

Hadoop Workflow System (HWS)

Specification

(PROPOSAL v1.0–2009MAR09)

Table of Contents

Hadoop Workflow System (HWS) Specification.....	1
Changelog.....	1
2009MAR09.....	1
2009FEB22.....	1
0 Definitions.....	1
1 Specification Highlights.....	1
2 Workflow Definition.....	2
2.1 Cycles in Workflow Definitions.....	2
3 Workflow Nodes.....	3
3.1 Control Flow Nodes.....	3
3.1.1 Start Control Node.....	3
3.1.2 End Control Node.....	3
3.1.3 Kill Control Node.....	4
3.1.4 Decision Control Node.....	4
3.1.5 Fork and Join Control Nodes.....	6
3.2 Workflow Action Nodes.....	7
3.2.1 Action Basis.....	7
3.2.1.1 Action Computation/Processing Is Always Remote.....	7
3.2.1.2 Actions Are Asynchronous.....	7
3.2.1.3 Actions have 2 Transitions, ok and error.....	7
3.2.1.4 Action Recovery.....	7
3.2.2 Map–Reduce Action.....	8
3.2.2.1 Java Map/Reduce Job.....	8
3.2.2.2 Map–Reduce Streaming Action.....	10
3.2.3 Pig Action.....	11
3.2.4 Fs action.....	13
3.2.5 Ssh Action.....	15
3.2.6 Sub–workflow Action.....	16
3.2.7 Http Action.....	17
3.2.8 Email Node.....	18
4 Parameterization of Workflows.....	19
4.1 Workflow Job Properties.....	19
4.2 Expression Language Functions.....	20
4.2.1 EL Constants.....	21
4.2.2 Basic EL Functions.....	21
4.2.3 Workflow EL Functions.....	21
4.2.4 Hadoop EL Functions.....	22
4.2.5 HDFS EL Functions.....	23
4.2.6 Action EL Functions.....	23
5 HWS Notifications.....	23
5.1 Action Start and End Notifications.....	24
5.2 Workflow Job End Notification.....	24
6 User Propagation.....	24
6.1 User Propagation for Map–Reduce Jobs.....	24
6.2 User Propagation for Pig Jobs.....	25
6.3 User Propagation for FS Operations.....	25
6.4 User Propagation for Ssh Jobs.....	26
7 Workflow Applications Packaging.....	26
8 Workflow Applications Lifecycle.....	27
9 Self Containment Model.....	27
10 External Data Assumptions.....	28
11 Workflow Jobs Lifecycle.....	28
12 Workflow Jobs Recovery.....	28
13 Workflow Applications API.....	29

Table of Contents

Hadoop Workflow System (HWS) Specification

14 Workflow Administration API.....	29
15 Customizing HWS with Extensions.....	30
16 Workflow Jobs Priority.....	30
17 Client API.....	30
18 Command Line Tools.....	30
19 Web UI Console.....	30
Appendixes.....	31
Appendix A, HWS XML–Schema.....	31
Appendix B, Workflow Examples.....	34
Fork and Join Example.....	34

Hadoop Workflow System (HWS) Specification

(PROPOSAL v1.0–2009MAR09)

This goal of this document is to define a workflow engine system specialized in coordinating the execution of Hadoop Map/Reduce and Pig jobs.

Changelog

2009MAR09

- Changed CREATED job state to PREP to have same states as Hadoop
- Renamed 'hadoop-workflow' element to 'workflow-app'
- Decision syntax changed to be 'switch/case' with no transition indirection
- Action nodes common root element 'action', with the action type as sub-element (to use a single built-in XML schema)
- Action nodes have 2 explicit transitions 'ok to' and 'error to' enforced by XML schema
- Renamed 'fail' action element to 'kill'
- Renamed 'hadoop' action element to 'map-reduce'
- Renamed 'hdfs' action element to 'fs'
- Updated all XML snippets and examples
- Made user propagation simpler and consistent
- Added HWS XML schema to Appendix A
- Added workflow example to Appendix B

2009FEB22

- Opened JIRA HADOOP-5303 attached Revision 3 of this doc.

0 Definitions

Action: An execution/computation task (Map-Reduce job, Pig job, a shell command). It can also be referred as task or 'action node'.

Workflow: A collection of actions arranged in a control dependency DAG (Direct Acyclic Graph). "control dependency" from one action to another means that the second action can't run until the first action has completed.

Workflow Definition: A programmatic description of a workflow that can be executed.

Workflow Definition Language: The language used to define a Workflow Definition.

Workflow Job: An executable instance of a workflow definition.

Workflow Engine: A system that executes workflows jobs. It can also be referred as a DAG engine.

1 Specification Highlights

A Workflow application is DAG that coordinates the following types of actions: Hadoop, Pig, Ssh, Http, Email and sub-workflows.

Flow control operations within the workflow applications can be done using decision, fork and join nodes. Cycles in workflows are not supported.

Actions and decisions can be parameterized with job properties, actions output (i.e. Hadoop counters, Ssh key/value pairs output) and file information (file exists, file size, etc). Formal parameters are expressed in the workflow definition as `${VAR}` variables.

A Workflow application is a ZIP file that contains the workflow definition (an XML file), all the necessary files to run all the actions: JAR files for Map/Reduce jobs, shells for streaming Map/Reduce jobs, native libraries, Pig scripts, and other resource files.

Before running a workflow job, the corresponding workflow application must be deployed in HWS.

Deploying workflow application and running workflow jobs can be done via command line tools, a WS API and a Java API.

Monitoring the system and workflow jobs can be done via a web console, command line tools, a WS API and a Java API.

When submitting a workflow job, a set of properties resolving all the formal parameters in the workflow definitions must be provided. This set of properties is a Hadoop configuration.

Possible states for a workflow jobs are: `PREP`, `RUNNING`, `SUSPENDED`, `SUCCEEDED`, `KILLED` and `FAILED`.

In the case of an action failure in a workflow job, depending on the type of failure, HWS will attempt automatic retries, it will request a manual retry or it will fail the workflow job.

HWS can make HTTP callback notifications on action start/end/failure events and workflow end/failure events.

In the case of workflow job failure, the workflow job can be resubmitted skipping previously completed actions. Before doing a resubmission the workflow application could be updated with a patch to fix a problem in the workflow application code.

2 Workflow Definition

A workflow definition is a DAG with control flow nodes (start, end, decision, fork, join, fail) or action nodes (map-reduce, pig, etc.), nodes are connected by transitions arrows.

The workflow definition language is XML based and it is called hPDL (Hadoop Process Definition Language).

2.1 Cycles in Workflow Definitions

HWS does not support cycles in workflow definitions, workflow definitions must be a strict DAG.

At workflow application deployment time, if HWS detects a cycle in the workflow definition it must fail the deployment.

3 Workflow Nodes

Workflow nodes are classified in control flow nodes and action nodes:

- **Control flow nodes:** nodes that control the start and end of the workflow and workflow job execution path.
- **Action nodes:** nodes that trigger the execution of a computation/processing task.

Node names and transitions must be conform to the following pattern `[a-zA-Z][\-_a-zA-Z0-0]*`, of up to 20 characters long.

3.1 Control Flow Nodes

Control flow nodes define the beginning and the end of a workflow (the `start`, `end` and `fail` nodes) and provide a mechanism to control the workflow execution path (the `decision`, `fork` and `join` nodes).

3.1.1 Start Control Node

The `start` node is the entry point for a workflow job, it indicates the first workflow node the workflow job must transition to.

When a workflow is started, it automatically transitions to the node specified in the `start`.

A workflow definition must have one `start` node.

Syntax:

```
<workflow-app name="[WF-DEF-NAME]" xmlns="uri:hws-app:1.0">
  ...
  <start to="[NODE-NAME]"/>
  ...
</workflow-app>
```

The `to` attribute is the name of first workflow node to execute.

Example:

```
<workflow-app name="foo-wf" xmlns="uri:hws-app:1.0">
  ...
  <start to="firstHadoopJob"/>
  ...
</workflow-app>
```

3.1.2 End Control Node

The `end` node is the end for a workflow job, it indicates that the workflow job has completed successfully.

When a workflow job reaches the `end` it finishes successfully (SUCCEEDED).

If one or more actions started by the workflow job are executing when the `end` node is reached, the actions will be killed. In this scenario the workflow job is still considered as successfully run.

A workflow definition must have one `end` node.

Syntax:

```

<workflow-app name="[WF-DEF-NAME]" xmlns="uri:hws-app:1.0">
  ...
  <end name="[NODE-NAME]" />
  ...
</workflow-app>

```

The name attribute is the name of the transition to do to end the workflow job.

Example:

```

<workflow-app name="foo-wf" xmlns="uri:hws-app:1.0">
  ...
  <end name="end" />
</workflow-app>

```

3.1.3 Kill Control Node

The kill node allows a workflow job to kill itself.

When a workflow job reaches the kill it finishes in error (KILLED).

If one or more actions started by the workflow job are executing when the kill node is reached, the actions will be killed.

A workflow definition may have zero or more kill nodes.

Syntax:

```

<workflow-app name="[WF-DEF-NAME]" xmlns="uri:hws-app:1.0">
  ...
  <kill name="[NODE-NAME]">
    <message>[MESSAGE-TO-LOG]</message>
  </kill>
  ...
</workflow-app>

```

The name attribute in the kill node is the name of the Kill action node.

