Spring Hadoop Reference Manual

Costin Leau

Spring Hadoop Reference Manual

by Costin Leau

1.0.0.M2

Table of Contents

Preface v
I. Introduction 1
1. Requirements 2
II. Spring and Hadoop 3
2. Hadoop Configuration, MapReduce, and Distributed Cache 4
2.1. Using the Spring for Apache Hadoop Namespace 4
2.2. Configuring Hadoop 5
2.3. Creating a Hadoop Job 8
Creating a Hadoop Streaming Job 9
Running a Hadoop Job 9
2.4. Using the Hadoop Job tasklet 10
2.5. Running a Hadoop Tool 10
Replacing Hadoop shell invocations with `tool` 11
2.6. Using the Hadoop Tool tasklet 12
2.7. Map Reduce Generic Options 12
2.8. Configuring the Hadoop `DistributedCache` 12
3. Working with the Hadoop File System 14
3.1. Configuring the file-system 14
3.2. Scripting the Hadoop API 15
Using scripts 17
3.3. Scripting implicit variables 17
3.4. File System Shell (FsShell) 18
DistCp API 19
3.5. Scripting Lifecycle 20
3.6. Using the Scripting tasklet 20
4. Working with HBase 22
4.1. Data Access Object (DAO) Support 22
5. Hive integration 24
5.1. Starting a Hive Server 24
5.2. Using the Hive Thrift Client 24
5.3. Using the Hive JDBC Client 25
5.4. Using the Hive tasklet 25
6. Pig support 27
6.1. Using the Pig tasklet 27
7. Cascading integration 28
7.1. Using the Cascading tasklet 29
7.2. Using Scalding 29
7.3. Spring-specific local `Tap`s 30
8. Security Support 32
8.1. HDFS permissions 32
8.2. User impersonation (Kerberos) 32
III. Developing Spring for Apache Hadoop Applications 34
9. Guidance and Examples 35

9.1. Scheduling 35
9.2. Batch Job Listeners 36
IV. Spring for Apache Hadoop sample applications 37
10. Sample prerequisites 38
11. Wordcount sample using the Spring Framework 39
11.1. Introduction 39
12. Wordcount sample using Spring Batch 40
12.1. Introduction 40
12.2. Basic Spring for Apache Hadoop configuration 40
12.3. Build and run the sample application 42
12.4. Run the sample application as a standlone Java application 42
V. Other Resources 44
13. Useful Links 45
VI. Appendices 46
A. Spring for Apache Hadoop Schema 47

Preface

Spring for Apache Hadoop provides extensions to Spring, Spring Batch, and Spring Integration to build manageable and robust pipeline solutions around Hadoop.

Spring for Apache Hadoop supports reading from and writing to HDFS, running various types of Hadoop jobs (Java MapReduce, Streaming), scripting and HBase, Hive and Pig interactions. An important goal is to provide excellent support for non-Java based developers to be productive using Spring for Apache Hadoop and not have to write any Java code to use the core feature set.

Spring for Apache Hadoop also applies the familiar Spring programming model to Java MapReduce jobs by providing support for dependency injection of simple jobs as well as a POJO based MapReduce programming model that decouples your MapReduce classes from Hadoop specific details such as base classes and data types.

This document assumes the reader already has a basic familiarity with the Spring Framework and Hadoop concepts and APIs.

While every effort has been made to ensure that this documentation is comprehensive and there are no errors, nevertheless some topics might require more explanation and some typos might have crept in. If you do spot any mistakes or even more serious errors and you can spare a few cycles during lunch, please do bring the error to the attention of the Spring for Apache Hadoop team by raising an issue. Thank you.

Part I. Introduction

Spring for Apache Hadoop provides integration with the Spring Framework to create and run Hadoop MapReduce, Hive, and Pig jobs as well as work with HDFS and HBase. If you have simple needs to work with Hadoop, including basic scheduling, you can add the Spring for Apache Hadoop namespace to your Spring based project and get going quickly using Hadoop. As the complexity of your Hadoop application increases, you may want to use Spring Batch and Spring Integration to regin in the complexity of developing a large Hadoop application.

This document is the reference guide for Spring for Apache Hadoop project (SHDP). It explains the relationship between the Spring framework and Hadoop as well as related projects such as Spring Batch and Spring Integration. The first part describes the integration wtih the Spring framework to define the base concepts and semantics of the integration and how they can be use effectively. The second part describes how you can build upon these base concepts and create workflow based solutions provided by the integration with Spring Batch.

1. Requirements

Spring for Apache Hadoop requires JDK level 6.0 (just like Hadoop) and above, Spring Framework 3.0 (3.1 recommended) and above and Hadoop 0.20.2 (1.0.0 recommended) and above. Regarding Hadoop-related projects, SDHP supports HBase 0.90.x, Hive 0.7.x and Pig 0.9.x and above. As a rule of thumb, when using Hadoop-related projects, such as Hive or Pig, use the required Hadoop version as a basis for discovering the supported versions.

Spring for Apache Hadoop also requires you have a Hadoop installation up and running. If you don't already have a Hadoop cluster up and running in your environment, a good first step is to create a single-node cluster. To install Hadoop 0.20.x+, the Getting Started page from the official Apache documentation is a good general guide. If you are running on Ubuntu, the tutorial from Michael G. Noll, "Running Hadoop On Ubuntu Linux (Single-Node Cluster)" provides more details. It is also convenience to download a Virtual Machine where Hadoop is setup and ready to go. Cloudera provides virtual machines of various formats here. You can also download the EMC Greenplum HD distribution or get a tech preview of the Hortonworks distribution.

Part II. Spring and Hadoop

Document structure

This part of the reference documentation explains the core functionality that Spring for Apache Hadoop (SHDP) provides to any Spring based application.

Chapter 2, *Hadoop Configuration, MapReduce, and Distributed Cache* describes the Spring support for bootstrapping, initializing and working with core Hadoop.

Chapter 3, *Working with the Hadoop File System* describes the Spring support for interacting with the Hadoop file system.

Chapter 4, *Working with HBase* describes the Spring support for HBase.

Chapter 5, *Hive integration* describes the Spring support for Hive.

Chapter 6, *Pig support* describes the Spring support for Pig.

2. Hadoop Configuration, MapReduce, and Distributed Cache

One of the common tasks when using Hadoop is interacting with its *runtime* - whether it is a local setup or a remote cluster, one needs to properly configure and bootstrap Hadoop in order to submit the required jobs. This chapter will focus on how Spring for Apache Hadoop (SHDP) leverages Spring's lightweight IoC container to simplify the interaction with Hadoop and make deployment, testing and provisioning easier and more manageable.

2.1 Using the Spring for Apache Hadoop Namespace

To simplify configuration, SHDP provides a dedicated namespace for most of its components. However, one can opt to configure the beans directly through the usual `<bean>` definition. For more information about XML Schema-based configuration in Spring, see this appendix in the Spring Framework reference documentation.

To use the SHDP namespace, one just needs to import it inside the configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:❶hdp="❷http://www.springframework.org/schema/hadoop"
   xsi:schemaLocation="
    http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xs
    http://www.springframework.org/schema/hadoop http://www.springframework.org/schema/hadoop/spring-hadoop

   <bean id ... >

   ❹<hdp:configuration ...>
</beans>
```

❶ Spring for Apache Hadoop namespace prefix. Any name can do but through out the reference documentation, the `hdp` will be used.

❷ The namespace URI.

❸ The namespace URI location. Note that even though the location points to an external address (which exists and is valid), Spring will resolve the schema locally as it is included in the Spring for Apache Hadoop library.

❹ Declaration example for the Hadoop namespace. Notice the prefix usage.

Once declared, the namespace elements can be declared simply by appending the aforementioned prefix. Note that is possible to change the default namespace, for example from `<beans>` to `<hdp>`. This is useful for configuration composed mainly of Hadoop components as it avoids declaring the prefix. To achieve this, simply swap the namespace prefix declaration above:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/hadoop"❶
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   ❷xmlns:beans="http://www.springframework.org/schema/beans"
   xsi:schemaLocation="
```

```
    http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/hadoop http://www.springframework.org/schema/gemfire/spring-hadoop

    ❸<beans:bean id ... >

    ❹<configuration ...>

</beans:beans>
```

❶ The default namespace declaration for this XML file points to the Spring for Apache Hadoop namespace.
❷ The beans namespace prefix declaration.
❸ Bean declaration using the `<beans>` namespace. Notice the prefix.
❹ Bean declaration using the `<hdp>` namespace. Notice the *lack* of prefix (as `hdp` is the default namespace).

For the remainder of this doc, to improve readability, the XML examples will simply refer to the `<hdp>` namespace without the namespace declaration, where possible.

2.2 Configuring Hadoop

In order to use Hadoop, one needs to first configure it namely by creating a `Configuration` object. The configuration holds information about the job tracker, the input, output format and the various other parameters of the map reduce job.

In its simplest form, the configuration definition is a one liner:

```
<hdp:configuration />
```

The declaration above defines a `Configuration` bean (to be precise a factory bean of type `ConfigurationFactoryBean`) named, by default, `hadoopConfiguration`. The default name is used, by conventions, by the other elements that require a configuration - this leads to simple and very concise configurations as the main components can automatically wire themselves up without requiring any specific configuration.

For scenarios where the defaults need to be tweaked, one can pass in additional configuration files:

```
<hdp:configuration resources="classpath:/custom-site.xml, classpath:/hq-site.xml">
```

In this example, two additional Hadoop configuration resources are added to the configuration.

Note

Note that the configuration makes use of Spring's `Resource` abstraction to locate the file. This allows various search patterns to be used, depending on the running environment or the prefix specified (if any) by the value - in this example the classpath is used.

In addition to referencing configuration resources, one can tweak Hadoop settings directly through Java `Properties`. This can be quite handy when just a few options need to be changed:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:hdp="http://www.springframework.org/schema/hadoop"
 xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/hadoop http://www.springframework.org/schema/hadoop/spring-had

    <hdp:configuration>
       fs.default.name=hdfs://localhost:9000
       hadoop.tmp.dir=/tmp/hadoop
       electric=sea
    </hdp:configuration>
</beans>
```

One can further customize the settings by avoiding the so called *hard-coded* values by externalizing them so they can be replaced at runtime, based on the existing environment without touching the configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:hdp="http://www.springframework.org/schema/hadoop"
    xmlns:context="http://www.springframework.org/schema/context"
 xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-c
       http://www.springframework.org/schema/hadoop http://www.springframework.org/schema/hadoop/spring-had

    <hdp:configuration>
       fs.default.name=${hd.fs}
       hadoop.tmp.dir=file://${java.io.tmpdir}
       hangar=${number:18}
    </hdp:configuration>

    <context:property-placeholder location="classpath:hadoop.properties" />
</beans>
```

Through Spring's property placeholder support, SpEL and the environment abstraction (available in Spring 3.1). one can externalize environment specific properties from the main code base easing the deployment across multiple machines. In the example above, the default file system is replaced based on the properties available in `hadoop.properties` while the temp dir is determined dynamically through `SpEL`. Both approaches offer a lot of flexbility in adapting to the running environment - in fact we use this approach extensivly in the Spring for Apache Hadoop test suite to cope with the differences between the different development boxes and the CI server.

Additionally, external `Properties` files can be loaded, `Properties` beans (typically declared through Spring's `util` namespace). Along with the nested properties declaration, this allows customized configurations to be easily declared:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:hdp="http://www.springframework.org/schema/hadoop"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:util="http://www.springframework.org/schema/util"
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/be
        http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-c
```

```
        http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util.xs
        http://www.springframework.org/schema/hadoop http://www.springframework.org/schema/hadoop/spring-had

   <!-- merge the local properties, the props bean and the two properties files -->
   <hdp:configuration properties-ref="props" properties-location="cfg-1.properties, cfg-2.properties">
      star=chasing
      captain=eo
   </hdp:configuration>

   <util:properties id="props" location="props.properties"/>
</beans>
```

When merging several properties, ones defined locally win. In the example above the configuration properties are the primary source, followed by the `props` bean followed by the external properties file based on their defined order. While it's not typical for a configuration to refer to use so many properties, the example showcases the various options available.

Note

For more properties utilities, including using the System as a source or fallback, or control over the merging order, consider using Spring's `PropertiesFactoryBean` (which is what Spring for Apache Hadoop and `util:properties` use underneath).

It is possible to create configuration based on existing ones - this allows one to create dedicated configurations, slightly different from the main ones, usable for certain jobs (such as streaming - more on that below). Simply use the `configuration-ref` attribute to refer to the *parent* configuration - all its properties will be inherited and overridden as specified by the child:

```
<!-- default name is 'hadoopConfiguration' -->
<hdp:configuration>
    fs.default.name=${hd.fs}
    hadoop.tmp.dir=file://${java.io.tmpdir}
</hdp:configuration>

<hdp:configuration id="custom" configuration-ref="hadoopConfiguration">
    fs.default.name=${custom.hd.fs}
</hdp:configuration>

...
```

Make sure though you specify a different name since otherwise, since both definitions will have the same name, the Spring container will interpret this as being the same definition (and will usually consider the last one found).

Another option worth mentioning is `register-url-handler` which, as the name implies, automatically registers an URL handler in the running VM. This allows urls referrencing *hdfs* resource (by using the `hdfs` prefix) to be properly resolved - if the handler is not registered, such an URL would through an exception since the VM does not know what `hdfs` mean.

Note

Since only one URL handler can be registered per VM, at most once, this option is turned off by default. Due to the reasons mentioned before, once enabled if it fails, it will log the error

but will not throw an exception. If your `hdfs` URLs stop working, make sure to investigate this aspect.

Last but not least a reminder that one can mix and match all these options to her preference. In general, consider externalizing configuration since it allows easier updates without interfering with the application configuration. When dealing with multiple, similar configuration use configuration *composition* as it tends to keep the definitions concise, in sync and easy to update.

2.3 Creating a Hadoop Job

Once the Hadoop configuration is taken care of, one needs to actually submit some work to it. SHDP makes it easy to configure and run Hadoop jobs whether they are vanilla map-reduce type or streaming. Let us start with an example:

```
<hdp:job id="mr-job"
  input-path="/input/" output-path="/ouput/"
  mapper="org.apache.hadoop.examples.WordCount.TokenizerMapper"
  reducer="org.apache.hadoop.examples.WordCount.IntSumReducer"/>
```

The declaration above creates a typical Hadoop `Job`: specifies its input and output, the mapper and the reducer classes. Notice that there is no reference to the Hadoop configuration above - that's because, if not specified, the default naming convention (`hadoopConfiguration`) will be used instead. Neither are the key or value types - these two are automatically determined through a best-effort attempt by analyzing the class information of the mapper and the reducer. Of course, these settings can be overridden: the former through the `configuration-ref` element, the latter through `key` and `value` attributes. There are plenty of options available not shown in the example (for simplicity) such as the jar (specified directly or by class), sort or group comparator, the combiner, the partitioner, the codecs to use or the input/output format just to name a few - they are supported, just take a look at the SHDP schema (Appendix A, *Spring for Apache Hadoop Schema*) or simply trigger auto-completion (usually `ALT+SPACE`) in your IDE; if it supports XML namespaces and is properly configured it will display the available elements. Additionally one can extend the default Hadoop configuration object and add any special properties not available in the namespace or its backing bean (`JobFactoryBean`).

It is worth pointing out that per-job specific configurations are supported by specifying the custom properties directly or referring to them (more information on the pattern is available here):

