Spring for Apache Hadoop
- Reference Documentation

1.1.0.RELEASE-phd1

Costin Leau Elasticsearch , Thomas Risberg Pivotal , Janne Valkealahti Pivotal

Table of Contents

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 regain on 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 with the Spring framework to define the base concepts and semantics of the integration and how they can be used 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 (3.2 recommended) and above and is built against Apache Hadoop 1.2.1. SHDP supports and is tested daily against Apache Hadoop 1.2.1 and also against 1.1.2 and 2.0.x alpha as well as against various Hadoop distributions:

- Pivotal HD 1.1
- Cloudera CDH4 (cdh4.3.1 MRv1) distributions
- Hortonworks Data Platform 1.3

Any distro compatible with Apache Hadoop 1.x stable should be supported.

We have recently added support to allow Hadoop 2.x based distributions to be used with the current functionality. We are running test builds against Apache Hadoop 2.0.x alpha, Pivotal HD 1.1 and Hortonworks Data Platform 2.0.

Note

Hadoop YARN support is only available in Spring for Apache Hadoop version 2.0 and later.

Spring Data Hadoop is provided out of the box and it is certified to work on Greenplum HD 1.2 and Pivotal HD 1.0 and 1.1 distributions. It is also certified to run against Hortonworks HDP 1.3.

Instructions for setting up project builds using various supported distributions are provided on the Spring for Apache Hadoop wiki -http://cascading.org/http://hbase.apache.org/http://hive.apache.org/ https://github.com/spring-projects/spring-hadoop/wiki

Regarding Hadoop-related projects, SDHP supports Cascading 2.1, HBase 0.90.x, Hive 0.8.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 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 1.2.1, 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 convenient to download a Virtual Machine where Hadoop is setup and ready to go. Cloudera, Hortonworks and Pivotal all provide virtual machines and provide VM downloads on their product pages. Additionally, the appendix provides information on how to use Spring for Apache Hadoop and setup Hadoop with cloud providers, such as Amazon Web Services.

2. Additional Resources

While this documentation acts as a reference for Spring for Hadoop project, there are number of resources that, while optional, complement this document by providing additional background and code samples for the reader to try and experiment with:

- **Spring for Apache Hadoop samples https://github.com/spring-projects/spring-hadoop-samples/.** Official repository full of SHDP samples demonstrating the various project features.
- **Spring Data Book http://shop.oreilly.com/product/0636920024767.do.** Guide to Spring Data projects, written by the committers behind them. Covers Spring Data Hadoop stand-alone but in tandem with its *siblings* projects. All author royalties from book sales are donated to Creative Commons organization.
- **Spring Data Book Hadoop examples https://github.com/spring-projects/spring-data-book/tree/master/hadoop.** Complete running samples for the Spring Data book. Note that some of them are available inside Spring for Apache Hadoop samples as well.

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

Chapter 4, *Working with the Hadoop File System* describes the Spring support for interacting with the Hadoop file system.

Chapter 5, *Working with HBase* describes the Spring support for HBase.

Chapter 6, *Hive integration* describes the Hive integration in SHDP.

Chapter 7, *Pig support* describes the Pig support in Spring for Apache Hadoop.

Chapter 8, *Cascading integration* describes the Cascading integration in Spring for Apache Hadoop.

Chapter 10, *Security Support* describes how to configure and interact with Hadoop in a secure environment.

3. 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.

3.1 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.xsd
    http://www.springframework.org/schema/hadoop http://www.springframework.org/schema/
hadoop/spring-hadoop.xsd❸">

   <bean id ... >

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

❶ Spring for Apache Hadoop namespace prefix. Any name can do but throughout the reference documentation, 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 imported, the namespace elements can be declared simply by using 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 declarations 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/
hadoop/spring-hadoop.xsd">

    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 may simply refer to the <hdp> namespace without the namespace declaration, where possible.

3.2 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/spring-beans.xsd
        http://www.springframework.org/schema/hadoop http://www.springframework.org/
schema/hadoop/spring-hadoop.xsd">

     <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/spring-beans.xsd
        http://www.springframework.org/schema/context http://www.springframework.org/
schema/context/spring-context.xsd
        http://www.springframework.org/schema/hadoop http://www.springframework.org/
schema/hadoop/spring-hadoop.xsd">

     <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/beans/spring-beans.xsd
        http://www.springframework.org/schema/context http://www.springframework.org/
schema/context/spring-context.xsd
        http://www.springframework.org/schema/util http://www.springframework.org/schema/
util/spring-util.xsd
        http://www.springframework.org/schema/hadoop http://www.springframework.org/
schema/hadoop/spring-hadoop.xsd">

   <!-- 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 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 configurations 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 that you specify a different name since otherwise, because 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 will throw an exception since the VM does not know what `hdfs` means.

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 Hadoop configuration since it allows easier updates without interfering with the application configuration. When dealing with multiple, similar configurations use configuration *composition* as it tends to keep the definitions concise, in sync and easy to update.

3.3 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 is there to 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 C, *Spring for Apache Hadoop Schema*) or simply trigger auto-completion (usually CTRL+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>
```

<hdp: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 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 it allows 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 a task - it's as easy and straightforward 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 they can pass the command line arguments (such as `-D` or `-cmdenv`). While the former customize the Hadoop configuration (which has been convered in the previous section), 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.

3.4 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 the `job-runner` element (backed by `JobRunner` class) which submits several jobs sequentially (and waits by default for their completion):

```
<hdp:job-runner id="myjob-runner" pre-action="cleanup-script" post-action="export-
results" job-ref="myjob" run-at-startup="true"/>

<hdp:job id="myjob"  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:

```
<hdp:job-runner id="myjobs-runner" pre-action="cleanup-script" job-ref="myjob1,
 myjob2" run-at-startup="true"/>

<hdp:job id="myjob1"  ... />
<hdp:streaming id="myjob2"  ... />
```

One or multiple Map-Reduce jobs can be specified through the `job` attribute in the order of the execution. The runner will trigger the execution during the application start-up (notice the `run-at-startup` flag which is by default `false`). Do note that the runner will not run unless triggered manually or if `run-at-startup` is set to `true`. Additionally the runner (as in fact do all runners in SHDP) allows one or multiple `pre` and `post` actions to be specified to be executed before and after each run. Typically other runners (such as other jobs or scripts) can be specified but any JDK `Callable` can be passed in. For more information on runners, see the dedicated chapter.