The content of the message element will be logged as the kill reason for the workflow job.

A kill node does not have transition elements because it ends the workflow job, as KILLED.

Example:

```

<workflow-app name="foo-wf" xmlns="uri:hws-app:1.0">
  ...
  <kill name="killBecauseNoInput">
    <message>Input unavailable</message>
  </kill>
  ...
</workflow-app>

```

3.1.4 Decision Control Node

A decision node enables a workflow to make a selection on the execution path to follow.

The behavior of a decision node can be seen as a switch-case statement.

A `decision` node consists of a list of predicates–transition pairs plus a default transition. Predicates are evaluated in order or appearance until one of them evaluates to `true` and the corresponding transition is taken. If none of the predicates evaluates to `true` the `default` transition is taken.

Predicates are JSP Expression Language (EL) expressions (refer to section 4.2 of this document) that resolve into a boolean value, `true` or `false`. For example:

```
${wf:nodeStatus('myfirstjob') == 'OK'}  
  
${hdfs:size('/usr/foo/myinputdir') gt 10 * GB}
```

Syntax:

```
<workflow-app name="[WF-DEF-NAME]" xmlns="uri:hws-app:1.0">  
  ...  
  <decision name="[NODE-NAME]">  
    <switch>  
      <case to="[NODE_NAME]">[PREDICATE]</case>  
      ...  
      <case to="[NODE_NAME]">[PREDICATE]</case>  
      <default name="[NODE_NAME]" />  
    </switch>  
  </decision>  
  ...  
</workflow-app>
```

The `name` attribute in the `decision` node is the name of the decision node.

Each `case` elements contains a predicate an a transition name. The predicate ELs are evaluated in order until one returns `true` and the corresponding transition is taken.

The `default` element indicates the transition to take if none of the predicates evaluates to `true`.

All decision nodes must have a `default` element to avoid bringing the workflow into an error state if none of the predicates evaluates to `true`.

Example:

```
<workflow-app name="foo-wf" xmlns="uri:hws-app:1.0">  
  ...  
  <decision name="mydecision">  
    <switch>  
      <case to="reconsolidatejob">  
        ${hdfs:size(wf:conf('secondjob.output.dir')) gt 10 * GB}  
      </case>  
      <case to="reexpandjob">${hdfs:size(wf:conf('secondjob.output.dir')) lt 100 * MB}</eval  
      <case to="recomputejob">  
        ${  
          hadoop:counters('secondjob')['org.apache.hadoop.mapred.Task.Counter']['REDUCE_OUTPUT_RECORD']  
          lt 1000000  
        }  
      </case>  
      <default name="end" />  
    </switch>  
  </decision>  
  ...  
</workflow-app>
```

3.1.5 Fork and Join Control Nodes

A fork node splits one path of execution into multiple concurrent paths of execution.

A join node waits until every concurrent execution path of a previous fork node arrives to it.

fork and join nodes must be used in pairs. The join node assumes concurrent execution paths are children of the same fork node.

Syntax:

```
<workflow-app name="[WF-DEF-NAME]" xmlns="uri:hws-app:1.0">
  ...
  <fork name="[FORK-NODE-NAME]">
    <transition to="[NODE-NAME]" />
    ...
    <transition to="[NODE-NAME]" />
  </fork>
  ...
  <join name="[JOIN-NODE-NAME]">
    <transition to="[NODE-NAME]" />
  </join>
  ...
</workflow-app>
```

The name attribute in the fork node is the name of the workflow fork node. The to attribute in the transition elements in the fork node indicate the name of the workflow node that will be part of the concurrent execution.

The name attribute in the join node is the name of the workflow join node. The to attribute in the transition element in the join node indicates the name of the workflow node that will be executed after all concurrent execution paths of the corresponding fork arrive to the join node.

Example:

```
<workflow-app name="sample-wf" xmlns="uri:hws-app:1.0">
  ...
  <fork name="forking">
    <transition to="firstparalleljob"/>
    <transition to="secondparalleljob"/>
  </fork>

  <action name="firstparalleljob">
    <map-reduce>
      <job-xml>job1.xml</job-xml>
    </map-reduce>
    <ok to="joining"/>
    <error to="fail"/>
  </action>

  <map-reduce name="secondparalleljob">
    <map-reduce>
      <job-xml>job2.xml</job-xml>
    </map-reduce>
    <ok to="joining"/>
    <error to="fail"/>
  </map-reduce>

  <join name="joining">
    <transition to="nextaction"/>
  </join>
  ...
</workflow-app>
```

</workflow-app>

3.2 Workflow Action Nodes

Action nodes are the mechanism by which a workflow triggers the execution of a computation/processing task.

3.2.1 Action Basis

The following sub-sections define common behavior and capabilities for all action types.

3.2.1.1 Action Computation/Processing Is Always Remote

All computation/processing tasks triggered by an action node are remote to HWS. No workflow application specific computation/processing task is executed within HWS.

3.2.1.2 Actions Are Asynchronous

All computation/processing tasks triggered by an action node are executed asynchronously by HWS. For most types of computation/processing tasks triggered by workflow action, the workflow job has to wait until the computation/processing task completes before transitioning to the following node in the workflow.

The exception is the `fs` action that is handled as a synchronous action.

HWS can detect completion of computation/processing tasks by two different means, callbacks and polling.

When a computation/processing task is started by HWS, HWS provides a unique callback URL to the task, the task should invoke the given URL to notify its completion.

For cases that the task failed to invoke the callback URL for any reason (i.e. a transient network failure) or when the type of task cannot invoke the callback URL upon completion, HWS has a mechanism to poll computation/processing tasks for completion.

3.2.1.3 Actions have 2 Transitions, `ok` and `error`

If a computation/processing task –triggered by a workflow– completes successfully, it transitions to `ok`.

If a computation/processing task –triggered by a workflow– fails to complete successfully, its transitions to `error`.

If a computation/processing task exits in error, there computation/processing task must provide `error-code` and `error-message` information to HWS. This information can be used from `decision` nodes to implement a fine grain error handling at workflow application level.

Each action type must clearly define all the error codes it can produce.

3.2.1.4 Action Recovery

Under certain conditions, HWS will provide a recovery mechanism for computation/processing task triggered by a workflow action node before the action exits in error.

Depending on the nature of the action failure HWS will have different recovery strategies.

If the action failure is of a recoverable transient nature, HWS must perform retries after a pre-defined (fixed or exponential) time interval. The number of retries and timer interval for a type of action must be

pre-configured at HWS level. Workflow jobs can override such configuration.

Examples of a transient failures are network problems or a remote system temporary unavailable.

If the action failure is of a recoverable non-transient nature, HWS will suspend the workflow job until an manual or programmatic intervention resumes the workflow job and the action is retried. It is the responsibility of an administrator or an external managing system to perform any necessary cleanup before retrying the action.

If the action failure is of a non-recoverable, HWS will fail the workflow job immediately.

3.2.2 Map-Reduce Action

The `map-reduce` action starts a Hadoop map/reduce job from a workflow. Hadoop jobs can be Java Map/Reduce jobs or streaming jobs.

A `map-reduce` action can be configured to perform HDFS files/directories cleanup before starting the Hadoop job. This capability enables HWS to retry a Hadoop job in the situation of a transient failure (Hadoop checks the non-existence of the job output directory and then creates it when the Hadoop job is starting, thus a retry without cleanup of the job output directory would fail).

The workflow job will wait until the Hadoop map/reduce job completes before continuing to the next action in the workflow execution path.

The counters of the Hadoop job and job exit status (`FAILED`, `KILLED` or `SUCCEEDED`) must be available to the workflow job after the Hadoop jobs ends. This information can be used from within decision nodes.

The `map-reduce` action has to be configured with all the necessary Hadoop JobConf properties to run the Hadoop map/reduce job.

The Hadoop JobConf properties can be specified in a JobConf XML file bundled with the workflow application or they can be indicated inline in the `map-reduce` action configuration. If both are used and the same property is in the JobConf XML file and in the inline configuration, the inline property value has precedence.

Inline property values can be parameterized (templated) using EL expressions.

Hadoop `mapred.job.tracker` and `fs.default.name` properties must always be present in the `hadoop` action configuration, inline or in the JobConf XML file.

3.2.2.1 Java Map/Reduce Job

Syntax:

```
<workflow-app name="[WF-DEF-NAME]" xmlns="uri:hws-app:1.0">
  ...
  <action name="[NODE-NAME]">
    <map-reduce>
      <prepare>
        <delete path="[PATH]" />
        ...
        <delete path="[PATH]" />
      </prepare>
      <job-xml>[JOB-XML-FILE]</job-xml>
      <configuration>
        <property>
          <name>[PROPERTY-NAME]</name>
```

```

        <value>[PROPERTY-VALUE]</value>
      </property>
    ...
  </configuration>
</map-reduce>

  <ok to="[NODE-NAME]" />
  <error to="[NODE-NAME]" />
</action>
...
</workflow-app>

```

The `prepare` element, if present, indicates a list of path do delete before starting the job. This should be used exclusively for directory cleanup for the job to be executed. The delete operation will be performed in the `fs.default.name` filesystem.

The `job-xml` element, if present, must refer to a Hadoop JobConf `job.xml` file bundled in the workflow application. The `job-xml` element is optional and if present it can be only one.

The `configuration` element, if present, contains JobConf properties for the Hadoop job.

