Spring for Apache Hadoop
- Reference Documentation

2.0.2.RELEASE-cdh5

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 2.0 is built and tested with JDK 7.0 (generated jars are usable in JDK 6.0 and above), Spring Framework 4.0 and is by default built against Apache Hadoop 2.2.0.

Spring for Apache Hadoop 2.0 supports the following versions and distributions:

- Apache Hadoop 1.2.1

- Apache Hadoop 2.2.0 *

- Apache Hadoop 2.4.0 *

- Pivotal HD 1.1

- Pivotal HD 2.0 *

- Cloudera CDH4 (CDH4.6.0)

- Cloudera CDH5 (2.3.0-cdh5.0.0) *

- Hortonworks Data Platform 1.3

- Hortonworks Data Platform 2.0 *

- Hortonworks Data Platform 2.1 *

* The distributions noted with and asterisk will include spring-yarn support in the build.

Any distribution compatible with Apache Hadoop 1.x or 2.2.x should be supported.

> **Note**
>
> Spring for Apache Hadoop has been certified to work on Pivotal HD 1.0 and 1.1, Hortonworks HDP 1.3 and Cloudera CDH 4.4 distributions. Further certifications will be done soon now that Spring for Apache 2.0 has reached GA.

Spring for Apache Hadoop 2.0 is tested daily against a number of Hadoop distributions. See the test plan page for current status.

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

Regarding Hadoop-related projects, SDHP supports HBase 0.94.11, Hive 0.10.0 and Pig 0.10.1 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.

To take full advantage of Spring for Apache Hadoop you need a running Hadoop cluster. If you don't already have one in your environment, a good first step is to create a single-node cluster. To install the most recent stable verision of Hadoop, the "Getting Started" page from the official Apache documentation is a good general guide. There should be a link for "Single Node Setup".

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.** Official repository full of SHDP samples demonstrating the various project features.

- **Spring Data Book.** 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 examples.** 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 6, *Working with HBase* describes the Spring support for HBase.

Chapter 7, *Hive integration* describes the Hive integration in SHDP.

Chapter 8, *Pig support* describes the Pig support 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
<?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:

Note

Usual configuration parameters for `fs.default.name`, `fs.defaultFS`, `mapred.job.tracker` and `yarn.resourcemanager.address` can be configured using tag attributes *file-system-uri*, *job-tracker-uri* and *rm-manager-uri* respectively.

```xml
<?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
<?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:

```xml
<!-- 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.

Table 3.1. `hdp:configuration` attributes

Name	Values	Description
`configuration-ref`	Bean Reference	Reference to existing *Configuration* bean
`properties-ref`	Bean Reference	Reference to existing *Properties* bean
`properties-location`	Comma delimited list	List or Spring *Resource* paths
`resources`	Comma delimited list	List or Spring *Resource* paths
`file-system-uri`	String	The HDFS filesystem address. Equivalent to *fs.default.name* and *fs.defaultFS* propertys.
`job-tracker-uri`	String	Job tracker address for HadoopV1. Equivalent to *mapred.job.tracker* property.
`rm-manager-uri`	String	The Yarn Resource manager address for HadoopV2. Equivalent to *yarn.resourcemanager.address* property.

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:

```xml
<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](cat) and [wc](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:

```xml
<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 [configuration]()s.

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:

```xml
<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:

```xml
<!-- 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.

```java
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:

```java
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 `HdfsResource`s.

If there is a need for *Spring Application Context* to be aware of `HdfsResourceLoader` it needs to be registered using `hdp:resource-loader-registrar` namespace tag.

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

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.

Table 4.3. `hdp:resource-loader-registrar` *attributes*

Name	Values	Description
loader-ref	Bean Reference	Reference to existing *Hdfs resource loader* bean. Default value is 'hadoopResourceLoader'.

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):

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

  <hdp:script id="inlined-js" language="javascript" run-at-startup="true">
    try {load("nashorn:mozilla_compat.js");} catch (e) {} // for Java 8
    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:

```xml
<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:

```xml
<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:

```python
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 4.4. 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

Name	Type	Description
org.springframework.data.hadoop.fs.DistributedCp `distcp`		Programmatic access to DistCp
org.apache.hadoop.fs.FileSystem `fs`		Hadoop File System (relies on 'hadoop-fs' bean or singleton type match, falls back to creating one based on 'cfg')
org.springframework.data.hadoop.FileSystem `fsh`		File System shell, exposing hadoop 'fs' commands as an API
org.springframework.data.hadoop.io.HdfsResources `hdfsRL`		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 4.5. `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 `String`s) 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. Writing and reading data using the Hadoop File System

The Store sub-project of Spring for Apache Hadoop provides abstractions for writing and reading various types of data residing in HDFS. We currently support different file types either via our own store accessors or by using the Dataset support in *Kite SDK*.

Currently, the Store sub-project doesn't have an XML namespace or javaconfig based configuration classes as it's considered to be a foundational library. However, this may change in future releases.

5.1 Store Abstraction

Native store abstractions provide various writer and reader interfaces so that the end user don't have to worry about the underlying implementation actually doing the work on files in HDFS. Implementations are usually strongly typed and provides constructors and setters for additional setup to work with naming, compression codecs and everything else defining the behaviour. Interfaces are meant to be used from integration components which don't need to know the internal workings of writers and readers.

Writing Data

Main interface writing into a store is a `DataWriter` which have one method `write` which simply writes an entity and the backing implementation will handle the rest.

```
public interface DataWriter<T> {
  void write(T entity) throws IOException;
}
```

The `DataStoreWriter` interface adds methods to close and flush a writer. Some of the writers have a property to close a stream after an idle time has been reached but generally this interface is meant for programmatic control of these operations.

```
public interface DataStoreWriter<T> extends DataWriter<T>, Flushable, Closeable {
}
```

File Naming

Different file naming strategies are used to automatically determine the name of a file to be used. Writers without additional naming configuration will usually use a given base path as is. As soon as any type of a strategy is configured, given base path is considered to be a base directory and the name of the file is resolved by file naming strategies.

For example, if defined base path is "`/tmp/path`" and the `StaticFileNamingStrategy` with "`data`" parameter is used then the actual file path resolved would be "`/tmp/path/data`".

```
Path path = new Path("/tmp/path");
Configuration config = new Configuration();
TextFileWriter writer = new TextFileWriter(config, path, null);
StaticFileNamingStrategy fileNamingStrategy = new StaticFileNamingStrategy("data")
writer.setFileNamingStrategy(fileNamingStrategy);
```

At first look this may feel a little complicated, but it will make sense after more file naming strategies are added. These will also provide facilities for using writers in parallel, or for a re-launched

writer to be able to create a new file based on already existing files in the directry. For example, `RollingFileNamingStrategy` will add a simple increasing value to a file name and will try to initialize itself with the correct position.

Built-in strategies currently supported are `StaticFileNamingStrategy`, `RollingFileNamingStrategy`, `UuidFileNamingStrategy` and `CodecFileNamingStrategy`. `ChainedFileNamingStrategy` can be used to chain multiple strategies together where each individual strategy will provide its own part.

File Rollover

File rolling strategy is used to determine a condition in a writer when a current stream should be automatically closed and the next file should be opened. This is usually done together with `RollingFileNamingStrategy` to rollover when a certain file size limit has been reached.

Currently, only one strategy `SizeRolloverStrategy` is supported.

Partitioning

Partitioning is a concept of choosing a target file on demand either based on content to be written or any other information available to a writer at the time of the write operation. While it would be perfectly alright to use multiple writers manually, the framework already does all the heavy lifting around partitioning. We work through interfaces and provide a generic default implementation still allowing to plug a customized version if there's a need for it.

`PartitionStrategy` is a strategy interface defining `PartitionResolver` and `PartitionKeyResolver`.

```java
public interface PartitionStrategy<T,K> {
  PartitionResolver<K> getPartitionResolver();
  PartitionKeyResolver<T, K> getPartitionKeyResolver();
}
```

`PartitionResolver` is an interface used to resolve arbitrary partition keys into a path. We don't force any specific partition key type in the interface level itself but usually the implementation needs to be aware of its type.

```java
public interface PartitionResolver<K> {
  Path resolvePath(K partitionKey);
}
```

`PartitionKeyResolver` is an interface which is responsible for creating a partition key from an entity. This is needed because writer interfaces allow us to write entities without an explicit partition key.

```java
public interface PartitionKeyResolver<T, K> {
  K resolvePartitionKey(T entity);
}
```

`PartitionDataStoreWriter` is an extension of `DataStoreWriter` adding a method to write an entity with a partition key. In this context the partition key is something what the partition strategy is able to use.

```java
public interface PartitionDataStoreWriter<T,K> extends DataStoreWriter<T> {
  void write(T entity, K partitionKey) throws IOException;
}
```

DefaultPartitionStrategy

`DefaultPartitionStrategy` is a generic default implementation meant to be used together with an expression using Spring's *SpEL* expression language. `PartitionResolver` used in `DefaultPartitionStrategy` expects partition key to be a type of Map<String,Object> and partition key created by `PartitionKeyResolver` is a `DefaultPartitionKey` which itself is a Map<String,Object>.

In order to make it easy to work with SpEL and partitioning, map values can be directly accessed with keys and additional partitioning methods has been registered.

Partition Path Expression

SpEL expression is evaluated against a partition key passed into a HDFS writer.

Accessing Properties

If partition key is a type of Map any property given to a SpEL expression is automatically resolved from a map.

Custom Methods

In addition to normal SpEL functionality, a few custom methods have been added to make it easier to build partition paths. These custom methods can be used to work with normal partition concepts like date formatting, lists, ranges and hashes.

path

```
path(String... paths)
```

You can concatenate paths together with a / delimiter. This method can be used to make the expression less verbose than using a native SpEL functionality to combine path parts together. To create a path *part1/part2*, expression *'part1' + '/' + 'part2'* is equivalent to *path('part1','part2')*.

Parameters

paths.   Any number of path parts

Return Value

Concatenated value of paths delimited with /.

dateFormat

```
dateFormat(String pattern)
dateFormat(String pattern, Long epoch)
dateFormat(String pattern, Date date)
dateFormat(String pattern, String datestring)
dateFormat(String pattern, String datestring, String dateformat)
```

Creates a path using date formatting. Internally this method delegates to `SimpleDateFormat` and needs a Date and a pattern.

Method signature with three parameters can be used to create a custom `Date` object which is then passed to `SimpleDateFormat` conversion using a dateformat pattern. This is useful in use cases

where partition should be based on a date or time string found from a payload content itself. Default dateformat pattern if omitted is *yyyy-MM-dd*.

Parameters

pattern.   Pattern compatible with SimpleDateFormat to produce a final output.

epoch.   Timestamp as Long which is converted into a Date.

date.   A Date to be formatted.

dateformat.   Secondary pattern to convert datestring into a Date.

datestring.   Date as a String

Return Value

A path part representation which can be a simple file or directory name or a directory structure.

list

```
list(Object source, List<List<Object>> lists)
```

Creates a partition path part by matching a source against a lists denoted by *lists*.

Lets assume that data is being written and it's possible to extract an *appid* from the content. We can automatically do a list based partition by using a partition method *list(appid, {{'1TO3','APP1','APP2','APP3'},{'4TO6','APP4','APP5','APP6'}})*. This method would create three partitions, *1TO3_list*, *4TO6_list* and *list*. The latter is used if no match is found from partition lists passed to lists.

Parameters

source.   An Object to be matched against lists.

lists.   A definition of list of lists.

Return Value

A path part prefixed with a matched key i.e. *XXX_list* or list if no match.

range

```
range(Object source, List<Object> list)
```

Creates a partition path part by matching a source against a list denoted by *list* using a simple binary search.

The partition method takes source as first argument and a list as the second argument. Behind the scenes this is using the JVM's binarySearch which works on an Object level so we can pass in anything. Remember that meaningful range match only works if passed in Object and types in list are of same type like `Integer`. Range is defined by a binarySearch itself so mostly it is to match against an upper bound

except the last range in a list. Having a list of *{1000,3000,5000}* means that everything above 3000 will be matched with 5000. If that is an issue then simply adding *Integer.MAX_VALUE* as last range would overflow everything above 5000 into a new partition. Created partitions would then be *1000_range*, *3000_range* and *5000_range*.

Parameters

source.   An Object to be matched against list.

list.   A definition of list.

Return Value

A path part prefixed with a matched key i.e. XXX_range.

hash

```
hash(Object source, int bucketcount)
```

Creates a partition path part by calculating hashkey using source`s hashCode and bucketcount. Using a partition method *hash(timestamp,2)* would then create partitions named *0_hash*, *1_hash* and *2_hash*. Number suffixed with _hash is simply calculated using *Object.hashCode() % bucketcount*.

Parameters

source.   An Object which hashCode will be used.

bucketcount.   A number of buckets

Return Value

A path part prefixed with a hash key i.e. *XXX_hash*.

Creating a Custom Partition Strategy

Creating a custom partition strategy is as easy as just implementing needed interfaces. Custom strategy may be needed in use cases where it is just not feasible to use SpEL expressions. This will then give total flexibility to implement partitioning as needed.

Below sample demonstrates how a simple customer id could be used as a base for partitioning.

```java
public class CustomerPartitionStrategy implements PartitionStrategy<String, String> {

  CustomerPartitionResolver partitionResolver = new CustomerPartitionResolver();
  CustomerPartitionKeyResolver keyResolver = new CustomerPartitionKeyResolver();

  @Override
  public PartitionResolver<String> getPartitionResolver() {
    return partitionResolver;
  }

  @Override
  public PartitionKeyResolver<String, String> getPartitionKeyResolver() {
    return keyResolver;
  }
}

public class CustomerPartitionResolver implements PartitionResolver<String> {

  @Override
  public Path resolvePath(String partitionKey) {
    return new Path(partitionKey);
  }
}

public class CustomerPartitionKeyResolver implements PartitionKeyResolver<String, String>
 {

  @Override
  public String resolvePartitionKey(String entity) {
    if (entity.startsWith("customer1")) {
      return "customer1";
    } else if (entity.startsWith("customer2")) {
      return "customer2";
    } else if (entity.startsWith("customer3")) {
      return "customer3";
    }
    return null;
  }
}
```

Writer Implementations

We provide a number of writer implementations to be used based on the type of file to write.

- **TextFileWriter.** an implementation meant to write a simple text data where entities are separated by a delimiter. Simple example for this is a text file with line terminations.

- **DelimitedTextFileWriter.** an extension atop of TextFileWriter where written entity itself is also delimited. Simple example for this is a csv file.

- **TextSequenceFileWriter.** a similar implementation to TextFileWriter except that backing file is a Hadoop's SequenceFile.

- **PartitionTextFileWriter.** wraps multiple TextFileWriters providing automatic partitioning functionality.

Reading Data

Main interface reading from a store is a DataReader.

```
public interface DataReader<T> {
  T read() throws IOException;
}
```

`DataStoreReader` is an extension of `DataReader` providing close method for a reader.

```
public interface DataStoreReader<T> extends DataReader<T>, Closeable {
}
```

Input Splits

Some of the HDFS storage and file formats can be read using an input splits instead of reading a whole file at once. This is a fundamental concept in Hadoop's MapReduce to parallelize data processing. Instead of reading a lot of small files, which would be a source of a Hadoop's "small file problem", one large file can be used. However one need to remember that not all file formats support input splitting especially when compression is used.

Support for reading input split is denoted via a Split interface which simply mark starting and ending positions.