Note

As the Hadoop job submission and execution (when `wait-for-completion` is `true`) is blocking, `JobRunner` uses a JDK `Executor` to start (or stop) a job. The default implementation, `SyncTaskExecutor` uses the calling thread to execute the job, mimicking the hadoop command line behaviour. However, as the hadoop jobs are time-consuming, in some cases this can lead to "application freeze", preventing normal operations or even application shutdown from occuring properly. Before going into production, it is recommended to double-check whether this strategy is suitable or whether a throttled or pooled implementation is better. One can customize the behaviour through the `executor-ref` parameter.

The job runner also allows running jobs to be cancelled (or killed) at shutdown. This applies only to jobs that the runner waits for (`wait-for-completion` is `true`) using a different executor then the default - that is, using a different thread then the calling one (since otherwise the calling thread has to wait for the job to finish first before executing the next task). To customize this behaviour, one should set the `kill-job-at-shutdown` attribute to `false` and/or change the `executor-ref` implementation.

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

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

3.5 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 opposed 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 -Dproperty=value
 someJar.jar org.foo.SomeTool data/in.txt data/out.txt
```

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" run-at-startup="true">
   <hdp:arg value="data/in.txt"/>
   <hdp:arg value="data/out.txt"/>

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

Additionally the runner (just like the job runner) allows one or multiple `pre` and `post` actions to be specified to be executed before and after each run. Typically other runners (such as other jobs or scripts) can be specified but any JDK `Callable` can be passed in. Do note that the runner will not run unless triggered manually or if `run-at-startup` is set to `true`. For more information on runners, see the dedicated chapter.

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">
   <hdp:tool>
      <bean class="org.foo.AnotherTool" p:input="data/in.txt" p:output="data/out.txt"/>
   </hdp:tool>
</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 execute 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-runner`

`tool-runner` 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-runner` element:

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

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" />
```

3.6 Running a Hadoop Jar

SHDP also provides support for executing vanilla Hadoop jars. Thus the famous WordCount example:

```
bin/hadoop jar hadoop-examples.jar wordcount /wordcount/input /wordcount/output
```

becomes

```
<hdp:jar-runner id="wordcount" jar="hadoop-examples.jar" run-at-startup="true">
    <hdp:arg value="wordcount"/>
    <hdp:arg value="/wordcount/input"/>
    <hdp:arg value="/wordcount/output"/>
</hdp:jar-runner>
```

Note

Just like the `hadoop jar` command, by default the jar support reads the jar's `Main-Class` if none is specified. This can be customized through the `main-class` attribute.

Additionally the runner (just like the job runner) allows one or multiple `pre` and `post` actions to be specified to be executed before and after each run. Typically other runners (such as other jobs or scripts) can be specified but any JDK `Callable` can be passed in. Do note that the runner will not run unless triggered manually or if `run-at-startup` is set to `true`. For more information on runners, see the dedicated chapter.

The `jar support` provides a nice and easy migration path from jar invocations from the command-line to SHDP (note that Hadoop generic options are also supported). Especially since SHDP enables Hadoop `Configuration` objects, created during the jar execution, to automatically inherit the context Hadoop configuration. In fact, just like other SHDP elements, the `jar` element allows configurations properties to be declared locally, just for the jar run. So for example, if one would use the following declaration:

```
<hdp:jar-runner id="wordcount" jar="hadoop-examples.jar" run-at-startup="true">
    <hdp:arg value="wordcount"/>
    ...
    speed=fast
</hdp:jar-runner>
```

inside the jar code, one could do the following:

```
assert "fast".equals(new Configuration().get("speed"));
```

This enabled basic Hadoop jars to use, without changes, the enclosing application Hadoop configuration.

And while we think it is a useful feature (that is why we added it in the first place), we strongly recommend using the tool support instead or migrate to it; there are several reasons for this mainly because there are *no contracts* to use, leading to very poor embeddability caused by:

- No standard `Configuration` injection

  While SHDP does a best effort to pass the Hadoop configuration to the jar, there is no guarantee the jar itself does not use a special initialization mechanism, ignoring the passed properties. After all, a vanilla `Configuration` is not very useful so applications tend to provide custom code to address this.

- `System.exit()` calls

  Most jar examples out there (including `WordCount`) assume they are started from the command line and among other things, call `System.exit`, to shut down the JVM, whether the code is succesful or not. SHDP prevents this from happening (otherwise the entire application context would shutdown abruptly) but it is a clear sign of poor code collaboration.

SHDP tries to use sensible defaults to provide the best integration experience possible but at the end of the day, without any contract in place, there are no guarantees. Hence using the `Tool` interface is a much better alternative.

Using the Hadoop Jar tasklet

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

```
<hdp:jar-tasklet id="jar-tasklet" jar="some-jar.jar" />
```

3.7 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.

Warning

Clients setting up a *classpath* in the `DistributedCache`, running on Windows platforms should set the `System` `path.separator` property to `:`. Otherwise the classpath will be set incorrectly and will be ignored; see HADOOP-9123 bug report for more information.

There are multiple ways to change the `path.separator` `System` property - a quick one being a simple `script` in Javascript (that uses the Rhino package bundled with the JDK) that runs at start-up:

```
<hdp:script language="javascript" run-at-startup="true">
    // set System 'path.separator' to ':' - see HADOOP-9123
    java.lang.System.setProperty("path.separator", ":")
</hdp:script>
```

3.8 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 throughout 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 needed, as the SHDP offers first-class support for defining and customizing Hadoop configurations.

4. 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.

4.1 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 than HDFS to be used.

The table below describes the common HDFS APIs in use:

Table 4.1. 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

What about FTP, Kosmos, S3 and the other file systems?

This chapter focuses on the core file-system protocols supported by Hadoop. S3 (see the Appendix), FTP and the rest of the other `FileSystem` implementations are supported as well - Spring for Apache Hadoop has no dependency on the underlying system rather just on the public Hadoop API.

`hdfs://` protocol should be familiar to most readers - 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 than 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. Furthermore, since `webhdfs://` is backed by a REST API, clients in other languages can use it with minimal effort.

