Next generation Puppet code for Bigtop deployment

- Introduction
- Proof of concept implementation (ongoing)
    - jcbollinger says:
- A detailed use case scenario
    - Testing release candidates of HBase
        - rpelavin questions:
    - An external wizard/orchestrator use case
- Issues to be resolved
    - Collection of modules vs. collection of classes in a single Bigtop module
        - rpelavin says:
    - Picking an overall 'theme' or 'paradigm' for the design of our modules.
    - Parameterized classes vs. external lookups
        - jcbollinger says:
    - Source of external configuration: extlookup/hiera
        - rvs says:
        - rpelavin says:
        - jcbollinger says:
    - What is the best tool for modeling the data that characterize the configuration to be deployed
        - jcbollinger says:
        - rpelavin says:
        - jcbollinger says:
    - How to solve a 'macro substitution of the puppet code' issue (or whether we need to solve it at all)
        - jcbollinger says:
    - How much of the DevOps community would we lose if we focus on Puppet 3?
        - rpelavin says:

Introduction

Our current Bigtop deployment code shows its signs of age (it originated in pre 2.X puppet). That code currently serves as the main engine for us to dynamically deploy Bigtop clusters on EC2 for testing purposes. However, given that the Bigtop distro is the foundation for the commercial distros, we would like our Puppet code to be the go-to place for all the puppet-driven Hadoop deployment needs.

What we strive at building is a set of reusable/encapsulated/modular building blocks that can be arranged in any legal “hadoop toplogy”, as opposed to automating a particular deployment use case. Our goal is to offer capabilities, not to dictate policies.

Thus at the highest level, our Puppet code needs to be:

1. self-contained, to a point where we might end up forking somebody else's module(s) and maitaining it (subject to licensing)
2. useful for as many versions of Puppet as possible from the code (manifest) compatibility point of view. Obviously, we shouldn't obsess too much over something like Puppet 0.24, but we should keep the goal in mind. Also, we're NOT solving a problem of different versions of Puppet master and Puppet agents here.
3. useful in a classical puppet-master driven setup where one has access to modules, hiera/extlookup, etc all nicely setup and maitained under /etc /puppet
4. useful in a masterless mode so that things like Puppet/Whirr integration can be utilized: WHIRR-385 This is the usecase where the Puppet classes are guaranteed to be delivered to each node out of band and --modulepath will be given to puppet apply. Everything else (hiera/extlookup files, etc) is likely to require additional out-of-band communications that we would like to minimize.
5. useful in orchestration scenarios (Apache Ambari, Reactor8) although this could be viewed as a subset of the previous one. We need to be absolutely clear though that we are NOT proposing to use Puppet itself as an orchestration platform, what is being proposed is to treat puppet as a tool for predictably stamping out the state of each node **when** and **only** when instructed by a higher level orchestration engine. The same orchestration engine will also make sure to provide all the required data for the puppet classes. We are leaving it up to the orchestration implementation to either require a Puppet master or not. The important part is the temporal sequence in which either puppet agent or puppet apply get invoked on every node.

Proof of concept implementation (ongoing)

The next generation puppet implementation will be a pretty radical departure from what we currently have in Bigtop. It feels like a POC with a subset of Bigtop component will be a useful first step. HBase deployment is one such subset that, on one hand, is still small enough, but on the other hand would require puppet deployment for 2 supporting components:

1. Zookeeper
2. HDFS

before HBase itself gets fully bootstrapped. For those unfamiliar with the HBase architecture the following provides a good introduction: http://hbase.apache.org/book.html#distributed

Also, there's now a GitHub repository for the code to evolve: https://github.com/rvs/bigtop-puppet

jcbollinger says:

The initial draft of the zookeeper stuff looks pretty clean and reasonable. I have attached a handful of detail-level critiques to the GH commit. At a higher level, I am a bit surprised by the amount of data in your classes, given some of our earlier discussions about separating data from puppet code. For example, I don't see anything in class bigtop::params that could not easily be externalized. It may be too soon to make decisions about what should and shouldn't be externalized, and about how the external data should be organized, but that's something to keep an eye on.