```
public interface Split {
  long getStart();
  long getLength();
  long getEnd();
}
```

Interface Splitter defines an contract how Split's are calculate from a given path.

```
public interface Splitter {
  List<Split> getSplits(Path path) throws IOException;
}
```

We provide few generic Splitter implementations to construct Split's.

`StaticLengthSplitter` is used to split input file with a given length.

`StaticBlockSplitter` is used to split input by used HDFS file block size. It's also possible to split further down the road within the blocks itself.

`SlopBlockSplitter` is an extension of `StaticBlockSplitter` which tries to estimate how much a split can overflow to a next block to taggle unnecessary overhead if last file block is very small compared to an actual split size.

Reader Implementations

We provide a number of reader implementations to be used based on the type of file to read.

- **TextFileReader.** used to read data written by a `TextFileWriter`.

- **DelimitedTextFileReader.** used to read data writte by a `DelimitedTextFileWriter`.

- **TextSequenceFileReader.** used to read data written by a `TextSequenceFileWriter`.

Using Codecs

Supported compression codecs are denoted via an interface `CodecInfo` which simply defines if codec supports splitting, what is it's fully qualified java class and what is its default file suffix.

```
public interface CodecInfo {
  boolean isSplittable();
  String getCodecClass();
  String getDefaultSuffix();
}
```

`Codecs` provides an enum for easy access to supported codecs.

- *GZIP* - `org.apache.hadoop.io.compress.GzipCodec`

- *SNAPPY* - `org.apache.hadoop.io.compress.SnappyCodec`

- *BZIP2* - `org.apache.hadoop.io.compress.BZip2Codec`

- *LZO* - `com.hadoop.compression.lzo.LzoCodec` (non-splittable)

- *SLZO* - `com.hadoop.compression.lzo.LzoCodec` (splittable)

5.2 Persisting POJO datasets using Kite SDK

One common requirement is to persist a large number of POJOs in serialized form using HDFS. The Kite SDK project provides a Kite Data Module that provides an API for working with datasets stored in HDFS. We are using this functionality and provide a some simple helper classes to aid in configuration and use in a Spring environment.

Data Formats

The Kite SDK project provides support for writing data using both the Avro and Parquet data formats. The data format you choose to use influences the data types you can use in your POJO classes. We'll discuss the basics of the Java type mapping for the two data formats but we recommend that you consult each project's documentation for additional details.

> ### Note
>
> Currently, you can't provide your own schema. This is something that we are considering changing in upcomming releases. We are also planning to provide better mapping support in line with the support we currently provide for NoSQL stores like MongoDB.

Using Avro

When using Avro as the data format the schema generation is based on reflection of thet POJO class used. Primitive data types and their corresponding wrapper classes are mapped to the corresponding Avro data type. More complex types, as well as the POJO itself, are mapped to a record type consisting of one or more fields.

The table below shows the mapping from some common types:

Table 5.1. Some common Java to Avro data types mapping

Java type	Avro type	Comment
String	string	
int / Integer	int	32-bit signed integer
long / Long	long	64-bit signed integer

Java type	Avro type	Comment
float / Float	float	32-bit floating point
double / Double	double	64-bit floating point
boolean / Boolean	boolean	
byte[]	bytes	byte array
java.util.Date	record	

Using Parquet

When using Parquet as the data format the schema generation is based on reflection of thet POJO class used. The POJO class must be a proper JavaBean and not have any nested types. We only support primitive data types and their corresponding wrapper classes plus byte arrays. We do rely on the Avro-to-Parquet mapping support that the Kite SDK uses, so the schema will be generated by Avro.

> **Note**
>
> The Parquet support we currently povide is considered experimental. We are planning to relax a lot of the restrictions on the POJO class in upcoming releases.

The table below shows the mapping from some common types:

Table 5.2. Some common Java to Parquet data types mapping

Java type	Parquet type	Comment
String	BINARY/UTF8	
int / Integer	INT32	32-bit signed integer
long / Long	INT64	64-bit signed integer
float / Float	FLOAT	32-bit floating point
double / Double	DOUBLE	64-bit floating point
boolean / Boolean	BOOLEAN	
byte[]	BINARY/BYTE_ARRAY	byte array

Configuring the dataset support

In order to use the dataset support you need to configure the following classes:

- `DatasetRepositoryFactory` that needs a `org.apache.hadoop.conf.Configuration` so we know how to connect to HDFS and a base path where the data will be written.

- `DatasetDefinition` that defines the dataset you are writing. Configuration options include the POJO class that is being stored, the type of format to use (Avro or Parquet). You can also specify whether to allow null values for all fields (default is *false*) and an optional partition strategy to use for the dataset (see below for partitioning).

The following example shows a simple configuration class:

Spring for Apache Hadoop
- Reference Documentation

```
@Configuration
@ImportResource("hadoop-context.xml")
public class DatasetConfig {

  private @Autowired org.apache.hadoop.conf.Configuration hadoopConfiguration;

  @Bean
  public DatasetRepositoryFactory datasetRepositoryFactory() {
    DatasetRepositoryFactory datasetRepositoryFactory = new DatasetRepositoryFactory();
    datasetRepositoryFactory.setConf(hadoopConfiguration);
    datasetRepositoryFactory.setBasePath("/user/spring");
    return datasetRepositoryFactory;
  }

  @Bean
  public DatasetDefinition fileInfoDatasetDefinition() {
    DatasetDefinition definition = new DatasetDefinition();
    definition.setFormat(Formats.AVRO.getName());
    definition.setTargetClass(FileInfo.class);
    definition.setAllowNullValues(false);
    return definition;
  }
}
```

Writing datasets

To write datasets to Hadoop you should use either the `AvroPojoDatasetStoreWriter` or the `ParquetDatasetStoreWriter` depending on the data format you want to use.

⭐ **Tip**

> To mark your fields as nullable use the `@Nullable` annotation (`org.apache.avro.reflect.Nullable`). This will result in the schema defining your field as a *union* of *null* and your datatype.

We are using a `FileInfo` POJO that we have defined to hold some information based on the files we read from our local file system. The dataset will be stored in a directory that is the name of the class using lowercase, so in this case it would be *fileinfo*. This directory is placed inside the *basePath* specified in the configuration of the `DatasetRepositoryFactory`.:

```
package org.springframework.samples.hadoop.dataset;

import org.apache.avro.reflect.Nullable;

public class FileInfo {
  private String name;
  private @Nullable String path;
  private long size;
  private long modified;

  public FileInfo(String name, String path, long size, long modified) {
    this.name = name;
    this.path = path;
    this.size = size;
    this.modified = modified;
  }

  public FileInfo() {
  }

  public String getName() {
      return name;
  }

  public String getPath() {
      return path;
  }

  public long getSize() {
      return size;
  }

  public long getModified() {
      return modified;
  }
}
```

To create a writer add the following bean definition to your configuration class:

```
@Bean
public DataStoreWriter<FileInfo> dataStoreWriter() {
  return new AvroPojoDatasetStoreWriter<FileInfo>(FileInfo.class,
      datasetRepositoryFactory(), fileInfoDatasetDefinition());
}
```

Next, have your class use the writer bean:

```
    private DataStoreWriter<FileInfo> writer;

    @Autowired
    public void setDataStoreWriter(DataStoreWriter dataStoreWriter) {
        this.writer = dataStoreWriter;
    }
```

Now we can use the writer, it will be opened automatically once we start writing to it:

```
      FileInfo fileInfo = new FileInfo(file.getName(),
          file.getParent(), (int)file.length(), file.lastModified());
      writer.write(fileInfo);
```

Once we are done writing we should close the writer:

```
    try {
      writer.close();
    } catch (IOException e) {
      throw new StoreException("Error closing FileInfo", e);
    }
```

We should now have dataset containing all the *FileInfo* entries in a `/user/spring/demo/fileinfo` directory:

```
$ hdfs dfs -ls /user/spring/*
Found 2 items
drwxr-xr-x   - spring supergroup          0 2014-06-09 17:09 /user/spring/
fileinfo/.metadata
-rw-r--r--   3 spring supergroup   13824695 2014-06-09 17:10 /user/spring/
fileinfo/6876f250-010a-404a-b8c8-0ce1ee759206.avro
```

The `.metadata` directory contains dataset information including the Avro schema:

```
$ hdfs dfs -cat /user/spring/fileinfo/.metadata/schema.avsc
{
  "type" : "record",
  "name" : "FileInfo",
  "namespace" : "org.springframework.samples.hadoop.dataset",
  "fields" : [ {
    "name" : "name",
    "type" : "string"
  }, {
    "name" : "path",
    "type" : [ "null", "string" ],
    "default" : null
  }, {
    "name" : "size",
    "type" : "long"
  }, {
    "name" : "modified",
    "type" : "long"
  } ]
}
```

Reading datasets

To read datasets to Hadoop we use the `DatasetTemplate` class.

To create a `DatasetTemplate` add the following bean definition to your configuration class:

```
  @Bean
  public DatasetOperations datasetOperations() {
    DatasetTemplate datasetOperations = new DatasetTemplate();
    datasetOperations.setDatasetRepositoryFactory(datasetRepositoryFactory());
    return datasetOperations;
  }
```

Next, have your class use the `DatasetTemplate`:

```
    private DatasetOperations datasetOperations;

    @Autowired
    public void setDatasetOperations(DatasetOperations datasetOperations) {
        this.datasetOperations = datasetOperations;
    }
```

Now we can read and count the entries using a `RecordCallback` callback interface that gets called once per retrieved record:

```
        final AtomicLong count = new AtomicLong();
        datasetOperations.read(FileInfo.class, new RecordCallback<FileInfo>() {
            @Override
            public void doInRecord(FileInfo record) {
                count.getAndIncrement();
            }
        });
        System.out.println("File count: " + count.get());
```

Partitioning datasets

To create datasets that are partitioned on one or more data fields we use the `PartitionStrategy.Builder` class that the *Kite SDK* project provides.

```
DatasetDefinition definition = new DatasetDefinition();
definition.setPartitionStrategy(new PartitionStrategy.Builder().year("modified").build());
```

This option lets you specify one or more paths that will be used to partition the files that the data is written to based on the content of the data. You can use any of the `FieldPartitioner`s that are available for the *Kite SDK* project. We simply use what is specified to create the corresponding partition strategy. The following partitioning functions are available:

- *year, month, day, hour, minute* creates partitions based on the value of a timestamp and creates directories named like "YEAR=2014" (works well with fields of datatype long)

  - specify function plus field name like:

    ```
    year("timestamp")
    ```

  - optionally, specify a partition name to replace the default one:

    ```
    year("timestamp", "YY")
    ```

- *dateformat* creates partitions based on a timestamp and a dateformat expression provided - creates directories based on the name provided (works well with fields of datatype long)

  - specify function plus field name, a name for the partition and the date format like:

    ```
    dateFormat("timestamp", "Y-M", "yyyyMM")
    ```

- *range* creates partitions based on a field value and the upper bounds for each bucket that is specified (works well with fields of datatype int and string)

  - specify function plus field name and the upper bounds for each partition bucket like:

    ```
    range("age", 20, 50, 80, Integer.MAX_VALUE)
    ```

- *identity* creates partitions based on the exact value of a field (works well with fields of datatype string, long and int)

  - specify function plus field name, a name for the partition, the type of the field (String or Integer) and the number of values/buckets for the partition like:

    ```
    identity("region", "R", String.class, 10)
    ```

- *hash* creates partitions based on the hash calculated from the value of a field divided into a number of buckets that is specified (works well with all data types)

  - specify function plus field name and number of buckets like:

    ```
    hash("lastname", 10)
    ```

Multiple expressions can be specified by simply chaining them like:

```
identity("region", "R", String.class, 10).year("timestamp").month("timestamp")
```

6. Working with HBase

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

```xml
<!-- 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:

```xml
<!-- 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):

```xml
<!-- 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](#):

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

6.1 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-liner*s) for common operations. Below are some examples of how the template usage looks like:

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

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

The first snippet 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.

7. 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.

7.1 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:

```xml
<!-- 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:

```xml
<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.

7.2 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:

```xml
<!-- 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:

```xml
<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.

7.3 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`:

```xml
<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 `Connection`s or better yet, use Spring's `JdbcTemplate` as in the example above.

7.4 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.

7.5 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"/>
```

```java
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 `DataAccessException`s. Thus the application only has to track only one exception hierarchy across all data technologies instead of one per technology.

8. 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.

8.1 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:

```xml
<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.

8.2 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:

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

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

```java
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 `DataAccessException`s. Thus the application only has to track only one exception hierarchy across all data technologies instead of one per technology.

9. Using the *runner* classes

Spring for Apache Hadoop provides for each Hadoop interaction type, whether it is vanilla Map/Reduce, 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 `jar`s	`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.
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 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*).

11. Yarn Support

You've propbably seen a lot of topics around Yarn and next version of Hadoop's Map Reduce called *MapReduce Version 2*. Originally Yarn was a component of MapReduce itself created to overcome some performance issues in Hadoop's original design. The fundamental idea of MapReduce v2 is to split up the two major functionalities of the JobTracker, resource management and job scheduling/monitoring, into separate daemons. The idea is to have a global *Resource Manager* (RM) and per-application *Application Master* (AM). An application is either a single job in the classical sense of Map-Reduce jobs or a group of jobs.

Let's take a step back and see how original *MapReduce Version 1* works. *Job Tracker* is a global singleton entity responsible for managing resources like per node *Task Trackers* and job life-cycle. *Task Tracker* is responsible for executing tasks from a *Job Tracker* and periodically reporting back the status of the tasks. Naturally there is a much more going on behind the scenes but the main point of this is that the *Job Tracker* has always been a bottleneck in terms of scalability. This is where Yarn steps in by splitting the load away from a global resource management and job tracking into per application masters. Global resource manager can then concentrate in its main task of handling the management of resources.

> ### 🍃 Note
>
> Yarn is usually referred as a synonym for *MapReduce Version 2*. This is not exactly true and it's easier to understand the relationship between those two by saying that *MapReduce Version 2* is an application running on top of *Yarn*.

As we just mentioned *MapReduce Version 2* is an application running of top of *Yarn*. It is possible to make similar custom *Yarn* based application which have nothing to do with *MapReduce*. *Yarn* itself doesn't know that it is running *MapReduce Version 2*. While there's nothing wrong to do everything from scratch one will soon realise that steps to learn how to work with *Yarn* are rather deep. This is where Spring Hadoop support for Yarn steps in by trying to make things easier so that user could concentrate on his own code and not having to worry about framework internals.

11.1 Using the Spring for Apache Yarn Namespace

To simplify configuration, SHDP provides a dedicated namespace for *Yarn* 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
<?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:❶yarn="❷http://www.springframework.org/schema/yarn"
  xmlns:❸yarn-int="❹http://www.springframework.org/schema/yarn/integration"
  xmlns:❺yarn-batch="❻http://www.springframework.org/schema/yarn/batch"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/yarn
    http://www.springframework.org/schema/yarn/spring-yarn.xsd❼
    http://www.springframework.org/schema/yarn/integration
    http://www.springframework.org/schema/yarn/integration/spring-yarn-integration.xsd❽
    http://www.springframework.org/schema/yarn/batch
    http://www.springframework.org/schema/yarn/batch/spring-yarn-batch.xsd❾">

  <bean id ... >

  ❿<yarn:configuration ...>
</beans>
```

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

❷   The namespace URI.