Note

Not all file systems 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 scheme has been decided upon, one can specify it through the standard Hadoop configuration, either through the Hadoop configuration files or 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 scheme:

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

<!-- creates 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).

4.2 Using HDFS Resource Loader

In Spring the `ResourceLoader` interface is meant to be implemented by objects that can return (i.e. load) Resource instances.

```
public interface ResourceLoader {
  Resource getResource(String location);
}
```

All application contexts implement the `ResourceLoader` interface, and therefore all application contexts may be used to obtain Resource instances.

When you call `getResource()` on a specific application context, and the location path specified doesn't have a specific prefix, you will get back a `Resource` type that is appropriate to that particular application context. For example, assume the following snippet of code was executed against a `ClassPathXmlApplicationContext` instance:

```
Resource template = ctx.getResource("some/resource/path/myTemplate.txt");
```

What would be returned would be a `ClassPathResource`; if the same method was executed against a `FileSystemXmlApplicationContext` instance, you'd get back a `FileSystemResource`. For a `WebApplicationContext`, you'd get back a `ServletContextResource`, and so on.

As such, you can load resources in a fashion appropriate to the particular application context.

On the other hand, you may also force `ClassPathResource` to be used, regardless of the application context type, by specifying the special `classpath:` prefix:

```
Resource template = ctx.getResource("classpath:some/resource/path/myTemplate.txt");
```

Note

More information about the generic usage of resource loading, check the *Spring Framework Documentation*.

`Spring Hadoop` is adding its own functionality into generic concept of resource loading. `Resource` abstraction in Spring has always been a way to ease resource access in terms of not having a need to know where there resource is and how it's accessed. This abstraction also goes beyond a single resource by allowing to use patterns to access multiple resources.

Lets first see how `HdfsResourceLoader` is used manually.

```
<hdp:file-system />
<hdp:resource-loader id="loader" file-system-ref="hadoopFs" />
<hdp:resource-loader id="loaderWithUser" user="myuser" uri="hdfs://localhost:8020" />
```

In above configuration we created two beans, one with reference to existing `Hadoop FileSystem` bean and one with impersonated user.

```
// get path '/tmp/file.txt'
Resource resource = loader.getResource("/tmp/file.txt");
// get path '/tmp/file.txt' with user impersonation
Resource resource = loaderWithUser.getResource("/tmp/file.txt");

// get path '/user/<current user>/file.txt'
Resource resource = loader.getResource("file.txt");
// get path '/user/myuser/file.txt'
Resource resource = loaderWithUser.getResource("file.txt");

// get all paths under '/tmp/'
Resource[] resources = loader.getResources("/tmp/*");
// get all paths under '/tmp/' recursively
Resource[] resources = loader.getResources("/tmp/**/*");
// get all paths under '/tmp/' using more complex ant path matching
Resource[] resources = loader.getResources("/tmp/?ile?.txt");
```

What would be returned in above examples would be instances of `HdfsResources`.

```
<hdp:file-system />
<hdp:resource-loader file-system-ref="hadoopFs" handle-noprefix="false" />
```

Note

On default the `HdfsResourceLoader` will handle all resource paths without prefix. Attribute `handle-noprefix` can be used to control this behaviour. If this attribute is set to *false*, non-prefixed resource uris will be handled by *Spring Application Context*.

```
// get 'default.txt' from current user's home directory
Resource[] resources = context.getResources("hdfs:default.txt");
// get all files from hdfs root
Resource[] resources = context.getResources("hdfs:/*");
// let context handle classpath prefix
Resource[] resources = context.getResources("classpath:cfg*properties");
```

What would be returned in above examples would be instances of `HdfsResources` and `ClassPathResource` for the last one. If requesting resource paths without existing prefix, this example would fall back into *Spring Application Context*. It may be advisable to let `HdfsResourceLoader` to handle paths without prefix if your application doesn't rely on loading resources from underlying context without prefixes.

Table 4.2. `hdp:resource-loader` attributes

Name	Values	Description
`file-system-ref`	Bean Reference	Reference to existing *Hadoop FileSystem* bean
`use-codecs`	Boolean(defaults to true)	Indicates whether to use (or not) the codecs found inside the Hadoop configuration when accessing the resource input stream.
`user`	String	The security user (ugi) to use for impersonation at runtime.
`uri`	String	The underlying HDFS system URI.
`handle-noprefix`	Boolean(defaults to true)	Indicates if loader should handle resource paths without prefix.

4.3 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 file systems; 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, straightforward 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 at 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" run-at-startup="true">
    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. Furthermore 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 then 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 seen before:

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

  <hdp:script language="javascript" run-at-startup="true">
    ...
    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.

As you might have noticed, the `script` element defines a runner for JVM scripts. And just like the rest of the SHDP runners, it allows one or multiple `pre` and `post` actions to be specified to be executed

before and after each run. Typically other runners (such as other jobs or scripts) can be specified but any JDK `Callable` can be passed in. Do note that the runner will not run unless triggered manually or if `run-at-startup` is set to `true`. For more information on runners, see the dedicated chapter.

Using scripts

Inlined scripting is quite handy for doing simple operations and coupled 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 than 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" run-at-startup="true"/>
```

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)
```

4.4 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 4.3. 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
ctxRL	`org.springframework.io.support.ResourcePatternResolver`	Enclosing application context ResourceLoader
distcp	`org.springframework.data.hadoop.fs.DistributedCopyUtil`	Programmatic access to DistCp
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

Name	Type	Description
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 by type), several log warnings will be made and none of the Hadoop-based variables (namely `cfg`, `distcp`, `fs`, `fsh`, `distcp` or `hdfsRL`) will be 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" run-at-startup="true">
   <hdp:property name="foo" value="bar"/>
   <hdp:property name="ref" ref="some-bean"/>
</hdp:script>
```

Running scripts

The `script` namespace provides various options to adjust its behaviour depending on the script content. By default the script is simply declared - that is, no execution occurs. One however can change that so that the script gets evaluated at startup (as all the examples in this section do) through the `run-at-startup` flag (which is by default `false`) or when invoked manually (through the `Callable`). Similarily, by default the script gets evaluated on each run. 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 4.4. `script` attributes

Name	Values	Description
`run-at-startup`	`false`(default), `true`	Wether the script is executed at startup or not
`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 once.

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"/>
```

4.5 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 existence of files, delete, move, copy directories or files or set up permissions. However the utility is only available from the command-line which makes it hard to use 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 than dealing with `System.in` or `System.out`, one deals with objects.