A detailed use case scenario

Testing release candidates of HBase

A user Alice wants to test the release candidate of HBase against the stable Zookeeper and HDFS. Alice uses Jenkins job to build packages for the HBase release candidate and wants to deploy on EC2. Alice decides that she wants a single topology to be tested and she wants the underlying OS to be Ubuntu Lucid. She uses Whirr to dynamically spin 4 EC2 machines and she uses Whir-Puppet integration to deploy Zookeeper, HDFS and the latest build of HBase packages and configure them with suitable configuration. She invokes Whirr with the attached properties file and expects a fully functioning HBase cluster at the end of the Whirr execution:

```
$ cat puppet-hbase.properties

whirr.cluster-name=hbase
# We would like to have 1 node running all the services listed
# and 3 nodes running a different set of services
whirr.instance-templates=1 puppet:bigtop::kerberos::kdc+      \
                           puppet:bigtop::kerberos::client+   \
                           puppet:bigtop::zookeeper::server+  \
                           puppet:bigtop::hdfs::namenode+     \
                           puppet:bigtop::hdfs::datanode+     \
                           puppet:bigtop::hbase::master+      \
                           puppet:bigtop::hbase::regionserver \
                                                              \
                         3 puppet:bigtop::kerberos::client+   \
                           puppet:bigtop::zookeeper::server+  \
                           puppet:bigtop::hdfs::datanode+     \
                           puppet:bigtop::hbase::regionserver \

# Where to fetch the module code from
puppet.bigtop.module=git://github.com/rvs/bigtop-puppet.git

# this parameter is applicable to the entire cluster
bigtop.auth=kerberos
# this one only concerns hbase master service
bigtop.hbase.master.heap_size=1024
# this one is HDFS specific (namenode and datanode services both will get it)
bigtop.hdfs.dirs=["/mnt"]

$ whirr launch-cluster --config puppet-hbase.properties
```

rpelavin questions:

Roman, is the whirr description something that works with current Puppet manifests or is a description of what you would like to work?

Also, if hiera is used, are you looking to get as close to one-to-one parameter mapping between parameters here (e.g., bigtop.auth, bigtop.hdfs.dirs, bigtop.hbase.master.heap_size) and 'hiera parameters'?

An external wizard/orchestrator use case

Another related perspective is if you had a wizard deployment tool that answered a few key questions to determine the characteristics of your hadoop deployment (i.e., what services are you running, the resiliency, whether secure mode, etc) you want to make it as easy as possible to build a mechanism that generate a site manifest that would then drive the Puppet automated configuration.

Issues to be resolved

Collection of modules vs. collection of classes in a single Bigtop module

A typical puppet module strives for the UNIX mantra of 'doing one thing and doing it well'. So it seems natural for Bigtop Puppet code to be split into independent modules each corresponding to a single Bigtop component. After all, that how the current Bigtop Puppet code is organized – into a collection of modules. Yet, I'd like to argue that given the level of coupling between different component of Bigtop we might as well be honest and admit that there's no way you can use them
in an independent fashion and you may as well create a single module called Bigtop with a collection of tightly coupled classes.

This also has a nice added benefit of simplifying the eventual publishing of our code on Puppet Forge – it is way easier to manage the versioning of a single module as opposed to multiple one.

rpelavin says:

If you have a single module it is going to get very large, much larger than any modules I have seen. I think it is useful to have multiple modules as away of organizing the classes much in the same way in a PostgreSQL database, rather than having many tables in one schema you can use multiple schemas to break up the tables into more manageable chunks. Now, as noted having multiple modules will have an added overhead of coordinating the versioning of the modules, but this may just give you more flexibility; you can always fallback on updating in lockstep. Also, someone developing related modules may just want to get certain parts of the Bigtop modules; they might just want hdfs/yarn, not anything else or just want zookeeper, etc

Picking an overall 'theme' or 'paradigm' for the design of our modules.

The Node->Role->Profile->Module->Resource approach articulated here http://www.craigdunn.org/2012/05/239/ looks pretty promising.

Parameterized classes vs. external lookups

Which style is prefered:

```
class { "bigtop::hdfs::secondarynamenode":
   hdfs_namenode_uri => "hdfs://nn.cluster.company.com:8020/"
}
```

vs.

```
   # all the data required for the class gets
   # to us via external lookups
   include bigtop::hdfs::secondarynamenode
```

jcbollinger says:

If you are going to rely on hiera or another external source for all class data, then you absolutely should use the 'include' form, not the parametrized form. The latter carries the constraint that it can be used only once for any given class, and that use must be the first one parsed. There are very good reasons, however, why sometimes you would like to declare a given class in more than one place. You can work around any need to do so with enough effort and code, but that generally makes your manifest set more brittle, and / or puts substantially greater demands on your ENC. I see from your subsequent comment (elided) that you recognize that, at least to some degree.

Source of external configuration: extlookup/hiera

Should we rely on extlookup or hiera? Should we somehow push that decision onto the consumer of our Puppet code so that they can use either one?

rvs says:

My biggest fear of embracing hiera 100% is compatibility concerns with older Pupppets

rpelavin says:

This is clunky compared to syntax in 3.x, but flexible. Why not introduce a level of indirection using a custom function like shown below that can be overwritten to bind to hiera or some other external source that lookups the passed qualified variable

```
class foo(
  $param1 = bigtop_lookup('foo::param1','default-val'),
...
```

jcbollinger says:

A level of indirection would be useful if you want to minimize manifest modifications in the event that you switch data binding mechanisms. Is that a likely eventuality? Hiera can be used with Puppet as old as v2.6, and it doesn't look likely to be pulled from any foreseeable future Puppet. On the cost side, an indirection layer either provides a least-common-denominator feature set (notably, hiera's `hiera_array()`, `hiera_hash()`, and `hiera_include()` functions are not covered by the example), or else ties the indirection layer tightly enough to a particular data provider that switching providers is difficult or even impossible in practice. (`extlookup()`, for example, has no associated analog of the additional hiera functions I just mentioned, and can supply only string values). Much depends on the nature of the data binding features you want/need to use. As a special case, if you end up relying on Puppet 3's automatic class parameter binding, then you might as well embrace Hiera all the way.