Properties specified in the `configuration` element override properties specified in the file specified in the `job-xml` element.

Example:

```

<workflow-app name="foo-wf" xmlns="uri:hws-app:1.0">
  ...
  <action name="myfirstHadoopJob">
    <map-reduce>
      <prepare>
        <delete path="/usr/tucu/output-data"/>
      </prepare>
      <job-xml>/myfirstjob.xml</job-xml>
      <configuration>
        <property>
          <name>mapred.input.dir</name>
          <value>/usr/tucu/input-data</value>
        </property>
        <property>
          <name>mapred.output.dir</name>
          <value>/usr/tucu/input-data</value>
        </property>
        <property>
          <name>mapred.reduce.tasks</name>
          <value>${wf:conf('first.job.reducers')}</value>
        </property>
      </configuration>
    </map-reduce>

    <ok to="myNextAction"/>
    <error to="errorCleanup"/>
  </action>
  ...
</workflow-app>

```

In the above example, the number of Reducers to be used by the Map/Reduce job has to be specified as a parameter of the workflow job configuration when creating the workflow job.

3.2.2.2 Map-Reduce Streaming Action

The map-reduce action provides support for Hadoop Streaming via the `streaming` element.

Syntax:

```
<workflow-app name="[WF-DEF-NAME]">
  ...
  <action name="[NODE-NAME]">
    <map-reduce>
      <prepare>
        <delete path="[PATH]" />
        ...
        <delete path="[PATH]" />
      </prepare>
      <streaming>
        <mapper>[MAPPER-PROCESS]</mapper>
        <reducer>[REDUCER-PROCESS]</reducer>
        <files>
          <file>[ANY-REQUIRED-FILES]</files>
          ....
          </files>
        <cacheFiles>
          <file>[ANY-REQUIRED-FILES]</files>
          ....
        </cacheFiles>
        <cacheArchives>
          <file>[ANY-REQUIRED-ARCHIVE-FILES]</files>
          ....
        </cacheArchives>
      </streaming>
      <job-xml>[JOB-XML-FILE]</job-xml>
      <configuration>
        <property>
          <name>[PROPERTY-NAME]</name>
          <value>[PROPERTY-VALUE]</value>
        </property>
        ...
      </configuration>
    </map-reduce>

    <ok to="[NODE-NAME]" />
    <error to="[NODE-NAME]" />
  </action>
  ...
</workflow-app>
```

The `mapper` and `reducer` elements are used to specify the executable/script to be used as mapper and reducer.

User defined scripts must be bundled with the workflow application and they must be declared in the `files` element of the streaming configuration. If they are not declared in the `files` element of the configuration it is assumed they will be available (and in the command PATH) of the Hadoop slave machines.

Some streaming jobs require Files found on HDFS to be available to the mapper/reducer scripts. This is done using the `cacheFiles` tag. The value should be of the form `/full/path/of/file#name-referred-to-by-script`. Contents of a whole archive present in HDFS can be made available to scripts using the `cacheArchives` tag. The value should be of the form `/full/dfs/path/of/archive.jar#name-referred-to-by-script`.

The Mapper/Reducer can be overridden by a `mapred.mapper.class` or `mapred.reducer.class` properties in the `job-xml` or `configuration` elements.

Example:

```
<workflow-app name="sample-wf" xmlns="uri:hws-app:1.0">
  ...
  <action name="firstjob">
    <map-reduce>
      <prepare>
        <delete path="{wf:conf('output')}" />
      </prepare>
      <streaming>
        <mapper>/bin/bash testarchive/bin/mapper.sh testfile</mapper>
        <reducer>/bin/bash testarchive/bin/reducer.sh</reducer>
        <cacheFiles>
          <file>/users/blabla/testfile#testfile</file>
        </cacheFiles>
        <cacheArchives>
          <file>/users/blabla/testarchive.jar#testarchive</file>
        </cacheArchives>
      </streaming>
      <configuration>
        <property>
          <name>mapred.job.tracker</name>
          <value>${wf:conf('jobtracker')}</value>
        </property>
        <property>
          <name>fs.default.name</name>
          <value>${wf:conf('namenode')}</value>
        </property>
        <property>
          <name>mapred.input.dir</name>
          <value>${wf:conf('input')}</value>
        </property>
        <property>
          <name>mapred.output.dir</name>
          <value>${wf:conf('output')}</value>
        </property>
        <property>
          <name>stream.num.map.output.key.fields</name>
          <value>3</value>
        </property>
      </configuration>
    </map-reduce>

    <ok to="end" />
    <error to="fail" />
  </action>
  ...
</workflow-app>
```

3.2.3 Pig Action

The `pig` action starts a Pig job on a remote machine as a remote secure shell in background. The Pig runtime must be present in the remote machine and it must be available for execution via the command `PATH`.

The workflow job will wait until the `pig` job completes before continuing to the next action.

The `pig` action has to be configured with the host that will run the pig job, the pig script and the necessary parameters to run the Pig job.

A `pig` action can be configured to perform HDFS files/directories cleanup before starting the Pig job. This capability enables HWS to retry a Pig job in the situation of a transient failure (Pig creates temporary directories for intermediate data, thus a retry without cleanup would fail).

Inline property values can be parameterized (templated) using EL expressions.

Hadoop `mapred.job.tracker` and `fs.default.name` properties must always be present in the `pig` action configuration, inline or in the JobConf XML file. These properties indicate the Hadoop system that will be used to run the Pig program.

Syntax:

```
<workflow-app name="[WF-DEF-NAME]" xmlns="uri:hws-app:1.0">
  ...
  <action name="[NODE-NAME]">
    <pig>
      <prepare>
        <delete path="[PATH]" />
        ...
        <delete path="[PATH]" />
      </prepare>
      <pig-host>[USER]@[HOST-RUNNING-PIG-JOB]</pig-host>
      <pig-script>[PIG-SCRIPT]</pig-script>
      <parameters>
        <param>[PARAM-VALUE]</param>
        ...
        <param>[PARAM-VALUE]</param>
      </parameters>
      <job-xml>[JOB-XML-FILE]</job-xml>
      <configuration>
        <property>
          <name>[PROPERTY-NAME]</name>
          <value>[PROPERTY-VALUE]</value>
        </property>
        ...
      </configuration>
    </pig>

    <ok to="[NODE-NAME]" />
    <error to="[NODE-NAME]" />
  </action>
  ...
</workflow-app>
```

The `prepare` element, if present, indicates a list of path do delete before starting the job. This should be used exclusively for directory cleanup for the job to be executed. The delete operation will be performed in the `fs.default.name` filesystem.

The `pig-host` indicates the host (and user) executing the pig script.

The `pig-script` element contains the pig script to execute. The pig script can be templated with variables of the form `${VARIABLE}`. The values of these variables can then be specified using the `params` element.

NOTE: HWS will perform the parameter substitution before firing the pig job. This is different from the parameter substitution mechanism provided by Pig, which has a few limitations.

The `params` element, if present, contains parameters to be passed to the pig script.

The `job-xml` element, if present, must refer to a Hadoop JobConf `job.xml` file bundled in the workflow application. The `job-xml` element is optional and if present it can be only one.

The `configuration` element, if present, contains JobConf properties for the underlying Hadoop jobs.

Properties specified in the `configuration` element override properties specified in the file specified in the `job-xml` element.

The inline and `job-xml` configuration properties are passed to the Hadoop jobs submitted by Pig runtime.

All the above elements can be parameterized (templated) using EL expressions.

Example:

```
<workflow-app name="sample-wf" xmlns="uri:hws-app:1.0">
  ...
  <action name="myfirstpigjob">
    <pig>
      <prepare>
        <delete path="${wf:conf('job.output')}" />
      </prepare>
      <pig-host>tucu@foo.com</pig-host>
      <pig-script>/mypigscript.pig</pig-script>
      <map-reduce-user>tucu</hadoop-user>
      <parameters>
        <param>InputDir=/home/tucu/input-data</param>
        <param>OutputDir=${wf:conf('job.output')}</param>
      </parameters>
      <configuration>
        <property>
          <name>fs.default.name</name>
          <value>foo.com:9000</value>
        </property>
        <property>
          <name>mapred.compress.map.output</name>
          <value>true</value>
        </property>
      </configuration>
    </pig>

    <ok to="myotherjob" />
    <error to="errorcleanup" />
  </action>
  ...
</workflow-app>
```

3.2.4 Fs action

The `fs` action allows to manipulate files and directories in HDFS from a workflow application. The supported commands are `move`, `delete` and `mkdir`.

The FS commands are executed synchronously from within the FS action, the workflow job will wait until the specified file commands are completed before continuing to the next action.

Path names specified in the `fs` action can be parameterized (templated) using EL expressions.

The `hfs` action has to specify the file system URI where the operations will be performed as well as the list of file commands together with the source and target locations.

IMPORTANT: All the commands within `fs` action do not happen atomically, if a `fs` action fails half way in the commands being executed, successfully executed commands are not rolled back. The `fs` action, before executing any command must check that source paths exist and target paths don't exist, thus failing before executing any command. Therefore the validity of all paths specified in one `fs` action are evaluated before any of the file operation are executed. Thus there is less chance of an error occurring while the `fs` action executes.