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

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

```
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 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 than `Strings`) making it easy to use them programmatically whether in Java or another language. Furthermore, 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-a232-2d5426c2bc4e
```

As you can see, not only can you 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 languages 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 and configuration options are the same however, one can use it from within a 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.

5. 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>
```

Additionally, one can specify the ZooKeeper port used by the HBase server - this is especially useful when connecting to a remote instance (note one can fully configure HBase including the ZooKeeper host and port through properties; the attributes here act as shortcuts for easier declaration):

```
<!-- specify ZooKeeper host/port -->
<hdp:hbase-configuration zk-quorum="${hbase.host}" zk-port="${hbase.port}">
```

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.properties"/>
```

5.1 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 than 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="hbaseConfiguration"/>
```

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-liners*) 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 showcases 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 or 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.

6. 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 support both of them.

6.1 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" configuration-ref="hadoopConfiguration">
  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.

6.2 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 'hiveClientFactory' -->
<hdp:hive-client-factory host="some-other-host" port="10001" />
```

Note that since Thrift clients are not thread-safe, `hive-client-factory` returns a factory (named `org.springframework.data.hadoop.hive.HiveClientFactory`) for creating `HiveClient` new instances for each invocation. Furthermore, the client definition also allows Hive scripts (either declared inlined or externally) to be executed during initialization, once the client connects; this is quite useful for doing Hive specific initialization:

```
<hive-client-factory 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">
       <arguments>ignore-case=true</arguments>
   </hdp:script>
</hive-client-factory>
```

In the example above, two scripts are executed each time a new Hive client is created (if the scripts need to be executed only once consider using a tasklet) by the factory. The first script is defined inline while the second is read from the classpath and passed one parameter. For more information on using parameters (or variables) in Hive scripts, see this section in the Hive manual.

6.3 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/spring-beans.xsd
       http://www.springframework.org/schema/context http://www.springframework.org/
schema/context/spring-context.xsd">

    <!-- 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 `Connections` or better yet, use Spring's `JdbcTemplate` as in the example above.

6.4 Running a Hive script or query

Like the rest of the Spring Hadoop components, a runner is provided out of the box for executing Hive scripts, either inlined or from various locations through `hive-runner` element:

```
<hdp:hive-runner id="hiveRunner" run-at-startup="true">
   <hdp:script>
     DROP TABLE IF EXITS testHiveBatchTable;
     CREATE TABLE testHiveBatchTable (key int, value string);
   </hdp:script>
   <hdp:script location="hive-scripts/script.q"/>
</hdp:hive-runner>
```

The runner will trigger the execution during the application start-up (notice the `run-at-startup` flag which is by default `false`). Do note that the runner will not run unless triggered manually or if `run-at-startup` is set to `true`. Additionally the runner (as in fact do all runners in SHDP) allows one or multiple `pre` and `post` actions to be specified to be executed before and after each run. Typically other runners (such as other jobs or scripts) can be specified but any JDK `Callable` can be passed in. For more information on runners, see the dedicated chapter.

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 straightforward:

```
<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.

6.5 Interacting with the Hive API

For those that need to programmatically interact with the Hive API, Spring for Apache Hadoop provides a dedicated template, similar to the aforementioned `JdbcTemplate`. The template handles the redundant, boiler-plate code, required for interacting with Hive such as creating a new `HiveClient`, executing the queries, catching any exceptions and performing clean-up. One can programmatically execute queries (and get the raw results or convert them to longs or ints) or scripts but also interact with the Hive API through the `HiveClientCallback`. For example:

```
<hdp:hive-client-factory ... />
<!-- Hive template wires automatically to 'hiveClientFactory'-->
<hdp:hive-template />

<!-- wire hive template into a bean -->
<bean id="someBean" class="org.SomeClass" p:hive-template-ref="hiveTemplate"/>
```

```
public class SomeClass {

  private HiveTemplate template;

  public void setHiveTemplate(HiveTemplate template) { this.template = template; }

  public List<String> getDbs() {
      return hiveTemplate.execute(new HiveClientCallback<List<String>>() {
         @Override
         public List<String> doInHive(HiveClient hiveClient) throws Exception {
            return hiveClient.get_all_databases();
         }
      }));
  }
}
```

The example above shows a basic container configuration wiring a `HiveTemplate` into a user class which uses it to interact with the `HiveClient` Thrift API. Notice that the user does not have to handle the lifecycle of the `HiveClient` instance or catch any exception (out of the many thrown by Hive itself and the Thrift fabric) - these are handled automatically by the template which converts them, like the rest of the Spring templates, into `DataAccessExceptions`. Thus the application only has to track only one exception hierarchy across all data technologies instead of one per technology.

7. 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 `org.springframework.data.hadoop.pig.PigServerFactory` instance, named `pigFactory`, a factory that creates `PigServer` instances on demand configured with a default `PigContext`, executing scripts in `MapReduce` mode. The factory is needed since `PigServer` is not thread-safe and thus cannot be used by multiple objects at the same time. 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-factory exec-type="LOCAL" job-name="pig-script" configuration-
ref="hadoopConfiguration" properties-location="pig-dev.properties"
   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-factory> />
```

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-factory/>` 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-factory` namespace offers several options pertaining to Pig configuration.

7.1 Running a Pig script

Like the rest of the Spring Hadoop components, a runner is provided out of the box for executing Pig scripts, either inlined or from various locations through `pig-runner` element:

```
<hdp:pig-runner id="pigRunner" run-at-startup="true">
   <hdp:script>
  A = LOAD 'src/test/resources/logs/apache_access.log' USING PigStorage() AS
 (name:chararray, age:int);
  ...
   </hdp:script>
   <hdp:script location="pig-scripts/script.pig"/>
</hdp:pig-runner>
```

The runner will trigger the execution during the application start-up (notice the `run-at-startup` flag which is by default `false`). Do note that the runner will not run unless triggered manually or if `run-at-startup` is set to `true`. Additionally the runner (as in fact do all runners in SHDP) allows one or multiple `pre` and `post` actions to be specified to be executed before and after each run. Typically other runners (such as other jobs or scripts) can be specified but any JDK `Callable` can be passed in. For more information on runners, see the dedicated chapter.