What is the best tool for modeling the data that characterize the configuration to be deployed

It seems like class parameters is an obvious choice here

jcbollinger says:

You are supposing that these will be modeled as class parameters in the first place. Certainly they are data that characterize the configuration to be deployed, and it is possible to model them as class parameters, but that is not the only – and maybe not the best – alternative available to you. Class parametrization is a protocol for declaring and documenting that data on which a class relies, and it enables mechanisms for obtaining that data that non-parametrized classes cannot use, but the same configuration objectives can be accomplished without them.

rpelavin says:

Seems like class parametrization should be a prime method, but allow some exceptions. One exception may be to handle what I would call "fine tuning parameters" to affect the configuration variables that may be in, for example, the hdfs-site, mapred-site . config files. If relied solely on class paramterization issue would be you may have classes with 30 - 50 parameters which is very cumbersome. Instead you might treat these with "global variables". Also related is keeping in mind easing the process of iterating on modules and adding new parameters; some of the signatures seem very set while other will evolve more rapidly.

jcbollinger says:

The implications of class parameterization should be understood before committing to that approach, and those implications depend in part on which Puppet versions will be supported. I have historically been very negative about using parameterized classes with Puppet 2.x, but I am fine with parameterized classes themselves in Puppet 3, as long as hiera or maybe an ENC is used for data binding. Parameterized-style class declarations (i.e. class { 'foo': param1 => 'bar' }) in Puppet 3 DSL still have all the problems that I took issue with in Puppet 2. This is a pernicious design issue, because although parameterized-style class declarations seem appealing and natural on the surface, they are in fact fundamentally inconsistent with Puppet's overall design. The inconsistencies play out in a variety of troublesome ways. Nevertheless, people certainly do use parameterized classes successfully, including in Puppet 2. If that is the direction chosen, however, then you will want to adopt appropriate coding practices to avoid the associated issues.

How to solve a 'macro substitution of the puppet code' issue (or whether we need to solve it at all)

Current Bigtop code uses quite a bit of dynamic lookups to solve a problem of multiple classes creating a pretty rich state in class-local variables and then calling onto a common piece of code to instantiate configuration files based on that information. Here's an example:

```
class one_of_many {
    if ($::this == 'that') {
       $var1 = 'something'
    } else {
        ....
   }

   include common_code
}
```

With older Puppets common_code class would have access to all the $varN variables simply by the virtue of doing dynamic scope lookups. With newer puppets the only alternative seems to be explicit ennumeration of all the variables that are required for common_code to perform its task. E.g.

```
class common_code($var1, ... $varN) {
}
....
   class { "common_code":
       var1 => $var1,
       ....
       varN => $var1
   }
```

jcbollinger says:

With newer Puppet, the key is that each class needs to know from where, proximally, the data it consumes come (and coding to that principle is good practice in *every* version of Puppet). Class parameters are probably the most obvious proximal source on which a class can rely for data (in 2.6+), but classes can also rely on class variables of other classes by referring to them by fully-qualified name, and it is discussed above how classes can obtain shared external data via an appropriate function, such as `hiera()` or `extlookup()`. To the extent that the current codebase appears generally to declare each class in just one place, the most direct accommodation for Puppet3 (and good practices generally) would probably be to just qualify all the variable references. Indeed, that was typical advice to anyone preparing to move from Puppet 2.x to Puppet 3. That is not meant to discount the opportunity to perform a deeper redesign, however.

How much of the DevOps community would we lose if we focus on Puppet 3?

Puppet 3 provides quite a bit of enchantments in precisely the areas where we need it (Hiera, etc.). The question is, if we focus on Puppet 3 too much, would we alienate tons of DevOps community and make our Puppet code less useful to them?

rpelavin says:

Answer to this depends on understanding in more detail what you want to deliver and context under which this system is being used.

First, is the "thing you are delivering" a system that builds and deploys Hadoop stacks or are you delivering (in addition) a set of reusable Hadoop stack modules.

If former is the goal, then the following is applicable: If the Puppet modules are being viewed as the implementation to provide higher-level functions to build and deploy Hadoop stacks then I think you can make a choice and use the Puppet version/features that makes it easiest for you to design the best and most modular components (Puppet 3 being the best choice I believe). If, on the other hand, the user of the system will want to integrate, for example, Puppet modules that build full applications that use Hadoop stack services, then issue of what Puppet version/features used becomes important. Related is whether the view of Bigtop-Puppet is that of a top level system hiding and controlling Puppet infrastructure or instead that of a solution where these modules are built such they can run on the Puppet environment currently used by the end user.

I think a good analogy is that to external or embedded databases. Are you looking towards a solution where Puppet is viewed as the "internal configuration management" system or conversely are you looking to Puppet as an "external configuration management system", one that is maintained to some extent by a Puppet expert/administrator (the analogy being to a DB admin)