```
<hdp:job id="mr-job"
  input-path="/input/" output-path="/ouput/"
  mapper="mapper class" reducer="reducer class"
  jar-by-class="class used for jar detection"
  properties-location="classpath:special-job.properties">
    electric=sea
</hdp:job>
```

Note

The job definition can validate the existance of the input and output paths before submitting the actual job (which is slow), to prevent its failure. Take a look at *validate-paths* attribute avoid these errors early on without having to touch the job tracker only to get an exception.

`job` provides additional properties, such as the generic options, however one that is worth mentioning is `jar` which allows a job (and its dependencies) to be loaded form entirely from a specified jar. This is useful for isolating jobs and avoiding classpath and versioning collisions. Note that provisioning of the jar into the cluster, still depends on the target environment - see the aforementioned section for more info (such as `libs`).

Creating a Hadoop Streaming Job

Hadoop Streaming job (or in short streaming), is a popular feature of Hadoop as they allow the creation of Map/Reduce jobs with any executable or script (the equivalent of using the previous counting words example is to use `cat` and `wc` commands). While it is rather easy to start up streaming from the command line, doing so programatically, such as from a Java environment, can be challenging due to the various number of parameters (and their ordering) that need to be parsed. SHDP simplifies such as tasks - it's as easy and straight-forward as declaring a `job` from the previous section; in fact most of the attributes will be the same:

```
<hdp:streaming id="streaming"
  input-path="/input/" output-path="/ouput/"
  mapper="${path.cat}" reducer="${path.wc}"/>
```

Existing users might be wondering how can they pass the command line arguments (such as `-D` or `-cmdenv`). These former customize the Hadoop configuration (which has been convered in the previous section) while the latter are supported through the `cmd-env` element:

```
<hdp:streaming id="streaming-env"
  input-path="/input/" output-path="/ouput/"
  mapper="${path.cat}" reducer="${path.wc}">
  <hdp:cmd-env>
     EXAMPLE_DIR=/home/example/dictionaries/
     ...
  </hdp:cmd-env>
</hdp:streaming>
```

Just like `job`, `streaming` supports the generic options; follow the link for more information.

Running a Hadoop Job

The jobs, after being created and configured, need to be submitted for execution to a Hadoop cluster. For non-trivial cases, a coordinating, workflow solution such as Spring Batch is recommended . However for basic job submission SHDP provides `JobRunner` class which submits several jobs sequentially (and waits by default for their completion):

```
<bean id="runner" class="org.springframework.data.hadoop.mapreduce.JobRunner" p:jobs-ref="job"/>

<hdp:job id="job"  input-path="/input/" output-path="/output/"
 mapper="org.apache.hadoop.examples.WordCount.TokenizerMapper"
 reducer="org.apache.hadoop.examples.WordCount.IntSumReducer" />
```

Multiple jobs can be specified and even nested if they are not used outside the runner:

```
<bean id="runner" class="org.springframework.data.hadoop.mapreduce.JobRunner">
```

```
   <property name="jobs"><list>
     <!-- reference to another job named 'job' -->
     <ref bean="streaming-job"/>
     <!-- nested bean definition -->
     <hdp:job id="nested-job" .... />
   </list></property>
</bean>

<hdp:job id="job"  ... />
```

2.4 Using the Hadoop Job tasklet

For Spring Batch environments, SHDP provides a dedicated tasklet to execute Hadoop jobs as a step in a Spring Batch workflow. An example declaration is shown below:

```
<hdp:tasklet id="hadoop-tasklet" job-ref="mr-job" wait-for-job="true" />
```

The tasklet above references a Hadoop job definition named "mr-job". By default, wait-for-job is true so that the tasklet will wait for the job to complete when it executes. Setting wait-for-job to false will submit the job to the Hadoop cluster but not wait for it to complete.

2.5 Running a Hadoop Tool

It is common for Hadoop utilities and libraries to be started from the command-line (ex: `hadoop jar` *some.jar*). SHDP offers generic support for such cases provided that the packages in question are built on top of Hadoop standard infrastructure, namely `Tool` and `ToolRunner` classes. As oppose to the command-line usage, `Tool` instances benefit from Spring's IoC features; they can be parameterized, created and destroyed on demand and have their properties (such as the Hadoop configuration) injected.

Consider the typical `jar` example - invoking a class with some (two in this case) arguments (notice that the Hadoop configuration properties are passed as well):

```
bin/hadoop jar -conf hadoop-site.xml -jt darwin:50020 -D property=value someJar.jar org.foo.SomeTool data/in
```

Since SHDP has first-class support for configuring Hadoop, the so called `generic options` aren't needed any more, even more so since typically there is only one Hadoop configuration per application. Through `tool-runner` element (and its backing `ToolRunner` class) one typically just needs to specify the `Tool` implementation and its arguments:

```
<hdp:tool-runner id="someTool" tool-class="org.foo.SomeTool" configuration-ref="hadoopConfiguration">
   <hdp:arg value="data/in.txt"/>
   <hdp:arg value="data/out.txt"/>

   property=value
</hdp:tool-runner>
```

The previous example assumes the `Tool` dependencies (such as its class) are available in the classpath. If that is not the case, `tool-runner` allows a jar to be specified:

```
<hdp:tool-runner ... jar="myTool.jar">
    ...
```

```
</hdp:tool-runner>
```

The jar is used to instantiate and start the tool - in fact all its dependencies are loaded from the jar meaning they no longer need to be part of the classpath. This mechanism provides proper isolation between tools as each of them might depend on certain libraries with different versions; rather then adding them all into the same app (which might be impossible due to versioning conflicts), one can simply point to the different jars and be on her way. Note that when using a jar, if the main class (as specified by the Main-Class entry) is the target `Tool`, one can skip specifying the tool as it will picked up automatically.

Like the rest of the SHDP elements, `tool-runner` allows the passed Hadoop configuration (by default `hadoopConfiguration` but specified in the example for clarity) to be customized accordingly; the snippet only highlights the property initialization for simplicity but more options are available. Since usually the `Tool` implementation has a default argument, one can use the `tool-class` attribute however it is possible to refer to another `Tool` instance or declare a nested one:

```
<hdp:tool-runner id="someTool" run-at-startup="true">
   <bean class="org.foo.AnotherTool" p:input="data/in.txt" p:output="data/out.txt"/>
</hdp:tool-runner>
```

This is quite convenient if the `Tool` class provides setters or richer constructors. Note that by default the `tool-runner` does not executes the `Tool`until its definition is actually called - this behavior can be changed through the `run-at-startup` attribute above.

Replacing Hadoop shell invocations with `tool`

`tool` is a nice way for migrating series or shell invocations or scripts into fully wired, managed Java objects. Consider the following shell script:

```
hadoop jar job1.jar -files fullpath:props.properties -Dconfig=config.properties ...
hadoop jar job2.jar arg1 arg2...
...
hadoop jar job10.jar ...
```

Each job is fully contained in the specified jar, including all the dependencies (which might conflict with the ones from other jobs). Additionally each invocation might provide some generic options or arguments but for the most part all will share the same configuration (as they will execute against the same cluster).

The script can be fully ported to SHDP, through the `tool` element:

```
<hdp:tool id="job1" tool-class="job1.Tool" jar="job1.jar" files="fullpath:props.properties" properties-loca
<hdp:tool id="job2" jar="job2.jar">
   <hdp:arg value="arg1"/>
   <hdp:arg value="arg2"/>
</hdp:tool>
<hdp:tool id="job3" jar="job3.jar"/>
...
```

All the features have been explained in the previous sections but let us review what happens here. As mentioned before, each tool gets autowired with the `hadoopConfiguration`; `job1` goes beyond this and uses its own properties instead. For the first jar, the `Tool` class is specified, however the

rest assume the jar *main class*es implement the `Tool` interface; the namespace will discover them automatically and use it accordingly. When needed (such as with `job1`), additional files or libs are provisioned in the cluster. Same thing with the job arguments.

However more things that go beyond scripting, can be applied to this configuration - each job can have multiple properties loaded or declared inlined - not just from the local file system, but also from the classpath or any url for that matter. In fact, the whole configuration can be externalized and parameterized (through Spring's property placeholder and/or Environment abstraction). Moreover, each job can be ran by itself (through the `JobRunner`) or as part of a workflow - either through Spring's `depends-on` or the much more powerful Spring Batch and `tool-tasklet`.

2.6 Using the Hadoop Tool tasklet

For Spring Batch environments, SHDP provides a dedicated tasklet to execute Hadoop tasks as a step in a Spring Batch workflow. The tasklet element supports the same configuration options as tool-runner except for `run-at-startup` (which does not apply for a workflow):

```
<hdp:tool-tasklet id="tool-tasklet" tool-ref="some-tool" />
```

2.7 Map Reduce Generic Options

The `job`, `streaming` and `tool` all support a subset of generic options, specifically `archives`, `files` and `libs`. `libs` is probably the most useful as it enriches a job classpath (typically with some jars) - however the other two allow resources or archives to be copied through-out the cluster for the job to consume. Whenver faced with provisioning issues, revisit these options as they can help up significantly. Note that the `fs`, `jt` or `conf` options are not supported - these are designed for command-line usage, for bootstrapping the application. This is no longer the case, as the SHDP offers first-class support for defining and customizing Hadoop configurations.

2.8 Configuring the Hadoop `DistributedCache`

DistributedCache is a Hadoop facility for distributing application-specific, large, read-only files (text, archives, jars and so on) efficiently. Applications specify the files to be cached via urls (`hdfs://`) using `DistributedCache` and the framework will copy the necessary files to the slave nodes before any tasks for the job are executed on that node. Its efficiency stems from the fact that the files are only copied once per job and the ability to cache archives which are un-archived on the slaves. Note that `DistributedCache` assumes that the files to be cached (and specified via hdfs:// urls) are already present on the Hadoop `FileSystem`.

SHDP provides first-class configuration for the distributed cache through its `cache` element (backed by `DistributedCacheFactoryBean` class), allowing files and archives to be easily distributed across nodes:

```
<hdp:cache create-symlink="true">
   <hdp:classpath value="/cp/some-library.jar#library.jar" />
   <hdp:cache value="/cache/some-archive.tgz#main-archive" />
   <hdp:cache value="/cache/some-resource.res" />
```

```
   <hdp:local value="some-file.txt" />
</hdp:cache>
```

The definition above registers several resources with the cache (adding them to the job cache or classpath) and creates symlinks for them. As described in the `DistributedCache` documentation, the declaration format is (`absolute-path#link-name`). The link name is determined by the URI fragment (the text following the # such as *#library.jar* or *#main-archive* above) - if no name is specified, the cache bean will infer one based on the resource file name. Note that one does not have to specify the `hdfs://node:port` prefix as these are automatically determined based on the configuration wired into the bean; this prevents environment settings from being hard-coded into the configuration which becomes portable. Additionally based on the resource extension, the definition differentiates between archives (`.tgz`, `.tar.gz`, `.zip` and `.tar`) which will be uncompressed, and regular files that are copied as-is. As with the rest of the namespace declarations, the definition above relies on defaults - since it requires a Hadoop `Configuration` and `FileSystem` objects and none are specified (through `configuration-ref` and `file-system-ref`) it falls back to the default naming and is wired with the bean named *hadoopConfiguration*, creating the `FileSystem` automatically.

3. Working with the Hadoop File System

A common task in Hadoop is interacting with its file system, whether for provisioning, adding new files to be processed, parsing results, or performing cleanup. Hadoop offers several ways to achieve that: one can use its Java API (namely `FileSystem`) or use the `hadoop` command line, in particular the file system shell. However there is no middle ground, one either has to use the (somewhat verbose, full of checked exceptions) API or fall back to the command line, outside the application. SHDP addresses this issue by bridging the two worlds, exposing both the `FileSystem` and the fs shell through an intuitive, easy-to-use Java API. Add your favorite JVM scripting language right inside your Spring for Apache Hadoop application and you have a powerful combination.

3.1 Configuring the file-system

The Hadoop file-system, HDFS, can be accessed in various ways - this section will cover the most popular protocols for interacting with HDFS and their pros and cons. SHDP does not enforce any specific protocol to be used - in fact, as described in this section any `FileSystem` implementation can be used, allowing even other implementations then HDFS to be used.

The table below describes the common HDFS APIs in use:

Table 3.1. HDFS APIs

File System	Comm. Method	Scheme / Prefix	Read / Write	Cross Version
HDFS	RPC	`hdfs://`	Read / Write	Same HDFS version only
HFTP	HTTP	`hftp://`	Read only	Version independent
WebHDFS	HTTP (REST)	`webhdfs://`	Read / Write	Version independent

`hdfs://` protocol should be familiar to most reader - most docs (and in fact the previous chapter as well) mention it. It works out of the box and it's fairly efficient however because it is RPC based, it requires both the client and the Hadoop cluster to share the same version. Upgrading one without the other causes serialization errors meaning the client cannot interact with the cluster. As an alternative one can use `hftp://` which is HTTP-based or its more secure brother `hsftp://` (based on SSL) which gives you a version independent protocol meaning you can use it to interact with clusters with an unknown or different version then that of the client. `hftp` is read only (write operations will fail right away) and it is typically used with `disctp` for reading data. `webhdfs://` is one of the additions in Hadoop 1.0 and is a mixture between `hdfs` and `hftp` protocol - it provides a version-independent, read-write, REST-based protocol which means that you can read and write to/from Hadoop clusters no matter their version. Further more, since `webhdfs://` is backed by a REST APIs, clients in other languages can use it with minimal effort.

Note

Not all file-system work out of the box. For example WebHDFS needs to be enabled first in the cluster (through `dfs.webhdfs.enabled` property) see this document for more information) while the secure `hftp`, `hsftp` requires the SSL configuration (such as certificates) to be specified. More about this (and how to use `hftp/hsftp` for proxying) in this page.

Once the schema has been decided upon, one can specify it through the standard Hadoop configuration, either through the Hadoop configuration files and its properties:

```
<hdp:configuration>
  fs.default.name=webhdfs://localhost
  ...
</hdp:configuration>
```

This instructs Hadoop (and automatically SHDP) what the default, implied file-system is. In SHDP, one can create additional file-systems (potentially to connect to other clusters) and specify a different schema:

```
<!-- manually creates the default SHDP file-system named 'hadoopFs' -->
<hdp:file-system uri="webhdfs://localhost"/>

<!-- create a different FileSystem instance -->
<hdp:file-system id="old-cluster" uri="hftp://old-cluster/"/>
```

As with the rest of the components, the file systems can be injected where needed - such as file shell or inside scripts (see the next section).

3.2 Scripting the Hadoop API

Supported scripting languages

SHDP scripting supports any JSR-223 (also known as `javax.scripting`) compliant scripting engine. Simply add the engine jar to the classpath and the application should be able to find it. Most languages (such as Groovy or JRuby) provide JSR-233 support out of the box; for those that do not see the scripting project that provides various adapters.

Since Hadoop is written in Java, accessing its APIs in a *native* way provides maximum control and flexibility over the interaction with Hadoop. This holds true for working with its files system; in fact all the other tools that one might use are built upon these. The main entry point is the `org.apache.hadoop.fs.FileSystem` abstract class which provides the foundation of most (if not all) of the actual file system implementations out there. Whether one is using a local, remote or distributed store through the `FileSystem` API she can query and manipulate the available resources or create new ones. To do so however, one needs to write Java code, compile the classes and configure them which is somewhat cumbersome especially when performing simple, straight-forward operations (like copy a file or delete a directory).

JVM scripting languages (such as Groovy, JRuby, Jython or Rhino to name just a few) provide a nice solution to the Java language; they run on the JVM, can interact with the Java code with no or few changes or restrictions and have a nicer, simpler, less *ceremonial* syntax; that is, there is no need to define a class or a method - simply write the code that you want to execute and you are done. SHDP combines the two, taking care of the configuration and the infrastructure so one can interact with the Hadoop environment from her language of choice

Let us take a look of a JavaScript example using Rhino (which is part of JDK 6 or higher, meaning one does not need any extra libraries):

```
<beans xmlns="http://www.springframework.org/schema/beans" ...>
  <hdp:configuration .../>

  <hdp:script id="inlined-js" language="javascript">
    importPackage(java.util);

    name = UUID.randomUUID().toString()
    scriptName = "src/test/resources/test.properties"
    // fs - FileSystem instance based on 'hadoopConfiguration' bean
    // call FileSystem#copyFromLocal(Path, Path)
    fs.copyFromLocalFile(scriptName, name)
    // return the file length
    fs.getLength(name)
  </hdp:script>

</beans>
```

The `script` element, part of the SHDP namespace, builds on top of the scripting support in Spring permitting script declarations to be evaluated and declared as normal bean definitions. Further more it automatically exposes Hadoop-specific objects, based on the existing configuration, to the script such as the `FileSystem` (more on that in the next section). As one can see, the script is fairly obvious: it generates a random name (using the `UUID` class from `java.util` package) and the copies a local file into HDFS under the random name. The last line returns the length of the copied file which becomes the value of the declaring bean (in this case `inlined-js`) - note that this might vary based on the scripting engine used.

Note

The attentive reader might have noticed that the arguments passed to the `FileSystem` object are not of type `Path` but rather `String`. To avoid the creation of `Path` object, SHDP uses a wrapper class (`SimplerFileSystem`) which automatically does the conversion so you don't have to. For more information see the implicit variables section.

Note that for inlined scripts, one can use Spring's property placeholder configurer to automatically expand variables at runtime. Using one of the examples before:

```
<beans ...>
  <context:property-placeholder location="classpath:hadoop.properties" />

  <hdp:script language="javascript">
    ...
    tracker=${hd.fs}
    ...
  </hdp:script>
```

```
</beans>
```

Notice how the script above relies on the property placeholder to expand `${hd.fs}` with the values from `hadoop.properties` file available in the classpath.

Using scripts

Inlined scripting is quite handy for doing simple operations and couple with the property expansion is quite a powerful tool that can handle a variety of use cases. However when more logic is required or the script is affected by XML formatting, encoding or syntax restrictions (such as Jython/Python for which white-spaces are important) one should consider externalization. That is rather then declaring the script directly inside the XML, one can declare it in its own file. And speaking of Python, consider the variation of the previous example:

```
<hdp:script location="org/company/basic-script.py"/>
```

The definition does not bring any surprises but do notice there is no need to specify the language (as in the case of a inlined declaration) since script extension (`py`) already provides that information. Just for completeness, the `basic-script.py` looks as follows:

```
from java.util import UUID
from org.apache.hadoop.fs import Path

print "Home dir is " + str(fs.homeDirectory)
print "Work dir is " + str(fs.workingDirectory)
print "/user exists " + str(fs.exists("/user"))

name = UUID.randomUUID().toString()
scriptName = "src/test/resources/test.properties"
fs.copyFromLocalFile(scriptName, name)
print Path(name).makeQualified(fs)
```

3.3 Scripting implicit variables

To ease the interaction of the script with its enclosing context, SHDP binds by default the so-called *implicit* variables. These are:

Table 3.2. Implicit variables

Name	Type	Description
cfg	`org.apache.hadoop.conf.Configuration`	Hadoop Configuration (relies on *hadoopConfiguration* bean or singleton type match)
cl	`java.lang.ClassLoader`	ClassLoader used for executing the script
ctx	`org.springframework.context.ApplicationContext`	Enclosing application context
rl	`org.springframework.io.support.ResourcePatternResolver`	Enclosing application context ResourceLoader
distcp	`org.springframework.data.hadoop.fs.DistributedCopyUtil`	Programmatic access to DistCp

Name	Type	Description
fs	org.apache.hadoop.fs.FileSystem	Hadoop File System (relies on 'hadoop-fs' bean or singleton type match, falls back to creating one based on 'cfg')
fsh	org.springframework.data.hadoop.fs.FsShell	File System Shell, exposing hadoop 'fs' commands as an API
hdfsRL	org.springframework.data.hadoop.io.HdfsResourceLoader	HDFS Resource Loader (relies on 'hadoop-resource-loader' or singleton type match, falls back to creating one automatically based on 'cfg')

Note

If no Hadoop `Configuration` can be detected (either by name `hadoopConfiguration` or type), several log warnings will be made and none of the Hadoop-based variables namely `cfg`, `distcp`, `fs`, `fsh`, `distcp` or `hdfsRL` bound.

As mentioned in the *Description* column, the variables are first looked (either by name or by type) in the application context and, in case they are missing, created on the spot based on the existing configuration. Note that it is possible to override or add new variables to the scripts through the `property` sub-element that can set values or references to other beans:

```
<hdp:script location="org/company/basic-script.js">
   <hdp:property name="foo" value="bar"/>
   <hdp:property name="ref" ref="some-bean"/>
</hdp:script>
```

3.4 File System Shell (FsShell)

A handy utility provided by the Hadoop distribution is the file system shell which allows UNIX-like commands to be executed against HDFS. One can check for the existance of files, delete, move, copy directories or files or setting up permissions. However the utility is only available from the command-line which makes it hard to use it from/inside a Java application. To address this problem, SHDP provides a lightweight, fully embeddable shell, called `FsShell` which mimics most of the commands available from the command line: rather then dealing with the `System.in` or `System.out`, one deals with objects.

Let us take a look of using `FsShell` by building on the previous scripting examples:

```
<hdp:script location="org/company/basic-script.groovy"/>
```

```
name = UUID.randomUUID().toString()
scriptName = "src/test/resources/test.properties"
fs.copyFromLocalFile(scriptName, name)

// use the shell (made available under variable fsh
dir = "script-dir"
if (!fsh.test(dir)) {
   fsh.mkdir(dir); fsh.cp(name, dir); fsh.chmodr(700, dir)
   println "File content is " + fsh.cat(dir + name).toString()
}
println fsh.ls(dir).toString()
```

```
fsh.rmr(dir)
```

As mentioned in the previous section, a `FsShell` instance is automatically created and for configured for scripts, under the name *fsh*. Notice how the entire block relies on the usual commands: `test`, `mkdir`, `cp` and so on. Their semantics are exactly the same as in the command-line version however one has access to a native Java API that returns actual objects (rather then `String`s) making it easy to use them programmatically whether in Java or another language. Further more, the class offers enhanced methods (such as `chmodr` which stands for *recursive* `chmod`) and multiple overloaded methods taking advantage of varargs so that multiple parameters can be specified. Consult the API for more information.

To be as close as possible to the command-line shell, `FsShell` mimics even the messages being displayed. Take a look at line 9 which prints the result of `fsh.cat()`. The method returns a `Collection` of Hadoop `Path` objects (which one can use programatically). However when invoking `toString` on the collection, the same printout as from the command-line shell is being displayed:

```
File content is some text
```

The same goes for the rest of the methods, such as `ls`. The same script in JRuby would look something like this:

```
require 'java'
name = java.util.UUID.randomUUID().to_s
scriptName = "src/test/resources/test.properties"
$fs.copyFromLocalFile(scriptName, name)

# use the shell
dir = "script-dir/"
...
print $fsh.ls(dir).to_s
```

which prints out something like this:

```
drwx------   - user     supergroup          0 2012-01-26 14:08 /user/user/script-dir
-rw-r--r--   3 user     supergroup        344 2012-01-26 14:08 /user/user/script-dir/520cf2f6-a0b6-427e-a23
```

As you can see, not only you can reuse the existing tools and commands with Hadoop inside SHDP, but you can also code against them in various scripting languages. And as you might have noticed, there is no special configuration required - this is automatically inferred from the enclosing application context.

Note

The careful reader might have noticed that besides the syntax, there are some minor differences in how the various langauges interact with the java objects. For example the automatic `toString` call called in Java for doing automatic `String` conversion is not necessarily supported (hence the `to_s` in Ruby or `str` in Python). This is to be expected as each language has its own semantics - for the most part these are easy to pick up but do pay attention to details.

DistCp API

Similar to the `FsShell`, SHDP provides a lightweight, fully embeddable `DistCp` version that builds on top of the `distcp` from the Hadoop distro. The semantics are configuration options are the same

however, one can use it from within an Java application without having to use the command-line. See the API for more information:

```
<hdp:script language="groovy">distcp.copy("${distcp.src}", "${distcp.dst}")</hdp:script>
```

The bean above triggers a distributed copy relying again on Spring's property placeholder variable expansion for its source and destination.

3.5 Scripting Lifecycle

The `script` namespace provides various options to adjust its behaviour depending on the script content. By default the script is executed in a lazy manner - that is when the declaring bean is being referred/used by another entity. One however can change that so that the script gets evaluated at startup through the `run-at-startup` flag (which is by default `false`). Similarily, by default the script gets evaluated every single time the bean is being invoked - that is the script is actually ran every time one refers to it. However for scripts that are expensive and return the same value every time one has various *caching* options, so the evaluation occurs only when needed through the `evaluate` attribute:

Table 3.3. `script` attributes

Name	Values	Description
`run-at-startup`	`false`(default), `true`	Wether the script is executed at startup or on demand (lazy)
`evaluate`	`ALWAYS`(default), `IF_MODIFIED`, `ONCE`	Wether to actually evaluate the script when invoked or used a previous value. `ALWAYS` means evaluate every time, `IF_MODIFIED` evaluate if the backing resource (such as a file) has been modified in the meantime and `ONCE` only one.

3.6 Using the Scripting tasklet

For Spring Batch environments, SHDP provides a dedicated tasklet to execute scripts.

```
<script-tasklet id="script-tasklet">
  <script language="groovy">
    inputPath = "/user/gutenberg/input/word/"
    outputPath = "/user/gutenberg/output/word/"
    if (fsh.test(inputPath)) {
      fsh.rmr(inputPath)
    }
    if (fsh.test(outputPath)) {
      fsh.rmr(outputPath)
    }
    inputFile = "src/main/resources/data/nietzsche-chapter-1.txt"
    fsh.put(inputFile, inputPath)
  </script>
</script-tasklet>
```

The tasklet above embedds the script as a nested element. You can also declare a reference to another script definition, using the script-ref attribute which allows you to externalize the scripting code to an external resource.

```
<script-tasklet id="script-tasklet" script-ref="clean-up"/>
<hdp:script id="clean-up" location="org/company/myapp/clean-up-wordcount.groovy"/>
```

4. Working with HBase

SHDP provides basic configuration for HBase through the `hbase-configuration` namespace element (or its backing `HbaseConfigurationFactoryBean`).

```
<!-- default bean id is 'hbaseConfiguration' that uses the existing 'hadoopCconfiguration' object -->
<hdp:hbase-configuration configuration-ref="hadoopCconfiguration" />
```

The above declaration does more than easily create an HBase configuration object; it will also manage the backing HBase connections: when the application context shuts down, so will any HBase connections opened - this behavior can be adjusted through the `stop-proxy` and `delete-connection` attributes:

```
<!-- delete associated connections but do not stop the proxies -->
<hdp:hbase-configuration stop-proxy="false" delete-connection="true">
  foo=bar
  property=value
</hdp:hbase-configuration>
```

Notice that like with the other elements, one can specify additional properties specific to this configuration. In fact `hbase-configuration` provides the same properties configuration `knobs` as hadoop configuration:

```
<hdp:hbase-configuration properties-ref="some-props-bean" properties-location="classpath:/conf/testing/hbase
```

4.1 Data Access Object (DAO) Support

One of the most popular and powerful feature in Spring Framework is the Data Access Object (or DAO) support. It makes dealing with data access technologies easy and consistent allowing easy switch or interconnection of the aforementioned persistent stores with minimal friction (no worrying about catching exceptions, writing boiler-plate code or handling resource acquisition and disposal). Rather then reiterating here the value proposal of the DAO support, we recommend the DAO section in the Spring Framework reference documentation

SHDP provides the same functionality for Apache HBase through its `org.springframework.data.hadoop.hbase` package: an `HbaseTemplate` along with several callbacks such as `TableCallback`, `RowMapper` and `ResultsExtractor` that remove the low-level, tedious details for finding the HBase table, run the query, prepare the scanner, analyze the results then clean everything up, letting the developer focus on her actual job (users familiar with Spring should find the class/method names quite familiar).

At the core of the DAO support lies `HbaseTemplate` - a high-level abstraction for interacting with HBase. The template requires an HBase configuration, once it's set, the template is thread-safe and can be reused across multiple instances at the same time:

```
// default HBase configuration
<hdp:hbase-configuration/>

// wire hbase configuration (using default name 'hbaseConfiguration') into the template
```

```
<bean id="htemplate" class="org.springframework.data.hadoop.hbase.HbaseTemplate" p:configuration-ref="hbase
```

The template provides generic callbacks, for executing logic against the tables or doing result or row extraction, but also utility methods (the so-called *one-liner*s) for common operations. Below are some examples of how the template usage looks like:

```
// writing to 'MyTable'
template.execute("MyTable", new TableCallback<Object>() {
  @Override
  public Object doInTable(HTable table) throws Throwable {
    Put p = new Put(Bytes.toBytes("SomeRow"));
    p.add(Bytes.toBytes("SomeColumn"), Bytes.toBytes("SomeQualifier"), Bytes.toBytes("AValue"));
    table.put(p);
    return null;
  }
});
```

```
// read each row from 'MyTable'
List<String> rows = template.find("MyTable", "SomeColumn", new RowMapper<String>() {
  @Override
  public String mapRow(Result result, int rowNum) throws Exception {
    return result.toString();
  }
}));
```

The first snippet show-cases the generic `TableCallback` - the most generic of the callbacks, it does the table lookup and resource cleanup so that the user code does not have to. Notice the callback signature - any exception thrown by the HBase API is automatically caught, converted to Spring's DAO exceptions and resource clean-up applied transparently. The second example, displays the dedicated lookup methods - in this case `find` which, as the name implies, finds all the rows matching the given criteria and allows user code to be executed against each of them (typically for doing some sort of type conversion of mapping). If the entire result is required, then one can use `ResultsExtractor` instead of `RowMapper`.

Besides the template, the package offers support for automatically binding HBase table to the current thread through `HbaseInterceptor` and `HbaseSynchronizationManager`. That is, each class that performs DAO operations on HBase can be *wrapped* by `HbaseInterceptor` so that each table in use, once found, is bound to the thread so any subsequent call to it avoids the lookup. Once the call ends, the table is automatically closed so there is no leakage between requests. Please refer to the Javadocs for more information.

5. Hive integration

When working with http://hive.apache.org from a Java environment, one can choose between the Thrift client or using the Hive JDBC-like driver. Both have their pros and cons but no matter the choice, Spring and SHDP supports both of them.

5.1 Starting a Hive Server

SHDP provides a dedicated namespace element for starting a Hive server as a Thrift service (only when using Hive 0.8 or higher). Simply specify the host, the port (the defaults are `localhost` and `10000` respectively) and you're good to go:

```
<!-- by default, the definition name is 'hive-server' -->
<hdp:hive-server host="some-other-host" port="10001" />
```

If needed the Hadoop configuration can be passed in or additional properties specified. In fact `hiver-server` provides the same properties configuration `knobs` as hadoop configuration:
:

```
<hdp:hive-server host="some-other-host" port="10001" properties-location="classpath:hive-dev.properties" co
  someproperty=somevalue
  hive.exec.scratchdir=/tmp/mydir
</hdp:hive-server>
```

The Hive server is bound to the enclosing application context life-cycle, that is it will automatically startup and shutdown along-side the application context.

5.2 Using the Hive Thrift Client

Similar to the server, SHDP provides a dedicated namespace element for configuring a Hive client (that is Hive accessing a server node through the Thrift). Likewise, simply specify the host, the port (the defaults are `localhost` and `10000` respectively) and you're done:

```
<!-- by default, the definition name is 'hive-client' -->
<hdp:hive-client host="some-other-host" port="10001" />
```

Just as well, the Hive client is bound to the enclosing application context life-cycle; it will automatically startup and shutdown along-side the application context. Further more, the client definition also allows Hive scripts ()either declared inlined or externally) to be executed at startup, once the client connects; this quite useful for doing Hive specific initialization:

```
<hive-client host="some-host" port="some-port" xmlns="http://www.springframework.org/schema/hadoop">
   <hdp:script>
     DROP TABLE IF EXITS testHiveBatchTable;
     CREATE TABLE testHiveBatchTable (key int, value string);
   </hdp:script>
   <hdp:script location="classpath:org/company/hive/script.q" />
</hive-client> />
```

5.3 Using the Hive JDBC Client

Another attractive option for accessing Hive is through its JDBC driver. This exposes Hive through the JDBC API meaning one can use the standard API or its derived utilities to interact with Hive, such as the rich JDBC support in Spring Framework.

Warning

Note that the JDBC driver is a work-in-progress and not all the JDBC features are available (and probably never will since Hive cannot support all of them as it is not the typical relational database). Do read the official documentation and examples.

SHDP does not offer any dedicated support for the JDBC integration - Spring Framework itself provides the needed tools; simply configure Hive as you would with any other JDBC `Driver`:

```
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:c="http://www.springframework.org/schema/c"
 xmlns:context="http://www.springframework.org/schema/context"
 xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-c

    <!-- basic Hive driver bean -->
    <bean id="hive-driver" class="org.apache.hadoop.hive.jdbc.HiveDriver"/>

    <!-- wrapping a basic datasource around the driver -->
    <!-- notice the 'c:' namespace (available in Spring 3.1+) for inlining constructor arguments,
         in this case the url (default is 'jdbc:hive://localhost:10000/default') -->
    <bean id="hive-ds" class="org.springframework.jdbc.datasource.SimpleDriverDataSource"
       c:driver-ref="hive-driver" c:url="${hive.url}"/>

    <!-- standard JdbcTemplate declaration -->
    <bean id="template" class="org.springframework.jdbc.core.JdbcTemplate" c:data-source-ref="hive-ds"/>

    <context:property-placeholder location="hive.properties"/>
</beans>
```

And that is it! Following the example above, one can use the `hive-ds` `DataSource` bean to manually get a hold of `Connection`s or better yet, use Spring's `JdbcTemplate` as in the example above.

5.4 Using the Hive tasklet

For Spring Batch environments, SHDP provides a dedicated tasklet to execute Hive queries, on demand, as part of a batch or workflow. The declaration is pretty straight forward:

```
<hdp:hive-tasklet id="hive-script">
   <hdp:script>
     DROP TABLE IF EXITS testHiveBatchTable;
     CREATE TABLE testHiveBatchTable (key int, value string);
   </hdp:script>
   <hdp:script location="classpath:org/company/hive/script.q" />
</hdp:hive-tasklet>
```

The tasklet above executes two scripts - one declared as part of the bean definition followed by another located on the classpath.

6. Pig support

For Pig users, SHDP provides easy creation and configuration of `PigServer` instances for registering and executing scripts either locally or remotely. In its simplest form, the declaration looks as follows:

```
<hdp:pig />
```

This will create a `PigServer` instance, named `hadoop-pig`, configured with a default `PigContext`, executing scripts in `MapReduce` mode. In typical scenarios however, one might want to connect to a remote Hadoop tracker and register some scripts automatically so let us take a look of how the configuration might look like:

```
<pig exec-type="LOCAL" job-name="pig-script" configuration-ref="hadoopConfiguration" properties-location="pi
   xmlns="http://www.springframework.org/schema/hadoop">
     source=${pig.script.src}
   <script location="org/company/pig/script.pig">
     <arguments>electric=sea</arguments>
   </script>
   <script>
     A = LOAD 'src/test/resources/logs/apache_access.log' USING PigStorage() AS (name:chararray, age:int);
     B = FOREACH A GENERATE name;
     DUMP B;
   </script>
</pig> />
```

The example exposes quite a few options so let us review them one by one. First the top-level pig definition configures the pig instance: the execution type, the Hadoop configuration used and the job name. Notice that additional properties can be specified (either by declaring them inlined or/and loading them from an external file) - in fact, `<hdp:pig/>` just like the rest of the libraries configuration elements, supports common properties attributes as described in the hadoop configuration section.

The definition contains also two scripts: `script.pig` (read from the classpath) to which one pair of arguments, relevant to the script, is passed (notice the use of property placeholder) but also an inlined script, declared as part of the definition, without any arguments.

As you can tell, the `pig` namespace offers several options pertaining to Pig configuration. And, as with the other Hadoop-related integration, the underlying `PigServer` is bound to the enclosing application context life-cycle; that is, it will automatically start and stop along-side the application so one does not have to worry about its management.

6.1 Using the Pig tasklet

For Spring Batch environments, SHDP provides a dedicated tasklet to execute Pig queries, on demand, as part of a batch or workflow. The declaration is pretty straight forward:

```
<hdp:pig-tasklet id="pig-script">
   <hdp:script location="org/company/pig/handsome.pig" />
</hdp:pig-tasklet>
```

The syntax of the scripts declaration is similar to that of the `pig` namespace.

7. Cascading integration

SHDP provides basic support for Cascading library through the `org.springframework.data.hadoop.cascading` package - one can create `Flow`s or `Cascade`s, either through XML or/and Java and execute them, either in a simplistic manner or as part of a Spring Batch job. In addition, dedicated `Tap`s for Spring environments are available.

As Cascading is aimed at code configuration, typically one would configure the library programatically. This type of configuration is supported through Spring's `@Configuration` and `@Bean` (see this chapter for more information). In short one use Java code (or any JVM language for that matter) to create beans. Below is an example of using that to create various Cascading components (do refer to the Cascading examples for more context):

```
@Configuration
public class CascadingAnalysisConfig {
    // fields that act as placeholders for externalized values
    @Value("${cascade.sec}") private String sec;
    @Value("${cascade.min}") private String min;

    @Bean public Pipe tsPipe() {
        DateParser dateParser = new DateParser(new Fields("ts"), "dd/MMM/yyyy:HH:mm:ss Z");
        return new Each("arrival rate", new Fields("time"), dateParser);
    }

    @Bean public Pipe tsCountPipe() {
        Pipe tsCountPipe = new Pipe("tsCount", tsPipe());
        tsCountPipe = new GroupBy(tsCountPipe, new Fields("ts"));
        return new Every(tsCountPipe, Fields.GROUP, new Count());
    }

    @Bean public Pipe tmCountPipe() {
        Pipe tmPipe = new Each(tsPipe(),
                new ExpressionFunction(new Fields("tm"), "ts - (ts % (60 * 1000))", long.class));
        Pipe tmCountPipe = new Pipe("tmCount", tmPipe);
        tmCountPipe = new GroupBy(tmCountPipe, new Fields("tm"));
        return new Every(tmCountPipe, Fields.GROUP, new Count());
    }

    @Bean public Map<String, Tap> sinks(){
        Tap tsSinkTap = new Hfs(new TextLine(), sec);
        Tap tmSinkTap = new Hfs(new TextLine(), min);
        return Cascades.tapsMap(Pipe.pipes(tsCountPipe(), tmCountPipe()), Tap.taps(tsSinkTap, tmSinkTap));
    }

    @Bean public String regex() {
        return "^([^ ]*) +[^ ]* +[^ ]* +\\[([^]]*)\\] +\\\"([^ ]*) ([^ ]*) [^ ]*\\\" ([^ ]*) ([^ ]*).*$";
    }

    @Bean public Fields fields() {
        return new Fields("ip", "time", "method", "event", "status", "size");
    }
}
```

The class above creates several objects (all part of the Cascading package) (named after the methods) which can be injected or wired just like any other bean (notice how the wiring is done between the beans

by point to their methods). One can mix and match (if needed) code and XML configurations inside the same application:

```
<!-- code configuration class -->
<bean class="org.springframework.data.hadoop.cascading.CascadingAnalysisConfig"/>

<!-- Tap created through XML rather then code (using Spring's 3.1 c: namespace)-->
<bean id="tap" class="cascading.tap.hadoop.Hfs" c:fields-ref="fields" c:string-path-value="${cascade.input}"

<bean id="cascade" class="org.springframework.data.hadoop.cascading.CascadeFactoryBean" p:configuration-ref=
  <property name="flows"><list>
    <bean class="org.springframework.data.hadoop.cascading.HadoopFlowFactoryBean"
      p:configuration-ref="hadoopConfiguration" p:source-ref="tap" p:sinks-ref="sinks">
     <property name="tails"><list>
         <ref bean="tsCountPipe"/>
         <ref bean="tmCountPipe"/>
     </list></property>
    </bean>
  </list></property>
</bean>

<bean id="cascade-runner" class="org.springframework.data.hadoop.cascading.CascadeRunner" p:unit-of-work-re
```

The XML above, whose main purpose is to illustrate possible ways of configuring, uses SHDP classes to create a `Cascade` with one nested `Flow` using the taps and sinks configured by the code class. Additionally it also shows how the cascade is ran (through `CascadeRunner`).

Whether XML or Java config is better is up to the user and is usually based on the type of the configuration required. Java config suits Cascading better but note that the `FactoryBean`s above handle the life-cycle and some default configuration for both the `Flow` and `Cascade` object. Either way, whatever option is used, SHDP fully supports it.

7.1 Using the Cascading tasklet

For Spring Batch environments, SHDP provides a dedicated tasklet (similar to `CascadeRunner` above) for executing `Cascade` or `Flow` instances, on demand, as part of a batch or workflow. The declaration is pretty straight forward:

```
<bean id="cascade-tasklet" class="org.springframework.data.hadoop.cascading.CascadeTasklet" p:unit-of-work-
```

7.2 Using Scalding

There are quite a number of DSLs built on top of Cascading, most noteably Cascalog (written in Clojure) and Scalding (written in Scala). This documentation will cover Scalding however the same concepts can be applied across the board to the DSLs.

As with the rest of the DSLs, Scalding offers a simplified, fluent syntax for creating units of code that built on top of Cascading. This in turn translate to Map Reduce jobs that get executed on Hadoop. Once compiled, the DSL gets translated into actual JVM classes that get executed by Scalding through its own `Tool` instance (namely `com.twitter.scalding.Tool`). One has the option or either deploy the Scalding jobs directly (by invoking the aforementioned `Tool`) or use Scalding's `scald.rb` script which does the same thing based on the various attributes passed to it. Both approaches can be used in

SHDP, the former through the Tool support (described below) and the latter by invoking the `scald.rb` script directly through the scripting feature.

For example, to run the tutorial examples (say Tutorial1), one can issue the following command:

```
scripts/scald.rb --local tutorial/Tutorial1.scala
```

which compiles Tutorial1, creates a bundled jar and runs it on a local Hadoop instance. When using the `Tool` support, the compilation and the library provisioning are external tasks (just as in the case of typical Hadoop jobs). The SHDP configuration to run the tutorial looks as follows:

```
<!-- the tool automatically is injected with 'hadoopConfiguration' -->
<hdp:tool-runner id="scalding" tool-class="com.twitter.scalding.Tool">
   <hdp:arg value="tutorial/Tutorial1"/>
   <hdp:arg value="--local"/>
</hdp:tool-runner>
```

7.3 Spring-specific local `Taps`

Why only local `Tap`?

Because Hadoop is designed a distributed file-system (HDFS) and splitable resources. Non-HDFS resources tend to not be cluster friendly: for example don't offer any notion of node locality, true chucking or even scalability (as there are no copies, partial or not made). These being said, the team is pursuing certain approaches to see whether they are viable or not. Feedback is of course welcome.

Besides dedicated configuration support, SHDP also provides *read-only* `Tap` implementations useful inside Spring environments. Currently they are meant for *local* use only such as testing or single-node Hadoop setups.

The `Taps` in `org.springframework.data.hadoop.cascading.tap.local` tap (pun intended) into the rich resource support from Spring Framework and Spring Integration allowing data to flow easily in and out of a Cascading flow.

Below is a list of the type of `Taps` available and their backing support.

Table 7.1. Local `Taps`

Tap Name	Tap Type	Backing Resource	Resource Description
`ResourceTap`	Source	Spring `Resource`	classpath, file-system, URL-based or even in-memory content
`MessageSourceTap`	Source	Spring Integration MessageSource	Inbound adapter for anything from arbitrary streams, FTP or JDBC to RSS/Atom and Twitter

Tap Name	Tap Type	Backing Resource	Resource Description
`MessageHandlerTap`	Sink	Spring Integration MessageHandler	The opposite of `MessageSourceTap`: Outbound adapter for Files, JMS, TCP, etc...

Note the `Tap`s do not require any special configuration and are fully compatible with the existing Cascading local `Scheme`s. To wit:

```
<bean id="cp-txt-files" class="org.springframework.data.hadoop.cascading.tap.local.ResourceTap">
 <constructor-arg><bean class="cascading.scheme.local.TextLine"/></constructor-arg>
 <constructor-arg><value>classpath:/data/*.txt</value></constructor-arg>
</bean>
```

The `Tap` above, reads all the text files in the classpath, under `data` folder, through Cascading `TextLine`. Simply wire that to a Cascading flow (as described in the previous section) and you are good to go.

8. Security Support

Spring for Apache Hadoop is aware of the security constraints of the running Hadoop environment and allows its components to be configured as such. For clarity, this document breaks down *security* into HDFS permissions and user impersonation (also known as *secure* Hadoop). The rest of this document discusses each component and the impact (and usage) it has on the various SHDP features.

8.1 HDFS permissions

HDFS layer provides file permissions designed to be similar to those present in *nix OS. The official guide explains the major components but in short, the access for each file (whether it's for reading, writing or in case of directories accessing) can be restricted to a certain users or groups. Depending on the user identity (which is typically based on the host operating system), code executing against the Hadoop cluster can see or/and interact with the file-system based on these permissions. Do note that each HDFS or `FileSystem` implementation can have slightly different semantics or implementation.

SHDP obeys the HDFS permissions, using the identity of the current user (by default) for interacting with the file system. In particular, the `HdfsResourceLoader` considers when doing pattern matching, only the files that its suppose to *see* and does not perform any privileged action. It is possible however to specify a different user, meaning the `ResourceLoader` interacts with HDFS using that user's rights - however this obeys the user impersonation rules. When using different users, it is recommended to create separate `ResourceLoader` instances (one per user) instead of assigning additional permissions or groups to one user - this makes it easier to manage and wire the different HDFS *views* without having to modify the ACLs. Note however that when using impersonation, the `ResourceLoader` might (and will typically) return *restricted* files that might not be consumed or seen by the callee.

8.2 User impersonation (Kerberos)

Securing a Hadoop cluster can be a difficult task - each machine can have a different set of users and groups, each with different passwords. Hadoop relies on Kerberos, a ticket-based protocol for allowing nodes to communicate over a non-secure network to prove their identity to one another in a secure manner. Unfortunately there is not a lot of documentation on this topic out there however the there are some resources to get you started.

SHDP does not require any extra configuration - it simply obeys the security system in place. By default, when running inside a *secure* Hadoop, SHDP uses the current user (as expected). It also supports *user impersonation*, that is, interacting with the Hadoop cluster with a different identity (this allows a superuser to submit job or access hdfs on behalf of another user in a secure way, without *leaking* permissions). The major MapReduce components, such as `job`, `streaming` and `tool` as well as `pig` support user impersonation through the `user` attribute. By default, this property is empty, meaning the current user is used - however one can specify the different identity (also known as *ugi*) to be used by the target component:

```
<hdp:job id="jobFromJoe" user="joe" .../>
```

Note that the user running the application (or the current user) must have the proper kerberos credentials to be able to impersonate the target user (in this case *joe*).

Part III. Developing Spring for Apache Hadoop Applications

This section provides some guidance on how one can use the Spring for Apache Hadoop project in conjunction with other Spring projects, starting with the Spring Framework itself, then Spring Batch, and then Spring Integration.

9. Guidance and Examples

Spring for Apache Hadoop provides integration with the Spring Framework to create and run Hadoop MapReduce, Hive, and Pig jobs as well as work with HDFS and HBase. If you have simple needs to work with Hadoop, including basic scheduling, you can add the Spring for Apache Hadoop namespace to your Spring based project and get going quickly using Hadoop.

As the complexity of your Hadoop application increases, you may want to use Spring Batch to regin in the complexity of developing a large Hadoop application. Spring Batch provides an extension to the Spring programming model to support common batch job scenarios characterized by the processing of large amounts of data from flat files, databases and messaging systems. It also provides a workflow style processing model, persistent tracking of steps within the workflow, event notification, as well as administrative functionality to start/stop/restart a workflow. As Spring Batch was designed to be extended, Spring for Apache Hadoop plugs into those extensibilty points, allowing for Hadoop related processing to be a first class citizen in the Spring Batch processing model.

Another project of interest to Hadoop developers is Spring Integration. Spring Integration provides an extension of the Spring programming model to support the well-known Enterprise Integration Patterns. It enables lightweight messaging *within* Spring-based applications and supports integration with external systems via declarative adapters. These adapters are of particular interest to Hadoop developers, as they directly support common Hadoop use-cases such as polling a directory or FTP folder for the presence of a file or group of files. Then once the files are present, a message is sent internal to the application to do additional processing. This additional processing can be calling a Hadoop MapReduce job directly or starting a more complex Spring Batch based workflow. Similarly, a step in a Spring Batch workflow can invoke functionality in Spring Integration, for example to send a message though an email adapter.

Not matter if you use the Spring Batch project with the Spring Framework by itself or with additional extentions such as Spring Batch and Spring Integration that focus on a particular domain, you will you benefit from the core values that Spring projects bring to the table, namely enabling modularity, reuse and extensive support for unit and integration testing.

9.1 Scheduling

Spring Batch integrates with a variety of job schedulers and is not a scheduling framework. There are many good enterprise schedulers available in both the commercial and open source spaces such as Quartz, Tivoli, Control-M, etc. It is intended to work in conjunction with a scheduler, not replace a scheduler. As a lightweight solution, you can use Spring's built in scheduling support that will give you cron like and other basic scheduling trigger functionality. See the Task Execution and Scheduling documention for more info. A middle ground it to use Spring's Quartz integration, see Using the OpenSymphony Quartz Scheduler for more information. The Spring Batch distribution contains an example, but this documentation will be updated to provide some more directed examples with Hadoop, check for updates on the main web site of Spring for Apache Hadoop.

9.2 Batch Job Listeners

Spring Batch let's you attached listeners at the job and step levels to perform additional processing. For example, at the end of a job you can perform some notification or perhaps even start another Spring Batch Jobs. As a brief example, implement the interface JobExecutionListener and configure it into the Spring Batch job as shown below.

```
<batch:job id="job1">
  <batch:step id="import" next="wordcount">
    <batch:tasklet ref="script-tasklet"/>
  </batch:step>

  <batch:step id="wordcount">
    <batch:tasklet ref="wordcount-tasklet" />
  </batch:step>

  <batch:listeners>
   <batch:listener ref="simpleNotificatonListener"/>
  </batch:listeners>

</batch:job>

<bean id="simpleNotificatonListener" class="com.mycompany.myapp.SimpleNotificationListener"/>
```

Part IV. Spring for Apache Hadoop sample applications

Document structure

This part of the reference documentation covers the sample applications included with Spring for Apache Hadoop that demonstrate features in a code centric manner.

Chapter 11, *Wordcount sample using the Spring Framework* describes a standard Spring application that executes the wordcount map-reduce job

Chapter 12, *Wordcount sample using Spring Batch* describes a Batch application that executes the wordcount map-reduce job

10. Sample prerequisites

In order to run the examples you need a working Hadoop installation and JDK 1.6+ installed on the machine that runs the samples.

For instructions on installing Hadoop refer to your distribution documentation or you can refer to the Getting Started section of this for instructions based off the Apache download distribution.

11. Wordcount sample using the Spring Framework

Please read the sample prerequistes before following these instructions.

11.1 Introduction

This sample demonstrates how to execute a MapReduce application and a script that interacts with HDFS inside a Spring based application. It does not use spring Batch or Spring Integration.

The example code is located in the distribution directory `<spring-hadoop-install-dir>/samples/wordcount`.

12. Wordcount sample using Spring Batch

Please read the sample prerequistes before following these instructions.

12.1 Introduction

This sample demonstrates how to execute the wordcount example in the context of a Spring Batch application. It serves as a starting point that will be expanded upon in other samples. The sample code is located in the distribution directory `<spring-hadoop-install-dir>/samples/batch-wordcount`

The sample uses the Spring for Apache Hadoop namespace to define a Spring Batch Tasklet that runs a Hadoop job. A Spring Batch Job (not to be confused with a Hadoop job) combines multiple steps together to create a flow of execution, a small workflow. Each step in the flow does some processing which can be as complex or as simple as your require. The configuration of the flow from one step to another can very simple, a linear sequence of steps, or complex using conditional and programmatic branching as well as sequential and parallel step execution. A Spring Batch Tasklet is the abstraction that represents the processing for a Step and is an extensible part of Spring Batch. You can write your own implementation of a Tasklet to perform arbitrary processing, but often you configure existing Tasklets provided by Spring Batch and Spring Hadoop.

Spring Batch provides Tasklets for reading, writing, and processing data from flat files, databases, messaging systems and executing system (shell) commands. Spring for Apache Hadoop provides Tasklets for running Hadoop MapReduce, Streaming, Hive, and Pig jobs as well as executing script files that have build in support for ease of use interaction with HDFS.

12.2 Basic Spring for Apache Hadoop configuration

The part of the configuration file that defines the Spring Batch Job flow is shown below and can be found in the file `wordcount-context.xml`. The elements `<batch:job/>, <batch:step/>, <batch:tasklet>` come from the XML Schema for Spring Batch

```
<batch:job id="job1">
    <batch:step id="import" next="wordcount">
      <batch:tasklet ref="script-tasklet"/>
    </batch:step>

    <batch:step id="wordcount">
       <batch:tasklet ref="wordcount-tasklet" />
    </batch:step>
</batch:job>
```

This configuration defines a Spring Batch Job named "job1" that contains two steps executed sequentially. The first one prepares HDFS with sample data and the second runs the Hadoop wordcount mapreduce job. The tasklet's reference to "script-tasklet" and "wordcount-tasklet" definitions that will be shown a little later.

The Spring Source Toolsuite is a free Eclipse-powered development environment which provides a nice visualization and authoring help for Spring Batch workflows as shown below.

The script tasklet shown below uses Groovy to remove any data that is in the input or output directories and puts the file "nietzsche-chapter-1.txt" into HDFS.

```
<script-tasklet id="script-tasklet">
  <script language="groovy">
    inputPath = "${wordcount.input.path:/user/gutenberg/input/word/}"
    outputPath = "${wordcount.output.path:/user/gutenberg/output/word/}"
    if (fsh.test(inputPath)) {
      fsh.rmr(inputPath)
    }
    if (fsh.test(outputPath)) {
      fsh.rmr(outputPath)
    }
    inputFile = "src/main/resources/data/nietzsche-chapter-1.txt"
    fsh.put(inputFile, inputPath)
  </script>
</script-tasklet>
```

The script makes use of the predefined variable fsh, which is the embedded Filesystem Shell that Spring for Apache Hadoop provides. It also uses Spring's Property Placeholder functionality so that the input and out paths can be configured external to the application, for example using property files. The syntax for variables recognized by Spring's Property Placeholder is `${key:defaultValue}`, so in this case `/user/gutenberg/input/word` and `/user/gutenberg/output/word` are the default input and output paths. Note you can also use Spring's Expression Language to configure values in the script or in other XML definitions.

The configuration of the tasklet to execute the Hadoop MapReduce jobs is shown below.

```
<hdp:tasklet id="hadoop-tasklet" job-ref="mr-job"/>

<job id="wordcount-job"
  input-path="${wordcount.input.path:/user/gutenberg/input/word/}"
  output-path="${wordcount.output.path:/user/gutenberg/output/word/}"
  mapper="org.apache.hadoop.examples.WordCount.TokenizerMapper"
  reducer="org.apache.hadoop.examples.WordCount.IntSumReducer" />
```

The `<hdp:tasklet>` and `<hdp:job/>` elements are from Spring Haddop XML Schema. The hadoop tasklet is the bridge between the Spring Batch world and the Hadoop world. The hadoop tasklet in turn refers to your map reduce job. Various common properties of a map reduce job can be set, such as the mapper and reducer. The section Creating a Hadoop Job describes the additional elements you can configure in the XML.

Note

If you look at the JavaDocs for the org.apache.hadoop.examples package (here), you can see the Mapper and Reducer class names for many examples you may have previously used from the hadoop command line runner.

The `configuration-ref` element in the job definition refers to common Hadoop configuration information that can be shared across many jobs. It is defined in the file `hadoop-context.xml` and is shown below

```
<!--  default id is 'hadoopConfiguration' -->
<hdp:configuration register-url-handler="false">
  fs.default.name=${hd.fs}
</hdp:configuration>
```

As mentioned before, as this is a configuration file processed by the Spring container, it supports variable substitution through the use of `${var}` style variables. In this case the location for HDFS is parameterized and no default value is provided. The property file `hadoop.properties` contains the definition of the hd.fs variable, change the value if you want to refer to a different name node location.

```
hd.fs=hdfs://localhost:9000
```

The entire application is put together in the configuration file `launch-context.xml`, shown below.

```
<!-- where to read externalized configuration values -->
<context:property-placeholder location="classpath:batch.properties,classpath:hadoop.properties"
  ignore-resource-not-found="true" ignore-unresolvable="true" />

<!-- simple base configuration for batch components, e.g. JobRepository -->
<import resource="classpath:/META-INF/spring/batch-common.xml" />
<!-- shared hadoop configuration -->
<import resource="classpath:/META-INF/spring/hadoop-context.xml" />
<!-- word count workflow -->
<import resource="classpath:/META-INF/spring/wordcount-context.xml" />
```

12.3 Build and run the sample application

In the directory `<spring-hadoop-install-dir>/samples/batch-wordcount` build and run the sample

```
$ ../../gradlew
```

If this is the first time you are using gradlew, it will download the Gradle build tool and all necessary dependencies to run the sample.

Note

If you run into some issues, drop us a line in the Spring Forums.

12.4 Run the sample application as a standlone Java application

You can use Gradle to export all required dependencies into a directory and create a shell script to run the application. To do this execute the command

```
$ ../../gradlew installApp
```

This places the shell scripts and dependencies under the `build/install/batch-wordcount` directory. You can zip up that directory and share the application with others.

The main Java class used is part of Spring Batch. The class is `org.springframework.batch.core.launch.support.CommandLineJobRunner`. This main app requires you to specify at least a Spring configuration file and a job instance name. You can read the `CommandLineJobRunner` JavaDocs for more information as well as this section in the reference docs for Spring Batch to find out more of what command line options it supports.