❸   Spring for Apache Hadoop Yarn namespace prefix for integration package. Any name can do but through out the reference documentation, the `yarn-int` will be used.

❹   The namespace URI.

❺   Spring for Apache Hadoop Yarn namespace prefix for batch package. Any name can do but through out the reference documentation, the `yarn-batch` will be used.

❻   The namespace URI.

❼   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 Yarn library.

❽   The namespace URI location.

❾   The namespace URI location.

❿   Declaration example for the Yarn namespace. Notice the prefix usage.

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

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/yarn"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/yarn
    http://www.springframework.org/schema/yarn/spring-yarn.xsd">

    ❸<beans:bean id ... >

    ❹<configuration ...>

</beans:beans>
```

❶   The default namespace declaration for this XML file points to the Spring for Apache Yarn namespace.

❷   The beans namespace prefix declaration.

❸   Bean declaration using the `<beans>` namespace. Notice the prefix.

❹   Bean declaration using the `<yarn>` namespace. Notice the *lack* of prefix (as `yarn` is the default namespace).

11.2 Using the Spring for Apache Yarn JavaConfig

It is also possible to work without XML configuration and rely on Annotation based configuration model. XML and JavaConfig for *Spring YARN* are not full replacement for each others but we try to mimic the behaviour as much as we can.

We basically rely on two concepts when working with JavaConfig. Firstly an annotation `@EnableYarn` is used to activate different parts of a Spring Configuration depending on *enable* attribute. We can enable configuration for *CONTAINER*, *APPMASTER* or *CLIENT*. Secondly when configuration is enabled one can use `SpringYarnConfigurerAdapter` whose callback methods can be used to do further configuration for components familiar from XML.

```
@Configuration
@EnableYarn(enable=Enable.CONTAINER)
public class ContainerConfiguration extends SpringYarnConfigurerAdapter {

  @Override
  public void configure(YarnContainerConfigurer container) throws Exception {
    container
      .containerClass(MultiContextContainer.class);
  }

}
```

In above example we enabled configuration for *CONTAINER* and used `SpringYarnConfigurerAdapter` and its `configure` callback method for `YarnContainerConfigurer`. In this method we instructed container class to be a `MultiContextContainer`.

```
@Configuration
@EnableYarn(enable=Enable.APPMASTER)
public class AppmasterConfiguration extends SpringYarnConfigurerAdapter {

  @Override
  public void configure(YarnAppmasterConfigurer master) throws Exception {
    master
      .withContainerRunner();
  }

}
```

In above example we enabled configuration for *APPMASTER* and because of this a callback method for `YarnAppmasterConfigurer` is called automatically.

```
@Configuration
@EnableYarn(enable=Enable.CLIENT)
@PropertySource("classpath:hadoop.properties")
public class ClientConfiguration extends SpringYarnConfigurerAdapter {

  @Autowired
  private Environment env;

  @Override
  public void configure(YarnConfigConfigurer config) throws Exception {
    config
      .fileSystemUri(env.getProperty("hd.fs"))
      .resourceManagerAddress(env.getProperty("hd.rm"));
  }

  @Override
  public void configure(YarnClientConfigurer client) throws Exception {
    Properties arguments = new Properties();
    arguments.put("container-count", "4");
    client
      .appName("multi-context-jc")
      .withMasterRunner()
        .contextClass(AppmasterConfiguration.class)
        .arguments(arguments);
}
```

In above example we enabled configuration for *CLIENT*. Here one will get yet another callback for `YarnClientConfigurer`. Additionally this shows how a Hadoop configuration can be customized using a callback for `YarnConfigConfigurer`.

11.3 Configuring Yarn

In order to use Hadoop and Yarn, one needs to first configure it namely by creating a `YarnConfiguration` object. The configuration holds information about the various parameters of the Yarn system.

Note

Configuration for `<yarn:configuration>` looks very similar than `<hdp:configuration>`. Reason for this is a simple separation for Hadoop's `YarnConfiguration` and `JobConf` classes.

_____

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

```
<yarn:configuration />
```

The declaration above defines a `YarnConfiguration` bean (to be precise a factory bean of type `ConfigurationFactoryBean`) named, by default, `yarnConfiguration`. 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:

```
<yarn: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
<?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:yarn="http://www.springframework.org/schema/yarn"
  xsi:schemaLocation="http://www.springframework.org/schema/beans http://
www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/yarn http://www.springframework.org/schema/yarn/
spring-yarn.xsd">

  <yarn:configuration>
    fs.defaultFS=hdfs://localhost:9000
    hadoop.tmp.dir=/tmp/hadoop
    electric=sea
  </yarn: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
<?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:yarn="http://www.springframework.org/schema/yarn"
  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/yarn http://www.springframework.org/schema/yarn/
spring-yarn.xsd">

  <yarn:configuration>
    fs.defaultFS=${hd.fs}
    hadoop.tmp.dir=file://${java.io.tmpdir}
    hangar=${number:18}
  </yarn: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
<?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:yarn="http://www.springframework.org/schema/yarn"
  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/yarn http://www.springframework.org/schema/yarn/
spring-yarn.xsd">

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

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

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

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

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

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

<yarn:configuration id="custom" configuration-ref="yarnConfiguration">
  fs.defaultFS=${custom.hd.fs}
</yarn:configuration>

...
```

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

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

Table 11.1. `yarn:configuration` attributes

Name	Values	Description
configuration-ref	Bean Reference	Reference to existing *Configuration* bean
properties-ref	Bean Reference	Reference to existing *Properties* bean
properties-location	Comma delimited list	List or Spring *Resource* paths
resources	Comma delimited list	List or Spring *Resource* paths
fs-uri	String	The HDFS filesystem address. Equivalent to *fs.defaultFS* property.

Name	Values	Description
`rm-address`	String	The Yarn Resource manager address. Equivalent to *yarn.resourcemanager.address* property.
`scheduler-address`	String	The Yarn Resource manager scheduler address. Equivalent to *yarn.resourcemanager.scheduler.address* property.

11.4 Local Resources

When *Application Master* or any other *Container* is run in a hadoop cluster, there are usually dependencies to various application and configuration files. These files needs to be localized into a running *Container* by making a physical copy. Localization is a process where dependent files are copied into node's directory structure and thus can be used within the *Container* itself. Yarn itself tries to provide isolation in a way that multiple containers and applications would not clash.

In order to use local resources, one needs to create an implementation of `ResourceLocalizer` interface. In its simplest form, resource localizer can be defined as:

```
<yarn:localresources>
  <yarn:hdfs path="/path/in/hdfs/my.jar"/>
</yarn:localresources>
```

The declaration above defines a `ResourceLocalizer` bean (to be precise a factory bean of type `LocalResourcesFactoryBean`) named, by default, *yarnLocalresources*. The default name is used, by conventions, by the other elements that require a reference to a resource localizer. It's explained later how this reference is used when container launch context is defined.

It is also possible to define path as pattern. This makes it easier to pick up all or subset of files from a directory.

```
<yarn:localresources>
  <yarn:hdfs path="/path/in/hdfs/*.jar"/>
</yarn:localresources>
```

Behind the scenes it's not enough to simple have a reference to file in a hdfs file system. Yarn itself when localizing resources into container needs to do a consistency check for copied files. This is done by checking file size and timestamp. This information needs to passed to yarn together with a file path. Order to do this the one who defines these beans needs to ask this information from hdfs prior to sending out resouce localizer request. This kind of behaviour exists to make sure that once localization is defined, *Container* will fail fast if dependant files were replaced during the process.

On default the hdfs base address is coming from a Yarn configuration and `ResourceLocalizer` bean will use configuration named *yarnLocalresources*. If there is a need to use something else than the default bean, *configuration* parameter can be used to make a reference to other defined configurations.

```
<yarn:localresources configuration="yarnConfiguration">
  <yarn:hdfs path="/path/in/hdfs/my.jar"/>
</yarn:localresources>
```

For example, client defining a launch context for *Application Master* needs to access dependent hdfs entries. Effectively hdfs entry given to resource localizer needs to be accessed from a *Node Manager*.

Spring for Apache Hadoop
- Reference Documentation

Yarn resource localizer is using additional parameters to define entry type and visibility. Usage is described below:

```
<yarn:localresources>
  <yarn:hdfs path="/path/in/hdfs/my.jar" type="FILE" visibility="APPLICATION"/>
</yarn:localresources>
```

For convenience it is possible to copy files into hdfs during the localization process using a *yarn:copy* tag. Currently base staging directory is */syarn/staging/xx* where *xx* is a unique identifier per application instance.

```
<yarn:localresources>
  <yarn:copy src="file:/local/path/to/files/*jar" staging="true"/>
  <yarn:hdfs path="/*" staging="true"/>
</yarn:localresources>
```

Table 11.2. `yarn:localresources` attributes

Name	Values	Description
configuration	Bean Reference	A reference to configuration bean name, default is *yarnConfiguration*
type	ARCHIVE, FILE, PATTERN	Global default if not defined in entry level
visibility	PUBLIC, PRIVATE, APPLICATION	Global default if not defined in entry level

Table 11.3. `yarn:hdfs` attributes

Name	Values	Description
path	HDFS Path	Path in hdfs
type	ARCHIVE, FILE(default), PATTERN	ARCHIVE - automatically unarchived by the Node Manager, FILE - regular file, PATTERN - hybrid between archive and file.
visibility	PUBLIC, PRIVATE, APPLICATION(default)	PUBLIC - Shared by all users on the node, PRIVATE - Shared among all applications of the *same user* on the node, APPLICATION - Shared only among containers of the *same application* on the node
staging	true, false(default)	Internal temporary stagind directory.

Table 11.4. `yarn:copy` attributes

Name	Values	Description
src	Copy sources	Comma delimited list of resource patterns
staging	true, false(default)	Internal temporary stagind directory.

11.5 Container Environment

One central concept in Yarn is to use environment variables which then can be read from a container. While it's possible to read those variable at any time it is considered bad design if one chooce to do so. Spring Yarn will pass variable into application before any business methods are executed, which makes things more clearly and testing becomes much more easier.

```
<yarn:environment/>
```

The declaration above defines a `Map` bean (to be precise a factory bean of type `EnvironmentFactoryBean`) named, by default, *yarnEnvironment*. The default name is used, by conventions, by the other elements that require a reference to a environment variables.

For conveniance it is possible to define a classpath entry directly into an environment. Most likely one is about to run some java code with libraries so classpath needs to be defined anyway.

```
<yarn:environment include-local-system-env="false">
  <yarn:classpath use-yarn-app-classpath="true" delimiter=":">
    ./*
  </yarn:classpath>
</yarn:environment>
```

If *use-yarn-app-classpath* parameter is set to *true*(default value) a default yarn entries will be added to classpath automatically. These entries are on default resolved from a normal `Hadoop Yarn Configuration` using its *yarn.application.classpath* property or if *site-yarn-app-classpath* has a any content entries are resolved from there.

Note

Be carefull if passing environment variables between different systems. For example if running a client on Windows and passing variables to Application Master running on Linux, execution wrapper in Yarn may silently fail.

Table 11.5. `yarn:environment` attributes

Name	Values	Description
include-local-system-env	true, false(default)	Defines whether system environment variables are actually added to this bean.

Table 11.6. `classpath` attributes

Name	Values	Description
use-yarn-app-classpath	false(default), true	Defines whether default yarn entries are added to classpath.
use-mapreduce-app-classpath	false(default), true	Defines whether default mr entries are added to classpath.

Name	Values	Description
`site-yarn-app-classpath`	Classpath entries	Defines a comma delimited list of default yarn application classpath entries.
`site-mapreduce-app-classpath`	Classpath entries	Defines a comma delimited list of default mr application classpath entries.
`delimiter`	Delimiter string, default is ":"	Defines delimiter used in a classpath string

11.6 Application Client

Client is always your entry point when interacting with a Yarn system whether one is about to submit a new application instance or just querying *Resource Manager* for running application(s) status. Currently support for client is very limited and a simple command to start *Application Master* can be defined. If there is just a need to query *Resource Manager*, command definition is not needed.

```xml
<yarn:client app-name="customAppName">
  <yarn:master-command>
    <![CDATA[
      /usr/local/java/bin/java
      org.springframework.yarn.am.CommandLineAppmasterRunner
      appmaster-context.xml
      yarnAppmaster
      container-count=2
      1><LOG_DIR>/AppMaster.stdout
      2><LOG_DIR>/AppMaster.stderr
    ]]>
  </yarn:master-command>
</yarn:client>
```

The declaration above defines a `YarnClient` bean (to be precise a factory bean of type `YarnClientFactoryBean`) named, by default, *yarnClient*. It also defines a command launching an *Application Master* using `<master-command>` entry which is also a way to define the raw commands. If this *yarnClient* instance is used to submit an application, its name would come from a *app-name* attribute.

```xml
<yarn:client app-name="customAppName">
  <yarn:master-runner/>
</yarn:client>
```

For a convinience entry `<master-runner>` can be used to define same command entries.

```
<yarn:client app-name="customAppName">
  <util:properties id="customArguments">
    container-count=2
  </util:properties>
  <yarn:master-runner
    command="java"
    context-file="appmaster-context.xml"
    bean-name="yarnAppmaster"
    arguments="customArguments"
    stdout="<LOG_DIR>/AppMaster.stdout"
    stderr="<LOG_DIR>/AppMaster.stderr" />
</yarn:client>
```

All previous three examples are effectively identical from Spring Yarn point of view.

Note

The <LOG_DIR> refers to Hadoop's dedicated log directory for the running container.

```
<yarn:client app-name="customAppName"
  configuration="customConfiguration"
  resource-localizer="customResources"
  environment="customEnv"
  priority="1"
  virtualcores="2"
  memory="11"
  queue="customqueue">
  <yarn:master-runner/>
</yarn:client>
```

If there is a need to change some of the parameters for the *Application Master* submission, `memory` and `virtualcores` defines the container settings. For submission, `queue` and `priority` defines how submission is actually done.

Table 11.7. `yarn:client` attributes

Name	Values	Description
app-name	Name as string, default is empty	Yarn submitted application name
configuration	Bean Reference	A reference to configuration bean name, default is *yarnConfiguration*
resource-localizer	Bean Reference	A reference to resource localizer bean name, default is *yarnLocalresources*
environment	Bean Reference	A reference to environment bean name, default is *yarnEnvironment*
template	Bean Reference	A reference to a bean implementing `ClientRmOperations`
memory	Memory as integer, default is "64"	Amount of memory for appmaster resource

Name	Values	Description
virtualcores	Cores as integer, default is "1"	Number of appmaster resource virtual cores
priority	Priority as integer, default is "0"	Submission priority
queue	Queue string, default is "default"	Submission queue

Table 11.8. `yarn:master-command`

Name	Values	Description
Entry content	List of commands	Commands defined in this entry are aggregated into a single command line

Table 11.9. `yarn:master-runner` attributes

Name	Values	Description
command	Main command as string, default is "java"	Command line first entry
context-file	Name of the Spring context file, default is "appmaster-context.xml"	Command line second entry
bean-name	Name of the Spring bean, default is "yarnAppmaster"	Command line third entry
arguments	Reference to Java's Properties	Added to command line parameters as key/value pairs separated by '='
stdout	Stdout, default is "<LOG_DIR>/ AppMaster.stdout"	Appended with 1>
stderr	Stderr, default is "<LOG_DIR>/ AppMaster.stderr"	Appended with 2>

11.7 Application Master

Application master is responsible for container allocation, launching and monitoring.

