.

To invoke this `trigger-function`, you will need an `OSWorkflow` built-in function provider to execute `trigger-functions` and to schedule a custom job—the `ScheduleJob` `FunctionProvider`.

More about Triggers

Quartz's triggers are of two types—the `SimpleTrigger` and the `CronTrigger`. The former, as its name implies, serves for very simple purposes while the latter is more complex and powerful; it allows for unlimited flexibility for specifying time periods.

SimpleTrigger

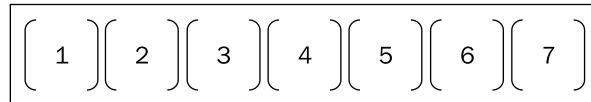
`SimpleTrigger` is more suited for job firing at specific points in time, such as Saturday 1st at 3.00 PM, or at an exact point in time repeating the triggering at fixed intervals. The properties for this trigger are the shown in the following table:

Property	Description
Start time	The fire time of the trigger.
End time	The end time of the trigger. If it is specified, then it overrides the repeat count.
Repeat interval	The interval time between repetitions. It can be 0 or a positive integer. If it is 0, then the repeat count will happen in parallel.
Repeat count	How many times the trigger will fire. It can be 0, a positive integer, or <code>SimpleTrigger.REPEAT_INDEFINITELY</code> .

CronTrigger

The `CronTrigger` is based on the concept of the UN*X Cron utility. It lets you specify complex schedules, like every Wednesday at 5.00 AM, or every twenty minutes, or every 5 seconds on Monday. Like the `SimpleTrigger`, the `CronTrigger` has a start time property and an optional end time.

A `CronExpression` is made of seven parts, each representing a time component:



Each number represents a time part:

- 1 represents seconds
- 2 represents minutes
- 3 represents hours
- 4 represents the day-of-month
- 5 represents month
- 6 represents the day-of-week
- 7 represents year (optional field)

Here are a couple of examples of cron expression:

0 0 6 ? * MON: This `CronExpression` means "Every Monday at 6 AM".

0 0 6 * *: This `CronExpression` mans "Every day at 6 am".

For more information about `CronExpressions` refer to the following website:

<http://www.opensymphony.com/quartz/wikidocs/CronTriggers%20Tutorial.html>.

Scheduling a Job

We will get a first taste of Quartz, by executing a very simple job. The following snippet of code shows how easy it is to schedule a job.

```
SchedulerFactory schedFact = new
                                org.quartz.impl.StdSchedulerFactory();

Scheduler sched = schedFact.getScheduler();
sched.start();
```

```

JobDetail jobDetail = new JobDetail("myJob", null, HelloJob.class);
Trigger trigger = TriggerUtils.makeHourlyTrigger();
                                // fire every hour
trigger.setStartTime(TriggerUtils.getEvenHourDate(new Date()));
                                // start on the next even hour
trigger.setName("myTrigger");
sched.scheduleJob(jobDetail, trigger);

```

The following code assumes a `HelloJob` class exists. It is a very simple class that implements the job interface and just prints a message to the console.

```

package packtpub.osw;
import org.quartz.Job;
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;

/**
 * Hello world job.
 */
public class HelloJob implements Job
{
    public void execute(JobExecutionContext ctx) throws
                                JobExecutionException
    {
        System.out.println("Hello Quartz world.");
    }
}

```

The first three lines of the following code create a `SchedulerFactory`, an object that creates schedulers, and then proceed to create and start a new scheduler.

```

SchedulerFactory schedFact = new
                                org.quartz.impl.StdSchedulerFactory();

Scheduler sched = schedFact.getScheduler();
sched.start();

```

This scheduler will fire the trigger and subsequently the jobs associated with the trigger. After creating the scheduler, we must create a `JobDetail` object that contains information about the job to be executed, the job group to which it belongs, and other administrative data.

```

JobDetail jobDetail = new JobDetail("myJob", null, HelloJob.class);
This JobDetail tells the Scheduler to instantiate a HelloJob object
when appropriate, has a null JobGroup, and has a Job name of "myJob".
After defining the JobDetail, we must create and define the Trigger,
that is, when the Job will be executed and how many times, etc.

