Nomic Documentation

Release

Zdenko Vrabel <vrabel.zdenko@gmail.com>

Dec 14, 2017

Contents

Nomic is small package manager-kind of applications that automatise our process of deployment and installation of analytics applications into Hadoop ecosystem. The analytics are packaged into archive file called **box**. These boxes have all files bundled inside and one descriptor file in root named `nomic.box` where is declared what and where is installed. The descriptor file is declarative DSL it's based on Groovy.

CHAPTER 1

Sitemap

1.1 Getting Started

Before you start creating own boxes and installing them into your Hadoop ecosystem, you have to get,install and setup the Nomic application.

Important: On first place check if your environment fulfills the most important requirement: **Java 1.8**

When the Java is present in your environment, we can get the application. The various distribution packages are available on Bintray. Pick the package depends on what kind of installation you will choose. For more details about installation on various systems, see the *Installation* section. For experimenting and playing around the Nomic, I recommend the TAR(or ZIP) distribution.

Download the latest version of application and unpack it to some working directory:

```
$ wget https://dl.bintray.com/sn3d/nomic-repo/nomic-{version}-bin.tar
$ tar -xzf ./nomic-{version}-bin.tar
$ cd ./nomic-{version}
```

1.1.1 Configuration

After unpacking you will need to configure Nomic application. First, you should copy the Hadoop `code-site.xml` and `hdfs-site.xml` into `./conf` folder. The Nomic use these files for connecting to HDFS. If you're not happy with copying these files, you can determine paths to files via configuration file `conf/nomic.conf`.

```
hdfs.core.site=/path/to/core-site.xml
hdfs.hdfs.site=/path/to/hdfs-site.xml
```

The default configuration is using user's home folder. The boxes are installed into `hdfs://${USER_HOME}/app` folder and nomic store metadata into `hdfs://${USER_HOME}/.nomic`.

You can test you configuration by executing

```
$ ./bin/nomic config
```

After execution you should see how your Nomic instance is configured. If this command failed, probably you have something wrong in `conf/nomic.conf` or your `core-site.xml` and `hdfs-site.xml` are invalid.

1.1.2 Your first box

Now it's time to create your first box. Let's imagine we've got simple Oozie `workflow.xml` and we want to deploy it as analytics application. We have to create a `nomic.box` nearby workflow file with content:

```
group = "examples"
name = "simple-workflow"
version = "1.0.0"

hdfs {
  resource "workflow.xml"
}
```

For more information about how to write box descriptor files you can visit *Nomic DSL Guide* section. Then we need to pack both files into archive bundle. For that purpose we can use java JAR utility:

```
$ jar cf simple-workflow.nomic ./workflow.xml ./nomic.box
```

Huray! You've got your first box ready for deployment. Let's deploy it.

1.1.3 Deploying and removing

In previous section we've created our first nomic box. We can deploy it easily by executing command:

```
$ ./bin/nomic install simple-workflow.nomic
```

In your HDFS, you should have `workflow.xml` available in application folder `${USER_HOME}/app/examples/simple-workflow`. Also after executing `./bin/nomic list` command, you will see the box was installed and what version of box is available. We should see output:

```
$ ./bin/nomic list
examples:simple-workflow:1.0.0
```

One of the primary goals of Nomic application is not only deploying but also safe removing of deployed boxes. Let's remove our box:

```
$ ./bin/nomic remove examples:simple-workflow:1.0.0
```

The remove command will erase only these resources, they were deployed. It's inverse of `deploy` command.

1.2 Installation

Important: On first place check if your environment fulfills the most important requirement: **Java 1.8**

1.2.1 Standalone Installation (Windows)

For Windows systems, you can yse standalone distribution available as ZIP archive. All you need is just download the ZIP from Bintray, unpack it into some folder and run `bin\nomic.bat`.

1.2.2 Standalone Installation (Linux/Mac OS)

For Linux and Mac OS systems, you can use standalone distribution available as TAR archive. All you need is just download it, unpack it and run.

```
$ wget https://dl.bintray.com/sn3d/nomic-repo/nomic-{version}-bin.tar
$ tar -xzf ./nomic-{version}-bin.tar
$ cd ./nomic-{version}
$ ./bin/nomic
```

All configuration files, libraries and shell scripts are placed in one folder.