Syntax:

```
<workflow-app name="[WF-DEF-NAME]" xmlns="uri:hws-app:1.0">
  ...
  <action name="[NODE-NAME]">
    <fs>
      <file-system>[FS-URI]</file-system>
      <user-name>[FS-USER]</user-name>
      <commands>
        <delete path=' [PATH] '/>
        <mkdir path=' [PATH] '/>
        <move source=' [SOURCE-PATH] ' target=' [TARGET-PATH] '/>
        ...
      </commands>
    </fs>

    <ok to="[NODE-NAME]"/>
    <error to="[NODE-NAME]"/>
  </action>
  ...
</workflow-app>
```

The `file-system` element indicates the URI of a FS file system where to perform the file commands.

The `user-name` element indicates the file system user ID to use for the file commands.

The `commands` element, contains all the file commands to perform as part of the `fs` action.

The `delete` command deletes the specified path, if it is a directory it deletes recursively all its content and then deletes the directory.

The `mkdir` command creates the specified directory, it creates all missing directories in the path. If the directory already exist it does a no-op.

In the `move` command the `source` path must exist and the `target` path must not exist. The parent directory of the `target` path must exist.

If relative paths are used it will be relative to the specified user home directory.

Example:

```
<workflow-app name="sample-wf" xmlns="uri:hws-app:1.0">
  ...
  <action name="hdfscommands">
    <fs>
      <file-system>hdfs://foo.corp:9000</file-system>
      <user-name>tucu</user-name>
      <commands>
        <delete path='/home/tucu/temp-data' />
        <mkdir path='archives/${wf:id()}' />
        <move source='${wf:conf("job.input")}' target='archives/${wf:id()}/processed-input' />
      </commands>
    </fs>

    <ok to="myotherjob"/>
    <error to="errorcleanup"/>
  </action>
  ...
</workflow-app>
```

In the above example, a directory named after the workflow job ID is created and the input of the job, passed

as workflow configuration parameter, is archived under the previously created directory.

3.2.5 Ssh Action

The `ssh` action starts a shell command on a remote machine as a remote secure shell in background. The workflow job will wait until the remote shell command completes before continuing to the next action.

The shell command must be present in the remote machine and it must be available for execution via the command path. The shell command cannot make any assumption regarding the current working directory.

The output (STDOUT) of the `ssh` job can be made available to the workflow job after the `ssh` job ends. This information could be used from within decision nodes. If the output of the `ssh` job is made available to the workflow job the shell command must follow the following requirements:

- The format of the output must be a valid Java Properties file.
- The size of the output must not exceed 2KB.

Syntax:

```
<workflow-app name="[WF-DEF-NAME]" xmlns="uri:hws-app:1.0">
  ...
  <action name="[NODE-NAME]">
    <ssh>
      <host>[USER]@[HOST]</host>
      <command>[SHELL]</command>
      <args>[ARGUMENTS]</args>
      ...
      <capture-output/>
    </ssh>

    <ok to="[NODE-NAME]"/>
    <error to="[NODE-NAME]"/>
  </action>
  ...
</workflow-app>
```

The `host` indicates the user and host where the shell will be executed.

The `command` element indicates the shell command to execute.

The `args` element, if present, contains parameters to be passed to the shell command. If more than one `args` element is present they are concatenated in order.

If the `capture-output` element is present, it indicates HWS to capture output of the STDOUT of the `ssh` command execution. The `ssh` command output must be in Java Properties file format and it must not exceed 2KB. From within the workflow definition, the output of an `ssh` action node is accessible via the `String action:output(String node, String key)` function (Refer to section '4.2.6 Action EL Functions').

The configuration of the `ssh` action can be parameterized (templated) using EL expressions.

Example:

```
<workflow-app name="sample-wf" xmlns="uri:hws-app:1.0">
  ...
  <action name="myssjob">
    </ssh>
    <host>foo@bar.com</host>
    <command>uploaddata</command>
```

```

        <args>jdbc:derby://bar.com:1527/myDB</args>
        <args>hdfs://foobar.com:9000/usr/tucu/myData</args>
    </ssh>

    <ok to="myotherjob"/>
    <error to="errorcleanup"/>
</action>
...
</workflow-app>

```

In the above example, the `uploaddata` shell command is executed with two arguments, `jdbc:derby://foo.com:1527/myDB` and `hdfs://foobar.com:9000/usr/tucu/myData`.

The `uploaddata` shell must be available in the remote host and available in the command path.

The output of the command will be ignored because the `capture-output` element is not present.

3.2.6 Sub-workflow Action

The `sub-workflow` action runs a child workflow job, the child workflow job can be in the same HWS system or in another HWS system.

The parent workflow job will wait until the child workflow job has completed.

Syntax:

```

<workflow-app name="[WF-DEF-NAME]" xmlns="uri:hws-app:1.0">
    ...
    <action name="[NODE-NAME]">
        <sub-workflow>
            <host-url>http://[HOST]:[PORT]/[HWS-PATH]</host-url>
            <workflow-app>[WF-DEFINITION-NAME]</workflow-app>
            <configuration>
                <property>
                    <name>[PROPERTY-NAME]</name>
                    <value>[PROPERTY-VALUE]</value>
                </property>
                ...
            </configuration>
        </sub-workflow>

        <ok to="[NODE-NAME]"/>
        <error to="[NODE-NAME]"/>
    </action>
    ...
</workflow-app>

```

The `host-url` element can be used to specify the URL of the target HWS system. If this tag is absent, it is assumed that the child workflow job runs in the same HWS system instance where the parent workflow job is running.

The `workflow-app` element specifies the name of the workflow application for the child workflow job.

The `configuration` section can be used to specify the job properties that are required to run the child workflow job.

The configuration of the `sub-workflow` action can be parameterized (templated) using EL expressions.

Example:

```

<workflow-app name="sample-wf" xmlns="uri:hws-app:1.0">
  ...
  <action name="a">
    <sub-workflow>
      <host-url>http://myhost:8080/oozie</host-url>
      <workflow-app>child-wf</workflow-app>
      <configuration>
        <property>
          <name>input.dir</name>
          <value>${wf:id()}/second-mr-output</value>
        </property>
      </configuration>
    </sub-workflow>

    <ok to="end"/>
    <error to="fail"/>
  </action>
  ...
</workflow-app>

```

In the above example, the workflow definition with the name `child-wf` will be run on the HWS instance at `http://myhost:8080/oozie`. The specified workflow application must be already deployed on the target HWS instance.

A configuration parameter `input.dir` is being passed as job property to the child workflow job.

3.2.7 Http Action

The `http` action makes an HTTP GET or POST call from within a workflow job.

The `http` action can be configured to make the workflow job to wait on a callback notification or to continue immediately to the next node in the workflow job execution path.

The output of the `http` action can be made available to the workflow job after the `http` action ends. This information could be used from within decision nodes. If the output of the `http` action is made available to the workflow job, it must follow the following requirements:

- The format of the output must be a valid Java Properties file.
- The size of the output must not exceed 2KB.

Syntax:

```

<workflow-app name="[WF-DEF-NAME]" xmlns="uri:hws-app:1.0">
  ...
  <action name="[NODE-NAME]">
    <http>
      <url>[EMAIL-ADDRESS, ...]</url>
      <post content-type="[CONTENT-TYPE]">
        [PAYLOAD]
      </post>
      <mode>CONTINUE|WAIT</mode>
      <capture-output/>
    </http>

    <ok to="[NODE-NAME]"/>
    <error to="[NODE-NAME]"/>
  </action>
  ...
</workflow-app>

```

The `url` element contains the URL to be used for the HTTP call.

If the `post` element is not present, a HTTP GET call will be done to the specified URL.

The `post` element is present, a HTTP POST call will be done to the specified URL using the content type and payload specified in the `post` element.

The `mode` element indicates the behavior of the `http` action.

If the mode is `CONTINUE`, the workflow job will transition to the next node in the execution path. If the `capture-output` element is present, it indicates HWS to capture the payload of the HTTP response as the action output.

If the mode is `WAIT`, the workflow job will wait for a callback before completing the `http` action. The `http` action must communicate the target HTTP system of the callback URL, this must be done using the `wf:callbackUrl(String stateVar)` function. If the `capture-output` element is present, it indicates HWS to capture HTTP callback request payload.

If the `capture-output` element is present, the HTTP call response or callback request payload (depending on the mode being `CONTINUE` or `WAIT`) will be stored in the workflow job. The payload must be in Java Properties file format, the content type must be `'x-application/java-properties'` and it must not exceed 2KB. From within the workflow definition, this payload is accessible via the `String action:output(String node, String key)` function (Refer to section '4.2.6 Action EL Functions').

The `url` and `post` and `mode` elements can be parameterized (templated) using EL expressions.

Example:

```
<workflow-app name="sample-wf" xmlns="uri:hws-app:1.0">
  ...
  <action name="httpNotification">
    <http>
      <url>http://foo.com/notify?callback=${urlencode(wf:callback('%EXITSTATE%'))}</url>
      <mode>WAIT</mode>
    </http>

    <ok to="onOK"/>
    <error to="onError"/>
  </action>
  ...
</workflow-app>
```

In the above example, an HTTP GET call it will done to the `http://foo.com/notify` URL and the query string of the URL will contain the callback URL for the external system to signal HWS that the action has completed. The external system must replace the `%EXITSTATE%` token in the callback URL with `OK` or `ERROR` before making the callback. Because the mode is set to `WAIT`, HWS will wait for the callback before completing the `Http` action.