```

```
Trigger trigger = TriggerUtils.makeHourlyTrigger();
                                // fire every hour
trigger.setStartTime(TriggerUtils.getEvenHourDate(new Date()));
                                // start on the next even hour
trigger.setName("myTrigger");
```

The `TriggerUtils` is a helper object used to simplify the trigger code. With the help of the `TriggerUtils`, we will create a trigger that will fire every hour. This trigger will start firing the next even hour after the trigger is registered with the Scheduler. The last line of code puts a name to the trigger for housekeeping purposes.

Finally, the last line of code associates the trigger with the job and puts them under the control of the Scheduler.

```
sched.scheduleJob(jobDetail, trigger);
```

When the next even hour arrives after this line of code is executed, the Scheduler will fire the trigger and it will execute the job by reading the `JobDetail` and instantiating the `HelloJob.class`. This requires that the class implementing the job interface must have a no-arguments constructor.

An alternative method is to use an XML file for declaring the jobs and triggers. This will not be covered in the book, but you can find more information about it in the Quartz documentation.

Scheduling from a Workflow Definition

The `ScheduleJob` `FunctionProvider` has two modes of operation, depending on whether you specify the `jobClass` parameter or not. If you declare the `jobClass` parameter, `ScheduleJob` will create a `JobDetail` with `jobClass` as the class implementing the job interface.

```
<pre-functions>
<function type="class">
  <arg name="class.name">com.opensymphony.workflow.util.ScheduleJob
</arg>
  <arg name="jobName">Scheduler Test
</arg>
  <arg name="triggerName">SchedulerTestTrigger</arg>
  <arg name="triggerId">10
</arg>
  <arg name="jobClass">packtpub.osw.SendMailIfActive
</arg>
  <arg name="schedulerStart">>true
```

```

    </arg>
    <arg name="local">true
  </arg>
</function>
</pre-functions>

```

This fragment will schedule a job based on the `SendMailIfActive` class with the current time as the start time. The `ScheduleJob` like any `FunctionProvider` can be declared as a pre or a post function.

On the other hand, if you don't declare the `jobClass`, `ScheduleJob` will use the `WorkflowJob.class` as the class implementing the job interface. This job executes a `trigger-function` on the instance that scheduled it when fired.

```

<pre-functions>
  <function type="class">
    <arg name="class.name">com.opensymphony.workflow.util.ScheduleJob
  </arg>
    <arg name="jobName">Scheduler Test
  </arg>
    <arg name="triggerName">SchedulerTestTrigger
  </arg>
    <arg name="triggerId">10
  </arg>
    <arg name="schedulerStart">true
  </arg>
    <arg name="local">true
  </arg>
  </function>
</pre-functions>

```

This definition fragment will execute the `trigger-function` with ID 10 as soon as possible, because no `CronExpression` or start time arguments have been specified.

This `FunctionProvider` has the arguments shown in the following table:

Argument	Description	Mandatory
<code>triggerId</code>	ID of the trigger function to be executed if no <code>jobClass</code> class name is set. If <code>jobClass</code> is specified, this argument is ignored.	Yes
<code>jobName</code>	The job name.	Yes
<code>triggerName</code>	The trigger name.	Yes
<code>groupName</code>	The group name to be shared between the trigger and the job.	No