1.2.3 Installation from RPM (RedHat)

If you're using Linux system with YUM or RPM, you can install application as RPM directly:

```
$ sudo rpm -i https://dl.bintray.com/sn3d/nomic-repo/nomic-{version}.noarch.rpm
$ nomic
```

Or you can add YUM repository with Nomic application:

```
$ wget https://bintray.com/sn3d/nomic-rhel/rpm -O bintray-sn3d-nomic-rhel.repo
$ sudo mv bintray-sn3d-nomic-rhel.repo /etc/yum.repos.d/
```

And then you can use YUM for installing/upgrading:

```
$ sudo yum install nomic
$ nomic
```

Now you can type `nomic` in shell. You should see the nomic's output. The configuration is placed in `/etc/nomic`. The main application is placed in `/usr/share/nomic` folder.

1.3 Configuration

The Nomic configuration can be divided into 2 parts:

- environment configuration
- application configuration

1.3.1 Environment configuration

The environment configuration is about setting few important envirnoment variables:

- **$JAVA_HOME:** path to java JRE which will be used
- **$NOMIC_HOME:** point to place where is Nomic application installed.

- **$NOMIC_CONF:** path to `nomic.conf` file. The default value is `$NOMIC_HOME/conf/nomic.conf` but you can specify alternative file.
- **$NOMIC_OPTS:** Java runtime options used when Nomic application is executed

All these environment variables can be set as global env. variables via system or you can override them in `$NOMIC_HOME/conf/setenv`.

1.3.2 Application configuration

The application configuration is more detailed configuration that is happening in the `nomic.conf` file. The configuration file is following HOCON format and Nomic application will look for this file in the `$NOMIC_HOME/conf` folder.

This example show all of the default values:

```
##############################################################################
# General Nomic configuration
##############################################################################

# user what will be used by Nomic application. Also folders and files in HDFS
# will be owned by this used.
#
# default value is system user you're logged with
nomic.user = ${user.name}

# Nomic home on FileSystem where nomic will look for configuration, write
# logs etc.
#
# default value is in user's home folder as '.nomic' folder
nomic.home = ${user.home}"/.nomic"

# This is home folder that will be used by Nomic in HDFS.
#
# default value is hdfs://server/user/{user.name}
nomic.hdfs.home = "/user/"${nomic.user}

# Point to 'app' folder in HDFS where are all boxes deployed
#
# default value is hdfs://server/user/{user.name}/app
nomic.hdfs.app.dir = ${nomic.hdfs.home}"/app"

# Point to 'repository' folder in HDFS where Nomic will store metadata
#
# default value is hdfs://server/user/{user.name}/.nomic
nomic.hdfs.repository.dir = ${nomic.hdfs.home}"/.nomic"


##############################################################################
# HDFS configuration
##############################################################################

# What kind of adapter to HDFS will be used. The possible values are 'hdfs' and
↪'simulator'.
# Simulator just simulate HDFS on your FileSystem. It's useful for debuging etc..
hdfs.adapter = "hdfs"

# The directory where simulator adapter will store all files. Make sence to set it
```

```
# if you set adater to 'simulator'.
hdfs.simulator.basedir = ${nomic.home}"/hdfs"

# Point to Hadoop core configuration file. It's relevant for real 'hdfs' adapter.
hdfs.core.site = ${nomic.home}"/conf/core-site.xml"

# Point to Hadoop HDFS configuration file. It's relevant for real 'hdfs' adapter.
hdfs.hdfs.site = ${nomic.home}"/conf/hdfs-site.xml"



################################################################################
# HIVE configuration
################################################################################

# host for Hive JDBC
hive.host = "localhost:10000"

# this is JDBC connection string to HIVE
hive.jdbc.url = "jdbc:hive2://"${hive.host}

# The default HIVE schema
hive.schema = ${nomic.user}

# Username for HIVE connection
hive.user = ${nomic.user}

# Password for HIVE connection
hive.password = ""



################################################################################
# OOZIE configuration
################################################################################

# URL that point to oozie server where is Oozie REST API available. It's
# just hostname and port without `/oozie/v1` postfix. This postfix is handled
# by application itself
oozie.url = "https://localhost:11000"

# Job tracker URL that will be used as pre-filled value when you submitting
# Oozie job (coordinator)
oozie.jobTracker = "localhost:8032"
```

1.4 Nomic DSL Guide

In this guide I would like to explain main cocepts of Nomic DSL and you will find informations how to write `nomic.box` descriptor files. We're using Groovy as main language here and we created declarative DSL. The descriptors are declarative way how to tell application what can be deployed and removed. The core of descriptor file is collection of items called `Facts`. Each fact can be installed and reverted. The minimum descriptor script contains 3 required facts: the box name, the box group and version.

```
group = "app"
name = "some-box"
```

```
version = "1.0.0"
```