```xml
<yarn:master>
  <yarn:container-allocator virtualcores="1" memory="64" priority="0"/>
  <yarn:container-launcher username="whoami"/>
  <yarn:container-command>
    <![CDATA[
      /usr/local/java/bin/java
      org.springframework.yarn.container.CommandLineContainerRunner
      container-context.xml
      1><LOG_DIR>/Container.stdout
      2><LOG_DIR>/Container.stderr
    ]]>
  </yarn:container-command>
</yarn:master>
```

The declaration above defines a `YarnAppmaster` bean (to be precise a bean of type `StaticAppmaster`) named, by default, *yarnAppmaster*. It also defines a command launching a *Container(s)* using `<container-command>` entry, parameters for allocation using `<container-allocator>` entry and finally a launcher parameter using `<container-launcher>` entry.

Currently there is a simple implementation of `StaticAppmaster` which is able to allocate and launch a number of containers. These containers are monitored by querying resource manager for container execution completion.

```xml
<yarn:master>
  <yarn:container-runner/>
</yarn:master>
```

For a convinience entry `<container-runner>` can be used to define same command entries.

```xml
<yarn:master>
  <util:properties id="customArguments">
    some-argument=myvalue
  </util:properties>
  <yarn:container-runner
    command="java"
    context-file="container-context.xml"
    bean-name="yarnContainer"
    arguments="customArguments"
    stdout="<LOG_DIR>/Container.stdout"
    stderr="<LOG_DIR>/Container.stderr" />
</yarn:master>
```

Table 11.10. `yarn:master` attributes

Name	Values	Description
configuration	Bean Reference	A reference to configuration bean name, default is *yarnConfiguration*
resource-localizer	Bean Reference	A reference to resource localizer bean name, default is *yarnLocalresources*
environment	Bean Reference	A reference to environment bean name, default is *yarnEnvironment*

Table 11.11. `yarn:container-allocator` *attributes*

Name	Values	Description
`virtualcores`	Integer	*number of virtual cpu cores* of the resource.
`memory`	Integer, as of MBs.	*memory* of the resource.
`priority`	Integer	Assigned priority of a request.
`locality`	Boolean	If set to true indicates that resources are not relaxed. Default is *FALSE*.

Table 11.12. `yarn:container-launcher` *attributes*

Name	Values	Description
`username`	String	Set the *user* to whom the container has been allocated.

Table 11.13. `yarn:container-runner` *attributes*

Name	Values	Description
`command`	Main command as string, default is "java"	Command line first entry
`context-file`	Name of the Spring context file, default is "container-context.xml"	Command line second entry
`bean-name`	Name of the Spring bean, default is "yarnContainer"	Command line third entry
`arguments`	Reference to Java's Properties	Added to command line parameters as key/value pairs separated by '='
`stdout`	Stdout, default is "<LOG_DIR>/Container.stdout"	Appended with 1>
`stderr`	Stderr, default is "<LOG_DIR>/Container.stderr"	Appended with 2>

11.8 Application Container

There is very little what Spring Yarn needs to know about the Container in terms of its configuration. There is a simple contract between `org.springframework.yarn.container.CommandLineContainerRunner` and a bean it's trying to run on default. Default bean name is *yarnContainer*.

There is a simple interface `org.springframework.yarn.container.YarnContainer` which container needs to implement.

```
public interface YarnContainer {
  void run();
  void setEnvironment(Map<String, String> environment);
  void setParameters(Properties parameters);
}
```

There are few different ways how Container can be defined in Spring xml configuration. Natively without using namespaces bean can be defined with a correct name:

```
<bean id="yarnContainer" class="org.springframework.yarn.container.TestContainer">
```

Spring Yarn namespace will make it even more simpler. Below example just defines class which implements needed interface.

```
<yarn:container container-class="org.springframework.yarn.container.TestContainer"/>
```

It's possible to make a reference to existing bean. This is usefull if bean cannot be instantiated with default constructor.

```
<bean id="testContainer" class="org.springframework.yarn.container.TestContainer"/>
<yarn:container container-ref="testContainer"/>
```

It's also possible to inline the bean definition.

```
<yarn:container>
  <bean class="org.springframework.yarn.container.TestContainer"/>
</yarn:container>
```

11.9 Application Master Services

It is fairly easy to create an application which launches a few containers and then leave those to do their tasks. This is pretty much what *Distributed Shell* example application in Yarn is doing. In that example a container is configured to run a simple shell command and *Application Master* only tracks when containers have finished. If only need from a framework is to be able to fire and forget then that's all you need, but most likely a real-world Yarn application will need some sort of collaboration with *Application Master*. This communication is initiated either from *Application Client* or *Application Container*.

Yarn framework itself doesn't define any kind of general communication API for *Application Master*. There are APIs for communicating with *Container Manager* and *Resource Manager* which are used on

within a layer not necessarily exposed to a user. Spring Yarn defines a general framework to talk to *Application Master* through an abstraction and currently a JSON based rpc system exists.

This chapter concentrates on developer concepts to create a custom services for *Application Master*, configuration options for built-in services can be found from sections below - [Appmaster Service](#) and [Appmaster Service Client](#).

Basic Concepts

Having a communication framework between *Application Master* and *Container/Client* involves few moving parts. Firstly there has to be some sort of service running on an *Application Master*. Secondly user of this service needs to know where it is and how to connect to it. Thirtly, if not creating these services from scratch, it'd be nice if some sort of abstraction already exist.

Contract for appmaster service is very simple, *Application Master Service* needs to implement `AppmasterService` interface be registered with Spring application context. Actual appmaster instance will then pick it up from a bean factory.

```java
public interface AppmasterService {
  int getPort();
  boolean hasPort();
  String getHost();
}
```

Application Master Service framework currently provides integration for services acting as service for a *Client* or a *Container*. Only difference between these two roles is how the *Service Client* gets notified about the address of the service. For the *Client* this information is stored within the Hadoop Yarn resource manager. For the *Container* this information is passed via environment within the launch context.

```xml
<bean id="yarnAmservice" class="AppmasterServiceImpl" />
<bean id="yarnClientAmservice" class="AppmasterClientServiceImpl" />
```

Example above shows a default bean names, *yarnAmservice* and *yarnClientAmservice* respectively recognised by Spring Yarn.

Interface `AppmasterServiceClient` is currently an empty interface just marking class to be a appmaster service client.

```java
public interface AppmasterServiceClient {
}
```

Using JSON

Default implementations can be used to exchange messages using a simple domain classes and actual messages are converted into json and send over the transport.

```xml
<yarn-int:amservice
  service-impl="org.springframework.yarn.integration.ip.mind.TestService"
  default-port="1234"/>
<yarn-int:amservice-client
  service-
impl="org.springframework.yarn.integration.ip.mind.DefaultMindAppmasterServiceClient"
  host="localhost"
  port="1234"/>
```

```
@Autowired
AppmasterServiceClient appmasterServiceClient;

@Test
public void testServiceInterfaces() throws Exception {
  SimpleTestRequest request = new SimpleTestRequest();
  SimpleTestResponse response =
  (SimpleTestResponse) ((MindAppmasterServiceClient)appmasterServiceClient).
    doMindRequest(request);
  assertThat(response.stringField, is("echo:stringFieldValue"));
}
```

Converters

When default implementations for Application master services are exchanging messages, converters are net registered automatically. There is a namespace tag *converters* to ease this configuration.

```xml
<bean id="mapper"
  class="org.springframework.yarn.integration.support.Jackson2ObjectMapperFactoryBean" />

<yarn-int:converter>
  <bean class="org.springframework.yarn.integration.convert.MindObjectToHolderConverter">
    <constructor-arg ref="mapper"/>
  </bean>
</yarn-int:converter>

<yarn-int:converter>
  <bean class="org.springframework.yarn.integration.convert.MindHolderToObjectConverter">
    <constructor-arg ref="mapper"/>
    <constructor-arg value="org.springframework.yarn.batch.repository.bindings"/>
  </bean>
</yarn-int:converter>
```

11.10 Application Master Service

This section of this document is about configuration, more about general concepts for see a Section 11.9, "Application Master Services".

Currently Spring Yarn have support for services using Spring Integration tcp channels as a transport.

```xml
<bean id="mapper"
  class="org.springframework.yarn.integration.support.Jackson2ObjectMapperFactoryBean" />

<yarn-int:converter>
  <bean class="org.springframework.yarn.integration.convert.MindObjectToHolderConverter">
    <constructor-arg ref="mapper"/>
  </bean>
</yarn-int:converter>

<yarn-int:converter>
  <bean class="org.springframework.yarn.integration.convert.MindHolderToObjectConverter">
    <constructor-arg ref="mapper"/>
    <constructor-arg value="org.springframework.yarn.integration.ip.mind"/>
  </bean>
</yarn-int:converter>

<yarn-int:amservice
  service-impl="org.springframework.yarn.integration.ip.mind.TestService"/>
```

If there is a need to manually configure the server side dispatch channel, a little bit more configuration is needed.

```xml
<bean id="serializer"
  class="org.springframework.yarn.integration.ip.mind.MindRpcSerializer" />
<bean id="deserializer"
  class="org.springframework.yarn.integration.ip.mind.MindRpcSerializer" />
<bean id="socketSupport"
  class="org.springframework.yarn.integration.support.DefaultPortExposingTcpSocketSupport"
 />

<ip:tcp-connection-factory id="serverConnectionFactory"
  type="server"
  port="0"
  socket-support="socketSupport"
  serializer="serializer"
  deserializer="deserializer"/>

<ip:tcp-inbound-gateway id="inboundGateway"
  connection-factory="serverConnectionFactory"
  request-channel="serverChannel" />

<int:channel id="serverChannel" />

<yarn-int:amservice
  service-impl="org.springframework.yarn.integration.ip.mind.TestService"
  channel="serverChannel"
  socket-support="socketSupport"/>
```

Table 11.14. `yarn-int:amservice` *attributes*

Name	Values	Description
service-impl	Class Name	Full name of the class implementing a service
service-ref	Bean Reference	Reference to a bean name implementing a service
channel	Spring Int channel	Custom message dispatching channel
socket-support	Socket support reference	Custom socket support class

11.11 Application Master Service Client

This section of this document is about configuration, more about general concepts for see a Section 11.9, "Application Master Services".

Currently Spring Yarn have support for services using Spring Integration tcp channels as a transport.

```xml
<bean id="mapper"
  class="org.springframework.yarn.integration.support.Jackson2ObjectMapperFactoryBean" />

<yarn-int:converter>
  <bean class="org.springframework.yarn.integration.convert.MindObjectToHolderConverter">
    <constructor-arg ref="mapper"/>
  </bean>
</yarn-int:converter>

<yarn-int:converter>
  <bean class="org.springframework.yarn.integration.convert.MindHolderToObjectConverter">
    <constructor-arg ref="mapper"/>
    <constructor-arg value="org.springframework.yarn.integration.ip.mind"/>
  </bean>
</yarn-int:converter>

<yarn-int:amservice-client
  service-
impl="org.springframework.yarn.integration.ip.mind.DefaultMindAppmasterServiceClient"
  host="${SHDP_AMSERVICE_HOST}"
  port="${SHDP_AMSERVICE_PORT}"/>
```

If there is a need to manually configure the server side dispatch channel, a little bit more configuration is needed.

```xml
<bean id="serializer"
  class="org.springframework.yarn.integration.ip.mind.MindRpcSerializer" />
<bean id="deserializer"
  class="org.springframework.yarn.integration.ip.mind.MindRpcSerializer" />

<ip:tcp-connection-factory id="clientConnectionFactory"
  type="client"
  host="localhost"
  port="${SHDP_AMSERVICE_PORT}"
  serializer="serializer"
  deserializer="deserializer"/>

<ip:tcp-outbound-gateway id="outboundGateway"
  connection-factory="clientConnectionFactory"
  request-channel="clientRequestChannel"
  reply-channel="clientResponseChannel" />

<int:channel id="clientRequestChannel" />
<int:channel id="clientResponseChannel" >
  <int:queue />
</int:channel>

<yarn-int:amservice-client
  service-
impl="org.springframework.yarn.integration.ip.mind.DefaultMindAppmasterServiceClient"
  request-channel="clientRequestChannel"
  response-channel="clientResponseChannel"/>
```

Table 11.15. `yarn-int:amservice-client` *attributes*

Name	Values	Description
service-impl	Class Name	Full name of the class implementing a service client

Name	Values	Description
`host`	Hostname	Host of the running appmaster service
`port`	Port	Port of the running appmaster service
`request-channel`	Reference to Spring Int request channel	Custom channel
`response-channel`	Reference to Spring Int response channel	Custom channel

11.12 Using Spring Batch

In this chapter we assume you are fairly familiar with concepts using *Spring Batch*. Many batch processing problems can be solved with single threaded, single process jobs, so it is always a good idea to properly check if that meets your needs before thinking about more complex implementations. When you are ready to start implementing a job with some parallel processing, Spring Batch offers a range of options. At a high level there are two modes of parallel processing: single process, multi-threaded; and multi-process.

Spring Hadoop contains a support for running Spring Batch jobs on a Hadoop cluster. For better parallel processing Spring Batch partitioned steps can be executed on a Hadoop cluster as remote steps.

Batch Jobs

Starting point running a *Spring Batch Job* is always the *Application Master* whether a job is just simple job with or without partitioning. In case partitioning is not used the whole job would be run within the *Application Master* and no *Containers* would be launched. This may seem a bit odd to run something on Hadoop without using *Containers* but one should remember that *Application Master* is also just a resource allocated from a Hadoop cluster.

Order to run Spring Batch jobs on a Hadoop cluster, few constraints exists:

- *Job Context* - Application Master is the main entry point of running the job.

- *Job Repository* - Application Master needs to have access to a repository which is located either in-memory or in a database. These are the two type natively supported by Spring Batch.

- *Remote Steps* - Due to nature how Spring Batch partitioning works, remote step needs an access to a job repository.

Configuration for Spring Batch Jobs is very similar what is needed for normal batch configuration because effectively that's what we are doing. Only difference is a way a job is launched which in this case is automatically handled by *Application Master*. Implementation of a job launching logic is very similar compared to `CommandLineJobRunner` found from a Spring Batch.

```
<bean id="transactionManager" class="org.springframework.batch.support.transaction.ResourcelessTransactionMa
>

<bean id="jobRepository" class="org.springframework.batch.core.repository.support.MapJobRepositoryFactoryBea
  <property name="transactionManager" ref="transactionManager"/>
</bean>

<bean id="jobLauncher" class="org.springframework.batch.core.launch.support.SimpleJobLauncher">
  <property name="jobRepository" ref="jobRepository"/>
</bean>
```

The declaration above define beans for `JobRepository` and `JobLauncher`. For simplisity we used in-memory repository while it would be possible to switch into repository working with a database if persistence is needed. A bean named `jobLauncher` is later used within the *Application Master* to launch jobs.

```
<bean id="yarnEventPublisher" class="org.springframework.yarn.event.DefaultYarnEventPublisher"/
>

<yarn-batch:master/>
```

The declaration above defines `BatchAppmaster` bean named, by default, `yarnAppmaster` and `YarnEventPublisher` bean named `yarnEventPublisher` which is not created automatically.

Final step to finalize our very simple batch configuration is to define the actual batch job.