```
$ ./build/install/wordcount/bin/wordcount classpath:/launch-context.xml job1
```

You can then verify the output from work count is present and cat it to the screen

```
$ hadoop dfs -ls /user/gutenberg/output/word
Warning: $HADOOP_HOME is deprecated.

Found 2 items
-rw-r--r--   3 mpollack supergroup          0 2012-01-31 19:29 /user/gutenberg/output/word/_SUCCESS
-rw-r--r--   3 mpollack supergroup     918472 2012-01-31 19:29 /user/gutenberg/output/word/part-r-00000

$ hadoop dfs -cat /user/gutenberg/output/word/part-r-00000 | more
Warning: $HADOOP_HOME is deprecated.

"'Spells 1
"'army'  1
"(1)     1
"(Lo)cra"1
"13      4
"1490    1
"1498,"  1
"35"     1
"40,"    1
"A       9
"AS-IS". 1
"AWAY    1
"A_      1
"Abide   1
"About   1
```

Part V. Other Resources

In addition to this reference documentation, there are a number of other resources that may help you learn how to use Hadoop and Spring framework. These additional, third-party resources are enumerated in this section.

13. Useful Links

- *Spring for Apache Hadoop* - Home Page
- *Spring Data* - Home Page
- *SpringSource* - Blog
- *Hadoop* - Home Page
- *Spring Hadoop Team on Twitter* - Costin

Part VI. Appendices

Appendix A. Spring for Apache Hadoop Schema

Spring for Apache Hadoop Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.springframework.org/schema/hadoop"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:beans="http://www.springframework.org/schema/beans"
 xmlns:tool="http://www.springframework.org/schema/tool"
 targetNamespace="http://www.springframework.org/schema/hadoop"
 elementFormDefault="qualified"
 attributeFormDefault="unqualified"
 version="1.0.0.M2">

 <xsd:import namespace="http://www.springframework.org/schema/beans" />
 <xsd:import namespace="http://www.springframework.org/schema/tool" />

 <xsd:annotation>
  <xsd:documentation><![CDATA[
Defines the configuration elements for Spring Data Hadoop.
  ]]></xsd:documentation>
 </xsd:annotation>

 <xsd:element name="tasklet">
  <xsd:annotation>
     <xsd:documentation><![CDATA[
Defines a Spring Batch tasklet for Hadoop Jobs.
    ]]>
          </xsd:documentation>
       <xsd:appinfo>
         <tool:annotation>
           <tool:exports type="org.springframework.data.hadoop.mapreduce.HadoopTasklet"/>
         </tool:annotation>
       </xsd:appinfo>
  </xsd:annotation>
  <xsd:complexType>
  <!-- the job reference -->
  <xsd:attribute name="job-ref">
   <xsd:annotation>
    <xsd:documentation source="java:org.apache.hadoop.mapreduce.Job"><![CDATA[
Hadoop Job]]></xsd:documentation>
    <xsd:appinfo>
     <tool:annotation kind="ref">
      <tool:expected-type type="org.apache.hadoop.mapreduce.Job" />
     </tool:annotation>
    </xsd:appinfo>
   </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="wait-for-job" type="xsd:string" use="optional" default="true">
   <xsd:annotation>
    <xsd:documentation><![CDATA[
Whether to synchronously wait for the job to finish (the default) or not.
    ]]></xsd:documentation>
   </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="id" type="xsd:ID" use="required" />
```

```
  <xsd:attribute name="scope" type="xsd:string" use="optional" />
 </xsd:complexType>
 </xsd:element>

 <!-- common attributes shared by properties based configurations
   NOT meant for extensibility - do NOT rely on this type as it might be removed in the future -->
 <xsd:complexType name="propertiesConfigurableType" mixed="true">
  <xsd:attribute name="properties-ref" type="xsd:string">
   <xsd:annotation>
    <xsd:documentation><![CDATA[
Reference to a Properties object.
    ]]></xsd:documentation>
    <xsd:appinfo>
     <tool:annotation kind="ref">
      <tool:expected-type type="java.util.Properties" />
     </tool:annotation>
    </xsd:appinfo>
   </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="properties-location" type="xsd:string">
   <xsd:annotation>
    <xsd:documentation><![CDATA[
Properties location(s). Multiple locations can be specified using comma (,) as a separator.
    ]]></xsd:documentation>
   </xsd:annotation>
  </xsd:attribute>
 </xsd:complexType>

 <xsd:element name="configuration">
  <xsd:annotation>
     <xsd:documentation><![CDATA[
Defines a Hadoop Configuration.
    ]]></xsd:documentation>
       <xsd:appinfo>
         <tool:annotation>
           <tool:exports type="org.apache.hadoop.conf.Configuration"/>
         </tool:annotation>
       </xsd:appinfo>
  </xsd:annotation>
  <xsd:complexType mixed="true">
  <xsd:complexContent>
  <xsd:extension base="propertiesConfigurableType">
    <xsd:attribute name="id" type="xsd:ID" use="optional">
    <xsd:annotation>
    <xsd:documentation><![CDATA[
Bean id (default is "hadoopConfiguration").
    ]]></xsd:documentation>
    </xsd:annotation>
    </xsd:attribute>
    <xsd:attribute name="configuration-ref">
    <xsd:annotation>
      <xsd:documentation source="java:org.apache.hadoop.conf.Configuration"><![CDATA[
Reference to another Hadoop configuration (useful for chaining)]]></xsd:documentation>
      <xsd:appinfo>
       <tool:annotation kind="ref">
        <tool:expected-type type="org.apache.hadoop.conf.Configuration" />
       </tool:annotation>
      </xsd:appinfo>
     </xsd:annotation>
    </xsd:attribute>
    <xsd:attribute name="resources">
    <xsd:annotation>
```

```
      <xsd:documentation source="java:org.springframework.core.io.Resource"><![CDATA[
Hadoop Configuration resources. Multiple resources can be specified, using comma (,) as a separator.]]></xs
      <xsd:appinfo>
       <tool:annotation kind="direct">
        <tool:expected-type type="org.springframework.core.io.Resource[]" />
       </tool:annotation>
      </xsd:appinfo>
     </xsd:annotation>
    </xsd:attribute>
    <xsd:attribute name="register-url-handler" use="optional" default="false">
    <xsd:annotation>
    <xsd:documentation><![CDATA[
Registers an HDFS url handler in the running VM. Note that this operation can be executed at most once
in a given JVM hence the default is false.
    ]]></xsd:documentation>
    </xsd:annotation>
    </xsd:attribute>
  </xsd:extension>
  </xsd:complexContent>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="file-system">
  <xsd:annotation>
     <xsd:documentation><![CDATA[
Defines a HDFS file system.
    ]]>
          </xsd:documentation>
       <xsd:appinfo>
         <tool:annotation>
           <tool:exports type="org.apache.hadoop.fs.FileSystem"/>
         </tool:annotation>
       </xsd:appinfo>
  </xsd:annotation>
  <xsd:complexType>
  <xsd:attribute name="id" type="xsd:ID" use="optional">
  <xsd:annotation>
  <xsd:documentation><![CDATA[
Bean id (default is "hadoopFs").
  ]]></xsd:documentation>
  </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="configuration-ref" use="optional" default="hadoopConfiguration">
  <xsd:annotation>
    <xsd:documentation source="java:org.apache.hadoop.conf.Configuration"><![CDATA[
Reference to the Hadoop Configuration. Defaults to 'hadoopConfiguration'.]]></xsd:documentation>
    <xsd:appinfo>
     <tool:annotation kind="ref">
      <tool:expected-type type="org.apache.hadoop.conf.Configuration" />
     </tool:annotation>
    </xsd:appinfo>
   </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="uri" use="optional">
  <xsd:annotation>
  <xsd:documentation><![CDATA[
The underlying HDFS system URI (by default the configuration settings will be used).
  ]]></xsd:documentation>
  </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="user" type="xsd:string">
   <xsd:annotation>
```

```
   <xsd:documentation><![CDATA[
The security user (ugi) to use for impersonation at runtime.
    ]]></xsd:documentation>
   </xsd:annotation>
  </xsd:attribute>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="resource-loader">
  <xsd:annotation>
     <xsd:documentation><![CDATA[
Defines a HDFS-aware resource loader.
    ]]>
          </xsd:documentation>
       <xsd:appinfo>
         <tool:annotation>
           <tool:exports type="org.springframework.data.hadoop.fs.HdfsResourceLoader"/>
         </tool:annotation>
       </xsd:appinfo>
  </xsd:annotation>
  <xsd:complexType>
  <xsd:attribute name="id" type="xsd:ID" use="optional">
  <xsd:annotation>
  <xsd:documentation><![CDATA[
Bean id (default is "hadoopResourceLoader").
  ]]></xsd:documentation>
  </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="configuration-ref" use="optional" default="hadoopConfiguration">
  <xsd:annotation>
    <xsd:documentation source="java:org.apache.hadoop.conf.Configuration"><![CDATA[
Reference to the Hadoop Configuration. Defaults to 'hadoopConfiguration'.]]></xsd:documentation>
    <xsd:appinfo>
     <tool:annotation kind="ref">
      <tool:expected-type type="org.apache.hadoop.conf.Configuration" />
     </tool:annotation>
    </xsd:appinfo>
   </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="uri" use="optional">
  <xsd:annotation>
  <xsd:documentation><![CDATA[
The underlying HDFS system URI (by default the configuration settings will be used).
  ]]></xsd:documentation>
  </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="file-system-ref">
   <xsd:annotation>
    <xsd:documentation source="java:org.apache.hadoop.fs.FileSystem"><![CDATA[
Reference to the Hadoop FileSystem. Overrides the 'uri' or 'configuration-ref' properties.]]></xsd:documenta
    <xsd:appinfo>
     <tool:annotation kind="ref">
      <tool:expected-type type="org.apache.hadoop.fs.FileSystem" />
     </tool:annotation>
    </xsd:appinfo>
    </xsd:annotation>
   </xsd:attribute>
  <xsd:attribute name="user" type="xsd:string">
   <xsd:annotation>
   <xsd:documentation><![CDATA[
The security user (ugi) to use for impersonation at runtime.
    ]]></xsd:documentation>
```

```
   </xsd:annotation>
  </xsd:attribute>
  </xsd:complexType>
 </xsd:element>

 <!-- generic options shared by the various jobs
  NOT meant for extensibility - do NOT rely on this type as it might be removed in the future -->
 <xsd:complexType name="genericOptionsType">
  <xsd:complexContent>
  <xsd:extension base="propertiesConfigurableType">
  <xsd:attribute name="archives">
  <xsd:annotation>
   <xsd:appinfo>
   <xsd:documentation source="java:org.springframework.core.io.Resource"><![CDATA[
Archives to be unarchived to the cluster. Multiple resources can be specified, using comma (,) as a separato
    <tool:annotation kind="direct">
     <tool:expected-type type="java:org.springframework.core.io.Resource[]" />
    </tool:annotation>
   </xsd:appinfo>
   </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="files">
  <xsd:annotation>
   <xsd:appinfo>
   <xsd:documentation source="java:org.springframework.core.io.Resource"><![CDATA[
File resources to be copied to the cluster. Multiple resources can be specified, using comma (,) as a separa
    <tool:annotation kind="direct">
     <tool:expected-type type="java:org.springframework.core.io.Resource[]" />
    </tool:annotation>
   </xsd:appinfo>
   </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="libs">
  <xsd:annotation>
   <xsd:appinfo>
   <xsd:documentation source="java:org.springframework.core.io.Resource"><![CDATA[
Jar resources to include in the classpath. Multiple resources can be specified, using comma (,) as a separat
    <tool:annotation kind="direct">
     <tool:expected-type type="java:org.springframework.core.io.Resource[]" />
    </tool:annotation>
   </xsd:appinfo>
   </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="user" type="xsd:string">
   <xsd:annotation>
   <xsd:documentation><![CDATA[
The security user (ugi) to use for impersonation at runtime.
    ]]></xsd:documentation>
   </xsd:annotation>
  </xsd:attribute>
  </xsd:extension>
  </xsd:complexContent>
 </xsd:complexType>

 <!-- common attributes shared by properties based configurations
   NOT meant for extensibility - do NOT rely on this type as it might be removed in the future -->
 <xsd:complexType name="jobType">
  <xsd:complexContent>
  <xsd:extension base="genericOptionsType">
  <xsd:attribute name="id" type="xsd:ID" use="required" />
  <xsd:attribute name="scope" type="xsd:string" use="optional" />
  <xsd:attribute name="mapper" default="org.apache.hadoop.mapreduce.Mapper">
```

```
  <xsd:annotation>
   <xsd:appinfo>
    <tool:annotation kind="direct">
     <tool:expected-type type="java.lang.Class" />
     <tool:assignable-to type="org.apache.hadoop.mapreduce.Mapper" />
    </tool:annotation>
   </xsd:appinfo>
   </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="reducer" default="org.apache.hadoop.mapreduce.Reducer">
  <xsd:annotation>
   <xsd:appinfo>
    <tool:annotation kind="direct">
     <tool:expected-type type="java.lang.Class" />
     <tool:assignable-to type="org.apache.hadoop.mapreduce.Reducer" />
    </tool:annotation>
   </xsd:appinfo>
   </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="combiner">
  <xsd:annotation>
   <xsd:documentation><![CDATA[
The combiner class name.
   ]]></xsd:documentation>
   <xsd:appinfo>
    <tool:annotation kind="direct">
     <tool:expected-type type="java.lang.Class" />
     <tool:assignable-to type="org.apache.hadoop.mapreduce.Reducer" />
    </tool:annotation>
   </xsd:appinfo>
   </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="input-format">
  <xsd:annotation>
   <xsd:appinfo>
    <tool:annotation kind="direct">
     <tool:expected-type type="java.lang.Class" />
     <tool:assignable-to type="org.apache.hadoop.mapreduce.InputFormat" />
    </tool:annotation>
   </xsd:appinfo>
   </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="output-format">
  <xsd:annotation>
   <xsd:appinfo>
    <tool:annotation kind="direct">
     <tool:expected-type type="java.lang.Class" />
     <tool:assignable-to type="org.apache.hadoop.mapreduce.OutputFormat" />