1.4.1 Variables in descriptor

In your descriptor scripts, you can also use some useful global variables:

- **group** box group can be any string. You can also structure groups with / character.
- **name** is box name that must be unique because it's used for identification
- **version** box version
- **user** username the nomic is using for installation/uploading files into HDFS configured in `nomic.conf`
- **homeDir** each used in HDFS might have his own home directory. It's usefull when you want to sandboxing your applications/analyses.
- **appDir** path to application directory in HDFS where are applications installed. The default value is `${homeDir}/app`
- **nameNode** the hostname of Name Node (it's value of `fs.defaultFS` parameter in your Hadoop configuration)

Also each module (Hive, Hdfs etc) can expose own parameters.

- **hiveSchema** contain default HIVE schema that is configure via `nomic.conf`. Also good if you want to sandboxing your apps.
- **hiveJdbcUrl** value of `hive.jdbc.url` in `nomic.conf` that is used by Hive facts.
- **hiveUser** value of `hive.user` in `nomic.conf` that is used by Hive facts.

1.4.2 Modules & dependencies

You can create an application box with multiple modules. This is useful especially for larger applications when you need to organize your content. There is also second use case of modules. Because facts inside the Box don't know nothing about dependencies, you can solve your dependency problem via modules as well.

Let's consider we've got our application 'best-analytics' with some resources and with `./nomic.box`:

```
group = "mycompany"
name = "best-analytics"
version = "1.0.0"

hdfs {
  ...
}
```

The box is build via command:

```
$ jar cf best-analytics.nomic ./*
```

Let's imagine we would like to split the content into two modules `kpi` and `rfm`. We will create a 2 new folders with own `nomic.box` they will represents our new modules.

The `./kpi/nomic.box`:

```
group = "mycompany"
name = "kpi"
version = "1.0.0"

...
```

and the `./rfm/nomic.box`:

```
group = "mycompany"
name = "rfm"
version = "1.0.0"

...
```

The final step is to declare these 2 new folders as modules in main `./nomic.box`:

```
group = "mycompany"
name = "best-analytics"
version = "1.0.0"

module 'kpi'
module 'rfm'
```

The `module` fact ensure the main application box will have 2 new dependencies they will be installed before any resource in main box. That means the installation install each module first and then the `best-analytics`. When we install this bundle, we should see 3 new modules:

```
$ ./bin/nomic install best-analytics.nomic
$ ./bin/nomic list
mycompany:best-analytics:1.0.0
mycompany:kpi:1.0.0
mycompany:rfm:1.0.0
```

Also removing of `best-analytics` will remove all modules in right order.

Sometimes we also need to tell that our `rfm` module depends on `kpi`. That can be achieved via `require` fact. Let's modify our `./rfm/nomic.box`:

```
group = "mycompany"
name = "rfm"
version = "1.0.0"

require name: "kpi", group: this.group, version: $this.version
```

Now the `rfm` module need `kpi` first what means the `kpi` module will be installed first.

1.4.3 Factions

Maybe you realized there is no way how to set order how facts are executed. The solution is *faction*. The Factions are small blocks/groups of facts. Each faction has own unique ID in box and might depend on another faction.

Let's imagine you want to ensure the resources first and then create some hive tables.

```
group = "mycompany"
name = "rfm"
version = "1.0.0"

```

```
faction ("resources") {
  resource 'file-1.csv'
  resource 'file-2.csv'
}

faction ("hivescripts", dependsOn = "resources") {
  table 'authors' from "create_authors_table.q"
}
```

Everything declared outside the faction blocks is considered as global facts and it's executed first. The factions are executed after all these global facts.

```
group = "mycompany"
name = "rfm"
version = "1.0.0"

faction ("resources") {
  resource 'file-2.csv'
}

resource "file-1.csv"
```

In this example, the `file-1.csv` fact will be applied first even it's declared after the faction.

1.4.4 Facts

Resource

The `resource` fact is declaring which resource from your box will be uploaded to where in HDFS. Let's imagine we've got box archive like:

```
/nomic.box
/some-file.xml
```

The descriptor below will install the `some-file.xml` into application's folder (depends how it's configured).

```
group = "app"
name = "some-box"
version = "1.0.0"

hdfs {
    resource 'some-file.xml'
}
```

With small modification you can place any resource to any path. E.g. following example will demonstrate how to place some file to root `/app`:

```
hdfs {
    resource 'some-file.xml' to '/app/workflow.xml'
}
```

If you don't place `/` character, the file will be paced into working directory that is basically `${appDir}`.

```
hdfs {
    resource 'some-file.xml' to 'workflows/some-workflow.xml'
}
```

The example above will ensure the file in `${appDir}/workflows/some-workflow.xml` where the `some-file.xml` content will be copied.