Argument	Description	Mandatory
Username	The user name to be used when executing the trigger function.	No
Password	The password to be used when executing the trigger function.	No
Jobclass	If this is specified, <code>ScheduleJob</code> will use this class when creating <code>JobDetail</code> . Otherwise, <code>ScheduleJob</code> will create a job with a class <code>WorkflowJob</code> , used to execute the trigger function with the ID of <code>triggerId</code> .	No
schedulerName	The scheduler name to be used.	No
schedulerStart	If this is set to true, <code>ScheduleJob</code> will create and start a new scheduler.	No
txHack	Set this parameter to true if you are having problems with deadlocks in transactions.	No
cronExpression	The cron expression. If this argument is set, a <code>CronTrigger</code> will be created. Otherwise, a <code>SimpleTrigger</code> will be instantiated.	No
startOffset	This is the offset from the time of execution of the <code>ScheduleJob</code> function provider to the next job. Default is 0.	No
endOffset	This is the offset from the time of execution of the <code>ScheduleJob</code> function provider to the end of the job. Default is no ending.	No
Repeat	The repeat count of the Job. The default can be 0 or <code>REPEAT_INDEFINITELY</code> .	No
Repeatdelay	The offset between repetitions.	No

Transactions in Quartz

Excluding a few minor exceptions, Quartz performs the same transactions in a standalone application or inside a full-blown J2EE Container. One of these exceptions is the use of global JTA transactions inside a JTA-complaint container.

To enable the creation of a new JTA transaction or to join to an existing JTA transaction, just set the `org.quartz.scheduler.wrapJobExecutionInUserTransaction` property inside the `quartz.properties` file to `true`. Enabling this parameter allows the Quartz job to participate inside a global JTA transaction. This in combination with a JTA workflow implementation puts the workflow step and the temporal task into one transaction, thus assuring the information integrity.

JobStores

The JobStore interface designed in Quartz is responsible for the persistence and retrieval of all job and trigger data. There are two built-in implementations of the JobStore interface, the RamJobStore and the JDBCJobStore.

The RamJobStore stores the job, trigger, and calendar data in memory, losing its contents after JVM restarts. On the other hand, JDBCJobStore uses the JDBC API to store the same data.

The JDBCJobStore uses a delegate to use specific functions of each database, for example, DB2, PostgreSQL, etc.

The JobStore configuration is located in the `quartz.properties` file. To set the JobStore, add the following line to the configuration file, if you want to use the RamJobStore:

```
org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore
```

The configuration of the JDBCJobStore is a little more complex as it involves datasources, transactions, and delegates:

To use local JDBC transactions, you only need to set the following parameters:

```
org.quartz.jobStore.class = org.quartz.impl.jdbcjobstore.JobStoreTX
org.quartz.jobStore.dataSource = jdbc/defaultDS
```

The datasource is your datasource JNDI name.

To use global JTA transactions, you need the following parameters:

```
org.quartz.jobStore.class = org.quartz.impl.jdbcjobstore.JobStoreCMT
org.quartz.jobStore.dataSource = jdbc/defaultDS
org.quartz.jobStore.nonManagedTXDataSource = jdbc/nonTXDataSource
```

This differs from the JDBC transaction mode in its use of a non-JTA managed datasource for internal JobStore use.

For both transaction modes you need to set the database delegate appropriate for your database.

```
org.quartz.jobStore.driverDelegateClass=
    org.quartz.impl.jdbcjobstore.StdJDBCDelegate
```

The delegates included with Quartz are as follows:

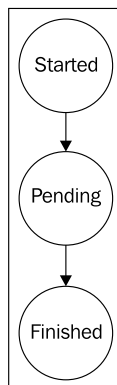
Database	Delegate class
Generic JDBC	<code>org.quartz.impl.jdbcjobstore.StdJDBCDelegate</code>
Microsoft SQL Server and Sybase	<code>org.quartz.impl.jdbcjobstore.MSSQLDelegate</code>
PostgreSQL	<code>org.quartz.impl.jdbcjobstore.PostgreSQLDelegate</code>
Oracle	<code>org.quartz.impl.jdbcjobstore.oracle.OracleDelegate</code>
Cloudscape	<code>org.quartz.impl.jdbcjobstore.CloudscapeDelegate</code>
DB2 v7	<code>org.quartz.impl.jdbcjobstore.DB2v7Delegate</code>
DB2 v8	<code>org.quartz.impl.jdbcjobstore.DB2v8Delegate</code>
HSQldb	<code>org.quartz.impl.jdbcjobstore.HSQldbDelegate</code>
Pointbase	<code>org.quartz.impl.jdbcjobstore.PointbaseDelegate</code>

For more detailed configuration options, refer to the Quartz documentation at <http://www.opensymphony.com/quartz/documentation.action>.

