Chapter 4. Configuration Management

Why Use Configuration Management?

Mike originally became enamored (*obsessed* might be a better word) with the idea of automatically configuring and deploying infrastructures after reading the classic “Bootstrapping an Infrastructure” paper from the LISA ’98 system administration conference.

The paper described the experiences of systems administrators and architects responsible for building and maintaining large Unix environments for financial trading floors, and outlined some philosophies they adopted as a result. While somewhat dated in terms of the software used, the underlying principles are still highly relevant to managing today’s cloud infrastructures:

We recognize that there really is no “standard” way to assemble or manage large infrastructures of UNIX machines. While the components that make up a typical infrastructure are generally well-known, professional infrastructure architects tend to use those components in radically different ways to accomplish the same ends. In the process, we usually write a great deal of code to glue those components together, duplicating each others’ work in incompatible ways. Because infrastructures are usually ad hoc, setting up a new infrastructure or attempting to harness an existing unruly infrastructure can be bewildering for new sysadmins. The sequence of steps needed to develop a comprehensive infrastructure is relatively straightforward, but the discovery of that sequence can be time-consuming and fraught with error. Moreover, mistakes made in the early stages of setup or migration can be difficult to remove for the lifetime of the infrastructure.

The authors of this paper recognized that automation is the key to effective system administration, and that a huge amount of time was being wasted by duplicating efforts to automate common tasks like installing software packages. By describing some battle-tested experiences, they hoped to reduce the amount of redundant work performed by their peers.

Configuration management software evolved out of a wider need to address this problem. Puppet, Chef, and Ansible are just a few examples of configuration management packages. They provide a framework for describing your application/server configuration in a text-based format. Instead of manually installing Apache on each of your web servers, you can write a configuration file that says, “All web servers must have Apache installed.”

As well as reducing manual labor, storing your configurations as text files means you can store them in a source-control system with your application code so that changes can be audited and reverted.

Your infrastructure effectively becomes *self-documenting*, as your server and application configurations can be reviewed at any time by browsing your source-control system. Of course, bits only ever answers how something is done, therefore authors must take care to document the why in their commit message.

Finally, the entire unit is a self-contained unit that can be deployed with minimal human interaction. Once the first server running the configuration management application is running, the rest of the infrastructure can be brought into service without having to manually configure operating systems or applications.

This is especially important when a small team is responsible for a large infrastructure, or in the case of consulting companies, a number of disparate infrastructures. Manually installing software does not scale up very well—if it takes you one hour to configure a server manually, it will take you two hours to configure two servers, to the extent that you are not parallel.

However, if it takes one hour to configure a server with configuration management software, that configuration can be reused for as many servers as you need.

Adopting the configuration management philosophy does involve an initial time investment if you have not used it before, but it will soon pay off by reducing the amount of time you spend configuring servers and deploying changes.

OpsWorks

Amazon recognizes how important configuration management tools are, and is doing its bit to make these tools more effective when working within AWS. In February 2013, they announced the OpsWorks service, bringing joy to the hearts of sysadmins everywhere.

OpsWorks made configuration management a core part of AWS, bringing support for Chef directly into the Management Console. It works by letting the users define the *layers* that make up their application—for example, clusters of web and database servers would be two separate layers. These layers consist of EC2 instances (or other AWS resources) that have been configured using Chef recipes. Once your layers have been defined, AWS will take care of provisioning all the required resources.

Your running application—and all of its layers—are visible in the Management Console, making it easy to see the overall health of your application.

This makes it a lot easier for people who are familiar with Chef to use it to manage their AWS infrastructure. More importantly, it makes configuration management tools a lot more discoverable for companies that do not have dedicated system administrators. A lot of people avoid implementing configuration management because the return on investment is not always clear in advance. OpsWorks will hopefully lead to a lot more people using professional system administration practices when setting up AWS applications.

Another advantage of OpsWorks is that it further commoditizes many parts of designing and deploying an application infrastructure. It is possible to find shared Chef recipes for installing common software packages such as PostgreSQL or HA-Proxy. Instead of manually designing your own database setup and writing custom Chef recipes, you can just take a working example and tweak it if necessary.