    </tool:annotation>
   </xsd:appinfo>
   </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="partitioner">
  <xsd:annotation>
   <xsd:appinfo>
    <tool:annotation kind="direct">
     <tool:expected-type type="java.lang.Class" />
     <tool:assignable-to type="org.apache.hadoop.mapreduce.Partitioner" />
    </tool:annotation>
   </xsd:appinfo>
   </xsd:annotation>
  </xsd:attribute>
```

```
  <xsd:attribute name="input-path" use="required">
  <xsd:annotation>
   <xsd:appinfo>
    <tool:annotation kind="direct">
     <tool:expected-type type="java.lang.String[]" />
    </tool:annotation>
   </xsd:appinfo>
   </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="output-path" use="required">
  <xsd:annotation>
   <xsd:appinfo>
    <tool:annotation kind="direct">
     <tool:expected-type type="java.lang.String" />
    </tool:annotation>
   </xsd:appinfo>
   </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="configuration-ref" default="hadoopConfiguration">
  <xsd:annotation>
    <xsd:documentation source="java:org.apache.hadoop.conf.Configuration"><![CDATA[
Reference to the Hadoop Configuration. Defaults to 'hadoopConfiguration'.]]></xsd:documentation>
    <xsd:appinfo>
     <tool:annotation kind="ref">
      <tool:expected-type type="org.apache.hadoop.conf.Configuration" />
     </tool:annotation>
    </xsd:appinfo>
   </xsd:annotation>
  </xsd:attribute>
  </xsd:extension>
  </xsd:complexContent>
 </xsd:complexType>

 <xsd:element name="job">
  <xsd:annotation>
     <xsd:documentation><![CDATA[
Defines a Hadoop Job.
    ]]></xsd:documentation>
       <xsd:appinfo>
         <tool:annotation>
           <tool:exports type="org.apache.hadoop.mapreduce.Job"/>
         </tool:annotation>
       </xsd:appinfo>
  </xsd:annotation>
  <xsd:complexType>
  <xsd:complexContent mixed="true">
  <xsd:extension base="jobType">
  <xsd:attribute name="sort-comparator">
  <xsd:annotation>
   <xsd:appinfo>
    <tool:annotation kind="direct">
     <tool:expected-type type="java.lang.Class" />
     <tool:assignable-to type="org.apache.hadoop.io.RawComparator" />
    </tool:annotation>
   </xsd:appinfo>
   </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="grouping-comparator">
  <xsd:annotation>
   <xsd:appinfo>
    <tool:annotation kind="direct">
     <tool:expected-type type="java.lang.Class" />
```

```
     <tool:assignable-to type="org.apache.hadoop.io.RawComparator" />
    </tool:annotation>
   </xsd:appinfo>
   </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="key">
  <xsd:annotation>
   <xsd:appinfo>
    <tool:annotation kind="direct">
     <tool:expected-type type="java.lang.Class" />
    </tool:annotation>
   </xsd:appinfo>
   </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="value">
  <xsd:annotation>
   <xsd:appinfo>
    <tool:annotation kind="direct">
     <tool:expected-type type="java.lang.Class" />
    </tool:annotation>
   </xsd:appinfo>
   </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="map-key">
  <xsd:annotation>
   <xsd:appinfo>
    <tool:annotation kind="direct">
     <tool:expected-type type="java.lang.Class" />
    </tool:annotation>
   </xsd:appinfo>
   </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="map-value">
  <xsd:annotation>
   <xsd:appinfo>
    <tool:annotation kind="direct">
     <tool:expected-type type="java.lang.Class" />
    </tool:annotation>
   </xsd:appinfo>
   </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="codec">
  <xsd:annotation>
   <xsd:appinfo>
    <tool:annotation kind="direct">
     <tool:expected-type type="java.lang.Class" />
    </tool:annotation>
   </xsd:appinfo>
   </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="jar">
  <xsd:annotation>
   <xsd:documentation><![CDATA[
Indicates the user jar for the map-reduce job.
   ]]></xsd:documentation>
   <xsd:appinfo>
    <tool:annotation>
     <tool:expected-type type="org.springframework.core.io.Resource" />
    </tool:annotation>
   </xsd:appinfo>
   </xsd:annotation>
  </xsd:attribute>
```

```
  <xsd:attribute name="jar-by-class">
  <xsd:annotation>
   <xsd:documentation><![CDATA[
Indicates the job's jar file by finding an example class location.
   ]]></xsd:documentation>
   <xsd:appinfo>
    <tool:annotation kind="direct">
     <tool:expected-type type="java.lang.Class" />
    </tool:annotation>
   </xsd:appinfo>
   </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="validate-paths" default="true">
  <xsd:annotation>
     <xsd:documentation><![CDATA[
Indicates whether the job input/output paths are validated before submitting. This
saves time as the validation is done locally without having to interact with the job
tracker. The validation checks whether the input path exists and the output does not.
    ]]></xsd:documentation>
  </xsd:annotation>
  </xsd:attribute>
  </xsd:extension>
  </xsd:complexContent>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="streaming">
  <xsd:annotation>
     <xsd:documentation><![CDATA[
Defines a Hadoop Streaming Job.
    ]]></xsd:documentation>
       <xsd:appinfo>
         <tool:annotation>
           <tool:exports type="org.apache.hadoop.mapreduce.Job"/>
         </tool:annotation>
       </xsd:appinfo>
  </xsd:annotation>
  <xsd:complexType>
  <xsd:complexContent mixed="true">
  <xsd:extension base="jobType">
  <xsd:sequence>
   <xsd:element name="cmd-env" minOccurs="0" maxOccurs="1">
    <xsd:annotation>
    <xsd:documentation><![CDATA[Environment variables (-cmdenv)]]></xsd:documentation>
    </xsd:annotation>
   </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="number-reducers">
  <xsd:annotation>
   <xsd:appinfo>
    <tool:annotation kind="direct">
     <tool:expected-type type="java.lang.Integer" />
    </tool:annotation>
   </xsd:appinfo>
   </xsd:annotation>
  </xsd:attribute>
  </xsd:extension>
  </xsd:complexContent>
  </xsd:complexType>
 </xsd:element>

 <!-- common attributes shared by properties based configurations
```

```
   NOT meant for extensibility - do NOT rely on this type as it might be removed in the future -->
 <xsd:complexType name="toolRunnerType">
  <xsd:complexContent mixed="true">
  <xsd:extension base="genericOptionsType">
   <xsd:sequence>
    <xsd:any namespace="##any" processContents="lax" minOccurs="0" maxOccurs="1"/>
    <xsd:element name="arg" minOccurs="0" maxOccurs="unbounded">
     <xsd:annotation>
     <xsd:documentation><![CDATA[
Tool argument.]]></xsd:documentation>
     </xsd:annotation>
     <xsd:complexType>
      <xsd:attribute name="value" type="xsd:string" use="required"/>
     </xsd:complexType>
    </xsd:element>
   </xsd:sequence>
   <xsd:attribute name="id" type="xsd:ID" use="required">
    <xsd:annotation>
    <xsd:documentation><![CDATA[
Bean id.]]></xsd:documentation>
    </xsd:annotation>
   </xsd:attribute>
         <xsd:attribute name="tool-class" type="xsd:string">
    <xsd:annotation>
     <xsd:documentation><![CDATA[
Indicates the Tool class name. This is useful when referring to an external jar (not required
in the classpath). If not specified, the Main-Class (specified in the MANIFEST.MF), if present,
is used instead.
     ]]></xsd:documentation>
     <xsd:appinfo>
      <tool:annotation kind="direct">
       <tool:expected-type type="java.lang.Class" />
      </tool:annotation>
     </xsd:appinfo>
     </xsd:annotation>
         </xsd:attribute>
         <xsd:attribute name="jar" type="xsd:string">
    <xsd:annotation>
     <xsd:documentation><![CDATA[
Indicates the jar (not required to be in the classpath) providing the Tool (and its dependencies).
     ]]></xsd:documentation>
     <xsd:appinfo>
      <tool:annotation>
       <tool:expected-type type="org.springframework.core.io.Resource" />
      </tool:annotation>
     </xsd:appinfo>
     </xsd:annotation>
         </xsd:attribute>
   <xsd:attribute name="tool-ref">
   <xsd:annotation>
     <xsd:documentation source="java:org.apache.hadoop.util.Tool"><![CDATA[
Reference to a Hadoop Tool instance.]]></xsd:documentation>
     <xsd:appinfo>
      <tool:annotation kind="ref">
       <tool:expected-type type="org.apache.hadoop.util.Tool" />
      </tool:annotation>
     </xsd:appinfo>
    </xsd:annotation>
    </xsd:attribute>
   <xsd:attribute name="configuration-ref" default="hadoopConfiguration">
   <xsd:annotation>
    <xsd:documentation source="java:org.apache.hadoop.conf.Configuration"><![CDATA[
```

```
Reference to the Hadoop Configuration. Defaults to 'hadoopConfiguration'.]]></xsd:documentation>
    <xsd:appinfo>
     <tool:annotation kind="ref">
      <tool:expected-type type="org.apache.hadoop.conf.Configuration" />
     </tool:annotation>
    </xsd:appinfo>
    </xsd:annotation>
   </xsd:attribute>
  </xsd:extension>
  </xsd:complexContent>
 </xsd:complexType>

 <xsd:element name="tool-runner">
  <xsd:annotation>
     <xsd:documentation><![CDATA[
Executes a Hadoop Tool.]]></xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
  <xsd:complexContent mixed="true">
  <xsd:extension base="toolRunnerType">
   <xsd:attribute name="run-at-startup" type="xsd:boolean" default="false">
    <xsd:annotation>
     <xsd:documentation><![CDATA[
Whether the Tool runs at startup or not (default).
     ]]></xsd:documentation>
           </xsd:annotation>
         </xsd:attribute>
  </xsd:extension>
  </xsd:complexContent>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="tool-tasklet">
  <xsd:annotation>
     <xsd:documentation><![CDATA[
Defines a Hadoop Tool Tasklet.]]></xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
   <xsd:complexContent>
    <xsd:extension base="toolRunnerType">
     <xsd:attribute name="scope" type="xsd:string" use="optional" />
    </xsd:extension>
   </xsd:complexContent>
  </xsd:complexType>
 </xsd:element>


 <xsd:complexType name="entryType">
  <xsd:attribute name="value" type="xsd:string" use="required"/>
 </xsd:complexType>

 <xsd:element name="cache">
   <xsd:annotation>
      <xsd:documentation><![CDATA[
Configures Hadoop Distributed Cache.
     ]]></xsd:documentation>
        <xsd:appinfo>
          <tool:annotation>
            <tool:exports type="org.apache.hadoop.io.DistributedCacheFactoryBean"/>
          </tool:annotation>
        </xsd:appinfo>
   </xsd:annotation>
```

```
  <xsd:complexType>
   <xsd:sequence minOccurs="1" maxOccurs="unbounded">
    <xsd:choice>
     <xsd:element name="classpath" type="entryType"/>
     <xsd:element name="cache" type="entryType"/>
     <xsd:element name="local" type="entryType"/>
    </xsd:choice>
   </xsd:sequence>
   <xsd:attribute name="id" type="xsd:ID" use="optional">
   <xsd:annotation>
    <xsd:documentation><![CDATA[
Bean id (default is "hadoopCache").
    ]]></xsd:documentation>
   </xsd:annotation>
   </xsd:attribute>
   <xsd:attribute name="create-symlink" type="xsd:boolean"/>
   <xsd:attribute name="configuration-ref" default="hadoopConfiguration">
   <xsd:annotation>
     <xsd:documentation source="java:org.apache.hadoop.conf.Configuration"><![CDATA[
Reference to the Hadoop Configuration. Defaults to 'hadoopConfiguration'.]]></xsd:documentation>
     <xsd:appinfo>
      <tool:annotation kind="ref">
       <tool:expected-type type="org.apache.hadoop.conf.Configuration" />
      </tool:annotation>
     </xsd:appinfo>
    </xsd:annotation>
    </xsd:attribute>
   <xsd:attribute name="file-system-ref">
   <xsd:annotation>
     <xsd:documentation source="java:org.apache.hadoop.fs.FileSystem"><![CDATA[
Reference to the Hadoop FileSystem.]]></xsd:documentation>
     <xsd:appinfo>
      <tool:annotation kind="ref">
       <tool:expected-type type="org.apache.hadoop.fs.FileSystem" />
      </tool:annotation>
     </xsd:appinfo>
    </xsd:annotation>
    </xsd:attribute>
  </xsd:complexType>
 </xsd:element>

 <xsd:complexType name="scriptType" mixed="true">
   <xsd:attribute name="location" type="xsd:string">
    <xsd:annotation>
     <xsd:documentation><![CDATA[
Location of the script. As an alternative one can inline the script by using a nested, text declaration.]]><
     <xsd:appinfo>
      <tool:annotation>
       <tool:expected-type type="org.springframework.core.io.Resource"/>
      </tool:annotation>
     </xsd:appinfo>
    </xsd:annotation>
   </xsd:attribute>
 </xsd:complexType>

 <xsd:complexType name="scriptWithArgumentsType" mixed="true">
  <xsd:complexContent>
   <xsd:extension base="scriptType">
   <xsd:sequence>
    <xsd:element name="arguments" minOccurs="0" maxOccurs="1">
     <xsd:annotation>
        <xsd:documentation><![CDATA[
```

```
Argument(s) to pass to this script. Defined in Properties format (key=value).
    ]]></xsd:documentation>
        </xsd:annotation>
    </xsd:element>
   </xsd:sequence>
   </xsd:extension>
  </xsd:complexContent>
 </xsd:complexType>

 <xsd:element name="pig">
  <xsd:annotation>
    <xsd:documentation><![CDATA[
Defines a PigServer 'template' (note that since PigServer is not thread-safe, each bean invocation will crea
    ]]>
         </xsd:documentation>
       <xsd:appinfo>
         <tool:annotation>
           <tool:exports type="org.apache.pig.PigServer"/>
         </tool:annotation>
       </xsd:appinfo>
  </xsd:annotation>
  <xsd:complexType>
  <xsd:complexContent mixed="true">
  <xsd:extension base="propertiesConfigurableType">
   <xsd:sequence>
     <xsd:element name="script" type="scriptWithArgumentsType" minOccurs="0" maxOccurs="unbounded">
      <xsd:annotation>
      <xsd:documentation><![CDATA[
Pig script.]]></xsd:documentation>
      </xsd:annotation>
     </xsd:element>
   </xsd:sequence>
   <xsd:attribute name="id" type="xsd:ID" use="optional">
    <xsd:annotation>
    <xsd:documentation><![CDATA[
Bean id (default is "pig").
    ]]></xsd:documentation>
    </xsd:annotation>
   </xsd:attribute>
   <xsd:attribute name="paths-to-skip">