```
<bean id="hello" class="org.springframework.yarn.examples.PrintTasklet">
  <property name="message" value="Hello"/>
</bean>

<batch:job id="job">
  <batch:step id="master">
    <batch:tasklet transaction-manager="transactionManager" ref="hello"/>
  </batch:step>
</batch:job>
```

The declaration above defines a simple job and tasklet. Job is named as `job` which is the default job name searched by *Application Master*. It is possible to use different name by changing the launch configuration.

Table 11.16. `yarn-batch:master` attributes

Name	Values	Description
configuration	Bean Reference	A reference to configuration bean name, default is *yarnConfiguration*
resource-localizer	Bean Reference	A reference to resource localizer bean name, default is *yarnLocalresources*
environment	Bean Reference	A reference to environment bean name, default is *yarnEnvironment*
job-name	Bean Name Reference	A name reference to Spring Batch job, default is *job*

Name	Values	Description
`job-launcher`	Bean Reference	A reference to job launcher bean name, default is *jobLauncher*. Target is a normal Spring Batch bean implementing `JobLauncher`.

Partitioning

Let's take a quick look how Spring Batch partitioning is handled. Concept of running a partitioned job involves three things, *Remote steps*, *Partition Handler* and a *Partitioner*. If we do a little bit of oversimplification a remote step is like any other step from a user point of view. Spring Batch itself does not contain implementations for any proprietary grid or remoting fabrics. Spring Batch does however provide a useful implementation of `PartitionHandler` that executes Steps locally in separate threads of execution, using the `TaskExecutor` strategy from Spring. Spring Hadoop provides implementation to execute Steps remotely on a Hadoop cluster.

> 🍃 **Note**
>
> For more background information about the Spring Batch Partitioning, read the Spring Batch reference documentation.

Configuring Master

As we previously mentioned a step executed on a remote host also need to access a job repository. If job repository would be based on a database instance, configuration could be similar on a container compared to application master. In our configuration example the job repository is in-memory based and remote steps needs access for it. Spring Yarn Batch contains implementation of a job repository which is able to proxy request via json requests. Order to use that we need to enable application client service which is exposing this service.

```xml
<bean id="jobRepositoryRemoteService" class="org.springframework.yarn.batch.repository.JobRepositoryRemoteSe
 >
  <property name="mapJobRepositoryFactoryBean" ref="&amp;jobRepository"/>
</bean>

<bean id="batchService" class="org.springframework.yarn.batch.repository.BatchAppmasterService"
 >
  <property name="jobRepositoryRemoteService" ref="jobRepositoryRemoteService"/>
</bean>

<yarn-int:amservice service-ref="batchService"/>
```

he declaration above defines `JobRepositoryRemoteService` bean named `jobRepositoryRemoteService` which is then connected into *Application Master Service* exposing job repository via Spring Integration Tcp channels.

As job repository communication messages are exchanged via custom json messages, converters needs to be defined.

```
<bean id="mapper" class="org.springframework.yarn.integration.support.Jackson2ObjectMapperFactoryBean"
  />

<yarn-int:converter>
  <bean class="org.springframework.yarn.integration.convert.MindObjectToHolderConverter">
    <constructor-arg ref="mapper"/>
  </bean>
</yarn-int:converter>

<yarn-int:converter>
  <bean class="org.springframework.yarn.integration.convert.MindHolderToObjectConverter">
    <constructor-arg ref="mapper"/>
    <constructor-arg value="org.springframework.yarn.batch.repository.bindings"/>
  </bean>
</yarn-int:converter>
```

Configuring Container

Previously we made a choice to use in-memore job repository running inside the application master. Now we need to talk to this repository via client service. We start by adding same converters as in application master.

```
<bean id="mapper" class="org.springframework.yarn.integration.support.Jackson2ObjectMapperFactoryBean"
  />

<yarn-int:converter>
  <bean class="org.springframework.yarn.integration.convert.MindObjectToHolderConverter">
    <constructor-arg ref="mapper"/>
  </bean>
</yarn-int:converter>

<yarn-int:converter>
  <bean class="org.springframework.yarn.integration.convert.MindHolderToObjectConverter">
    <constructor-arg ref="mapper"/>
    <constructor-arg value="org.springframework.yarn.batch.repository.bindings"/>
  </bean>
</yarn-int:converter>
```

We use general client implementation able to communicate with a service running on *Application Master*.

```
<yarn-int:amservice-client
  service-
impl="org.springframework.yarn.integration.ip.mind.DefaultMindAppmasterServiceClient"
  host="${SHDP_AMSERVICE_HOST}"
  port="${SHDP_AMSERVICE_PORT}" />
```

Remote step is just like any other step.

```
<bean id="hello" class="org.springframework.yarn.examples.PrintTasklet">
  <property name="message" value="Hello"/>
</bean>

<batch:step id="remoteStep">
  <batch:tasklet transaction-manager="transactionManager" start-limit="100" ref="hello"/>
</batch:step>
```

We need to have a way to locate the step from an application context. For this we can define a step locator which is later configured into running container.

```
<bean id="stepLocator" class="org.springframework.yarn.batch.partition.BeanFactoryStepLocator"/
>
```

Spring Hadoop contains a custom job repository implementation which is able to talk back to a remote instance via custom json protocol.

```
<bean id="transactionManager" class="org.springframework.batch.support.transaction.ResourcelessTransactionM
>

<bean id="jobRepository" class="org.springframework.yarn.batch.repository.RemoteJobRepositoryFactoryBean">
  <property name="transactionManager" ref="transactionManager"/>
  <property name="appmasterScOperations" ref="yarnAmserviceClient"/>
</bean>

<bean id="jobExplorer" class="org.springframework.yarn.batch.repository.RemoteJobExplorerFactoryBean">
  <property name="repositoryFactory" ref="&amp;jobRepository" />
</bean>
```

Finally we define a *Container* understanding how to work with a remote steps.

```
<bean id="yarnContainer" class="org.springframework.yarn.batch.container.DefaultBatchYarnContainer">
  <property name="stepLocator" ref="stepLocator"/>
  <property name="jobExplorer" ref="jobExplorer"/>
  <property name="integrationServiceClient" ref="yarnAmserviceClient"/>
</bean>
```

11.13 Using Spring Boot Application Model

We have additional support for leveraging *Spring Boot* when creating applications using *Spring YARN*. All dependencies for this exists in a sub-module named `spring-yarn-boot` which itself depends on *Spring Boot*.

Spring Boot extensions in *Spring YARN* are used to ease following issues:

- Create a clear model how application is built, packaged and run on *Hadoop YARN*.

- Automatically configure components depending whether we are on *Client*, *Appmaster* or *Container*.

- Create an easy to use externalized configuration model based on Boot's `ConfigurationProperties`.

Before we get into details let's go through how simple it is to create and deploy a custom application to a Hadoop cluster. Notice that there are no need to use XML.

```
@Configuration
@EnableAutoConfiguration
public class ContainerApplication {

  public static void main(String[] args) {
    SpringApplication.run(ContainerApplication.class, args);
  }

  @Bean
  public HelloPojo helloPojo() {
    return new HelloPojo();
  }

}
```

In above `ContainerApplication`, notice how we added `@Configuration` in a class level itself and `@Bean` for a `helloPojo()` method.

```
@YarnComponent
public class HelloPojo {

  private static final Log log = LogFactory.getLog(HelloPojo.class);

  @Autowired
  private Configuration configuration;

  @OnContainerStart
  public void publicVoidNoArgsMethod() {
    log.info("Hello from HelloPojo");
    log.info("About to list from hdfs root content");
    FsShell shell = new FsShell(configuration);
    for (FileStatus s : shell.ls(false, "/")) {
      log.info(s);
    }
  }

}
```

`HelloPojo` class is a simple POJO in a sense that it doesn't extend any *Spring YARN* base classes. What we did in this class:

- We've added a class level `@YarnComponent` annotation.

- We've added a method level `@OnContainerStart` annotation.

- We've `@Autowired` a Hadoop's Configuration class.

To demonstrate that we actually have some real functionality in this class, we simply use Spring Hadoop's `FsShell` to list entries from a root of a HDFS file system. For this we need to have access to Hadoop's Configuration which is prepared for you so that you can just autowire it.

```
@EnableAutoConfiguration
public class ClientApplication {

  public static void main(String[] args) {
    SpringApplication.run(ClientApplication.class, args)
      .getBean(YarnClient.class)
      .submitApplication();
  }

}
```

- `@EnableAutoConfiguration` tells Spring Boot to start adding beans based on classpath setting, other beans, and various property settings.

- Specific auto-configuration for Spring YARN components takes place since Spring YARN is on the classpath.

The `main()` method uses Spring Boot's `SpringApplication.run()` method to launch an application. From there we simply request a bean of type `YarnClient` and execute its `submitApplication()` method. What happens next depends on application configuration, which we go through later in this document.

```
@EnableAutoConfiguration
public class AppmasterApplication {

  public static void main(String[] args) {
    SpringApplication.run(AppmasterApplication.class, args);
  }

}
```

Application class for `YarnAppmaster` looks even simpler than what we just did for `ClientApplication`. Again the `main()` method uses Spring Boot's `SpringApplication.run()` method to launch an application.

In real life, you most likely need to start adding more custom functionality to your application component and you'd do that by start adding more beans. To do that you need to define a Spring `@Configuration` or `@ComponentScan`. AppmasterApplication would then act as your main starting point to define more custom functionality.

```
spring:
  hadoop:
    fsUri: hdfs://localhost:8020
    resourceManagerHost: localhost
  yarn:
    appName: yarn-boot-simple
    applicationDir: /app/yarn-boot-simple/
    client:
      files:
        - "file:build/libs/yarn-boot-simple-container-0.1.0.jar"
        - "file:build/libs/yarn-boot-simple-appmaster-0.1.0.jar"
      launchcontext:
        archiveFile: yarn-boot-simple-appmaster-0.1.0.jar
    appmaster:
      containerCount: 1
      launchcontext:
        archiveFile: yarn-boot-simple-container-0.1.0.jar
```

Final part for your application is its runtime configuration which glues all the components together which then can be called as a Spring YARN application. This configuration act as source for Spring Boot's `@ConfigurationProperties` and contains relevant configuration properties which cannot be auto-discovered or otherwise needs to have an option to be overwritten by an end user.

You can then write your own defaults for your own environment. Because these `@ConfigurationProperties` are resolved at runtime by Spring Boot, you even have an easy option to overwrite these properties either by using command-line options or provide additional configuration property files.

Auto Configuration

Spring Boot is heavily influenced by auto-configuration trying to predict what user wants to do. These decisions are based on configuration properties, what's currently available from a classpath and generally everything what auto-configurers are able to see.

Auto-configuration is able to see if it's currently running on a YARN cluster and can also differentiate between *YarnContainer* and *YarnAppmaster*. Parts of the auto-configuration which cannot be automatically detected are guarded by a flags in configuration properties which then allows end-user to either enable or disable these functionalities.

Application Files

As we already mentioned *Spring Boot* creates a clear model how you would work with your application files. Most likely what you need in your application is jar or zip file(s) having needed application code and optional configuration properties to customize the application logic. Customization via an external properties files makes it easier to change application functionality and reduce a need to hard-code application logic.

Running an application on YARN needs an instance of *YarnAppmaster* and instances of *YarnContainer*s. Both of these containers will need a set of files and instructions how to execute a container. Based on auto-configuration and configuration properties we will make few assumptions how a container is executed.

We are fundamentally supporting three different type of combinations:

- If a container main archive file is a jar file we expect it to be packaged with Boot and be self container executable jar archive.

- If a container main archive is a zip file we expect it to be packages with Boot. In this case we use a special runner which knows how to run this exploded archive.

- User defines a main class to be run and everything this class will need is already setup.

More detailed functionality can be found from a below sections; the section called "Application Classpath", the section called "Container Runners" and the section called "Configuration Properties".

Application Classpath

Let's go through as an examples how a classpath is configured on different use cases.

Simple Executable Jar

Running a container using an executable jar archive is the most simple scenario due to classpath limitation imposed by a JVM. Everything needed for the classpath needs to be inside the archive itself. Boot plugins for maven and gradle will greatly help to package all library dependencies into this archive.

```
spring:
  yarn:
    client:
      launchcontext:
        archiveFile: yarn-boot-appmaster-0.1.0.jar
    appmaster:
      launchcontext:
        archiveFile: yarn-boot-container-0.1.0.jar
```

Simple Zip Archive

Using a zip archive is basically needed in two use cases. In first case you want to re-use existing libraries in YARN cluster for your classpath. In second case you want to add custom classpath entries from an exploded zip archive.

```
spring:
  yarn:
    siteYarnAppClasspath: "/path/to/hadoop/libs/*"
    appmaster:
      launchcontext:
        useYarnAppClasspath: true
        archiveFile: yarn-boot-container-0.1.0.zip
```

In above example you can have a zip archive which doesn't bundle all dependant Hadoop YARN libraries. Default classpath entries are then resolved from `siteYarnAppClasspath` property.

```
spring:
  yarn:
    appmaster:
      launchcontext:
        archiveFile: yarn-boot-container-0.1.0.zip
        containerAppClasspath:
          - "./yarn-boot-container-0.1.0.zip/config"
          - "./yarn-boot-container-0.1.0.zip/lib"
```

In above example you needed to use custom classpath entries from an exploded zip archive.

Container Runners

Using a propertys `spring.yarn.client.launchcontext.archiveFile` and `spring.yarn.appmaster.launchcontext.archiveFile` respectively, will indicate that container is run based on an archive file and Boot runners are used. These runner classes are either used manually when constructing an actual raw command for container or internally within an executable jar archive.

However there are times when you may need to work on much lower level. Maybe you are having trouble using an executable jar archive or Boot runner is not enough what you want to do. For this use case you would use propertys `spring.yarn.client.launchcontext.runnerClass` and `spring.yarn.appmaster.launchcontext.runnerClass`.

Custom Runner

```
spring:
  yarn:
    appmaster:
      launchcontext:
        runnerClass: com.example.MyMainClazz
```

Resource Localizing

Order for containers to use application files, a YARN resource localization process needs to do its tasks. We have a few configuration properties which are used to determine which files are actually localized into container's working directory.

```
spring:
  yarn:
    client:
      localizer:
        patterns:
         - "*appmaster*jar"
         - "*appmaster*zip"
        zipPattern: "*zip"
        propertiesNames: [application]
        propertiesSuffixes: [properties, yml]
    appmaster:
      localizer:
        patterns:
         - "*container*jar"
         - "*container*zip"
        zipPattern: "*zip"
        propertiesNames: [application]
        propertiesSuffixes: [properties, yml]
```

Above is an example which equals a default functionality when localized resources are chosen. For example for a container we automatically choose all files matching a simple patterns `*container*jar` and `*container*zip`. Additionally we choose configuration properties files matching names `application.properties` and `application.yml`. Property *zipPattern* is used as an pattern to instruct YARN resource localizer to triet file as an archive to be automatically exploded.

If for some reason the default functionality and how it can be configured via configuration properties is not suiteable, one can define a custom bean to change how things work. Interface `LocalResourcesSelector` is used to find localized resources.