Over time, AWS may build on this platform to offer entire “off-the-shelf” application stacks—for example, a “social media stack” that contains all the elements required to run a typical social media website, such as web servers, a database cluster, and caching servers.

Amazon maintains a choice of multiple overlapping services for template-based application provisioning. OpsWorks is the best suited to users who have already adopted Chef as their configuration management tool of choice and therefore do not feel limited by this choice. Outside of this group, OpsWorks is best thought of as a service abstracting the creation of web application stacks to a lesser degree than the easy to use AWS Elastic Beanstalk. OpsWorks provides more customization of application instances at the cost of potentially higher complexity, thanks to the integration of Chef. The authors prefer a third application template option for most automation tasks: AWS CloudFormation. CloudFormation is a lower-level foundation service that delivers complete control of the application template to the administrator, allowing us to tailor every detail. The effort caused by the additional complexity is offset by the need to operate only one tool, and our use of it through API automation, which makes this a setup-time cost only.

For more information about OpsWorks, see Amazon’s OpsWorks page.

Choosing a Configuration Management Package

A plethora of tools are available in the configuration management software space. We believe the number of options is a result of the fact that many of their users are system administrators who can code. When a tool does not quite meet the users’ needs, they begin working on their own version to scratch their individual itch, as eloquently illustrated by XKCD’s (Figure 4-1).

The top-tier tools all have their own approaches and architecture decisions, but share a common fundamental set of features. For example, they all provide the capability to install software packages and ensure services are running, or to create files on client hosts (such as Apache configuration files).

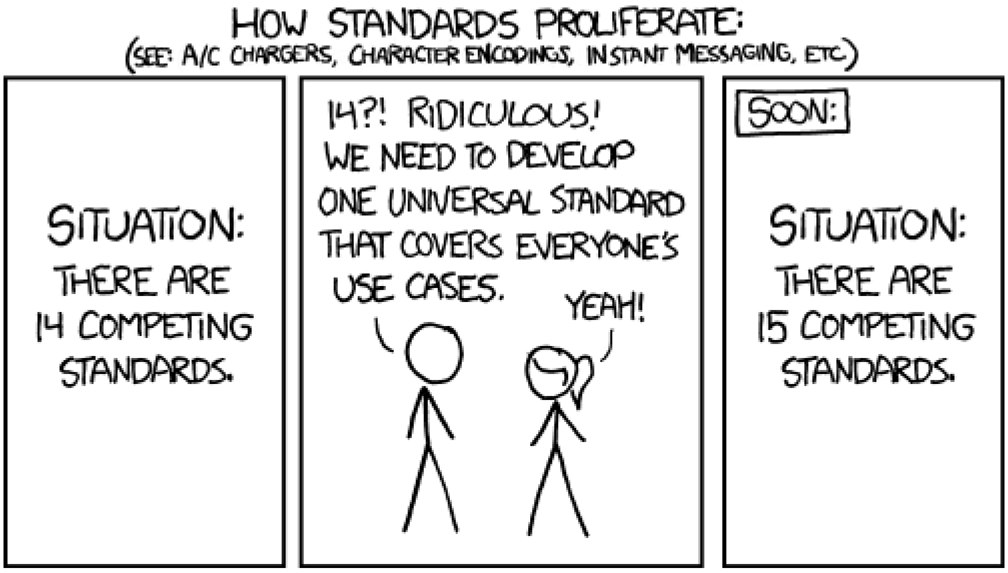
HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

Figure 4-1. XKCD standards, courtesy of Randall Munroe

There are a few things to keep in mind when choosing a package.

Nearly all of the available configuration management tools have the concept of reusable pieces of code that can be shared to save even more time. Instead of doing all of the work yourself, you can take some prewritten modules and customize them to fit your needs. In Puppet, these are known as modules, Chef calls them recipes, and Ansible calls them playbooks.

Puppet, Inc operates the Puppet Forge, a site where users from the Community can share Puppet modules written for common tasks.

The availability of external modules should factor into your decision; building on the work of others will usually be easier than building from scratch.

The language used to build the tool might also come into play, if you anticipate needing to extend it in some way (such as creating your own managed resource types). Chef and Puppet are both written in Ruby, whereas Ansible is a Python tool.