Example Application—Customer Support

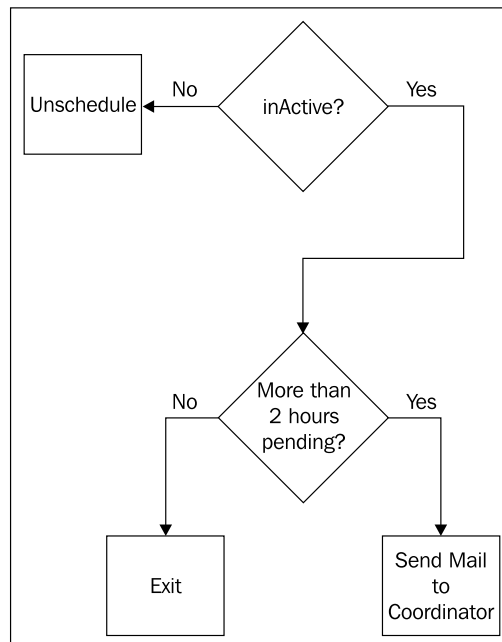
In this section we will develop an example to show the capabilities of the OSWorkflow-Quartz duo. In every company, the customer plays a central role. If not attended to correctly, he or she can turn around and buy services or products from a competitor. Every company also has a customer support department and a good performance indicator for this department would be the number of customer requests attended to.

Some customer support requests come from mail or web interfaces. Suppose you have an web application that receives customer support requests. A typical customer support process is as follows:



This is the most simple of processes and the most commonly implemented. While in the pending state, the request can be forwarded to many people to finally reach completion. If the request is stalled in this state, the process doesn't add value to the business and doesn't match customer expectations, thereby downgrading the company image and customer loyalty.

So a good approach to the process would be to reduce the percentage of support requests in the pending state. If a support request is in the same state for two hours, an e-mail to the customer support coordinator is send, and if a six hour threshold is exceeded an email is send directly to the customer support manager for notification purposes. These notifications assure the request will never be accidentally forgotten. The decision flow is depicted in the following figure:



To implement this process logic, we need temporal support in our business process. Obviously this is done by Quartz. The workflow definition is as follows:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE workflow PUBLIC "-//OpenSymphony Group//DTD OSWorkflow 2.6
//EN" "http://www.opensymphony.com/
osworkflow/workflow_2_8.dtd">

<workflow>
  <initial-actions>
    <action id="100" name="Start Workflow">
      <pre-functions>
        <function type="class">

```

```
<arg name="class.name">packtpub.osw.ScheduleJob
</arg>
<arg name="jobName">TwoHourMail
</arg>
<arg name="jobClass">packtpub.osw.SendMailIfActive
</arg>
<arg name="triggerName">MailIfActive2hTrigger
</arg>
<arg name="triggerId">10
</arg>
<arg name="schedulerStart">>false
</arg>
<arg name="local">>true
</arg>
<arg name="groupName">CustomerSupportJobs
</arg>
<arg name="cronExpression">0 0 */2 * * ?
</arg>
</function>
<function type="class">
  <arg name="class.name">packtpub.osw.ScheduleJob
  </arg>
  <arg name="jobName">SixHourMail
  </arg>
  <arg name="jobClass">packtpub.osw.SendMailIfActive
  </arg>
  <arg name="triggerName">MailIfActive6hTrigger
  </arg>
  <arg name="triggerId">10
  </arg>
  <arg name="schedulerStart">>false
  </arg>
  <arg name="local">>true
  </arg>
  <arg name="groupName">CustomerSupportJobs
  </arg>
  <arg name="cronExpression">0 0 */6 * * ?
  </arg>
</function>
</pre-functions>
<results>
  <unconditional-result old-status="Finished"
                        status="Pending" step="1" />
</results>
```

