Java Cookbook

This document comprehensively describes the procedure of running Java code using Oozie. Its targeted audience is all forms of users who will install, use and operate Oozie.

Java Action specification

The Java action will execute the public static void main(String[] args) method of the specified main Java class. The required Java class(es) should be packaged in the form of a JAR and placed within your workflow application's lib directory.

The directory structure looks like this:

- wf-app-dir/workflow.xml
- wf-app-dir/lib
- wf-app-dir/lib/myJavaClasses.JAR

When executing a java application without oozie, one specifies the class having the main function (main-class), run-time JVM options and the arguments to be passed.

```
$ java -Xms512m a.b.c.MyMainClass arg1 arg2
```

Now with Oozie, they are specified as inline tags in the workflow.xml file.

```
<action name='java1'>
    <java>
  ...
         <main-class> a.b.c.MyJavaMain </main-class>
         <java-opts> -Xms512m </java-opts>
         <arg> arg1 </arg>
         <arg> arg2 </arg>
  ...
    </java>
  </action>
```

The `java-opts` element, if present, contains the command line parameters which are to be used to start the JVM that will execute the Java application. For multiple command line parameters, there may instead be a `java-opt` element for each.

The `arg` elements, if present, contains arguments for the main function. The value of each arg element is considered a single argument and they are passed to the main method *in the same order*.

Java applications are executed in the Hadoop cluster as map-reduce job with a single Mapper task. Hence the Java action has to be configured with the following properties in the form of XML tags:

- `<job-tracker>`
- `<name-node>`
- `<configuration>` element is used to specify key/value properties for the map-reduce job. Some common properties include:
    - `mapred.job.queue.name` specifies the queue-name that the job will be submitted to since the Java application is run from within a Map-Reduce job. If not mentioned, the default queue *default* is assumed.

```
<action name='java1'>
          <java>
              <job-tracker>foo.bar:8021</job-tracker>
              <name-node>foo1.bar:8020</name-node>
                                 ...
              <configuration>
               <property>
                    <name>abc</name>
                    <value>def</value>
               </property>
            </configuration>
          </java>
</action>
```

Prepare block

A java action can be configured to perform HDFS files/directories cleanup such as deleting an existing output directory (`<delete>`) or creating a new one (`<mkdir>`) before starting the Java application. This capability enables Oozie to retry a Java application in the situation of a transient or non-transient failure (This can be used to cleanup any temporary data which may have been created by the Java application in case of failure).

The prepare element, if present, indicates a list of paths to do file operations upon, before starting the Java application. This should be used exclusively for directory cleanup for the Java application to be executed.

Capture-output element

The capture-output element can be used to propagate values back into Oozie context, which can then be accessed via EL-functions. Thus, in a workflow application with multiple actions, the output of the Java action can be accessed by subsequent actions.

- This needs to be written out as a Java properties format file specified via the property **`oozie.action.output.properties`**.
- A useful EL function for capturing action output data is
    - Map wf:actionData(String node)

This utility is only applicable to action nodes that produce output data on completion.

The output data is in a Java Properties format and via this EL function it is available as a Map.

From Oozie **v3.3** onwards, more EL functions are available so your Java code can process this output in numerous formats (e.g. Properties, JSON etc.)

For more details, refer to the section on EL functions in the Workflow Functional Specification

Example

The example below illustrates the use of `<capture-output>` element and the corresponding java main function definition.

- In this example, we pass a the PASS_ME variable between the java action and the pig1 action.
- The PASS_ME variable is given the value 123456 in the java-main action named java1.
- The pig1 action subsequently reads the value of the PASS_ME variable and passes it to the PIG script.

```
<workflow-app xmlns='uri:oozie:workflow:0.1' name='java-wf'>
    <start to='java1' />

    <action name='java1'>
        <java>
            <job-tracker>${jobTracker}</job-tracker>
            <name-node>${nameNode}</name-node>
            <configuration>
               <property>
                    <name>mapred.job.queue.name</name>
                    <value>${queueName}</value>
                </property>
            </configuration>
            <main-class>org.apache.oozie.test.MyTest</main-class>
            <arg>${outputFileName}</arg>
            <capture-output/>
        </java>
        <ok to="pig1" />
        <error to="fail" />
    </action>


    <action name='pig1'>
        <pig>
            <job-tracker>${jobTracker}</job-tracker>
            <name-node>${nameNode}</name-node>
            <configuration>
                <property>
                    <name>mapred.job.queue.name</name>
                    <value>${queueName}</value>
                </property>
            </configuration>
            <script>script.pig</script>
            <param>MY_VAR=${wf:actionData('java1')['PASS_ME']}</param>
        </pig>
        <ok to="end" />
        <error to="fail" />
    </action>

    <kill name="fail">
        <message>Pig failed, error message[${wf:errorMessage(wf:lastErrorNode())}]</message>
    </kill>
    <end name='end' />
</workflow-app>
```

Java Main Class

The `main()` method writes a Property file to the path specified in the 'oozie.action.output.properties' ENVIRONMENT variable.