Most of the mature tools come from the pre-cloud days, and have evolved to support the needs of a more dynamic infrastructure. Chef and Puppet in particular have very good integration with AWS.

Given the relative similarity of features, choosing the right tool is often merely a case of finding the one that you feel most comfortable using. Amazon’s choice of Chef as the backend to OpsWorks will add to Chef’s popularity in the future, but that does not necessarily mean it is the right tool for everyone.

Our recommendation, especially if you are new to configuration management, is to try out a few packages and see which suits your team’s workflow.

Puppet on AWS

Instead of dedicating half of the book to configuration management tools, we use Puppet to demonstrate the key concepts in the rest of this chapter. It has a good amount of overlap with other tools in terms of the available features, so all of the core principles can be applied with any of the available configuration management packages.

A Quick Introduction to Puppet

Initially launched by Luke Kanies in 2005, Puppet is perhaps the most popular Open Source configuration management tool. It uses a declarative language to let users describe the configuration and state of Unix or Windows hosts in text files known as Puppet *manifests*. These manifests describe the desired state of the system—*this* package should be installed, and *this* service should be running.

Typically, these manifests are stored on a central server known as the Puppet *master*. Client hosts periodically connect to the master server and describe their current system state. The Puppet master calculates the changes required to move from the current state to the desired state, as described in the manifests for that host, and lets the client know which changes it needs to make. Finally, the Puppet client performs these actions.

Because clients connect to the master server at regular intervals, configuration changes can be made on the master server and they propagate throughout your network as each client connects and picks up the new configuration.

The */etc/puppet/manifests/sites.pp* file is used to map server hostnames to configuration manifests, which are known as *node definitions*. The best way to illustrate this is with an example, which contains two node definitions:

```
# demo sites.pp with two nodes

node "www.example.com" {
    package { "nginx":
        ensure => installed
    }
}

node "db.example.com" {
    package { "postgresql":
        ensure => installed
    }
}
```

When a client named *www.example.com* connects to the Puppet master, the Nginx package will be installed. When *db.example.com* requests its configuration from the Puppet master, the PostgreSQL package will be installed.

In addition to matching explicit hostnames, regular expressions can be used in node definitions: `www-\d+\.example\.com` would match*www-01.example.com* and *www-999.example.com*.

Puppet supports a module-based system for creating reusable Puppet manifests. Consider the common example of needing to ensure that user accounts are automatically managed by Puppet across all of your servers. Rather than explicitly listing all of your users over and over again, they can be listed once in a `users` module that is reused in multiple node definitions.

Modules are Puppet manifests that exist in a particular directory, usually */etc/puppet/modules*. A simple `users` module, stored in */etc/puppet/modules/users/manifests/init.pp*, might look like this:

```
user { 'mike':
  ensure => present
  }

user { 'federico':
  ensure => present
  }
```

This is applied to nodes by including it in their node definition, for example:

```
node 'www.example.com' {
  include users
  package { 'nginx':
  ensure => installed
  }
}
node 'db.example.com' {
  include users
  package { 'postgresql':
  ensure => installed
  }
}
```

This would ensure that users named `mike` and `federico` are created on both *www.example.com* and *db.example.com*.

Tip

Puppet offers multiple ways to test a module locally during development, without having to wait for the server (known as the *master* in Puppet lingo). One of the simplest is to manually apply a manifest matching the current node locally:

```
puppet apply manifest.pp
```

The client's parser can also validate syntax without actually executing any configuration changes:

```
puppet parser validate module.pp
```

Additionally, a very conventient validation tool called `puppet-lint` is available. `puppet-lint` will go beyond reporting errors and will also deliver style warnings and flag potential inconsistencies. Found in Ubuntu's universe repository, it is installed thus:

```
sudo apt install puppet-lint
```

One last option is to cause the client to immediately contact the server and apply any outstanding action. While more involved a strategy and time consuming, this is the "all up" test that should provide final validation to your manifests:

```
sudo puppet agent --test
```

Another way of achieving this is to use the default node definition, which applies to every client that connects to the Puppet master. This configuration in the default node is applied to clients that do not have a node definition explicitly assigned to them.