```
public interface LocalResourcesSelector {
  List<Entry> select(String dir);
}
```

Below you see a logic how a default `BootLocalResourcesSelector` is created during the auto-configuration. You would then create a custom implementation and create it as a bean in your Configuration class. You would not need to use any Conditionals but not how in auto-configuration we use `@ConditionalOnMissingBean` to check if user have already created his own implementation.

```
@Configuration
@EnableConfigurationProperties({ SpringYarnAppmasterLocalizerProperties.class })
public static class LocalResourcesSelectorConfig {

  @Autowired
  private SpringYarnAppmasterLocalizerProperties syalp;

  @Bean
  @ConditionalOnMissingBean(LocalResourcesSelector.class)
  public LocalResourcesSelector localResourcesSelector() {
    BootLocalResourcesSelector selector = new BootLocalResourcesSelector(Mode.CONTAINER);
    if (StringUtils.hasText(syalp.getZipPattern())) {
      selector.setZipArchivePattern(syalp.getZipPattern());
    }
    if (syalp.getPropertiesNames() != null) {
      selector.setPropertiesNames(syalp.getPropertiesNames());
    }
    if (syalp.getPropertiesSuffixes() != null) {
      selector.setPropertiesSuffixes(syalp.getPropertiesSuffixes());
    }
    selector.addPatterns(syalp.getPatterns());
    return selector;
  }
}
```

Your configuration could then look like:

```
@EnableAutoConfiguration
public class AppmasterApplication {

  @Bean
  public LocalResourcesSelector localResourcesSelector() {
    return MyLocalResourcesSelector();
  }

  public static void main(String[] args) {
    SpringApplication.run(AppmasterApplication.class, args);
  }

}
```

Container as POJO

In Boot application model if *YarnContainer* is not explicitly defined it defaults to `DefaultYarnContainer` which expects to find a `POJO` created as a bean having a specific annotations instructing the actual functionality.

`@YarnComponent` is a stereotype annotation itself having a Spring's `@Component` defined in it. This is automatically marking a class to be a candidate having a `@YarnComponent` functionality.

Within a `POJO` class we can use `@OnContainerStart` annotation to mark a public method to act as an activator for a method endpoint.

```
@OnContainerStart
public void publicVoidNoArgsMethod() {
}
```

Returning type of `int` participates in a *YarnContainer* exit value.

```
@OnContainerStart
public int publicIntNoArgsMethod() {
  return 0;
}
```

Returning type of `boolean` participates in a *YarnContainer* exit value where *true* would mean complete and *false* failed container.

```
@OnContainerStart
public boolean publicBooleanNoArgsMethod() {
  return true;
}
```

Returning type of `String` participates in a *YarnContainer* exit value by matching `ExitStatus` and getting exit value from `ExitCodeMapper`.

```
@OnContainerStart
public String publicStringNoArgsMethod() {
  return "COMPLETE";
}
```

If method throws any `Exception` *YarnContainer* is marked as failed.

```
@OnContainerStart
public void publicThrowsException() {
  throw new RuntimeException("My Error");
}
```

Method parameter can be bound with `@YarnEnvironments` to get access to current *YarnContainer* environment variables.

```
@OnContainerStart
public void publicVoidEnvironmentsArgsMethod(@YarnEnvironments Map<String,String> env) {
}
```

Method parameter can be bound with `@YarnEnvironment` to get access to specific *YarnContainer* environment variable.

```
@OnContainerStart
public void publicVoidEnvironmentArgsMethod(@YarnEnvironment("key") String value) {
}
```

Method parameter can be bound with `@YarnParameters` to get access to current *YarnContainer* arguments.

```
@OnContainerStart
public void publicVoidParametersArgsMethod(@YarnParameters Properties properties) {
}
```

Method parameter can be bound with `@YarnParameter` to get access to a specific *YarnContainer* arguments.

```
@OnContainerStart
public void publicVoidParameterArgsMethod(@YarnParameter("key") String value) {
}
```

Configuration Properties

Configuration properties can be defined using various methods. See a Spring Boot dodumentation for details.

Table 11.17. `spring.hadoop` configuration properties

Property Name	Required	Type	Default Value
spring.hadoop.fsUri	Yes	String	null
spring.hadoop.resourceManagerAddress	No	String	null
spring.hadoop.resourceManagerSchedulerAddress	No	String	null
spring.hadoop.resourceManagerHost	No	String	null
spring.hadoop.resourceManagerPort	No	Integer	8032
spring.hadoop.resourceManagerSchedulerPort	No	Integer	8030
spring.hadoop.resources	No	List	null

spring.hadoop.fsUri
    A hdfs file system uri for a namenode.

spring.hadoop.resourceManagerAddress
    Address of a YARN resource manager.

`spring.hadoop.resourceManagerSchedulerAddress`
    Address of a YARN resource manager scheduler.

`spring.hadoop.resourceManagerHost`
    Hostname of a YARN resource manager.

`spring.hadoop.resourceManagerPort`
    Port of a YARN resource manager.

`spring.hadoop.resourceManagerSchedulerPort`
    Port of a YARN resource manager scheduler. This property is only needed for an application master.

`spring.hadoop.resources`
    List of Spring resource locations to be initialized in Hadoop configuration. These resources should be in Hadoop's own site xml format and location format can be anything Spring supports. For example, *classpath:/myentry.xml* from a classpath or *file:/myentry.xml* from a file system.

Table 11.18. `spring.yarn` configuration properties

Property Name	Required	Type	Default Value
`spring.yarn.applicationDir`	No	String	null
`spring.yarn.applicationBaseDir`	No	String	null
`spring.yarn.applicationVersion`	No	String	null
`spring.yarn.stagingDir`	No	String	/spring/staging
`spring.yarn.appName`	No	String	null
`spring.yarn.appType`	No	String	YARN
`spring.yarn.siteYarnAppClasspath`	No	String	null
`spring.yarn.siteMapreduceAppClasspath`	No	String	null

`spring.yarn.applicationDir`
    An application home directory in hdfs. If client copies files into a hdfs during an application submission, files will end up in this directory. If this property is omitted, a staging directory will be used instead.

`spring.yarn.applicationBaseDir`
    An applications base directory where build-in application deployment functionality would create a new application instance. For a normal application submit operation, this is not needed.

`spring.yarn.applicationVersion`
    An application version identifier used together with `applicationBaseDir` in deployment scenarios where `applicationDir` cannot be hard coded.

`spring.yarn.stagingDir`
    A global staging base directory in hdfs.

`spring.yarn.appName`
    Defines a registered application name visible from a YARN resource manager.

`spring.yarn.appType`
    Defines a registered application type used in YARN resource manager.

`spring.yarn.siteYarnAppClasspath`
    Defines a default base YARN application classpath entries.

`spring.yarn.siteMapreduceAppClasspath`
    Defines a default base MR application classpath entries.

Table 11.19. `spring.yarn.appmaster` configuration properties

Property Name	Required	Type	Default Value
`spring.yarn.appmaster.appmasterClass`	No	Class	null
`spring.yarn.appmaster.containerCount`	No	Integer	1
`spring.yarn.appmaster.keepContextAlive`	No	Boolean	true

`spring.yarn.appmaster.appmasterClass`
    Fully qualified classname which auto-configuration can automatically instantiate as a custom application master.

`spring.yarn.appmaster.containerCount`
    Property which is automatically kept in configuration as a hint which an application master can choose to use when determing how many containers should be launched.

`spring.yarn.appmaster.keepContextAlive`
    Setting for an application master runner to stop main thread to wait a latch before continuing. This is needed in cases where main thread needs to wait event from other threads to be able to exit.

Table 11.20. `spring.yarn.appmaster.launchcontext` configuration properties

Property Name	Required	Type	Default Value
`spring.yarn.appmaster.launchcontext.archiveFile`	No	String	null
`spring.yarn.appmaster.launchcontext.runnerClass`	No	Class	null
`spring.yarn.appmaster.launchcontext.options`	No	List	null
`spring.yarn.appmaster.launchcontext.arguments`	No	Map	null
`spring.yarn.appmaster.launchcontext.containerAppClasspath`	No	List	null
`spring.yarn.appmaster.launchcontext.pathSeparator`	No	String	:
`spring.yarn.appmaster.launchcontext.includeBaseDirectory`	No	Boolean	true
`spring.yarn.appmaster.launchcontext.useYarnAppClasspath`	No	Boolean	true
`spring.yarn.appmaster.launchcontext.useMapreduceAppClasspath`	No	Boolean	true
`spring.yarn.appmaster.launchcontext.includeSystemEnv`	No	Boolean	true
`spring.yarn.appmaster.launchcontext.locality`	No	Boolean	false

`spring.yarn.appmaster.launchcontext.archiveFile`
    Indicates that a container main file is treated as executable jar or exploded zip.

`spring.yarn.appmaster.launchcontext.runnerClass`
    Indicates a fully qualified class name for a container runner.

`spring.yarn.appmaster.launchcontext.options`
    JVM system options.

`spring.yarn.appmaster.launchcontext.arguments`
    Application arguments.

`spring.yarn.appmaster.launchcontext.containerAppClasspath`
    Additional classpath entries.

`spring.yarn.appmaster.launchcontext.pathSeparator`
    Separator in a classpath.

`spring.yarn.appmaster.launchcontext.includeBaseDirectory`
    If base directory should be added in a classpath.

`spring.yarn.appmaster.launchcontext.useYarnAppClasspath`
    If default yarn application classpath should be added.

`spring.yarn.appmaster.launchcontext.useMapreduceAppClasspath`
    If default mr application classpath should be added.

`spring.yarn.appmaster.launchcontext.includeSystemEnv`
    If system environment variables are added to a container environment.

`spring.yarn.appmaster.launchcontext.locality`
    If set to true indicates that resources are not relaxed.

Table 11.21. `spring.yarn.appmaster.localizer` configuration properties

Property Name	Required	Type	Default Value
`spring.yarn.appmaster.localizer.patterns`	No	List	null
`spring.yarn.appmaster.localizer.zipPattern`	No	String	null
`spring.yarn.appmaster.localizer.propertiesNames`	No	List	null
`spring.yarn.appmaster.localizer.propertiesSuffixes`	No	List	null

`spring.yarn.appmaster.localizer.patterns`
    A simple patterns to choose localized files.

`spring.yarn.appmaster.localizer.zipPattern`
    A simple pattern to mark a file as archive to be exploded.

`spring.yarn.appmaster.localizer.propertiesNames`
    Base name of a configuration files.

`spring.yarn.appmaster.localizer.propertiesSuffixes`
    Suffixes for a configuration files.

Table 11.22. `spring.yarn.appmaster.resource` configuration properties

Property Name	Required	Type	Default Value
`spring.yarn.appmaster.resource.priority`	No	String	null
`spring.yarn.appmaster.resource.memory`	No	String	null
`spring.yarn.appmaster.resource.virtualCores`	No	String	null

`spring.yarn.appmaster.resource.priority`
    Container priority.

`spring.yarn.appmaster.resource.memory`
    Container memory allocation.

`spring.yarn.appmaster.resource.virtualCores`
    Container cpu allocation.

Table 11.23. `spring.yarn.client` configuration properties

Property Name	Required	Type	Default Value
`spring.yarn.client.files`	No	List	null
`spring.yarn.client.priority`	No	Integer	null
`spring.yarn.client.queue`	No	String	null
`spring.yarn.client.clientClass`	No	Class	null
`spring.yarn.client.startup.action`	No	String	null

`spring.yarn.client.files`
    Files to copy into hdfs during application submission.

`spring.yarn.client.priority`
    Application priority.

`spring.yarn.client.queue`
    Application submission queue.

`spring.yarn.client.clientClass`
    Fully qualified classname which auto-configuration can automatically instantiate as a custom client.

`spring.yarn.client.startup.action`
    Default action to perform on `YarnClient`. Currently only one action named *submit* is supported.
    This action is simply calling `submitApplication` method on `YarnClient`.

Table 11.24. `spring.yarn.client.launchcontext` configuration properties

Property Name	Required	Type	Default Value
`spring.yarn.client.launchcontext.archiveFile`	No	String	null
`spring.yarn.client.launchcontext.runnerClass`	No	Class	null

Property Name	Required	Type	Default Value
`spring.yarn.client.launchcontext.options`	No	List	null
`spring.yarn.client.launchcontext.arguments`	No	Map	null
`spring.yarn.client.launchcontext.containerAppClasspath`	No	List	null
`spring.yarn.client.launchcontext.pathSeparator`	No	String	:
`spring.yarn.client.launchcontext.includeBaseDirectory`	No	Boolean	true
`spring.yarn.client.launchcontext.useYarnAppClasspath`	No	Boolean	true
`spring.yarn.client.launchcontext.useMapreduceAppClasspath`	No	Boolean	true
`spring.yarn.client.launchcontext.includeSystemEnv`	No	Boolean	true

`spring.yarn.client.launchcontext.archiveFile`
    Indicates that a container main file is treated as executable jar or exploded zip.

`spring.yarn.client.launchcontext.runnerClass`
    Indicates a fully qualified class name for a container runner.

`spring.yarn.client.launchcontext.options`
    JVM system options.

`spring.yarn.client.launchcontext.arguments`
    Application arguments.

`spring.yarn.client.launchcontext.containerAppClasspath`
    Additional classpath entries.

`spring.yarn.client.launchcontext.pathSeparator`
    Separator in a classpath.

`spring.yarn.client.launchcontext.includeBaseDirectory`
    If base directory should be added in a classpath.

`spring.yarn.client.launchcontext.useYarnAppClasspath`
    If default yarn application classpath should be added.

`spring.yarn.client.launchcontext.useMapreduceAppClasspath`
    If default mr application classpath should be added.

`spring.yarn.client.launchcontext.includeSystemEnv`
    If system environment variables are added to a container environment.

Table 11.25. `spring.yarn.client.localizer` *configuration properties*

Property Name	Required	Type	Default Value
`spring.yarn.client.localizer.patterns`	No	List	null
`spring.yarn.client.localizer.zipPattern`	No	String	null
`spring.yarn.client.localizer.propertiesNames`	No	List	null

Property Name	Required	Type	Default Value
spring.yarn.client.localizer.propertiesSuffixes	No	List	null

spring.yarn.client.localizer.patterns
    A simple patterns to choose localized files.

spring.yarn.client.localizer.zipPattern
    A simple pattern to mark a file as archive to be exploded.

spring.yarn.client.localizer.propertiesNames
    Base name of a configuration files.

spring.yarn.client.localizer.propertiesSuffixes
    Suffixes for a configuration files.

Table 11.26. `spring.yarn.client.resource` configuration properties

Property Name	Required	Type	Default Value
spring.yarn.client.resource.memory	No	String	null
spring.yarn.client.resource.virtualCores	No	String	null

spring.yarn.client.resource.memory
    Application master memory allocation.

spring.yarn.client.resource.virtualCores
    Application master cpu allocation.

Table 11.27. `spring.yarn.container` configuration properties

Property Name	Required	Type	Default Value
spring.yarn.container.keepContextAlive	No	Boolean	true
spring.yarn.container.containerClass	No	Class	null

spring.yarn.container.keepContextAlive
    Setting for an application container runner to stop main thread to wait a latch before continuing. This
    is needed in cases where main thread needs to wait event from other threads to be able to exit.

spring.yarn.container.containerClass
    Fully qualified classname which auto-configuration can automatically instantiate as a custom
    container.

Table 11.28. `spring.yarn.batch` configuration properties

Property Name	Required	Type	Default Value
spring.yarn.batch.name	No	String	null
spring.yarn.batch.enabled	No	Boolean	false
spring.yarn.batch.jobs	No	List	null

`spring.yarn.batch.name`
    Comma-delimited list of search patterns to find jobs to run defined either locally in application context or in job registry.

`spring.yarn.batch.enabled`
    Indicates if batch processing on yarn is enabled.

`spring.yarn.batch.jobs`
    Indicates a list of individual configuration properties for jobs.

Table 11.29. `spring.yarn` configuration properties

Property Name	Required	Type	Default Value
`spring.yarn.batch.jobs.name`	No	String	null
`spring.yarn.batch.jobs.enabled`	No	Boolean	false
`spring.yarn.batch.jobs.next`	No	Boolean	false
`spring.yarn.batch.jobs.failNext`	No	Boolean	false
`spring.yarn.batch.jobs.restart`	No	Boolean	false
`spring.yarn.batch.jobs.failRestart`	No	Boolean	false
`spring.yarn.batch.jobs.parameters`	No	String	null

Descriptions

`spring.yarn.batch.jobs.name`
    Name of a job to configure.

`spring.yarn.batch.jobs.enabled`
    Indicates if job is enabled.

`spring.yarn.batch.jobs.next`
    Indicates if job parameters incrementer is used to prepare a job for next run.

`spring.yarn.batch.jobs.failNext`
    Indicates if job execution should fail if job cannot be prepared for next execution.

`spring.yarn.batch.jobs.restart`
    Indicates of job should be restarted.

`spring.yarn.batch.jobs.failRestart`
    Indicates if job execution should fail if job cannot be restarted.

`spring.yarn.batch.jobs.parameters`
    Defines a Map of additional job parameters. Keys and values are in normal format supported by Batch.

Controlling Applications

We've already talked about how resources are localized into a running container. These resources are always localized from a HDFS file system which effectively means that the whole process of getting application files into a newly launched YARN application is a two phase process; firstly files are copied into HDFS and secondly files are localized from a HDFS.

When application instance is submitted into YARN, there are two ways how these application files can be handled. First which is the most obvious is to just copy all the necessary files into a known location in HDFS and then instruct YARN to localize files from there. Second method is to split this into two different stages, first install application files into HDFS and then submit application from there. At first there seem to be no difference with these two ways to handle application deployment. However if files are always copied into HDFS when application is submitted, you need a physical access to those files. This may not always be possible so it's easier if you have a change to prepare these files by first installing application into HDFS and then just send a submit command to a YARN resource manager.

To ease a process of handling a full application life cycle, few utility classes exist which are meant to be used with Spring Boot. These classes are considered to be a foundational Boot application classes, not a ready packaged Boot executable jars. Instead you would use these from your own application whether that application is a Boot or other Spring based application.

Generic Usage

Internally these applications are executed using a `SpringApplicationBuilder` and a dedicated *Spring Application Context*. This allows to isolate Boot application instance from your current context if you have one. One fundamental idea in these applications is to make it possible to work with Spring profiles and Boot configuration properties. If your existing application is already using profiles and configuration properties, simply launching a new Boot would most likely derive those settings automatically which is something what you may not want.

`AbstractClientApplication` which all these built-in applications are based on contains methods to work with *Spring profiles* and additional configuration properties.

Let's go through all this using an example:

Using Configuration Properties

Below sample is pretty much a similar from all other examples except of two settings, `applicationBaseDir` and `clientClass`. Property `applicationBaseDir` defines where in HDFS a new app will be installed. `DefaultApplicationYarnClient` defined using `clientClass` adds better functionality to guard against starting app which doesn't exist or not overwriting existing apps in HDFS.