```

    </action>
  </initial-actions>
  <steps>
    <step id="1" name="Pending">
      <actions>
        <action id="1" name="Finish request" finish="true">
          <results>
            <unconditional-result old-status="Finished" step="2"
                                  status="Finished" />
          </results>
        </action>
      </actions>
    </step>
  </steps>
</workflow>

```

So the process definition is very easy, as we have two steps, but the key of the solution lies in the `ScheduleJob2` `FunctionProvider`. This `FunctionProvider` is a slightly modified version of `OSWorkflow`'s built-in `ScheduleJob`; the only difference is that the new implementation puts the function provider's arguments in the `JobDataMap` of the job. The difference from the original `ScheduleJob` code is as follows:

```
dataMap.putAll (args) ;
```

There is just one line to put the arguments of the `FunctionProvider` into the `JobDataMap`.

The process definition schedules a custom `SendMailIfActive` job every two hours and a `SendMailIfActive` job every six hours. If the process is still in pending state, then a mail is sent, otherwise the job is unscheduled. The job code is as follows:

```

package packtpub.osw;
import java.util.Date;
import java.util.Properties;
import javax.mail.Message;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;
import org.quartz.Job;
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;
import org.quartz.SchedulerException;
import com.opensymphony.workflow.Workflow;

```

```
import com.opensymphony.workflow.basic.BasicWorkflow;
/**
 * Quartz job that send an email if the specified workflow is active.
 */
public class SendMailIfActive implements Job
{
    public void execute(JobExecutionContext ctx) throws
        JobExecutionException
    {
        long wfId = ctx.getJobDetail().getJobDataMap()
            .getLong("entryId");
        String username = ctx.getJobDetail().getJobDataMap()
            .getString("username");
        String to = ctx.getJobDetail().getJobDataMap().getString("to");
        String from = ctx.getJobDetail().getJobDataMap()
            .getString("from");
        String subject = ctx.getJobDetail().getJobDataMap()
            .getString("subject");
        String text = ctx.getJobDetail().getJobDataMap()
            .getString("text");
        String smtpHost = ctx.getJobDetail().getJobDataMap()
            .getString("smtpHost");
        String triggerName = ctx.getJobDetail().getJobDataMap()
            .getString("triggerName");
        String groupName = ctx.getJobDetail().getJobDataMap()
            .getString("groupName");

        Workflow workflow = new BasicWorkflow(username);
        long state = workflow.getEntryState(wfId);
        System.out.println("State:" + state + " for wf:" + wfId);
        if(state != 4)
        {
            sendMail(smtpHost, from, to, subject, text);
        } else
        {
            try
            {
                ctx.getScheduler().unscheduleJob(triggerName, groupName);
            } catch (SchedulerException e)
            {
                e.printStackTrace();
            }
        }
    }
}
```

```
private void sendMail(String smtpHost, String from, String to,
                    String subject, String text)
{
    Properties props = new Properties();
    props.put("mail.smtp.host", smtpHost);
    Session sendMailSession = Session.getInstance(props, null);
    try
    {
        Transport transport = sendMailSession.getTransport("smtp");
        Message message = new MimeMessage(sendMailSession);
        message.setFrom(new InternetAddress(from));
        message.setRecipient(Message.RecipientType.TO, new
                                InternetAddress(to));