    <xsd:annotation>
      <xsd:documentation><![CDATA[
The path to be skipped while automatically shipping binaries for streaming. Multiple resources can be specif
      ]]></xsd:documentation>
     </xsd:annotation>
   </xsd:attribute>
   <xsd:attribute name="parallelism" type="xsd:integer"/>
   <xsd:attribute name="validate-each-statement" type="xsd:boolean"/>
   <xsd:attribute name="job-priority" type="xsd:string"/>
   <xsd:attribute name="job-name" type="xsd:string"/>
   <xsd:attribute name="job-tracker" type="xsd:string"/>
   <xsd:attribute name="configuration-ref">
    <xsd:annotation>
     <xsd:documentation source="java:org.apache.hadoop.conf.Configuration"><![CDATA[
Reference to the Hadoop Configuration. Can be tweaked through the 'configuration' element or the other attri
     <xsd:appinfo>
      <tool:annotation kind="ref">
       <tool:expected-type type="org.apache.hadoop.conf.Configuration" />
      </tool:annotation>
     </xsd:appinfo>
     </xsd:annotation>
   </xsd:attribute>
```

```
   <xsd:attribute name="exec-type" default="MAPREDUCE">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
       <xsd:enumeration value="MAPREDUCE"/>
       <xsd:enumeration value="LOCAL"/>
      </xsd:restriction>
    </xsd:simpleType>
   </xsd:attribute>
   <xsd:attribute name="user" type="xsd:string">
    <xsd:annotation>
    <xsd:documentation><![CDATA[
The security user (ugi) to use for impersonation at runtime.
    ]]></xsd:documentation>
    </xsd:annotation>
   </xsd:attribute>
  </xsd:extension>
  </xsd:complexContent>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="pig-tasklet">
  <xsd:annotation>
     <xsd:documentation><![CDATA[
Defines a PigTasklet.
    ]]></xsd:documentation>
       <xsd:appinfo>
         <tool:annotation>
           <tool:exports type="org.springframework.data.hadoop.batch.PigTasket"/>
         </tool:annotation>
       </xsd:appinfo>
  </xsd:annotation>
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="script" type="scriptWithArgumentsType" minOccurs="1" maxOccurs="unbounded">
     <xsd:annotation>
     <xsd:documentation><![CDATA[
Pig script.]]></xsd:documentation>
     </xsd:annotation>
    </xsd:element>
   </xsd:sequence>
   <xsd:attribute name="id" type="xsd:ID" use="required">
    <xsd:annotation>
    <xsd:documentation><![CDATA[
Bean id.]]></xsd:documentation>
    </xsd:annotation>
   </xsd:attribute>
   <xsd:attribute name="pig-server-ref" type="xsd:string" use="optional" default="pig">
    <xsd:annotation>
    <xsd:documentation source="java:org.apache.pig.PigServer"><![CDATA[
Reference to a PigServer instance. Defaults to 'pig'.
    ]]></xsd:documentation>
     <xsd:appinfo>
      <tool:annotation kind="ref">
       <tool:expected-type type="org.apache.pig.PigServer" />
      </tool:annotation>
     </xsd:appinfo>
    </xsd:annotation>
   </xsd:attribute>
   <xsd:attribute name="scope" type="xsd:string" use="optional" />
  </xsd:complexType>
 </xsd:element>
```

```
 <!-- HBase -->
 <xsd:element name="hbase-configuration">
  <xsd:complexType>
  <xsd:complexContent mixed="true">
  <xsd:extension base="propertiesConfigurableType">
   <xsd:annotation>
      <xsd:documentation><![CDATA[
Defines an HBase configuration.
     ]]></xsd:documentation>
        <xsd:appinfo>
          <tool:annotation>
            <tool:exports type="org.apache.hadoop.conf.Configuration"/>
          </tool:annotation>
        </xsd:appinfo>
   </xsd:annotation>
   <xsd:attribute name="id" type="xsd:ID" use="optional">
   <xsd:annotation>
    <xsd:documentation><![CDATA[
Bean id (default is "hbaseConfiguration").
    ]]></xsd:documentation>
   </xsd:annotation>
   </xsd:attribute>
   <xsd:attribute name="stop-proxy" type="xsd:boolean" default="true"/>
   <xsd:attribute name="delete-connection" type="xsd:boolean" default="true"/>
   <xsd:attribute name="configuration-ref">
   <xsd:annotation>
     <xsd:documentation source="java:org.apache.hadoop.conf.Configuration"><![CDATA[
Reference to the Hadoop configuration.]]></xsd:documentation>
     <xsd:appinfo>
      <tool:annotation kind="ref">
       <tool:expected-type type="org.apache.hadoop.conf.Configuration" />
      </tool:annotation>
     </xsd:appinfo>
    </xsd:annotation>
    </xsd:attribute>
  </xsd:extension>
  </xsd:complexContent>
  </xsd:complexType>
 </xsd:element>

 <!-- Hive -->
 <xsd:element name="hive-client">
  <xsd:complexType>
   <xsd:annotation>
      <xsd:documentation><![CDATA[
Defines a Hive client for connecting to a Hive server through the Thrift protocol.
     ]]></xsd:documentation>
        <xsd:appinfo>
          <tool:annotation>
            <tool:exports type="org.apache.hadoop.hive.service.HiveClient"/>
          </tool:annotation>
        </xsd:appinfo>
   </xsd:annotation>
   <xsd:sequence>
    <xsd:element name="script" type="scriptType" minOccurs="0" maxOccurs="unbounded">
     <xsd:annotation>
     <xsd:documentation><![CDATA[
Hive script to be executed during start-up.]]></xsd:documentation>
     </xsd:annotation>
    </xsd:element>
   </xsd:sequence>
   <xsd:attribute name="id" type="xsd:ID" use="optional">
```

```
   <xsd:annotation>
    <xsd:documentation><![CDATA[
Bean id (default is "hiveClient").
    ]]></xsd:documentation>
   </xsd:annotation>
   </xsd:attribute>
   <xsd:attribute name="host" type="xsd:string" default="localhost"/>
   <xsd:attribute name="port" type="xsd:string" default="10000"/>
   <xsd:attribute name="auto-startup" type="xsd:boolean" default="true"/>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="hive-server">
  <xsd:complexType>
  <xsd:complexContent mixed="true">
  <xsd:extension base="propertiesConfigurableType">
   <xsd:annotation>
      <xsd:documentation><![CDATA[
Defines an embedded Hive Server instance opened for access through the Thrift protocol.
     ]]></xsd:documentation>
        <xsd:appinfo>
          <tool:annotation>
            <tool:exports type="org.apache.thrift.server.TServer"/>
          </tool:annotation>
        </xsd:appinfo>
   </xsd:annotation>
   <xsd:attribute name="id" type="xsd:ID" use="optional">
   <xsd:annotation>
    <xsd:documentation><![CDATA[
Bean id (default is "hiveServer").
    ]]></xsd:documentation>
   </xsd:annotation>
   </xsd:attribute>
   <xsd:attribute name="port" type="xsd:string" default="10000"/>
   <xsd:attribute name="min-threads" type="xsd:string" default="5"/>
   <xsd:attribute name="max-threads" type="xsd:string" default="100"/>
   <xsd:attribute name="auto-startup" type="xsd:boolean" default="true"/>
   <xsd:attribute name="configuration-ref">
   <xsd:annotation>
     <xsd:documentation source="java:org.apache.hadoop.conf.Configuration"><![CDATA[
Reference to the Hadoop configuration.]]></xsd:documentation>
     <xsd:appinfo>
      <tool:annotation kind="ref">
       <tool:expected-type type="org.apache.hadoop.conf.Configuration" />
      </tool:annotation>
     </xsd:appinfo>
    </xsd:annotation>
    </xsd:attribute>
  </xsd:extension>
  </xsd:complexContent>
  </xsd:complexType>
 </xsd:element>

 <xsd:element name="hive-tasklet">
  <xsd:annotation>
     <xsd:documentation><![CDATA[
Defines a HiveTasklet.
    ]]></xsd:documentation>
       <xsd:appinfo>
         <tool:annotation>
           <tool:exports type="org.springframework.data.hadoop.batch.HiveTasket"/>
         </tool:annotation>
```

```
      </xsd:appinfo>
  </xsd:annotation>
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="script" type="scriptType" minOccurs="1" maxOccurs="unbounded">
     <xsd:annotation>
     <xsd:documentation><![CDATA[
Hive script.]]></xsd:documentation>
     </xsd:annotation>
    </xsd:element>
   </xsd:sequence>
   <xsd:attribute name="id" type="xsd:ID" use="required">
    <xsd:annotation>
    <xsd:documentation><![CDATA[
Bean id.]]></xsd:documentation>
    </xsd:annotation>
   </xsd:attribute>
   <xsd:attribute name="hive-client-ref" type="xsd:string" use="optional" default="hiveClient">
    <xsd:annotation>
    <xsd:documentation source="java:org.apache.hadoop.hive.service.HiveClient"><![CDATA[
Reference to a HiveClient instance. Defaults to 'hiveClient'.
    ]]></xsd:documentation>
     <xsd:appinfo>
      <tool:annotation kind="ref">
       <tool:expected-type type="org.apache.hadoop.hive.service.HiveClient" />
      </tool:annotation>
     </xsd:appinfo>
    </xsd:annotation>
   </xsd:attribute>
   <xsd:attribute name="scope" type="xsd:string" use="optional" />
  </xsd:complexType>
 </xsd:element>

 <!-- Script type - NOT mean to be reused outside this schema -->
 <xsd:complexType name="scriptingType" abstract="true" mixed="true">
   <xsd:sequence>
    <xsd:element name="property" type="beans:propertyType"  minOccurs="0" maxOccurs="unbounded">
     <xsd:annotation>
      <xsd:documentation><![CDATA[
Property to pass to the script. Can be used to enhance or override the default properties.
      ]]></xsd:documentation>
     </xsd:annotation>
    </xsd:element>
   </xsd:sequence>
   <xsd:attribute name="location" type="xsd:string">
   <xsd:annotation>
    <xsd:documentation><![CDATA[
The location of the script. Can be any resource on the local filesystem, web or even hdfs.
    ]]>
          </xsd:documentation>
   </xsd:annotation>
   </xsd:attribute>
   <xsd:attribute name="language" type="xsd:string">
   <xsd:annotation>
    <xsd:documentation><![CDATA[
The language used for executing the script. If no value is given, the script source extension
is used to determine the scripting engine.
    ]]></xsd:documentation>
         </xsd:annotation>
   </xsd:attribute>
   <xsd:attribute name="evaluate" default="ALWAYS">
   <xsd:annotation>
```

```
    <xsd:documentation><![CDATA[
When to evaluate the script. 'ALWAYS' (default) evaluates the script on all invocations,
'IF_MODIFIED' if the script source has been modified since the last invocation and 'ONCE'
only once for the duration of the application.
    ]]></xsd:documentation>
          </xsd:annotation>
          <xsd:simpleType>
           <xsd:restriction base="xsd:string">
            <xsd:enumeration value="ONCE"/>
            <xsd:enumeration value="IF_MODIFIED"/>
            <xsd:enumeration value="ALWAYS"/>
           </xsd:restriction>
          </xsd:simpleType>
   </xsd:attribute>
 </xsd:complexType>

 <xsd:element name="script">
  <xsd:complexType mixed="true">
   <xsd:complexContent>
    <xsd:annotation>
       <xsd:documentation><![CDATA[
Dedicated scripting facility for interacting with Hadoop. Allows Groovy, JavaScript (Rhino), Ruby (JRuby), P
or any JSR-223 scripting language to be used for executing commands against Hadoop, in particular its file s
     ]]></xsd:documentation>
         <xsd:appinfo>
           <tool:annotation>
             <tool:exports type="java.lang.Object"/>
           </tool:annotation>
         </xsd:appinfo>
    </xsd:annotation>
    <xsd:extension base="scriptingType">
    <xsd:attribute name="id" type="xsd:ID" use="optional">
     <xsd:annotation>
      <xsd:documentation><![CDATA[
Bean id (if no value is given, a name will be generated).
      ]]></xsd:documentation>
     </xsd:annotation>
    </xsd:attribute>
    <xsd:attribute name="run-at-startup" type="xsd:boolean" default="false">
     <xsd:annotation>
      <xsd:documentation><![CDATA[
Whether the script is evaluated automatically once the application context initializes or only when in use
      ]]></xsd:documentation>
           </xsd:annotation>
          </xsd:attribute>
    <xsd:attribute name="scope" type="xsd:string" use="optional" />
    <xsd:attribute name="configuration-ref" default="hadoopConfiguration">
    <xsd:annotation>
     <xsd:documentation source="java:org.apache.hadoop.conf.Configuration"><![CDATA[
Reference to the Hadoop Configuration. Defaults to 'hadoopConfiguration'.]]></xsd:documentation>
     <xsd:appinfo>
      <tool:annotation kind="ref">
       <tool:expected-type type="org.apache.hadoop.conf.Configuration" />
      </tool:annotation>
     </xsd:appinfo>
     </xsd:annotation>
    </xsd:attribute>
    </xsd:extension>
   </xsd:complexContent>
  </xsd:complexType>
 </xsd:element>
```

```
 <xsd:element name="script-tasklet">
  <xsd:complexType>
   <xsd:annotation>
      <xsd:documentation><![CDATA[
Defines a scripting Tasklet for interacting with Hadoop. Allows Groovy, JavaScript (Rhino), Ruby (JRuby), Py
or any JSR-223 scripting language to be used for executing commands against Hadoop, in particular its file s
     ]]>
           </xsd:documentation>
        <xsd:appinfo>
          <tool:annotation>
            <tool:exports type="java.lang.Object"/>
          </tool:annotation>
        </xsd:appinfo>
   </xsd:annotation>
   <xsd:sequence>
    <xsd:element name="script" minOccurs="0" maxOccurs="1">
    <xsd:annotation>
      <xsd:documentation><![CDATA[
Nested script declaration.]]></xsd:documentation>
     </xsd:annotation>
     <xsd:complexType mixed="true">
      <xsd:complexContent>
       <xsd:extension base="scriptingType"/>
      </xsd:complexContent>
     </xsd:complexType>
    </xsd:element>
   </xsd:sequence>
   <xsd:attribute name="id" type="xsd:ID" use="required">
   <xsd:annotation>
    <xsd:documentation><![CDATA[
Bean id.]]></xsd:documentation>
   </xsd:annotation>
   </xsd:attribute>
   <xsd:attribute name="script-ref" type="xsd:string">
    <xsd:annotation>
      <xsd:documentation><![CDATA[
Reference to a script declaration.]]></xsd:documentation>
      <xsd:appinfo>
       <tool:annotation kind="ref">
        <tool:expected-type type="java.lang.Object" />
       </tool:annotation>
      </xsd:appinfo>
     </xsd:annotation>
   </xsd:attribute>
   <xsd:attribute name="scope" type="xsd:string" use="optional" />
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```