```yaml
spring:
  hadoop:
    fsUri: hdfs://localhost:8020
    resourceManagerHost: localhost
  yarn:
    appType: GS
    appName: gs-yarn-appmodel
    applicationBaseDir: /app/
    applicationDir: /app/gs-yarn-appmodel/
    client:
      clientClass: org.springframework.yarn.client.DefaultApplicationYarnClient
      files:
        - "file:build/libs/gs-yarn-appmodel-container-0.1.0.jar"
        - "file:build/libs/gs-yarn-appmodel-appmaster-0.1.0.jar"
      launchcontext:
        archiveFile: gs-yarn-appmodel-appmaster-0.1.0.jar
    appmaster:
      containerCount: 1
      launchcontext:
        archiveFile: gs-yarn-appmodel-container-0.1.0.jar
```

Using YarnPushApplication

YarnPushApplication is used to push your application into HDFS.

```
public void doInstall() {
  YarnPushApplication app = new YarnPushApplication();
  app.applicationVersion("version1");
  Properties instanceProperties = new Properties();
  instanceProperties.setProperty("spring.yarn.applicationVersion", "version1");
  app.configFile("application.properties", instanceProperties);
  app.run();
}
```

In above example we simply created a `YarnPushApplication`, set its `applicationVersion` and executed a run method. We also instructed `YarnPushApplication` to write used `applicationVersion` into a configuration file named *application.properties* so that it'd be available to an application itself.

Using YarnSubmitApplication

YarnSubmitApplication is used to submit your application from HDFS into YARN.

```
public void doSubmit() {
  YarnSubmitApplication app = new YarnSubmitApplication();
  app.applicationVersion("version1");
  ApplicationId applicationId = app.run();
}
```

In above example we simply created a `YarnSubmitApplication`, set its `applicationVersion` and executed a run method.

Using YarnInfoApplication

YarnInfoApplication is used to query application info from a YARN Resource Manager and HDFS.

```
public void doListPushed() {
  YarnInfoApplication app = new YarnInfoApplication();
  Properties appProperties = new Properties();

 appProperties.setProperty("spring.yarn.internal.YarnInfoApplication.operation", "PUSHED");
  app.appProperties(appProperties);
  String info = app.run();
  System.out.println(info);
}

public void doListSubmitted() {
  YarnInfoApplication app = new YarnInfoApplication();
  Properties appProperties = new Properties();

 appProperties.setProperty("spring.yarn.internal.YarnInfoApplication.operation", "SUBMITTED");
  appProperties.setProperty("spring.yarn.internal.YarnInfoApplication.verbose", "true");
  appProperties.setProperty("spring.yarn.internal.YarnInfoApplication.type", "GS");
  app.appProperties(appProperties);
  String info = app.run();
  System.out.println(info);
}
```

In above example we simply created a `YarnInfoApplication`, and used it to list installed and running applications. By adding `appProperties` will make Boot to pick these properties after every other

source of configuration properties but still allows to pass command-line options to override everything which is a normal way in Boot.

Using YarnKillApplication

YarnKillApplication is used to kill running application instances.

```
public void doKill() {
  YarnKillApplication app = new YarnKillApplication();
  Properties appProperties = new Properties();

 appProperties.setProperty("spring.yarn.internal.YarnKillApplication.applicationId", "application_139505803
  app.appProperties(appProperties);
  String info = app.run();
  System.out.println(info);
}
```

In above example we simply created a `YarnKillApplication`, and used it to send a application kill request into a YARN resource manager.

12. Testing Support

Hadoop testing has always been a cumbersome process especially if you try to do testing phase during the normal project build process. Traditionally developers have had few options like running Hadoop cluster either as a local or pseudo-distributed mode and then utilise that to run MapReduce jobs. Hadoop project itself is using a lot of mini clusters during the tests which provides better tools to run your code in an isolated environment.

Spring Hadoop and especially its Yarn module faced similar testing problems. Spring Hadoop provides testing facilities order to make testing on Hadoop much easier especially if code relies on Spring Hadoop itself. These testing facilities are also used internally to test Spring Hadoop, although some test cases still rely on a running Hadoop instance on a host where project build is executed.

Two central concepts of testing using Spring Hadoop is, firstly fire up the mini cluster and secondly use the configuration prepared by the mini cluster to talk to the Hadoop components. Now let's go through the general testing facilities offered by Spring Hadoop.

Testing for MapReduce and Yarn in Spring Hadoop is separated into different packages mostly because these two components doesn't have hard dependencies with each others. You will see a lot of similarities when creating tests for MapReduce and Yarn.

12.1 Testing MapReduce

Mini Clusters for MapReduce

Mini clusters usually contain testing components from a Hadoop project itself. These are clusters for *MapReduce Job* handling and *HDFS* which are all run within a same process. In Spring Hadoop mini clusters are implementing interface `HadoopCluster` which provides methods for lifecycle and configuration. *Spring Hadoop* provides transitive maven dependencies against different *Hadoop* distributions and thus mini clusters are started using different implementations. This is mostly because we want to support *HadoopV1* and *HadoopV2* at a same time. All this is handled automatically at runtime so everything should be transparent to the end user.

```
public interface HadoopCluster {
  Configuration getConfiguration();
  void start() throws Exception;
  void stop();
  FileSystem getFileSystem() throws IOException;
}
```

Currently one implementation named *StandaloneHadoopCluster* exists which supports simple cluster type where a number of nodes can be defined and then all the nodes will contain utilities for *MapReduce Job* handling and *HDFS*.

There are few ways how this cluster can be started depending on a use case. It is possible to use `StandaloneHadoopCluster` directly or configure and start it through `HadoopClusterFactoryBean`. Existing `HadoopClusterManager` is used in unit tests to cache running clusters.

Note

It's advisable not to use `HadoopClusterManager` outside of tests because literally it is using static fields to cache cluster references. This is a same concept used in *Spring Test* order to cache application contexts between the unit tests within a jvm.

```
<bean id="hadoopCluster" class="org.springframework.data.hadoop.test.support.HadoopClusterFactoryBean">
  <property name="clusterId" value="HadoopClusterTests"/>
  <property name="autoStart" value="true"/>
  <property name="nodes" value="1"/>
</bean>
```

Example above defines a bean named *hadoopCluster* using a factory bean `HadoopClusterFactoryBean`. It defines a simple one node cluster which is started automatically.

Configuration

Spring Hadoop components usually depend on *Hadoop* configuration which is then wired into these components during the application context startup phase. This was explained in previous chapters so we don't go through it again. However this is now a catch-22 because we need the configuration for the context but it is not known until mini cluster has done its startup magic and prepared the configuration with correct values reflecting current runtime status of the cluster itself. Solution for this is to use other bean named `ConfigurationDelegatingFactoryBean` which will simply delegate the configuration request into the running cluster.

```
<bean id="hadoopConfiguredConfiguration" class="org.springframework.data.hadoop.test.support.ConfigurationD
  <property name="cluster" ref="hadoopCluster"/>
</bean>

<hdp:configuration id="hadoopConfiguration" configuration-
ref="hadoopConfiguredConfiguration"/>
```

In the above example we created a bean named *hadoopConfiguredConfiguration* using `ConfigurationDelegatingFactoryBean` which simple delegates to *hadoopCluster* bean. Returned bean *hadoopConfiguredConfiguration* is type of *Hadoop's* `Configuration` object so it could be used as it is.

Latter part of the example show how *Spring Hadoop* namespace is used to create another `Configuration` object which is using *hadoopConfiguredConfiguration* as a reference. This scenario would make sense if there is a need to add additional configuration options into running configuration used by other components. Usually it is suiteable to use cluster prepared configuration as it is.

Simplified Testing

It is perfecly all right to create your tests from scratch and for example create the cluster manually and then get the runtime configuration from there. This just needs some boilerplate code in your context configuration and unit test lifecycle.

Spring Hadoop adds additional facilities for the testing to make all this even easier.

```
@RunWith(SpringJUnit4ClassRunner.class)
public abstract class AbstractHadoopClusterTests implements ApplicationContextAware {
  ...
}

@ContextConfiguration(loader=HadoopDelegatingSmartContextLoader.class)
@MiniHadoopCluster
public class ClusterBaseTestClassTests extends AbstractHadoopClusterTests {
  ...
}
```

Above example shows the `AbstractHadoopClusterTests` and how `ClusterBaseTestClassTests` is prepared to be aware of a mini cluster. `HadoopDelegatingSmartContextLoader` offers same base functionality as the default `DelegatingSmartContextLoader` in a spring-test package. One additional thing what `HadoopDelegatingSmartContextLoader` does is to automatically handle running clusters and inject `Configuration` into the application context.

```
@MiniHadoopCluster(configName="hadoopConfiguration", clusterName="hadoopCluster", nodes=1,
  id="default")
```

Generally `@MiniHadoopCluster` annotation allows you to define injected bean name for mini cluster, its Configurations and a number of nodes you like to have in a cluster.

Spring Hadoop testing is dependant of general facilities of *Spring Test framework* meaning that everything what is cached during the test are reuseable withing other tests. One need to understand that if *Hadoop* mini cluster and its `Configuration` is injected into an Application Context, caching happens on a mercy of a Spring Testing meaning if a test Application Context is cached also mini cluster instance is cached. While caching is always prefered, one needs to understant that if tests are expecting vanilla environment to be present, test context should be dirtied using `@DirtiesContext` annotation.

Wordcount Example

Let's study a proper example of existing *MapReduce Job* which is executed and tested using *Spring Hadoop*. This example is the Hadoop's classic wordcount. We don't go through all the details of this example because we want to concentrate on testing specific code and configuration.

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

<hdp:job id="wordcountJob"
  input-path="${wordcount.input.path}"
  output-path="${wordcount.output.path}"
  libs="file:build/libs/mapreduce-examples-wordcount-*.jar"
  mapper="org.springframework.data.hadoop.examples.TokenizerMapper"
  reducer="org.springframework.data.hadoop.examples.IntSumReducer" />

<hdp:script id="setupScript" location="copy-files.groovy">
  <hdp:property name="localSourceFile" value="data/nietzsche-chapter-1.txt" />
  <hdp:property name="inputDir" value="${wordcount.input.path}" />
  <hdp:property name="outputDir" value="${wordcount.output.path}" />
</hdp:script>

<hdp:job-runner id="runner"
  run-at-startup="false"
  kill-job-at-shutdown="false"
  wait-for-completion="false"
  pre-action="setupScript"
  job-ref="wordcountJob" />
```

In above configuration example we can see few differences with the actual runtime configuration. Firstly you can see that we didn't specify any kind of configuration for hadoop. This is because it's is injected automatically by testing framework. Secondly because we want to explicitely wait the job to be run and finished, *kill-job-at-shutdown* and *wait-for-completion* are set to *false*.

```
@ContextConfiguration(loader=HadoopDelegatingSmartContextLoader.class)
@MiniHadoopCluster
public class WordcountTests extends AbstractMapReduceTests {
  @Test
  public void testWordcountJob() throws Exception {
    // run blocks and throws exception if job failed
    JobRunner runner = getApplicationContext().getBean("runner", JobRunner.class);
    Job wordcountJob = getApplicationContext().getBean("wordcountJob", Job.class);

    runner.call();

    JobStatus finishedStatus = waitFinishedStatus(wordcountJob, 60, TimeUnit.SECONDS);
    assertThat(finishedStatus, notNullValue());

    // get output files from a job
    Path[] outputFiles = getOutputFilePaths("/user/gutenberg/output/word/");
    assertEquals(1, outputFiles.length);
    assertThat(getFileSystem().getFileStatus(outputFiles[0]).getLen(), greaterThan(0l));

    // read through the file and check that line with
    // "themselves 6" was found
    boolean found = false;
    InputStream in = getFileSystem().open(outputFiles[0]);
    BufferedReader reader = new BufferedReader(new InputStreamReader(in));
    String line = null;
    while ((line = reader.readLine()) != null) {
      if (line.startsWith("themselves")) {
        assertThat(line, is("themselves\t6"));
        found = true;
      }
    }
    reader.close();
    assertThat("Keyword 'themselves' not found", found);
  }
}
```

In above unit test class we simply run the job defined in xml, explicitly wait it to finish and then check the output content from *HDFS* by searching expected strings.