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 straightforward:

```
<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.

7.2 Interacting with the Pig API

For those that need to programmatically interact directly with Pig , Spring for Apache Hadoop provides a dedicated template, similar to the aforementioned `HiveTemplate`. The template handles the redundant, boiler-plate code, required for interacting with Pig such as creating a new `PigServer`, executing the scripts, catching any exceptions and performing clean-up. One can programmatically execute scripts but also interact with the Hive API through the `PigServerCallback`. For example:

```
<hdp:pig-factory ... />
<!-- Pig template wires automatically to 'pigFactory'-->
<hdp:pig-template />

<!-- use component scanning-->
<context:component-scan base-package="some.pkg" />
```

```
public class SomeClass {
  @Inject
  private PigTemplate template;

  public Set<String> getDbs() {
      return pigTemplate.execute(new PigCallback<Set<String>() {
         @Override
         public Set<String> doInPig(PigServer pig) throws ExecException, IOException {
            return pig.getAliasKeySet();
         }
      });
  }
}
```

The example above shows a basic container configuration wiring a `PigTemplate` into a user class which uses it to interact with the `PigServer` API. Notice that the user does not have to handle the lifecycle of the `PigServer` instance or catch any exception - these are handled automatically by the template which converts them, like the rest of the Spring templates, into `DataAccessExceptions`. Thus the application only has to track only one exception hierarchy across all data technologies instead of one per technology.

8. Cascading integration

SHDP provides basic support for Cascading library through the `org.springframework.data.hadoop.cascading` package - one can create `Flows` or `Cascades`, 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 `Taps` for Spring environments are available.

As Cascading is aimed at code configuration, typically one would configure the library programatically. Such code can easily be integrated into Spring in various ways - through `factory methods` or `@Configuration` and `@Bean` (see this chapter for more information). In short one uses Java code (or any JVM language for that matter) to create beans.

For example, looking at the official Cascading sample (Cascading for the Impatient, Part2) one can simply call the Cascading setup method from within the Spring container (original vs updated):

```
public class Impatient {
    public static FlowDef createFlowDef(String docPath, String wcPath) {
        // create source and sink taps
        Tap docTap = new Hfs(new TextDelimited(true, "\t"), docPath);
        Tap wcTap = new Hfs(new TextDelimited(true, "\t"), wcPath);

        // specify a regex operation to split the "document" text lines into a token
 stream
        Fields token = new Fields("token");
        Fields text = new Fields("text");
        RegexSplitGenerator splitter = new RegexSplitGenerator(token, "[ \\[\\]\\(\
\),.]");
        // only returns "token"
        Pipe docPipe = new Each("token", text, splitter, Fields.RESULTS);

        // determine the word counts
        Pipe wcPipe = new Pipe("wc", docPipe);
        wcPipe = new GroupBy(wcPipe, token);
        wcPipe = new Every(wcPipe, Fields.ALL, new Count(), Fields.ALL);

        // connect the taps, pipes, etc., into a flow
        FlowDef flowDef = FlowDef.flowDef().setName("wc").addSource(docPipe,
 docTap).addTailSink(wcPipe, wcTap);
        return flowDef; }
}
```

The entire Cascading configuration (defining the `Flow`) is encapsulated within one method, which can be called by the container:

```
<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:c="http://www.springframework.org/schema/c"
    xmlns:p="http://www.springframework.org/schema/p"
    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/hadoop/spring-hadoop.xsd
        http://www.springframework.org/schema/context http://www.springframework.org/
schema/context/spring-context.xsd">

    <!-- factory-method approach called with two parameters available as property
 placeholders -->
    <bean id="flowDef" class="impatient.Main" factory-
method="createFlowDef" c:_0="${in}" c:_1="${out}"/>

    <hdp:cascading-flow id="wc" definition-ref="flowDef" write-dot="dot/wc.dot"/>
    <hdp:cascading-cascade id="cascade" flow-ref="wc"/>
    <hdp:cascading-runner unit-of-work-ref="cascade" run-at-startup="true"/>
</beans>
```

Note that no jar needs to be setup - the Cascading namespace (in particular `cascading-flow`, backed by `HadoopFlowFactoryBean`) tries to automatically setup the resulting job classpath. By default, it will automatically add the Cascading library and its dependency to Hadoop `DistributedCache` so that when the job runs inside the Hadoop cluster, the jars are properly found. When using custom jars (for example to add custom Cascading functions) or when running against a cluster that is already provisioned, one can customize this behaviour through the `jar-setup`, `jar` and `jar-by-class`. For Cascading users, these settings are the equivalent of the `AppProps.setApplicationJarClass()`.

Furthermore, one can break down the configuration method in multiple pieces which is useful for reusing the components between multiple flows/cascades. This goes hand in hand with Spring `@Configuration` feature - see the example below that configures a Cascade pipes and taps as individual beans (see the original example):

```
@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}"/>

<!-- standard bean declaration used to showcase the container flexibility -->
<!-- note the tap and sinks are imported from the CascadingAnalysisConfig bean -->
<bean id="analysisFlow" class="org.springframework.data.hadoop.cascading.HadoopFlowFactoryBean" p:configura
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>

<hdp:cascading-cascade flow="analysisFlow" />
<hdp:cascading-runner unit-of-work-ref="cascade" run-at-startup="true"/>
```

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 `cascading-runner`). The runner will trigger the execution during the application start-up (notice the `run-at-startup` flag which is by default `false`). Do note that the runner will not run unless triggered manually or if `run-at-startup` is set to `true`. Additionally the runner (as in fact do all runners in SHDP) allows one or multiple `pre` and `post` actions to be specified to be executed before and after each run. Typically other runners (such as other jobs or scripts) can be specified but any JDK `Callable` can be passed in. For more information on runners, see the dedicated chapter.

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 `FactoryBeans` above handle the lifecycle and some default configuration for both the `Flow` and `Cascade` objects. Either way, whatever option is used, SHDP fully supports it.

8.1 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 straightforward:

```
<hdp:tasklet p:unit-of-work-ref="cascade" />
```

8.2 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 all DSLs.

As with the rest of the DSLs, Scalding offers a simplified, fluent syntax for creating units of code that build on top of Cascading. This in turn translates 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 of 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>
```

8.3 Spring-specific local Taps

Why only local Tap?