MyTest.java

```
package org.apache.oozie.test;

import java.io.*;
import java.util.Properties;

public class MyTest {

   ////////////////////////////////
   // Do whatever you want in here
   ////////////////////////////////
   public static void main (String[] args)
   {
      String fileName = args[0];
      try{
         File file = new File(System.getProperty("oozie.action.output.properties"));
         Properties props = new Properties();
         props.setProperty("PASS_ME", "123456");

         OutputStream os = new FileOutputStream(file);
         props.store(os, "");
         os.close();
         System.out.println(file.getAbsolutePath());
      }
      catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

Launcher configuration properties

Oozie executes the Java action within a Launcher mapper on the compute node. Some commonly used `<configuration>` properties passed for the java action can be as follows:

- `<oozie.mapred.child.java.opts>` similar to using the `<java-opts>` described before
- Setting environment variables

```
      <property>
          <name>oozie.mapred.child.env</name>
          <value>A=foo</value>
      </property>
```

Thus summarily, mapred properties are applied to the java-action map job by prefixing "oozie." to those property names.

Java Action Transition

The workflow job will wait until the java application completes its execution before continuing to the next action. To indicate an **ok** action transition, the main Java class must complete gracefully the main method invocation.

To indicate an **`error`** action transition, the main Java class must throw an exception.

e.g. If the main Java class calls System.exit(int n) where n is **non-zero**, this will make the java action to do an **error transition** regardless of the used exit code.

Non-zero system exit will be error whereas a System.exit(0) will lead to **ok** transition.

Debugging inside Java classes

- If the Java class function has any System.out.print commands, their output is seen on the HDFS Jobtracker task log (accessible via Oozie web-console by clicking on the job in the list and then using the console URL for jobtracker to view the log for your specific task).
- Alternately, if you know your job-id from CLI, you can look up your job's logs from the jobtracker webpage and then the 'map' task logs. The Oozie launcher has some very informative log statements such as job properties and exception traces in these task logs.

- If you are performing incremental changes to your java classes and want to see the changes reflected, you need to update the JAR that is supposed to include your java class. In this case, the JAR is the one inside your workflow application's lib directory.

```
$ jar uf /path/to/jar -C /path/to/a/ a/b/c/myMainClass.class
```

where a.b.c.myMainClass is the package structure you wish to maintain in the JAR.

Java System Properties

Within your Java Main class, you can query for the following system properties pertaining to your Oozie job.

1. **oozie.job.id** : Workflow ID
2. **oozie.action.id** : Action ID
3. **oozie.action.conf.xml** : local path to the resolved action configuration
4. **oozie.action.output.properties** : Action properties output as a Java Properties file (described more in Java Action Specification section. See the use-cases for illustration)

This provides for a convenient way to access these values directly within the action code.

Java action XML Schema

The XML schema definition for Oozie workflows specifies an 'xs:sequence' for the tags within each action block. Hence, for a valid workflow application, the various tags e.g. prepare, configuration, java-opts, arg, file, archive etc. should be given in the right order. Note that some of the tags are optional, but if used, the order should be kept in mind. For e.g. if using <file> or <archive> to copy some class files from known hdfs locations to your task cache, they should be specified **after** any <configuration> or <main-class> tags.

1. job-tracker (required)
2. name-node (required)
3. prepare
4. configuration
5. main-class (required)
6. java-opts
7. arg
8. file
9. archive
10. capture-output

Some tags are required. Rest all are optional.

For more information the XSD for Java action looks like the following:

```
<xs:complexType name="JAVA">
    <xs:sequence>
        <xs:element name="job-tracker" type="xs:string" minOccurs="1" maxOccurs="1"/>
        <xs:element name="name-node" type="xs:string" minOccurs="1" maxOccurs="1"/>
        <xs:element name="prepare" type="workflow:PREPARE" minOccurs="0" maxOccurs="1"/>
        <xs:element name="job-xml" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="configuration" type="workflow:CONFIGURATION" minOccurs="0" maxOccurs="1"/>
        <xs:element name="main-class" type="xs:string" minOccurs="1" maxOccurs="1"/>
        <xs:element name="java-opts" type="xs:string" minOccurs="0" maxOccurs="1"/>
        <xs:element name="arg" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="file" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="archive" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="capture-output" type="workflow:FLAG" minOccurs="0" maxOccurs="1"/>
    </xs:sequence>
</xs:complexType>
```

Authenticating on a Kerberos-enabled cluster

In order for a Java action to succeed on a secure cluster, it must propagate the Hadoop delegation token like in the following code snippet (this is benign on non-secure clusters):