12.2 Testing Yarn

Mini Clusters for Yarn

Mini cluster usually contain testing components from a *Hadoop* project itself. These are `MiniYARNCluster` for Resource Manager and `MiniDFSCluster` for Datanode and Namenode which are all run within a same process. In *Spring Hadoop* mini clusters are implementing interface `YarnCluster` which provides methods for lifecycle and configuration.

```
public interface YarnCluster {
  Configuration getConfiguration();
  void start() throws Exception;
  void stop();
  File getYarnWorkDir();
}
```

Currently one implementation named `StandaloneYarnCluster` exists which supports simple cluster type where a number of nodes can be defined and then all the nodes will have *Yarn Node Manager* and *Hdfs Datanode*, additionally a *Yarn Resource Manager* and *Hdfs Namenode* components are started.

There are few ways how this cluster can be started depending on a use case. It is possible to use `StandaloneYarnCluster` directly or configure and start it through `YarnClusterFactoryBean`. Existing `YarnClusterManager` is used in unit tests to cache running clusters.

Note

> It's advisable not to use `YarnClusterManager` outside of tests because literally it is using static fields to cache cluster references. This is a same concept used in *Spring Test* order to cache application contexts between the unit tests within a jvm.

```
<bean id="yarnCluster" class="org.springframework.yarn.test.support.YarnClusterFactoryBean">
  <property name="clusterId" value="YarnClusterTests"/>
  <property name="autoStart" value="true"/>
  <property name="nodes" value="1"/>
</bean>
```

Example above defines a bean named *yarnCluster* using a factory bean `YarnClusterFactoryBean`. It defines a simple one node cluster which is started automatically. Cluster working directories would then exist under below paths:

```
target/YarnClusterTests/
target/YarnClusterTests-dfs/
```

Note

> We rely on base classes from a *Hadoop* distribution and target base directory is hardcoded in Hadoop and is not configurable.

Configuration

Spring Yarn components usually depend on *Hadoop* configuration which is then wired into these components during the application context startup phase. This was explained in previous chapters so we don't go through it again. However this is now a catch-22 because we need the configuration for the context but it is not known until mini cluster has done its startup magic and prepared the configuration with correct values reflecting current runtime status of the cluster itself. Solution for this is to use other factory bean class named `ConfigurationDelegatingFactoryBean` which will simple delegate the configuration request into the running cluster.

```
<bean id="yarnConfiguredConfiguration" class="org.springframework.yarn.test.support.ConfigurationDelegatingF
  <property name="cluster" ref="yarnCluster"/>
</bean>

<yarn:configuration id="yarnConfiguration" configuration-
ref="yarnConfiguredConfiguration"/>
```

In the above example we created a bean named *yarnConfiguredConfiguration* using `ConfigurationDelegatingFactoryBean` which simple delegates to *yarnCluster* bean. Returned bean *yarnConfiguredConfiguration* is type of *Hadoop's* `Configuration` object so it could be used as it is.

Latter part of the example show how *Spring Yarn* namespace is used to create another `Configuration` object which is using *yarnConfiguredConfiguration* as a reference. This scenario would make sense if there is a need to add additional configuration options into running configuration used by other components. Usually it is suiteable to use cluster prepared configuration as it is.

Simplified Testing

It is perfecly all right to create your tests from scratch and for example create the cluster manually and then get the runtime configuration from there. This just needs some boilerplate code in your context configuration and unit test lifecycle.

Spring Hadoop adds additional facilities for the testing to make all this even easier.

```
@RunWith(SpringJUnit4ClassRunner.class)
public abstract class AbstractYarnClusterTests implements ApplicationContextAware {
  ...
}

@ContextConfiguration(loader=YarnDelegatingSmartContextLoader.class)
@MiniYarnCluster
public class ClusterBaseTestClassTests extends AbstractYarnClusterTests {
  ...
}
```

Above example shows the `AbstractYarnClusterTests` and how `ClusterBaseTestClassTests` is prepared to be aware of a mini cluster. `YarnDelegatingSmartContextLoader` offers same base functionality as the default `DelegatingSmartContextLoader` in a spring-test package. One additional thing what `YarnDelegatingSmartContextLoader` does is to automatically handle running clusters and inject `Configuration` into the application context.

```
@MiniYarnCluster(configName="yarnConfiguration", clusterName="yarnCluster", nodes=1,
  id="default")
```

Generally `@MiniYarnCluster` annotation allows you to define injected bean names for mini cluster, its Configurations and a number of nodes you like to have in a cluster.

Spring Hadoop Yarn testing is dependant of general facilities of *Spring Test* framework meaning that everything what is cached during the test are reuseable withing other tests. One need to understand that if *Hadoop* mini cluster and its `Configuration` is injected into an Application Context, caching happens on a mercy of a Spring Testing meaning if a test Application Context is cached also mini cluster instance is cached. While caching is always prefered, one needs to understant that if tests are expecting vanilla environment to be present, test context should be dirtied using `@DirtiesContext` annotation.

Spring Test Context configuration works exactly like you'd work with any other *Spring Test* based tests. It defaults on finding xml based config and fall back to Annotation based config. For example if one is working with *JavaConfig* a simple static configuration class can be used within the test class.

For test cases where additional context configuration is not needed a simple helper annotation `@MiniYarnClusterTest` can be used.

```
@MiniYarnClusterTest
public class ActivatorTests extends AbstractBootYarnClusterTests {
  @Test
  public void testSomething(){
    ...
  }
}
```

In above example a simple test case was created using annontation *@MiniYarnClusterTest* Behind a scenes it's using junit and prepares a YARN minicluster for you and injects needed configuration for you.

Drawback of using a composed annotation like this is that the *@Configuration* is then applied from an annotation class itself and user can't no longer add a static *@Configuration* class in a test class itself and expect Spring to pick it up from there which is a normal behaviour in Spring testing support. If user wants to use a simple composed annotation and use a custom *@Configuration*, one can simply duplicate functionality of this *@MiniYarnClusterTest* annotation.

```java
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@ContextConfiguration(loader=YarnDelegatingSmartContextLoader.class)
@MiniYarnCluster
public @interface CustomMiniYarnClusterTest {

  @Configuration
  public static class Config {
    @Bean
    public String myCustomBean() {
      return "myCustomBean";
    }
  }

}

@RunWith(SpringJUnit4ClassRunner.class)
@CustomMiniYarnClusterTest
public class ComposedAnnotationTests {

  @Autowired
  private ApplicationContext ctx;

  @Test
  public void testBean() {
    assertTrue(ctx.containsBean("myCustomBean"));
  }

}
```

In above example a custom composed annotation *@CustomMiniYarnClusterTest* was created and then used within a test class. This a great way to put your configuration is one place and still keep your test class relatively non-verbose.

Multi Context Example

Let's study a proper example of existing Spring Yarn application and how this is tested during the build process. Multi Context Example is a simple Spring Yarn based application which simply launches Application Master and four Containers and withing those containers a custom code is executed. In this case simply a log message is written.

In real life there are different ways to test whether Hadoop Yarn application execution has been succesful or not. The obvious method would be to check the application instance execution status reported by Hadoop Yarn. Status of the execution doesn't always tell the whole truth so i.e. if application is about to write something into HDFS as an output that could be used to check the proper outcome of an execution.

This example doesn't write anything into HDFS and anyway it would be out of scope of this document for obvious reason. It is fairly straightforward to check file content from HDFS. One other interesting method is simply to check to application log files that being the Application Master and Container logs. Test methods can check exceptions or expected log entries from a log files to determine whether test is succesful or not.

In this chapter we don't go through how Multi Context Example is configured and what it actually does, for that read the documentation about the examples. However we go through what needs to be done order to test this example application using testing support offered by Spring Hadoop.

In this example we gave instructions to copy library dependencies into Hdfs and then those entries were used within resouce localizer to tell Yarn to copy those files into Container working directory. During the unit testing when mini cluster is launched there are no files present in Hdfs because cluster is initialized from scratch. Furtunalety Spring Hadoop allows you to copy files into Hdfs during the localization process from a local file system where Application Context is executed. Only thing we need is the actual library files which can be assembled during the build process. Spring Hadoop Examples build system rely on Gradle so collecting dependencies is an easy task.

```
<yarn:localresources>
  <yarn:hdfs path="/app/multi-context/*.jar"/>
  <yarn:hdfs path="/lib/*.jar"/>
</yarn:localresources>
```

Above configuration exists in application-context.xml and appmaster-context.xml files. This is a normal application configuration expecting static files already be present in Hdfs. This is usually done to minimize latency during the application submission and execution.

```
<yarn:localresources>
  <yarn:copy src="file:build/dependency-libs/*" dest="/lib/"/>
  <yarn:copy src="file:build/libs/*" dest="/app/multi-context/"/>
  <yarn:hdfs path="/app/multi-context/*.jar"/>
  <yarn:hdfs path="/lib/*.jar"/>
</yarn:localresources>
```

Above example is from MultiContextTest-context.xml which provides the runtime context configuration talking with mini cluster during the test phase.

When we do context configuration for YarnClient during the testing phase all we need to do is to add copy elements which will transfer needed libraries into Hdfs before the actual localization process will fire up. When those files are copied into Hdfs running in a mini cluster we're basically in a same point if using a real Hadoop cluster with existing files.

Note

Running tests which depends on copying files into Hdfs it is mandatory to use build system which is able to prepare these files for you. You can't do this within IDE's which have its own ways to execute unit tests.

The complete example of running the test, checking the application execution status and finally checking the expected state of log files:

```
@ContextConfiguration(loader=YarnDelegatingSmartContextLoader.class)
@MiniYarnCluster
public class MultiContextTests extends AbstractYarnClusterTests {
  @Test
  @Timed(millis=70000)
  public void testAppSubmission() throws Exception {
    YarnApplicationState state = submitApplicationAndWait();
    assertNotNull(state);
    assertTrue(state.equals(YarnApplicationState.FINISHED));

    File workDir = getYarnCluster().getYarnWorkDir();

    PathMatchingResourcePatternResolver resolver = new
 PathMatchingResourcePatternResolver();
    String locationPattern = "file:" + workDir.getAbsolutePath() + "/**/*.std*";
    Resource[] resources = resolver.getResources(locationPattern);

    // appmaster and 4 containers should
    // make it 10 log files
    assertThat(resources, notNullValue());
    assertThat(resources.length, is(10));

    for (Resource res : resources) {
      File file = res.getFile();
      if (file.getName().endsWith("stdout")) {
        // there has to be some content in stdout file
        assertThat(file.length(), greaterThan(0l));
        if (file.getName().equals("Container.stdout")) {
          Scanner scanner = new Scanner(file);
          String content = scanner.useDelimiter("\\A").next();
          scanner.close();
          // this is what container will log in stdout
          assertThat(content, containsString("Hello from MultiContextBeanExample"));
        }
      } else if (file.getName().endsWith("stderr")) {
        // can't have anything in stderr files
        assertThat(file.length(), is(0l));
      }
    }
  }
}
```

12.3 Testing Boot Based Applications

In previous sections we showed a generic concepts of unit testing in *Spring Hadoop* and *Spring YARN*. We also have a first class support for testing *Spring Boot* based applications made for YARN.

```
@MiniYarnClusterTest
public class AppTests extends AbstractBootYarnClusterTests {

  @Test
  public void testApp() throws Exception {
    ApplicationInfo info = submitApplicationAndWait(ClientApplication.class, new
 String[0]);
    assertThat(info.getYarnApplicationState(), is(YarnApplicationState.FINISHED));

    List<Resource> resources = ContainerLogUtils.queryContainerLogs(
      getYarnCluster(), info.getApplicationId());
    assertThat(resources, notNullValue());
    assertThat(resources.size(), is(4));

    for (Resource res : resources) {
      File file = res.getFile();
      String content = ContainerLogUtils.getFileContent(file);
      if (file.getName().endsWith("stdout")) {
        assertThat(file.length(), greaterThan(0l));
        if (file.getName().equals("Container.stdout")) {
          assertThat(content, containsString("Hello from HelloPojo"));
        }
      } else if (file.getName().endsWith("stderr")) {
        assertThat("stderr with content: " + content, file.length(), is(0l));
      }
    }
  }

}
```

Let's go through step by step what's happening in this JUnit class. As already mentioned earlier we don't
need any existing or running Hadoop instances, instead testing framework from Spring YARN provides
an easy way to fire up a mini cluster where your tests can be run in an isolated environment.

- `@ContextConfiguration` together with `YarnDelegatingSmartContextLoader` tells Spring to
  prepare a testing context for a mini cluster. `EmptyConfig` is a simple helper class to use if there are
  no additional configuration for tests.

- `@MiniYarnCluster` tells Spring to start a Hadoop's mini cluster having components for *HDFS* and
  YARN. Hadoop's configuration from this minicluster is automatically injected into your testing context.

- `@MiniYarnClusterTest` is basically a replacement of `@MiniYarnCluster` and
  `@ContextConfiguration` having an empty context configuration.

- `AbstractBootYarnClusterTests` is a class containing a lot of base functionality what you need
  in your tests.

Then it's time to deploy the application into a running minicluster

- `submitApplicationAndWait()` method simply runs your `ClientApplication` and expects it
  to an application deployment. On default it will wait 60 seconds an application to finish and returns
  an current state.

- We make sure that we have a correct application state

We use `ContainerLogUtils` to find our container logs files from a minicluster.

- We assert count of a log files

- We expect some specified content from log file

- We expect stderr files to be empty

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.

13. 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.

13.1 Scheduling

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

13.2 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 and there is also some instructions in the *README* file of each example.

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.

14. Useful Links

- *Spring for Apache Hadoop - [Home Page](#)*

- *Spring Data - [Home Page](#)*

- *Spring Data Book - [Home Page](#)*

- *Spring - [Blog](#)*

- *Apache Hadoop - [Home Page](#)*

- *Pivotal HD - [Home Page](#)*

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:
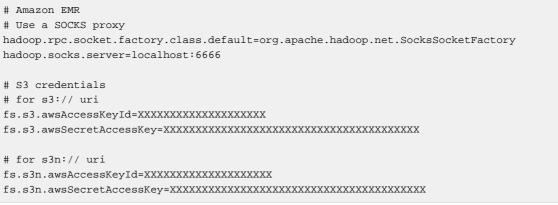
- Start an *alive* cluster using the bootstrap action to guarantees the cluster does NOT use a socks proxy. Open a SSH tunnel, in SOCKS mode, to the EMR cluster.
  Start the cluster for an indefinite period. Once the server is up, create an SSH tunnel,in SOCKS mode, to the remote cluster. This allows 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

```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=XXXXXXXXXXXXXXXXXXXXX
fs.s3.awsSecretAccessKey=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

# for s3n:// uri
fs.s3n.awsAccessKeyId=XXXXXXXXXXXXXXXXXXXXX
fs.s3n.awsSecretAccessKey=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

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">
```