Because Hadoop is designed as 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 8.1. 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 `Taps` do not require any special configuration and are fully compatible with the existing Cascading local `Schemes`. 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.

9. Using the *runner* classes

Spring for Apache Hadoop provides for each Hadoop interaction type, whether it is vanilla Map/Reduce, Cascading, Hive or Pig, a *runner*, a dedicated class used for declarative (or programmatic) interaction. The list below illustrates the existing *runner* classes for each type, their name and namespace element.

Table 9.1. Available Runners

Type	Name	Namespace element	Description
Map/ Reduce `Job`	`JobRunner`	`job-runner`	Runner for Map/Reduce jobs, whether vanilla M/R or streaming
Hadoop `Tool`	`ToolRunner`	`tool-runner`	Runner for Hadoop `Tool`s (whether stand-alone or as jars).
Hadoop `jars`	`JarRunner`	`jar-runner`	Runner for Hadoop jars.
Hive queries and scripts	`HiveRunner`	`hive-runner`	Runner for executing Hive queries or scripts.
Pig queries and scripts	`PigRunner`	`pig-runner`	Runner for executing Pig scripts.
Cascading `Cascades`	`CascadeRunner`	-	Runner for executing Cascading `Cascades`.
JSR-223/ JVM scripts	`HdfsScriptRunner`	`script`	Runner for executing JVM 'scripting' languages (implementing the JSR-223 API).

While most of the configuration depends on the underlying type, the runners share common attributes and behaviour so one can use them in a predictive, consistent way. Below is a list of common features:

- declaration does **not** imply execution

  The runner allows a script, a job, a cascade to run but the execution can be triggered either programmatically or by the container at start-up.

- `run-at-startup`

  Each runner can execute its action at start-up. By default, this flag is set to `false`. For multiple or on demand execution (such as scheduling) use the `Callable` contract (see below).

- JDK `Callable` interface

  Each runner implements the JDK `Callable` interface. Thus one can inject the runner into other beans or its own classes to trigger the execution (as many or as little times as she wants).

- `pre` and `post` actions

  Each runner allows one or multiple, pre or/and post actions to be specified (to chain them together such as executing a job after another or perfoming clean up). Typically other runners can be used but *any* `Callable` can be specified. The actions will be executed before and after the main action, in the declaration order. The runner uses a *fail-safe* behaviour meaning, any exception will interrupt the run and will propagated immediately to the caller.

- consider Spring Batch

  The runners are meant as a way to execute basic tasks. When multiple executions need to be coordinated and the flow becomes non-trivial, we strongly recommend using Spring Batch which provides all the features of the runners and more (a complete, mature framework for batch execution).

10. 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.

10.1 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 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 it's supposed 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.

10.2 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 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.

11. 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 regain on 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 internally 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.

No 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 benefit from the core values that Spring projects bring to the table, namely enabling modularity, reuse and extensive support for unit and integration testing.

11.1 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 How do I schedule a job with Spring Batch? documention for more info. A middle ground it to use Spring's Quartz integration, see Using the OpenSymphony Quartz Scheduler for more information.

11.2 Batch Job Listeners

Spring Batch lets you attach 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 job. 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

The sample applications have been moved into their own repository so they can be developed independently of the Spring for Apache Hadoop release cycle. They can be found on GitHub https://github.com/spring-projects/spring-hadoop-samples.

The wiki page for the Spring for Apache Hadoop project has more documentation for building and running the examples.

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.

12. Useful Links

- *Spring for Apache Hadoop* - http://www.springframework.org/spring-data/hadoop
- *Spring Data* - http://www.springframework.org/spring-data
- *Spring Data Book* - http://shop.oreilly.com/product/0636920024767.do
- *Spring* - http://spring.io/blog/
- *Apache Hadoop* - http://hadoop.apache.org/
- *Pivotal HD* - http://gopivotal.com/pivotal-products/pivotal-data-fabric/pivotal-hd

Part VI. Appendices

Appendix A. Using Spring for Apache Hadoop with Amazon EMR

A popular option for creating on-demand Hadoop cluster is Amazon Elastic Map Reduce or Amazon EMR service. The user can through the command-line, API or a web UI configure, start, stop and manage a Hadoop cluster in the *cloud* without having to worry about the actual set-up or hardware resources used by the cluster. However, as the setup is different then a *locally* available cluster, so does the interaction between the application that want to use it and the target cluster. This section provides information on how to setup Amazon EMR with Spring for Apache Hadoop so the changes between a using a local, pseudo-distributed or owned cluster and EMR are minimal.

Important

This chapter assumes the user is familiar with Amazon EMR and the cost associated with it and its related services - we strongly recommend getting familiar with the official EMR documentation.

One of the big differences when using Amazon EMR versus a local cluster is the lack of access of the file system server and the job tracker. This means submitting jobs or reading and writing to the file-system isn't available out of the box - which is understandable for security reasons. If the cluster would be open, if could be easily abused while charging its rightful owner. However, it is fairly straight-forward to get access to both the file system and the job tracker so the deployment flow does not have to change.

Amazon EMR allows clusters to be created through the management console, through the API or the command-line. This documentation will focus on the command-line but the setup is not limited to it - feel free to adjust it according to your needs or preference. Make sure to properly setup the credentials so that the S3 file-system can be properly accessed.

A.1 Start up the cluster

Important

Make sure you read the whole chapter before starting up the EMR cluster

A nice feature of Amazon EMR is starting a cluster for an indefinite period. That is rather then submitting a job and creating the cluster until it finished, one can create a cluster (along side a job) but request to be kept *alive* even if there is no work for it. This is easily done through the `--create --alive` parameters:

```
./elastic-mapreduce --create --alive
```

The output will be similar to this:

```
Created job flowJobFlowID
```

One can verify the results in the console through the `list` command or through the web management console. Depending on the cluster setup and the user account, the Hadoop cluster initialization should be complete anywhere between 1 to 5 minutes. The cluster is ready once its state changes from `STARTING/PROVISIONING` to `WAITING`.