3.2.8 Email Node

The `email` action sends an email from within a workflow job.

Syntax:

```
<workflow-app name="[WF-DEF-NAME]" xmlns="uri:hws-app:1.0">
  ...
  <action name="[NODE-NAME]">
    <email>
      <to>[EMAIL-ADDRESS, ...]</to>
```

```

        <subject>[SUBJECT]</subject>
        <message>[MESSAGE]</message>
    </email>

    <ok to="[NODE-NAME]"/>
    <error to="[NODE-NAME]"/>
</action>
...
</workflow-app>

```

The `to` element contains the emails addresses to send the message, they must be separated by commas.

The `subject` element contains the subject for the message.

the `message` element contains the body of the message.

The `to`, `subject` and `message` elements can be parameterized (templated) using EL expressions.

Example:

```

<workflow-app name="sample-wf" xmlns="uri:hws-app:1.0">
    ...
    <action name="notify">
        <email>
            <to>foo@bar.com, {$wf:conf('admin.email')}</to>
            <subject>Workflow {$wf:id()} finished second phase on {$timestamp()}</subject>
            <message/>
        </email>

        <ok to="next"/>
        <error to="next"/>
    </action>
    ...
</workflow-app>

```

4 Parameterization of Workflows

Workflow definitions can be parameterized.

When workflow node is executed by HWS all the ELs are resolved into concrete values.

The parameterization of workflow definitions it done using JSP Expression Language syntax from the JSP 2.0 Specification (JSP.2.3), allowing not only to support variables as parameters but also functions and complex expressions.

EL expressions can be used in the configuration values of action and decision nodes. They can be used in XML attribute values and in XML element and attribute values.

They cannot be used in XML element and attribute names. They cannot be used in the name of a node and they cannot be used within the `transition` elements of a node.

4.1 Workflow Job Properties

When a workflow job is submitted to HWS, the submitter may specify as many workflow job properties as required (similar to Hadoop JobConf properties).

Workflow application may define default values for the workflow job parameters. They must be defined in a `configuration.xml` file bundled with the workflow application archive (refer to section '7 Workflow

Applications Packaging'). Workflow job properties have precedence over the default values.

From within the workflow definition all these properties are available via the `String wf:conf(String name)` function. This use of a function to expose the workflow job properties is required to avoid the JSP EL requirement that variables must be valid Java identifiers (for example: a property named `mapper.class` is not a valid Java identifier).

Example:

Parameterized Workflow definition:

```
<workflow-app name='hello-wf' xmlns="uri:hws-app:1.0">
  ...
  <action name='firstjob'>
    <map-reduce>
      <configuration>
        <property>
          <name>mapred.job.tracker</name>
          <property>${wf:conf('hadoop.job.tracker')}</property>
        </property>
        <property>
          <name>mapred.mapper.class</name>
          <property>com.foo.FirstMapper</property>
        </property>
        <property>
          <name>mapred.reducer.class</name>
          <property>com.foo.FirstReducer</property>
        </property>
        <property>
          <name>mapred.input.dir</name>
          <property>${wf:conf('wfjob.input.dir')}</property>
        </property>
        <property>
          <name>mapred.output.dir</name>
          <property>${wf:conf('wfjob.output.dir')}</property>
        </property>
      </configuration>
    </map-reduce>

    <ok to='secondjob' />
    <error to='failedcleanup' />
  </action>
  ...
</workflow-app>
```

When submitting a workflow job for the workflow definition above, 3 workflow job properties must be specified:

- `hadoop.job.tracker:`
- `wfjob.input.dir:`
- `wfjob.output.dir:`

4.2 Expression Language Functions

HWS, besides allowing the use of workflow job properties to parameterize workflow jobs, it provides a set of build in EL functions that enable a more complex parameterization of workflow action nodes as well as the predicates in decision nodes.

4.2.1 EL Constants

- **KB**: 1024, one kilobyte.
- **MB**: 1024 * KB, one megabyte.
- **GB**: 1024 * MB, one gigabyte.
- **TB**: 1024 * GB, one terabyte.
- **PB**: 1024 * TG, one petabyte.

All the above constants are of type long.

4.2.2 Basic EL Functions

String firstNotNull(String value1, String value2)

It returns the first not null value, or null if both are null.

Note that if the output of this function is null and it is used as string, the EL library converts it to an empty string. This is the common behavior when using `firstNotNull()` in node configuration sections.

String concat(String s1, String s2)

It returns the concatenation of 2 strings. A string with null value is considered as an empty string.

String concat(String s1, String s2, String s3)

It returns the concatenation of 3 strings. A string with null value is considered as an empty string.

String trim(String s)

It returns the trimmed value of the given string. A string with null value is considered as an empty string.

String urlEncode(String s)

It returns the URL UTF-8 encoded value of the given string. A string with null value is considered as an empty string.

String timestamp()

It returns the UTC current date and time in W3C format down to the second (YYYY-MM-DDThh:mm:ss.sZ). I.e.: 1997-07-16T19:20:30.45Z

4.2.3 Workflow EL Functions

String wf:appName()

It returns the workflow application name for the current workflow job.

String wf:id()

It returns the workflow job ID for the current workflow job.

String wf:conf(String name)

It returns the value of the workflow job configuration property for the current workflow job, or null if

undefined.

String wf:callback(String stateVar)

It returns the callback URL for the current workflow action node, `stateVar` can be a valid exit state (OK or ERROR) for the action or a token to be replaced with the exit state by the remote system executing the task.

String wf:exitState(String node)

It returns the exit state for the specified workflow action node, or `null` if the action has not being executed yet or it has not completed yet.

String wf:errorNode()

It returns the name of the last workflow action node that exit with an ERROR exit state, or `null` if no action has exited with ERROR state in the current workflow job.

String wf:errorCode(String node)

It returns the error code for the specified action node, or `null` if the action node has not exited with ERROR state.

Each type of action node must define its complete error code list.

String wf:errorMessage(String message)

It returns the error message for the specified action node, or `null` if no action node has not exited with ERROR state.

The error message can be useful for debugging and notification purposes.

int wf:recovery()

It returns 0 if the workflow job is not in recovery mode, otherwise it will return the recovery run attempt.

4.2.4 Hadoop EL Functions

String hadoop:jobId(String node)

It returns the `jobId` for a job submitted by a Hadoop action node, or `null` if the action has not started yet.

String hadoop:jobTracker(String node)

It returns the Hadoop job tracker for a job submitted by a Hadoop action node, or `null` if the action has not started yet.

String hadoop:jobStatus(String node)

It returns the Hadoop job status for a job submitted by a Hadoop action node, or `null` if the action has not started yet.

Valid values are: PREP, RUNNING, SUCCEEDED, FAILED, KILLED or UNKNOWN.

Map < String, Map < String, Long > > hadoop:jobCounters(String node)

It returns the counters for a job submitted by a Hadoop action node. It returns `null` if the Hadoop job has not started yet.

The outer Map key is a counter group name. The inner Map value is a Map of counter names and counter values.

4.2.5 HDFS EL Functions

For all the functions in this section:

- The `fsUri` parameter must be of the form `hdfs://HOST:PORT`
- The path must be absolute paths

boolean hdfs:exists(String fsUri, String path)

It returns `true` or `false` depending if the specified path URI exists or not.

boolean hdfs:isDir(String fsUri, String path)

It returns `true` if the specified path URI exists and it is a directory, otherwise it returns `false`.

boolean hdfs:dirSize(String fsUri, String path)

It returns the size in bytes of all the files in the specified path. If the path is not a directory, or if it does not exist it returns `-1`. It does not work recursively, only computes the size of the files under the specified path.

boolean hdfs:fileSize(String fsUri, String path)

It returns the size in bytes of specified file. If the path is not a file, or if it does not exist it returns `-1`.

boolean hdfs:blockSize(String fsUri, String path)

It returns the block size in bytes of specified file. If the path is not a file, or if it does not exist it returns `-1`.

4.2.6 Action EL Functions

String action:output(String node, String key)

This function is only applicable to Http and Ssh action nodes.

It returns the value for a key from the captured output of the action node. The output will be stored by HWS only if the node has been configured with the `capture-output` element. If the action has not been executed yet, the action has not been set to capture its output or the key is not defined in the captured output the function returns `null`.

5 HWS Notifications

Workflow jobs can be configured to make an HTTP GET notification upon start and end of a workflow action node and upon the completion of a workflow job.

HWS will make a best effort to deliver the notifications, in case of failure it will retry the notification a pre-configured number of times at a pre-configured interval before giving up.

5.1 Action Start and End Notifications

If the `hws.action.notification.url` property is present in the workflow job properties when submitting the job, HWS will make a notification to the provided URL every time the workflow job enters and exits an action node. For decision nodes, HWS will send a single notification with the name of the first evaluation that resolved to `true`.

If the URL contains any of the following tokens, they will be replaced with the actual values by HWS before making the notification:

- `$wfId` : The workflow job ID
- `$nodeName` : The name of the workflow node
- `$nodeStatus` : For action nodes the actual value will be `started`, `automatic-retry`, `manual-retry`, `OK` or `ERROR`. For decision nodes is the name of the evaluation that resolved to `true`.

5.2 Workflow Job End Notification

If the `hws.workflow.notification.url` property is present in the workflow job properties when submitting the job, HWS will make a notification to the provided URL when the workflow job ends.