```
// propagate delegation related props from launcher job to MR job
if (System.getenv("HADOOP_TOKEN_FILE_LOCATION") != null) {
    jobConf.set("mapreduce.job.credentials.binary", System.getenv("HADOOP_TOKEN_FILE_LOCATION"));
}
```

Examples and Use-Cases

Following are example workflow applications that illustrate use-cases of the Oozie Java action.

I Using Java-Main action to copy local file to HDFS

Assume a local file $filename can be accessed by all cluster nodes.

- Define a java action in your workflow.xml

```
<workflow-app xmlns='uri:oozie:workflow:0.3' name='java-filecopy-wf'>
    <start to='java1' />
        <action name='java1'>
        <java>
            <job-tracker>${jobTracker}</job-tracker>
            <name-node>${nameNode}</name-node>
            <configuration>
                <property>
                    <name>mapred.job.queue.name</name>
                    <value>${queueName}</value>
                </property>
            </configuration>
            <main-class>testCopyFromLocal</main-class>
            <arg>${filename}</arg>
            <arg>${nameNode}${testDir}</arg>
            <capture-output/>
        </java>
        <ok to="end" />
        <error to="fail" />
    </action>
    <kill name="fail">
        <message>Java failed, error message[${wf:errorMessage(wf:lastErrorNode())}]
            </message>
    </kill>
    <end name='end' />
</workflow-app>
```

- Here is a sample java main class.

TestCopyFromLocal.java

```
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.Configuration;
import java.io.File;
import java.io.IOException;

public class TestCopyFromLocal {

        public static void main (String[] args) throws IOException {
                String src = args[0];
                String dst = args[1];
                System.out.println("testCopyFromLocal, source= " + src);
                System.out.println("testCopyFromLocal, target= " + dst);
                Configuration conf = new Configuration();
                Path src1 = new Path(src);
                Path dst1 = new Path(dst);
                FileSystem fs = FileSystem.get(conf);
                try{
                   //delete local file after copy
                   fs.copyFromLocalFile(true, true, src1, dst1);
                }
                catch(IOException ex) {
                   System.err.println("IOException during copy operation " +
```

```
                                    ex.toString());
                ex.printStackTrace();
                System.exit(1);
            }
        }
}
```

II Java-Main Action decision nodes

This example is to illustrate how action output data captured using capture output can be used in decision nodes.

```
<workflow-app xmlns='uri:oozie:workflow:0.3' name='java-actionprops-wf'>
    <start to='java3' />
        <action name='java3'>
        <java>
            <job-tracker>${jobTracker}</job-tracker>
            <name-node>${nameNode}</name-node>
            <configuration>
                <property>
                    <name>mapred.job.queue.name</name>
                    <value>${queueName}</value>
                </property>
            </configuration>
            <main-class>exampleDecision</main-class>
            <arg>yes</arg>
            <capture-output/>
        </java>
        <ok to="end" />
        <error to="fail" />
    </action>

        <decision name="decision1">
       <switch>
           <case to="end">${(wf:actionData('java3')['key1'] == "value1") and (wf:actionData('java3')['key2'] ==
"value2")}</case>
           <default to="fail" />
       </switch>
    </decision>

    <kill name="fail">
        <message>Java failed, error message[${wf:errorMessage(wf:lastErrorNode())}]
         </message>
    </kill>
    <end name='end' />
<workflow-app>
```

The corresponding Java class can be as below:

ExampleDecision.java

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.util.Properties;

public class ExampleDecision {
    public static void main (String[] args) {
       String text = args[0];
        try{
          File file = new File(System.getProperty("oozie.action.output.properties"));
          Properties props = new Properties();
```

```
          if (text.equals("yes")) {
                props.setProperty("key1", "value1");
                props.setProperty("key2", "value2");
          } else {
                props.setProperty("key1", "novalue");
                props.setProperty("key2", "novalue");
          }

          OutputStream os = new FileOutputStream(file);
          props.store(os, "");
          os.close();
          System.out.println(file.getAbsolutePath());
        }
        catch (Exception e) {
          e.printStackTrace();
        }
    }
}
```

FAQ

<to be compiled>