        message.setSubject(subject);
        message.setSentDate(new Date());
        message.setText(text);
        message.saveChanges();
        transport.connect();
        transport.sendMessage(message, message.getAllRecipients());
        transport.close();
    } catch (Exception e)
    {
        e.printStackTrace();
    }
}
```

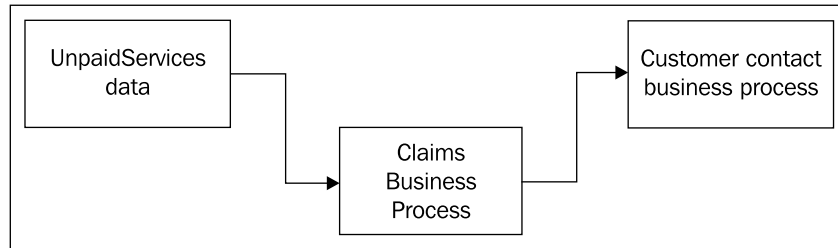
This completes the proactive workflow solution commonly requested by users.

Example Application—Claims Processing

Every service organization has a business process to claim for unpaid services. This process is commonly called claim processing. For every unpaid service a new claim is issued to a representative, to contact the customer for payment.

This process is suited for system-oriented BPM, because it's a batch process that goes through every unpaid service, creates a new claim associated with that service, and assigns this claim to a customer representative. This process runs every day and can be represented with an OSWorkflow `workflow` definition. In this `workflow` definition, there's no human intervention; the representative sees only the end result—the customers he or she has to contact.

Additionally this customer contact is a business process in itself, but this time it is a human-oriented business process. The customer representative has a list of customers he or she has to contact each day.



The components of this business solution are as follows:

- A Quartz job that runs every night and creates new instances of the claim processing workflow.
- A workflow definition that uses auto-actions and needs no human interaction at all. This definition reflects the real-business process. It gets all the unpaid services from a web service and creates a new customer contact workflow.
- A `FunctionProvider` to call the unpaid services from a web service and create a new customer contact workflow for each one of them.
- A workflow definition for the customer contact business process.

By creating a new workflow every day, this workflow is fully automatic and has a little intelligence for identifying failed states.

The first component of the solution is a Quartz job to instantiate a new workflow. We will call this job `WorkflowInitJob`. It is described in the following snippet of code:

```
package packtpub.osw;
import java.util.Map;
import org.quartz.Job;
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;
import com.opensymphony.workflow.Workflow;
import com.opensymphony.workflow.basic.BasicWorkflow;
import com.opensymphony.workflow.config.Configuration;
import com.opensymphony.workflow.config.DefaultConfiguration;
/**
 * Creates a new workflow instance.
 */
public class WorkflowInitJob implements Job
{
    public void execute(JobExecutionContext ctx) throws
```

```

                                                                    JobExecutionException
{
    String userName = ctx.getJobDetail().getJobDataMap()
                                                                    .getString("user");
    String wfName = ctx.getJobDetail().getJobDataMap()
                                                                    .getString("wfName");
    Map inputs = (Map) ctx.getJobDetail().getJobDataMap()
                                                                    .get("inputs");
    int initAction = ctx.getJobDetail().getJobDataMap()
                                                                    .getInt("initAction");
    Workflow workflow = new BasicWorkflow(userName);
    Configuration config = new DefaultConfiguration();
    workflow.setConfiguration(config);
    try
    {
        long workflowId = workflow.initialize(wfName, initAction,
                                                                    inputs);
        System.out.println("Instantiated new workflow with id:" +
                                                                    workflowId);
    } catch (Exception e)
    {
        throw new JobExecutionException(e);
    }
}
}

```

The second component of the claims processing workflow solution is a workflow definition. It is as follows:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE workflow PUBLIC "-//OpenSymphony Group//DTD OSWorkflow 2.6
                                                                    //EN" "http://www.opensymphony.com/
                                                                    osworkflow/workflow_2_8.dtd">
<workflow>
  <initial-actions>
    <action id="100" name="Start Workflow">
      <results>
        <unconditional-result old-status="Finished"
                                                                    status="Pending" step="1" />
      </results>
    </action>
  </initial-actions>
  <steps>
    <step id="1" name="Get unpaid and create contact">
      <actions>