If the URL contains any of the following tokens, they will be replaced with the actual values by HWS before making the notification:

- `$wfId` : The workflow job ID
- `$wfstatus` : the workflow end state: `SUCCEEDED`, `FAILED` or `KILLED`

6 User Propagation

The user name that submits a job must be present in the workflow job configuration in the `user.name` property.

Unless specified otherwise, HWS uses the same user name to submit Map-Reduce, Fs and Pig jobs. For cases where the HWS user name does not match the Hadoop user name, the workflow application can override and force another user name.

For other action types, HWS does not propagate the user name automatically, it has to be explicitly done in the action node configuration (i.e. in an `ssh` node). The reason behind this is that there may not be direct correlation between HWS users and other systems users. For example, a HWS user may not be a valid Unix user to start `ssh` shells via a secure remote shell in another machine.

6.1 User Propagation for Map-Reduce Jobs

To set the user to use for a Map Reduce job, the `user.name` property must be set in the configuration of the `map-reduce` action.

The `user.name` property can be set either in the `job.xml` file specified in the `job-xml` element or inline in the `map-reduce` action.

For example, to use the user `tucu`:

```
<workflow-app name="sample-wf" xmlns="uri:hws-app:1.0">
  ...
  <action name="myfirstmrjob">
```

```

    <map-reduce>
      <configuration>
        <property>
          <name>user.name</name>
          <value>tucu</value>
        </property>
        ...
      </configuration>
    </map-reduce>

    <ok to="myotherjob"/>
    <error to="errorcleanup"/>
  </action>
  ...
</workflow-app>

```

6.2 User Propagation for Pig Jobs

There are 2 users involved in running a Pig job. The Unix user of the remote system starting the Pig runtime and the Hadoop user running the compiled Pig script in the Hadoop cluster.

Setting the Unix user to start the Pig runtime is identical to a `ssh` action node, the user must be specified in the `host` element of the Pig node using the standard ssh notation `USER@HOST`.

To set the user to use for the Hadoop jobs started by Pig runtime, the `user.name` property must be set in the configuration of the Pig node.

For example, to use the Unix user `tucu` for the Pig runtime and the user `pepe` for Hadoop:

```

<workflow-app name="sample-wf" xmlns="uri:hws-app:1.0">
  ...
  <action name="myfirstpigjob">
    <pig>
      <pig-host>tucu@foo.com</pig-host>
      <pig-script>/mypigscript.pig</pig-script>
      <parameters>
        <param>InputDir=/home/tucu/input-data</param>
        <param>OutputDir=${wf:conf('job.output')}</param>
      </parameters>
      <configuration>
        <property>
          <name>user.name</name>
          <value>pepe</value>
        </property>
      </configuration>
    </pig>

    <ok to="myotherjob"/>
    <error to="errorcleanup"/>
  </action>
  ...
</workflow-app>

```

6.3 User Propagation for FS Operations

To set the user to use for FS file system operations, the `user` element must be set in the `hdfs` node.

For example, to use the user `tucu`:

```

<workflow-app name="sample-wf" xmlns="uri:hws-app:1.0">
  ...
  <fs name="hdfscommands">

```

```

    <fs>
      <file-system>hdfs://foo.corp:9000</file-system>
      <user-name>tucu</user-name>
      <commands>
        <delete path='/home/tucu/temp-data' />
        <mkdir path='archives/${wf:id()}' />
        <move source='${wf:conf("job.input")}' target='archives/${wf:id()}/processed-input' />
      </commands>
    </fs>

    <ok to="myotherjob"/>
    <error to="errorcleanup"/>
  </action>
  ...
</workflow-app>

```

6.4 User Propagation for Ssh Jobs

The user to use for a ssh shell must be specified in the `host` element of the `ssh` action using the standard ssh notation `USER@HOST`.

For example, to use the user `tucu`:

```

<workflow-app name="sample-wf" xmlns="uri:hws-app:1.0">
  ...
  <action name="mysshjob">
    <ssh>
      <host>tucu@foo.com</host>
      <command>uploaddata</command>
      <args>jdbc:derby://bar.com:1527/myDB</args>
      <args>hdfs://foobar.com:9000/usr/tucu/myData</args>
    </ssh>

    <ok to="myotherjob"/>
    <error to="errorcleanup"/>
  </action>
  ...
</workflow-app>

```

7 Workflow Applications Packaging

Workflow application are self-contained contained application (identical to the J2EE application model).

A workflow application is packaged as a ZIP archive file containing the workflow definition, `workflow.xml`, all configuration files and scripts (Pig and shell) needed by the workflow action nodes, all the necessary JAR files and native libraries needed to execute all the hadoop and pig actions specified in the workflow definition.

The ZIP file must have the following layout:

```

- /workflow.xml
- /configuration.xml
|
- /lib/ (*.jar;*.so)

```

The `workflow.xml` file is the workflow definition in hPDL.

The `configuration.xml` file defines, if any, default values for the workflow job parameters. This file must be use the Hadoop Configuration XML format.

The `lib/` direct must also contain all the necessary JAR files and SO files (native library).

Any other resources like `job.xml` files referenced from a workflow action action node must be included under the corresponding path in the ZIP file. Absolute and Relative paths always start from the root of the ZIP file.

8 Workflow Applications Lifecycle

To create workflow jobs for a given workflow application, first the workflow application package (the ZIP file described in section '7 Workflow Applications Packaging') must be deployed in HWS.

When a workflow application is deployed in HWS, the workflow application state is `INACTIVE`.

Before submitting any workflow job for a workflow application its state must be changed to `ACTIVE`.

An `ACTIVE` workflow application can be changed to `SUSPENDED` state. For suspended workflow applications, existing workflow jobs continue to run normally until their completion but HWS does not allow the creation of any new workflow job for suspended workflow application.

A `SUSPENDED` workflow application can be changed to `INACTIVE` state only if all the running workflow jobs have completed. A workflow application in `SUSPENDED` state can be changed back to `ACTIVE` state.

An `INACTIVE` workflow application can be undeployed.

It is not possible to deploy a new workflow application with the same name of an existing workflow application. First the existing workflow application must be undeployed.

HWS does not provide any mechanism for doing workflow applications versioning. It is left to HWS administrator to put the necessary deployment processes in place.

Workflow applications state valid transitions:

- `--> INACTIVE`
- `INACTIVE --> ACTIVE | undeployed`
- `ACTIVE --> SUSPENDED`
- `SUSPENDED --> ACTIVE | INACTIVE`

9 Self Containment Model

Section "7 Workflow Applications Packaging" defines the packaging of a workflow application. A direct consequence of this packaging model is application self-containment is made easy for workflow developers while application assembly at runtime is made difficult.

The self containment model works well with Hadoop and Pig computation/processing jobs, which are the primary type of tasks for HWS. All resources required to run Hadoop and Pig jobs must be bundled with the workflow application and they will be made available to the Hadoop tasks via the Hadoop Distributed Cache.

For SSH and HTTP actions, the workflow application developer is responsible for deploying all necessary components in the target boxes. Using `ssh` and `http` action for computation/processing tasks is highly discouraged. They should only be used to perform very specialized tasks, like for example HOD provisioning.

10 External Data Assumptions

HWS runs workflow jobs under the assumption all necessary data to execute an action is readily available at the time the workflow job is about to executed the action.

In addition, it is assumed, but it is not the responsibility of HWS, that all input data used by a workflow job is immutable for the duration of the workflow job.

11 Workflow Jobs Lifecycle

A workflow job can have be in any of the following states:

PREP: When a workflow job is first create it will be in PREP state. The workflow job is defined but it is not running.

RUNNING: When a CREATED workflow job is started it goes into RUNNING state, it will remain in RUNNING state while it does not reach its end state, ends in error or it is suspended.

SUSPENDED: A RUNNING workflow job can be suspended, it will remain in SUSPENDED state until the workflow job is resumed or it is killed.

SUCCEDED: When a RUNNING workflow job reaches the end node it ends reaching the SUCCEDED final state.

KILLED: When a CREATED, RUNNING or SUSPENDED workflow job is killed by an administrator or the owner via a request to HWS the workflow job ends reaching the KILLED final state.

FAILED: When a RUNNING workflow job fails due to an unexpected error it ends reaching the FAILED final state.

Workflow job state valid transitions:

- --> PREP
- PREP --> RUNNING | KILLED
- RUNNING --> SUSPENDED | SUCCEDED | KILLED | FAILED
- SUSPENDED --> RUNNING | KILLED

12 Workflow Jobs Recovery

HWS must provide a mechanism by which a a failed workflow job can be resubmitted and executed starting after any action node that has completed its execution in the prior run. This is specially useful when the already executed action of a workflow job are too expensive to be re-executed.

It is the responsibility of the user resubmitting the workflow job to do any necessary cleanup to ensure that the rerun will not fail due to not cleaned up data from the previous run.

When starting a workflow job in recovery mode, the user must indicate what workflow nodes in the workflow should be skipped. All workflow nodes indicated as skipped must have completed in the previous run. If a workflow node has not completed its execution in its previous run, and during the recovery submission is flagged as a node to be skipped, the recovery submission must fail.

The recovery workflow job will run under the same workflow job ID as the original workflow job.

To submit a recovery workflow job the target workflow job to recover must be in an end state (SUCCEEDED, FAILED or KILLED).