Note

By default, each newly created cluster has a new public IP that is not typically reused. To simplify the setup, one can use Amazon Elastic IP, that is a static, predefined IP, so that she knows

before-hand the cluster address. Refer to this section inside the EMR documentation for more information. As an alternative, one can use the EC2 API in combinatioon with the EMR API to retrieve the private IP of address of the master node of her cluster or even programatically configure and start the EMR cluster on demand without having to hard-code the private IPs.

However, to remotely access the cluster from outside (as oppose to just running a jar within the cluster), one needs to tweak the cluster settings just a tiny bit - as mentioned below.

A.2 Open an SSH Tunnel as a SOCKS proxy

Due to security reasons, the EMR cluster is not exposed to the outside world and is bound only to the machine internal IP. While you can open up the firewall to allow access (note that you also have to do some port forwarding since again, Hadoop is bound to the cluster internal IP rather then all available network cards), it is recommended to use a SSH tunnel instead. The SSH tunnel provides a secure connection between your machine on the cluster preventing any snooping or man-in-the-middle attacks. Further more it is quite easy to automate and be executed along side the cluster creation, programmatically or through some script. The Amazon EMR docs have dedicated sections on SSH Setup and Configuration and on opening a SSH Tunnel to the master node so please refer to them. Make sure to setup the SSH tunnel as a SOCKS proxy, that is to redirect all calls to remote ports - this is crucial when working with Hadoop (or other applications) that use a range of ports for communication.

A.3 Configuring Hadoop to use a SOCKS proxy

Once the tunnel or the SOCKS proxy is in place, one needs to configure Hadoop to use it. By default, Hadoop makes connections directly to its target which is fine for regular use, but in this case, we need to use the SOCKS proxy to pass through the firewall. One can do so through the `hadoop.rpc.socket.factory.class.default` and `hadoop.socks.server` properties:

```
hadoop.rpc.socket.factory.class.default=org.apache.hadoop.net.SocksSocketFactory
# this configure assumes the SOCKS proxy is opened on local port 6666
hadoop.socks.server=localhost:6666
```

At this point, all Hadoop communication will go through the SOCKS proxy at localhost on port 6666. The main advantage is that all the IPs, domain names, ports are resolved on the 'remote' side of the proxy so one can just start using the remote cluster IPs. However, only the Hadoop client needs to use the proxy - to avoid having the client configuration be read by the cluster nodes (which would mean the nodes would try to use a SOCKS proxy on the remote side as well), make sure the master node (and thus all its nodes) `hadoop-site.xml` marks the default network setting as final (see this blog post for a detailed explanation):

```
<property>
    <name>hadoop.rpc.socket.factory.class.default</name>
    <value>org.apache.hadoop.net.StandardSocketFactory</value>
    <final>true</final>
</property>
```

Simply pass this configuration (and other options that you might have) to the master node using a bootstrap action. One can find this file ready for usage, already deployed to Amazon S3 at s3://dist.springframework.org/release/SHDP/emr-settings.xml. Simply pass the file to command-line used for firing up the EMR cluster:

```
./elastic-mapreduce --create --alive --bootstrap-action s3://elasticmapreduce/bootstrap-
actions/configure-hadoop --args "--site-config-file,s3://dist.springframework.org/release/
SHDP/emr-settings.xml"
```

Note

For security reasons, we recommend copying the 'emr-settings.xml' file to one of your S3 buckets and use that location instead.

A.4 Accessing the file-system

Amazon EMR offers Simple Storage Service, also known as S3 service, as means for durable read-write storage for EMR. While the cluster is active, one can write additional data to HDFS but unless S3 is used, the data will be lost once the cluster shuts down. Note that when using an S3 location for the first time, the proper access permissions needs to be setup. Accessing S3 is easier then the job tracker - in fact the Hadoop distribution provides not one but two file-system implementations for S3:

Table A.1. Hadoop S3 File Systems

Name	URI Prefix	Access Method	Description
S3 Native FS	`s3n://`	S3 Native	Native access to S3. The recommended file-system as the data is read/written in its native format and can be used not just by Hadoop but also other systems without any translation. The down-side is that it does not support large files (5GB) out of the box (though there is a work-around through the multipart upload feature).
S3 Block FS	`s3://`	Block Based	The files are stored as blocks (similar to the underlying structure in HDFS). This is somewhat more efficient in terms of renames and file sizes but requires a dedicated bucket and is not inter-operable with other S3 tools.

To access the data in S3 one can either use an HDFS file-system on top of it, which requires no extra setup, or copy the data from S3 to the HDFS cluster using manual tools, distcp with S3, its dedicated version s3distcp, Hadoop DistributedCache (which SHDP supports as well) or third-party tools such as s3cmd.

For newbies and development we recommend accessing the S3 directly through the File-System abstraction as in most cases, its performance is close to that of the data inside the native HDFS. When dealing with data that is read multiple times, copying the data from S3 locally inside the cluster might improve performance but we advice running some performance tests first.

A.5 Shutting down the cluster

Once the cluster is no longer needed for a longer period of time, one can shut it down fairly straight forward:

```
./elastic-mapreduce --terminate JobFlowID
```

Note that the EMR cluster is billed by the hour and since the time is rounded upwards, starting and shutting down the cluster repeateadly might end up being more expensive then just keeping it alive. Consult the documentation for more information.

A.6 Example configuration

To put it all together, to use Amazon EMR one can use the following work-flow with SHDP:

- Start an *alive* cluster using the bootstrap action to guarantees the cluster does NOT use a socks proxy. Open a SSH tunnel, in SOCKS mode, to the EMR cluster.
  Start the cluster for an indefinite period. Once the server is up, create an SSH tunnel,in SOCKS mode, to the remote cluster. This allows the client to communicate directly with the remote nodes as if they are part of the same network.This step does not have to be repeated unless the cluster is terminated - one can (and should) submit multiple jobs to it.

