DevOps in Practice

Reliable and automated software delivery

Code Crushing

DANILO SATO

Code Crushing
Books and programming
Rua Vergueiro, 3185 - 8º andar
04101-300 – Vila Mariana – São Paulo – SP – Brasil

Chapter 5

Puppet beyond the basics

By the end of chapter 4, the Puppet code to configure the online store's infrastructure is not very well organized. The only separation that we did was creating two manifest files, one for each server: `web` and `db`. However, in each file, the code is simply a long list of resources. In addition, the configuration and template files are written without any structure. In this chapter we will learn new Puppet concepts and features while refactoring the online store infrastructure code to make it more idiomatic and better factored.

5.1 Classes and defined types

Puppet manages a single instance of each resource defined in a manifest, making them similar to a *singleton*. Likewise, a **class** is a collection of singleton resources in the system. If you know object-oriented languages, do not get confused with this terminology. A Puppet class cannot be instantiated multi-

ple times. Classes are just a way of giving a name to a collection of resources that will be applied as a unit.

A good use case for Puppet classes is when you are configuring services that you need to install in the system only once. For example, in the `db.pp` file we install and setup MySQL server and then create the application-specific user and schema for the online store. In a real scenario, we could have several schemas and users in the same underlying database, but we do not install MySQL multiple times in the same system. MySQL server is a good candidate to be setup in a generic class, which we will call `mysql-server`. Refactoring our `db.pp` file to declare and use this new class, we will have:

```
class mysql-server {
  exec    { "apt-update": ... }
  package { "mysql-server": ... }
  file    { "/etc/mysql/conf.d/allow_external.cnf": ... }
  service { "mysql": ... }
  exec    { "remove-anonymous-user": ... }
}

include mysql-server

exec { "store-schema": ...,
  require => Class["mysql-server"],
}
exec { "store-user": ... }
```

Notice that we moved the `Exec["remove-anonymous-user"]` resource – that revokes access to the anonymous user – into the `mysql-server` class because this is something that should happen only once when MySQL server is installed. Another change you may notice is that the `Exec["store-schema"]` resource now depends on `Class["mysql-server"]` instead of the previous `Service["mysql"]` resource. This is a way to encapsulate implementation details within a class, isolating this knowledge from the rest of the code, which can declare dependencies on something more abstract and stable.

To define a new class, just choose its name and define all of its resources in a `class <class name> { ... }` declaration. To use a class, you can

use the `include` syntax or a version that's more similar to how we have been defining other resources: `class { "<class name>": ... }`.

On the other hand, the resources to create the online store schema and user can be reused to create schemas and users for other applications running in the same database server. Putting them in a class would be the wrong way to encapsulate them, because Puppet would force them to be singletons. For situations like this, Puppet has another form of encapsulation known as "defined types."

A **defined type** is a collection of resources that can be used multiple times in the same manifest. They help you eliminate duplication by grouping related resources that can be reused together. You can think of them as the equivalent to macros in a programming language. Moreover, they can be parameterized and define default values for optional parameters. Grouping the two `exec` resources that create the database schema and user in a defined type called `mysql-db`, we will have:

```
class mysql-server { ... }

define mysql-db($schema, $user = $title, $password) {
  Class['mysql-server'] -> Mysql-db[$title]

  exec { "$title-schema":
    unless  => "mysql -uroot $schema",
    command => "mysqladmin -uroot create $schema",
    path    => "/usr/bin/",
  }

  exec { "$title-user":
    unless  => "mysql -u$user -p$password $schema",
    command => "mysql -uroot -e \"GRANT ALL PRIVILEGES ON \
                                  $schema.* TO '$user'@'%' \
                                  IDENTIFIED BY '$password';\"",
    path    => "/usr/bin/",
    require => Exec["$title-schema"],
  }
}
```

```
include mysql-server

mysql-db { "store":
  schema   => "store_schema",
  password => "storesecret",
}
```

The syntax to declare a defined type is `define <defined type name>(<parameters>) { ... }`. In our example, the `mysql-db` defined type accepts three parameters: `$schema`, `$user` and `$password`. The `$user` parameter, unless otherwise specified, will take the default value of the special `$title` parameter. In order to understand the value of `$title` parameter – which does not need to be explicitly declared – just look at the syntax used when instantiating a defined type: `mysql-db { "store": schema => "store_schema", ... }`. The resource name that instantiates the defined type, in this case `store`, will be passed as the value for the `$title` parameter. The other parameters are passed following the same syntax used in other native Puppet resources: the parameter name and its value separated by an arrow `=>`.

We moved the two `exec` resources that create the schema and the user to a defined type. We also replaced all the hard-coded references with the respective parameters, paying attention to use double quotes in strings so Puppet can expand the values correctly. In order to use the defined type more than once, we need to parameterize the names of the `exec` resources to make them unique, using the `$title` parameter. The last update was to promote the dependency with the `mysql-server` class to the top of the defined type. With that, the individual resources inside the defined type don't have to declare any dependencies with external resources, making our code easier to maintain in the future.

Using classes and defined types, we managed to refactor our Puppet code to make it more reusable. However, we have not yet changed how the files are organized. Everything continues to be declared in a single file. We need to learn a better way to organize our files.

5.2 Using modules for packaging and distribution

Puppet has a standard for packaging and structuring your code: They are called **modules.** Modules define a standard directory structure in which you should place your files, as well as some naming conventions. Modules are also a way to share Puppet code with the community. The Puppet Forge (http://forge.puppetlabs.com/) is a website maintained by Puppet Labs where you can find many modules written by the community, or you can register and share a module you wrote.

Depending on your experience with different languages, the equivalent of a Puppet module in Ruby, Java, Python and .NET would be a *gem*, a *jar*, an *egg* and a *DLL*, respectively. The simplified directory structure for a Puppet module is:

```
<module name>/
 files
    ...
 manifests
    init.pp
 templates
    ...
 tests
     init.pp
```

First of all, the name of the root directory defines the module name. The most important directory is the *manifests* directory, because it is where you place your manifest files (with a `.pp` extension). Inside it, there must be at least one file called `init.pp`, which is loaded as the entry point for the module. The *files* directory contains static configuration files that can be used by a `file` resource in a manifest using a special URL: `puppet:///modules/<module name>/<file>`. The *templates* directory contains ERB files that can be referenced in a manifest using the module name: `template('<module name>/<ERB file>')`.

Finally, the *tests* directory contains examples of how to use the Classes, as well as defined types exposed by the module. These tests do not per-