A recovery run could be done using a new version of the workflow application under certain constraints (see next paragraph). This is to allow users to patch and redeploy the workflow application before doing the recovery.

If the workflow application is patched, the patched workflow application used for the recovery must match the execution flow, node types, node names and node configuration for all executed nodes that will be skipped during recovery. This cannot be checked by HWS, it is the responsibility of the user to ensure this is the case.

HWS does not attempt workflow job recovery automatically, this has to be done explicitly by a user.

HWS provides the `int wf:recovery()` function to allow workflow applications to perform custom logic at workflow definition level (i.e. in a `decision` node) or at action node level (i.e. by passing the value of the recovery function as a parameter to the task). The `wf:recovery()` function value is 0 if the workflow job is not in recovery mode, otherwise it will return the recovery run attempt (as it may be possible that multiple recoveries are attempted on the same workflow job).

13 Workflow Applications API

The Workflow Application API is a JSON HTTP REST based API.

The following operations are supported:

- Workflow Applications
 - ◆ Deploy: deploys a workflow application as inactive
 - ◆ Undeploy: undeploys a workflow application
 - ◆ Activate: activates an inactive or suspended workflow application
 - ◆ Deactivate: deactivates a suspended workflow application
 - ◆ Suspend: suspends a active workflow application
 - ◆ Details: returns the details of a workflow application (name, definition, status)
 - ◆ List: lists all deployed workflow applications (provides filters and pagination)

- Workflow Jobs
 - ◆ Submit: submits a workflow job for an active workflow application
 - ◆ Start: starts a submitted workflow job
 - ◆ Kill: kills a submitted/running/suspended workflow job
 - ◆ Suspend: suspends a running workflow job
 - ◆ Resume: resumes a suspended workflow job
 - ◆ Details: returns the details of a workflow job (id, user, start, actions state, etc)
 - ◆ List: list all workflow jobs, running and completed (provides filters and pagination)
 - ◆ ActionRetry: retries a failed action that requires manual retry
 - ◆ JobResubmit: resubmits a workflow job using a existing workflow job from a previously executed state

14 Workflow Administration API

The Workflow Administration API is a JSON HTTP REST based API.

The following operations are supported:

- Activate: activates HWS

- Deactivate: deactivates HWS (application API stops working)
- Status: returns HWS status (active/inactive)
- Instrumentation: returns the current instrumentation values of HWS
- OSEnvironment: returns the shell environment use to start HWS
- JavaSystemProperties: returns the Java System Properties of HWS
- DBPurge: purges HWS DB (cleans up data from completed workflow jobs older than a specified threshold)

15 Customizing HWS with Extensions

Out of the box HWS provides support for a predefined set of action node types and Expression Language functions.

HWS will provide a well defined API to add support for additional action node types and Expression Language functions.

Extending HWS to support additional action node types and expression language function it will require using an extension API to be provided by an HWS SDK. It will not require any code change on the HWS codebase. It will require adding the JAR files providing the new functionality and declaring them in HWS configuration.

16 Workflow Jobs Priority

HWS does not handle workflow jobs priority. As soon as a workflow job is ready to do a transition, HWS will trigger the transition. Workflow transitions and action triggering are assumed to be fast and lightweight operations.

HWS assumes that the remote systems are properly sized to handle the amount of remote jobs HWS will trigger.

Any prioritization of jobs in the remote systems is outside of the scope of HWS.

Workflow applications can influence the remote systems priority via configuration if the remote systems support it.

17 Client API

HWS will provide a (Java) Client API that allows to perform all Workflow Application API and Workflow Administration API operations.

The Client API will be implemented as a client of the Web Services API.

18 Command Line Tools

HWS will provide command line tools that allow to perform all Workflow Application API and Workflow Administration API operations.

The command line tools will be implemented as a client of the Web Services API.

19 Web UI Console

HWS will provide a read-only Web based console that allows to allow to monitor HWS system status, workflow applications status and workflow jobs status.

The Web base console will be implemented as a client of the Web Services API.

Appendixes

Appendix A, HWS XML–Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:hws="uri:hws:1"
  elementFormDefault="qualified" targetNamespace="uri:hws:1">

  <xs:element name="workflow-app" type="hws:WORKFLOW-APP"/>

  <xs:simpleType name="IDENTIFIER">
    <xs:restriction base="xs:string">
      <xs:pattern value="([a-zA-Z]([_\a-zA-Z0-9])*){1,19}"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="WORKFLOW-APP">
    <xs:sequence>
      <xs:element name="start" type="hws:START" minOccurs="1" maxOccurs="1"/>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="decision" type="hws:DECISION" minOccurs="1" maxOccurs="1"/>
        <xs:element name="fork" type="hws:FORK" minOccurs="1" maxOccurs="1"/>
        <xs:element name="join" type="hws:JOIN" minOccurs="1" maxOccurs="1"/>
        <xs:element name="kill" type="hws:KILL" minOccurs="1" maxOccurs="1"/>
        <xs:element name="action" type="hws:ACTION" minOccurs="1" maxOccurs="1"/>
      </xs:choice>
      <xs:element name="end" type="hws:END" minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
    <xs:attribute name="name" type="hws:IDENTIFIER" use="required"/>
  </xs:complexType>

  <xs:complexType name="START">
    <xs:attribute name="to" type="hws:IDENTIFIER" use="required"/>
  </xs:complexType>

  <xs:complexType name="END">
    <xs:attribute name="name" type="hws:IDENTIFIER" use="required"/>
  </xs:complexType>

  <xs:complexType name="DECISION">
    <xs:sequence>
      <xs:element name="switch" type="hws:SWITCH" minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
    <xs:attribute name="name" type="hws:IDENTIFIER" use="required"/>
  </xs:complexType>

  <xs:element name="switch" type="hws:SWITCH"/>

  <xs:complexType name="SWITCH">
    <xs:sequence>
      <xs:sequence>
        <xs:element name="case" type="hws:CASE" minOccurs="1" maxOccurs="unbounded"/>
        <xs:element name="default" type="hws:DEFAULT" minOccurs="1" maxOccurs="1"/>
      </xs:sequence>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="CASE">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="to" type="hws:IDENTIFIER" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:schema>
```

```

</xs:complexType>

<xs:complexType name="DEFAULT">
  <xs:attribute name="to" type="hws:IDENTIFIER" use="required"/>
</xs:complexType>

<xs:complexType name="FORK_TRANSITION">
  <xs:attribute name="start" type="hws:IDENTIFIER" use="required"/>
</xs:complexType>

<xs:complexType name="FORK">
  <xs:sequence>
    <xs:element name="path" type="hws:FORK_TRANSITION" minOccurs="2" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="name" type="hws:IDENTIFIER" use="required"/>
</xs:complexType>

<xs:complexType name="JOIN">
  <xs:attribute name="name" type="hws:IDENTIFIER" use="required"/>
  <xs:attribute name="to" type="hws:IDENTIFIER" use="required"/>
</xs:complexType>

<xs:element name="kill" type="hws:KILL"/>

<xs:complexType name="KILL">
  <xs:sequence>
    <xs:element name="message" type="xs:string" minOccurs="1" maxOccurs="1"/>
  </xs:sequence>
  <xs:attribute name="name" type="hws:IDENTIFIER" use="required"/>
</xs:complexType>

<xs:complexType name="ACTION_TRANSITION">
  <xs:attribute name="to" type="hws:IDENTIFIER" use="required"/>
</xs:complexType>

<xs:complexType name="ACTION">
  <xs:sequence>
    <xs:choice minOccurs="1" maxOccurs="1">
      <xs:element name="map-reduce" type="hws:MAP-REDUCE" minOccurs="1" maxOccurs="1"/>
      <xs:element name="pig" type="hws:PIG" minOccurs="1" maxOccurs="1"/>
      <xs:element name="ssh" type="hws:SSH" minOccurs="1" maxOccurs="1"/>
      <xs:element name="sub-workflow" type="hws:SUB-WORKFLOW" minOccurs="1" maxOccurs="1"/>
      <xs:element name="fs" type="hws:FS" minOccurs="1" maxOccurs="1"/>
      <xs:element name="http" type="hws:HTTP" minOccurs="1" maxOccurs="1"/>
      <xs:element name="email" type="hws:EMAIL" minOccurs="1" maxOccurs="1"/>
      <xs:any namespace="##other" minOccurs="1" maxOccurs="1"/>
    </xs:choice>
    <xs:element name="ok" type="hws:ACTION_TRANSITION" minOccurs="1" maxOccurs="1"/>
    <xs:element name="error" type="hws:ACTION_TRANSITION" minOccurs="1" maxOccurs="1"/>
  </xs:sequence>
  <xs:attribute name="name" type="hws:IDENTIFIER" use="required"/>
</xs:complexType>

<xs:complexType name="MAP-REDUCE">
  <xs:sequence>
    <xs:element name="job-xml" type="xs:string" minOccurs="0" maxOccurs="1"/>
    <xs:element name="configuration" type="hws:CONFIGURATION" minOccurs="0" maxOccurs="1"/>
    <xs:element name="streaming" type="hws:STREAMING" minOccurs="0" maxOccurs="1"/>
    <xs:element name="prepare" type="hws:PREPARE" minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="PIG">
  <xs:sequence>
    <xs:element name="pig-host" type="xs:string" minOccurs="1" maxOccurs="1"/>
    <xs:element name="pig-script" type="xs:string" minOccurs="1" maxOccurs="1"/>
    <xs:element name="param" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

```