- Configure SHDP

- Once the cluster is up and the SSH tunnel/SOCKS proxy is in place, point SHDP to the new configuration. The example below shows how the configuration can look like:
  hadoop-context.xml

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

<!-- property placeholder backed by hadoop.properties -->
<context:property-placeholder location="hadoop.properties"/>

<!-- Hadoop FileSystem using a placeholder and emr.properties -->
<hdp:configuration properties-location="emr.properties" file-system-uri="${hd.fs}" job-
tracker-uri="${hd.jt}/>
```

  hadoop.properties

```
# Amazon EMR
# S3 bucket backing the HDFS S3 fs
hd.fs=s3n://my-working-bucket/
# job tracker pointing to the EMR internal IP
hd.jt=10.123.123.123:9000
```

  emr.properties

```
# Amazon EMR
# Use a SOCKS proxy
hadoop.rpc.socket.factory.class.default=org.apache.hadoop.net.SocksSocketFactory
hadoop.socks.server=localhost:6666

# S3 credentials
# for s3:// uri
fs.s3.awsAccessKeyId=XXXXXXXXXXXXXXXXXXX
fs.s3.awsSecretAccessKey=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

# for s3n:// uri
fs.s3n.awsAccessKeyId=XXXXXXXXXXXXXXXXXXX
fs.s3n.awsSecretAccessKey=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Spring Hadoop is now ready to talk to your Amazon EMR cluster. Try it out!

Note

The inquisitive reader might wonder why the example above uses two properties file, `hadoop.properties` and `emr.properties` instead of just one. While one file is enough, the example tries to isolate the EMR configuration into a separate configuration (especially as it contains security credentials).

- Shutdown the tunnel and the cluster

  Once the jobs submitted are completed, unless new jobs are shortly scheduled, one can shutdown the cluster. Just like the first step, this is optional. Again, make sure you understand the billing process first.

Appendix B. Using Spring for Apache Hadoop with EC2/Apache Whirr

As mentioned above, those interested in using on-demand Hadoop clusters can use Amazon Elastic Map Reduce (or Amazon EMR) service. An alternative to that, for those that want maximum control over the cluster, is to use Amazon Elastic Compute Cloud or EC2. EC2 is in fact the service on top of which Amazon EMR runs and that is, a resizable, configurable compute capacity in the cloud.

Important

This chapter assumes the user is familiar with Amazon EC2 and the cost associated with it and its related services - we strongly recommend getting familiar with the official EC2 documentation.

Just like Amazon EMR, using EC2 means the Hadoop cluster (or whatever service you run on it) runs in the cloud and thus 'development' access to it, is different then when running the service in local network. There are various tips and tools out there that can handle the initial provisioning and configure the access to the cluster. Such a solution is Apache Whirr which is a set of libraries for running cloud services. Though it provides a Java API as well, one can easily configure, start and stop services from the command-line.

B.1 Setting up the Hadoop cluster on EC2 with Apache Whirr

The Whirr documentation provides more detail on how to interact with the various cloud providers out-there through Whirr. In case of EC2, one needs Java 6 (which is required by Apache Hadoop), an account on EC2 and an SSH client (available out of the box on *nix platforms and freely downloadable (such as PuTTY) on Windows). Since Whirr does most of the heavy lifting, one needs to tell Whirr what Cloud provider and account is used, either by setting some environment properties or through the `~/.whirr/credentials file`:

```
whirr.provider=aws-ec2
whirr.identity=your-aws-key
whirr.credential=your-aws-secret
```

Now instruct Whirr to configure a Hadoop cluster on EC2 - just add the following properties to a configuration file (say `hadoop.properties`):

```
whirr.cluster-name=myhadoopcluster
whirr.instance-templates=1 hadoop-jobtracker+hadoop-namenode,1 hadoop-datanode+hadoop-
tasktracker
whirr.provider=aws-ec2
whirr.private-key-file=${sys:user.home}/.ssh/id_rsa
whirr.public-key-file=${sys:user.home}/.ssh/id_rsa.pub
```

The configuration above assumes the SSH keys for your user have been already generated. Now start your Hadoop cluster:

```
bin/whirr launch-cluster --config hadoop.properties
```

As with Amazon EMR, one cannot correct to the Hadoop cluster from outside - however Whirr provides out of the box the feature to create an SSH tunnel to create a SOCKS proxy (on port 6666). When

a cluster is created, Whirr creates a script to launch the cluster which may be found in `~/.whirr/cluster-name`. Run it as a follows (in a new terminal window):

```
~/.whirr/myhadoopcluster/hadoop-proxy.sh
```

At this point, one can just the SOCKS proxy configuration from the Amazon EMR section to configure the Hadoop client.

To destroy the cluster, one can use the Amazon EMR console or Whirr itself:

```
bin/whirr destroy-cluster --config hadoop.properties
```

Appendix C. 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">

 <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>

 <!-- common attributes shared by Job executors
   NOT meant for extensibility - do NOT rely on this type as it might be removed in the
 future -->
 <xsd:complexType name="jobRunnerType">
  <xsd:attribute name="id" type="xsd:ID" use="optional">
   <xsd:annotation>
   <xsd:documentation><![CDATA[
Bean id.]]></xsd:documentation>
   </xsd:annotation>
  </xsd:attribute>
  <!-- the job reference -->
  <xsd:attribute name="job-ref">
   <xsd:annotation>
    <xsd:documentation source="java:org.apache.hadoop.mapreduce.Job"><![CDATA[
Hadoop Job. Multiple names can be specified using comma (,) as a separator.]]></
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-
completion" type="xsd:string" use="optional" default="true">
   <xsd:annotation>
    <xsd:documentation><![CDATA[
Whether to synchronously wait for the job(s) to finish (the default) or not.
    ]]></xsd:documentation>
   </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="verbose" type="xsd:string" use="optional" default="true"/>
  <xsd:attribute name="kill-job-at-
shutdown" type="xsd:string" use="optional" default="true">
   <xsd:annotation>
    <xsd:documentation><![CDATA[
Whether the configured jobs should be 'killed' when the application shuts down (default)
 or not.
For long-running or fire-and-forget jobs that live beyond the starting application, set
 this to false.

Note that if 'wait-for-job' is true, this flag is considered to be true as otherwise the
 application
cannot shut down (since it has to keep waiting for the job).
    ]]></xsd:documentation>
   </xsd:annotation>
  </xsd:attribute>
  <xsd:attribute name="executor-ref" type="xsd:string" use="optional">
   <xsd:annotation>
```