```

```
<action id="1" name="process data" finish="true" auto="true">
  <pre-functions>
    <function type="class">
      <arg name="class.name">packtpub.osw
                                     .ClaimsWebServiceProvider
      </arg>
      <arg name="url">http://localhost:8080/ws/unpaid
      </arg>
      <arg name="username">${caller}
      </arg>
    </function>
  </pre-functions>
  <results>
    <unconditional-result old-status="Finished"
                          step="1" status="Created" />
  </results>
</action>
</actions>
</step>
</steps>
</workflow>
```

The customer contact definition is simpler and is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE workflow PUBLIC "-//OpenSymphony Group//DTD OSWorkflow 2.6
                             //EN" "http://www.opensymphony.com/
                             osworkflow/workflow_2_8.dtd">
<workflow>
  <initial-actions>
    <action id="100" name="Start Workflow">
      <results>
        <unconditional-result old-status="Finished" status="Pending"
                              step="1" />
      </results>
    </action>
  </initial-actions>
  <steps>
    <step id="1" name="Contact">
      <actions>
        <action id="1" name="Finish contact" finish="true">
          <results>
            <unconditional-result old-status="Finished" step="1"
                                  status="Contacted" />
          </results>
        </action>
      </actions>
    </step>
  </steps>
</workflow>
```

```
        </action>
    </actions>
</step>
</steps>
</workflow>
```

The third component is a `FunctionProvider`, which calls the unpaid services from a web service and for each unpaid service generates a new customer contact workflow. This `FunctionProvider` uses the Apache Axis Web Service Framework to call a standard SOAP web service. You can find more information about the Apache Axis framework at the following website: <http://ws.apache.org/axis/java/index.html>.

The `FunctionProvider` code is as follows:

```
package packtpub.osw;

import java.util.Iterator;
import java.util.List;
import java.util.Map;

import com.opensymphony.module.propertyset.PropertySet;
import com.opensymphony.workflow.FunctionProvider;
import com.opensymphony.workflow.Workflow;
import com.opensymphony.workflow.WorkflowException;
import com.opensymphony.workflow.basic.BasicWorkflow;
import com.opensymphony.workflow.config.Configuration;
import com.opensymphony.workflow.config.DefaultConfiguration;

/**
 * Gets unpaid services data from web service and
 * creates a new customer contact workflow for each one.
 */
public class ClaimsWebServiceProvider implements FunctionProvider
{
    public void execute(Map arg0, Map arg1, PropertySet arg2)
        throws WorkflowException
    {
        Workflow workflow = new BasicWorkflow("test");
        Configuration config = new DefaultConfiguration();
        workflow.setConfiguration(config);
        List unpaidData = getUnpaidDataFromWebService();
        for (Iterator iter = unpaidData.iterator(); iter.hasNext();)
        {
            UnpaidService service = (UnpaidService) iter.next();
            Map serviceData = serviceToMap(service);
        }
    }
}
```

```
        workflow.initialize("customer-contact", 100, serviceData);
    }
}
private Map serviceToMap(UnpaidService service)
{
    ...
}
private List getUnpaidDataFromWebService()
{
    ...
}
}
```

Finally and to integrate all the solution, there's a Java class designed specifically to call the Quartz Scheduler and schedule the Quartz job for running every night.

Quartz has many more features worth exploring. For more information about Quartz check its website and the Quartz Wiki at <http://wiki.opensymphony.com/display/QRTZ1/Quartz+1>.

Summary

In this chapter, we covered the integration of the Quartz job-scheduling system with OSWorkflow, which provided temporal capabilities to OSWorkflow. We also took a look at the `trigger`-functions, which are executed when a Quartz trigger fires. We also learned how to schedule a job from a Workflow definition by using `ScheduleJob`. Finally, we showed the capabilities of the Quartz-OSWorkflow duo with the help of two sample applications.