```

        <xs:element name="job-xml" type="xs:string" minOccurs="0" maxOccurs="1"/>
        <xs:element name="configuration" type="hws:CONFIGURATION" minOccurs="0" maxOccurs="1"/>
        <xs:element name="prepare" type="hws:PREPARE" minOccurs="0" maxOccurs="1"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="SSH">
    <xs:sequence>
        <xs:element name="ssh-host" type="xs:string" minOccurs="1" maxOccurs="1"/>
        <xs:element name="command" type="xs:string" minOccurs="1" maxOccurs="1"/>
        <xs:element name="args" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="capture-output" type="hws:FLAG" minOccurs="0" maxOccurs="1"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="SUB-WORKFLOW">
    <xs:sequence>
        <xs:element name="hws-uri" type="xs:string" minOccurs="1" maxOccurs="1"/>
        <xs:element name="workflow-app" type="hws:IDENTIFIER" minOccurs="1" maxOccurs="1"/>
        <xs:element name="configuration" type="hws:CONFIGURATION" minOccurs="0" maxOccurs="1"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="FS">
    <xs:sequence>
        <xs:element name="fs-uri" type="xs:string" minOccurs="1" maxOccurs="1"/>
        <xs:element name="user-name" type="xs:string" minOccurs="0" maxOccurs="1"/>
        <xs:element name="commands" type="hws:COMMANDS" minOccurs="1" maxOccurs="1"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="HTTP">
    <xs:sequence>
        <xs:element name="http-url" type="xs:string" minOccurs="1" maxOccurs="1"/>
        <xs:element name="post" type="hws:HTTP-POST" minOccurs="0" maxOccurs="1"/>
        <xs:choice minOccurs="1" maxOccurs="1">
            <xs:element name="wait" type="hws:FLAG" minOccurs="1" maxOccurs="1"/>
            <xs:element name="continue" type="hws:FLAG" minOccurs="1" maxOccurs="1"/>
        </xs:choice>
        <xs:element name="capture-output" type="hws:FLAG"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="EMAIL">
    <xs:sequence>
        <xs:element name="to" type="xs:string" minOccurs="1" maxOccurs="1"/>
        <xs:element name="subject" type="xs:string" minOccurs="1" maxOccurs="1"/>
        <xs:element name="message" type="xs:string" minOccurs="0" maxOccurs="1"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="HTTP-POST">
    <xs:simpleContent>
        <xs:extension base="xs:string">
            <xs:attribute name="content-type" type="xs:string" use="required"/>
        </xs:extension>
    </xs:simpleContent>
</xs:complexType>

<xs:complexType name="FLAG"/>

<xs:complexType name="CONFIGURATION">
    <xs:sequence>
        <xs:element name="property" minOccurs="1" maxOccurs="unbounded">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="name" minOccurs="1" maxOccurs="1" type="xs:string"/>

```

```

        <xs:element name="value" minOccurs="1" maxOccurs="1" type="xs:string"/>
        <xs:element name="description" minOccurs="0" maxOccurs="1" type="xs:string"/>
    </xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>

<xs:complexType name="STREAMING">
    <xs:sequence>
        <xs:element name="mapper" type="xs:string" minOccurs="0" maxOccurs="1"/>
        <xs:element name="reducer" type="xs:string" minOccurs="0" maxOccurs="1"/>
        <xs:element name="file" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="cached-file" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="cached-archive" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="PREPARE">
    <xs:sequence>
        <xs:element name="delete" type="hws:DELETE" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="DELETE">
    <xs:attribute name="path" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="COMMANDS">
    <xs:sequence>
        <xs:element name="delete" type="hws:DELETE" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="mkdir" type="hws:MKDIR" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="move" type="hws:MOVE" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="MKDIR">
    <xs:attribute name="path" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="MOVE">
    <xs:attribute name="source" type="xs:string" use="required"/>
    <xs:attribute name="target" type="xs:string" use="required"/>
</xs:complexType>
</xs:schema>

```

Appendix B, Workflow Examples

Fork and Join Example

The following workflow definition example executes 4 Map–Reduce jobs in 3 steps, 1 job, 2 jobs in parallel and 1 job.

The output of the jobs in the previous step are use as input for the next jobs.

Required workflow job parameters:

- jobtracker : JobTracker HOST:PORT
- namenode : NameNode HOST:PORT
- input : input directory
- output : output directory

```

<workflow-app name='example-forkjoinwf' xmlns="uri:hws-app:1.0">

  <start to='firstjob' />

  <action name="firstjob">
    <map-reduce xmlns='uri:hws-actions:1.0'>
      <configuration>
        <property>
          <name>mapred.mapper.class</name>
          <value>com.yahoo.oozie.example.IdMapper</value>
        </property>
        <property>
          <name>mapred.reducer.class</name>
          <value>com.yahoo.oozie.example.IdReducer</value>
        </property>
        <property>
          <name>mapred.map.tasks</name>
          <value>1</value>
        </property>
        <property>
          <name>mapred.input.dir</name>
          <value>${wf:conf('input')}</value>
        </property>
        <property>
          <name>mapred.output.dir</name>
          <value>/usr/foo/${wf:id()}/templ</value>
        </property>
        <property>
          <name>mapred.job.tracker</name>
          <value>${wf:conf('jobtracker')}</value>
        </property>
        <property>
          <name>fs.default.name</name>
          <value>${wf:conf('namenode')}</value>
        </property>
      </configuration>
    </map-reduce>
    <ok to="fork" />
    <error to="kill" />
  </action>

  <fork name='fork'>
    <transition to='secondjob' />
    <transition to='thirdjob' />
  </fork>

  <action name="secondjob">
    <map-reduce xmlns='uri:hws-actions:1.0'>
      <configuration>
        <property>
          <name>mapred.mapper.class</name>
          <value>com.yahoo.oozie.example.IdMapper</value>
        </property>
        <property>
          <name>mapred.reducer.class</name>
          <value>com.yahoo.oozie.example.IdReducer</value>
        </property>
        <property>
          <name>mapred.map.tasks</name>
          <value>1</value>
        </property>
        <property>
          <name>mapred.input.dir</name>
          <value>/usr/foo/${wf:id()}/templ</value>
        </property>
        <property>
          <name>mapred.output.dir</name>

```

```

        <value>/usr/foo/${wf:id()}/temp2</value>
    </property>
    <property>
        <name>mapred.job.tracker</name>
        <value>${wf:conf('jobtracker')}</value>
    </property>
    <property>
        <name>fs.default.name</name>
        <value>${wf:conf('namenode')}</value>
    </property>
</configuration>
</map-reduce>
<ok to="join" />
<error to="kill" />
</action>

<action name="thirdjob">
    <map-reduce xmlns='uri:hws-actions:1.0'>
        <configuration>
            <property>
                <name>mapred.mapper.class</name>
                <value>com.yahoo.oozie.example.IdMapper</value>
            </property>
            <property>
                <name>mapred.reducer.class</name>
                <value>com.yahoo.oozie.example.IdReducer</value>
            </property>
            <property>
                <name>mapred.map.tasks</name>
                <value>1</value>
            </property>
            <property>
                <name>mapred.input.dir</name>
                <value>/usr/foo/${wf:id()}/templ</value>
            </property>
            <property>
                <name>mapred.output.dir</name>
                <value>/usr/foo/${wf:id()}/temp3</value>
            </property>
            <property>
                <name>mapred.job.tracker</name>
                <value>${wf:conf('jobtracker')}</value>
            </property>
            <property>
                <name>fs.default.name</name>
                <value>${wf:conf('namenode')}</value>
            </property>
        </configuration>
    </map-reduce>
    <ok to="join" />
    <error to="fail" />
</action>

<join name='join'>
    <transition to='finaljob' />
</join>

<action name="finaljob">
    <map-reduce xmlns='uri:hws-actions:1.0'>
        <configuration>
            <property>
                <name>mapred.mapper.class</name>
                <value>com.yahoo.oozie.example.IdMapper</value>
            </property>
            <property>
                <name>mapred.reducer.class</name>
                <value>com.yahoo.oozie.example.IdReducer</value>
            </property>
        </configuration>
    </map-reduce>
    <ok to="join" />
    <error to="fail" />
</action>

```

```

    </property>
    <property>
      <name>mapred.map.tasks</name>
      <value>1</value>
    </property>
    <property>
      <name>mapred.input.dir</name>
      <value>/usr/foo/${wf:id()}/temp2,/usr/foo/${wf:id()}/temp3
      </value>
    </property>
    <property>
      <name>mapred.output.dir</name>
      <value>${wf:conf('output')}</value>
    </property>
    <property>
      <name>mapred.job.tracker</name>
      <value>${wf:conf('jobtracker')}</value>
    </property>
    <property>
      <name>fs.default.name</name>
      <value>${wf:conf('namenode')}</value>
    </property>
  </configuration>
</map-reduce>
<ok to="end" />
<ok to="kill" />
</action>

<kill name="kill">
  <message>Map/Reduce failed, error message[${wf:errorMessage()}]</message>
</kill>

<end name='end' />
</workflow-app>

```

Last revised by: tucu on 09 Mar 2009

ATTACHMENT LIST

```
%ATTACHMENTLIST{format="$fileName?"}%
```