Also you can redefine the default working directory:

```
hdfs("/path/to/app") {
    resource 'some-file.xml'
}
```

This example above will install `some-file.xml` into `/path/to/app/some-file.xml`

As I mentioned, the facts are can be installed and uninstalled. In the `resource` case, uninstall means the file will be removed. Anyway you can mark file by setting property `keepIt` to `true` and uninstall will keep the file:

```
hdfs("/path/to/app") {
    resource 'some-file.xml' keepIt true
}
```

Dir

You can also declare presence of directory via `dir` fact. The declaration will create empty new directory if is not present yet.

```
hdfs {
    dir "data"
}
```

Because path start without `/` character, the directory will be created in current working directory. This declaration also ensure uninstalling that means the folder will be removed when uninstall or upgrade. If you wish to keep it, you can use the `keepIt` parameter:

```
hdfs {
    dir "data" keepIt true
}
```

Table

You can declare in descriptor also facts for HIVE. You can declare tables, schemes, you can also ensure the Hive scripts executions. Everything for Hive must be wrapped in `hive`.

Following example show how to create simple table in default schema you have configured in `nomic.conf`:

```
group = "app"
name = "some-box"
version = "1.0.0"

hive {
    table 'authors' from "create_authors_table.q"
}
```

In you box, you need to have the hive qurey file `create_authors_table.q` that will create table if it's not present in system:

```
CREATE EXTERNAL TABLE authors(
  NAME STRING,
  SURNAME STRING
```

```
)
STORED AS PARQUET
LOCATION '/data/authors';
```

In your hive scripts you can use placeholders they will be replaced with values from descriptor. Values are declared via `fields`. This is sometime usefull when you want e.g. place table into some schema.

```
hive {
    fields 'APP_DATA_DIR': "${appDir}/data", 'DATABASE_SCHEMA': defaultSchema
    table 'authors' from "create_authors_table.q"
}
```

The `create_authors_table.q` then use these placeholders:

```
CREATE EXTERNAL TABLE ${DATABASE_SCHEMA}.authors(
  NAME STRING,
  SURNAME STRING
)
STORED AS PARQUET
LOCATION '${APP_DATA_DIR}/authors';
```

Schema

This fact create Hive schema during installation and drop this schema during uninstall procedure. This fact is useful if you want to declare multiple schemas or if you don't want to rely on default schema.

```
hive {
   schema 'my_schema'
}
```

As I mentioned the example above will drop the schema during uninstall process that means also during upgrading. If you want to prevent this, you can mark schema with `keepIt`.

```
hive {
   schema 'my_schema' keepIt true
}
```

You can also declare schemas in `hive` block. In this case, the schema will be used as default schema across all facts inside hive block. Also you might have multiple blocks. The example below demonstrate more complex usage of schemas.

```
hive("${user}_${name}_staging") {
    table 'some_table' from 'some_script.q'
}

hive("${user}_${name}_processing") {
    fields 'DATABASE_SCHEMA': "${user}_${name}_processing"
    table 'some_table' from 'some_script.q'
}

hive("${user}_${name}_archive") {
    table 'some_table' from 'some_script.q'
}
```

This descriptor script will ensure 3 schemas where name of schema will be created as composition of user name, box name and some postfix. As you can see, each section might have own `fields` declaration.

Coordinator

The Nomic application is also integrate Oozie. You can declare the Oozie `coordinator` that is acting similar as `resource` but also submitting the coordinator with parameters. This fact also ensure the coordinator will be stoped during removing.

Let's assume we've got simple coordinator available as `coordinator.xml` in our Box. In description file we will declare:

```
group = "examples"
name = "oozieapp"
version = "1.0.0"

oozie {
    coordinator "coordinator.xml" parameters SOME_PARAMETER: "value 1", "another.
↪parameter": "value 2"
}
```

This example copy the XML into HDFS, into application folder and submit a coordinator job with given parameters like `SOME_PARAMETER` and also with following pre-filled parameters:

name	value
user.name	The user from Nomic configuration (e.g `me`)
nameNode	The nameNode URL (e.g. `hdfs://server:8020`)
jobTracker	Job tracker hostname from configuration with port (e.g. `server:8032`)
oozie.coord.application.path	Path to coordinator XML in HDFS (e.g. `/app/examples/oozieapp/coordinator.xml`)