To see Puppet in action, we will launch two EC2 instances—one to act as the master, the other as a client. The initial Puppet manifest will simply install the Nginx web server package and make sure it is running—something akin to a "Hello, World" for Puppet.

Example 4-1 shows a CloudFormation stack that will create the two EC2 instances, as well as two security groups. These are required to access both instances with SSH, and to allow the Puppet client to contact the master on TCP port 8140.

Example 4-1. Puppet master and client CloudFormation stack

```
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Description": "Example Puppet master and client stack (manual install)",

    "Parameters" : {
        "KeyName": {
            "Description" : "EC2 KeyPair name",
            "Type": "String",
            "MinLength": "1",
            "MaxLength": "255",
            "AllowedPattern" : "[\\x20-\\x7E]*",
            "ConstraintDescription" : "can contain only ASCII characters."
        },
        "AMI" : {
            "Description" : "AMI ID",
            "Type": "String"
        }
    },

    "Resources": {
        "PuppetClientGroup": {
            "Type": "AWS::EC2::SecurityGroup",
            "Properties": {
                "SecurityGroupIngress": [
                    {
                        "ToPort": "22",
                        "IpProtocol": "tcp",
                        "CidrIp": "0.0.0.0/0",
                        "FromPort": "22"
                    }
                ],
                "GroupDescription": "Group for Puppet clients"
            }
        },
        "PuppetMasterGroup": {
            "Type": "AWS::EC2::SecurityGroup",
            "Properties": {
                "SecurityGroupIngress": [
                    {
                        "ToPort": "8140",
                        "IpProtocol": "tcp",
                        "SourceSecurityGroupName" : { "Ref" :
"PuppetClientGroup" },
                        "FromPort": "8140"
                    },
                    {
                        "ToPort": "22",
                        "IpProtocol": "tcp",
                        "CidrIp": "0.0.0.0/0",
                        "FromPort": "22"
                    }
                ],
                "GroupDescription": "Group for Puppet master"
            }
        },
        "PuppetMasterInstance": {
            "Type": "AWS::EC2::Instance",
            "Properties": {
                "ImageId" : { "Ref" : "AMI"},
                "KeyName" : { "Ref" : "KeyName" },
                "SecurityGroups": [
                    {
                        "Ref": "PuppetMasterGroup"
                    }
                ],
                "InstanceType": "t2.micro"
            }
        },
        "PuppetClientInstance": {
            "Type": "AWS::EC2::Instance",
            "Properties": {
                "ImageId" : { "Ref" : "AMI"},
                "KeyName" : { "Ref" : "KeyName" },
                "SecurityGroups": [
                    {
                        "Ref": "PuppetClientGroup"
                    }
                ],
                "InstanceType": "t2.micro",
                "UserData": {
                    "Fn::Base64": {
                        "Fn::GetAtt": [ "PuppetMasterInstance", "PrivateDnsName"
]
                    }
                }
            }
        }
    },
    "Outputs" : {
        "PuppetMasterIP" : {
          "Description" : "Public IP of the Puppet master instance",
          "Value" : { "Fn::GetAtt" : [ "PuppetMasterInstance", "PublicIp" ] }
        },
        "PuppetClientIP" : {
          "Description" : "Public IP of the Puppet client instance",
          "Value" : { "Fn::GetAtt" : [ "PuppetClientInstance", "PublicIp" ] }
        },
        "PuppetMasterPrivateDNS" : {
          "Description" : "Private DNS of the Puppet master instance",
          "Value" : { "Fn::GetAtt" : [ "PuppetMasterInstance",
"PrivateDnsName" ] }
        },
        "PuppetClientPrivateDNS" : {
          "Description" : "Private DNS of the Puppet client instance",
          "Value" : { "Fn::GetAtt" : [ "PuppetMasterInstance",
"PrivateDnsName" ] }
        }
    }
}
```

Save this stack template to a file named *puppet-stack.json*. Next, create the stack with the AWS command-line tools, remembering to replace the `KeyName` parameter to match your own key's:

```
aws cloudformation create-stack --region us-east-1 --stack-name puppet-stack \
    --template-body file://puppet-stack.json \
    --parameters ParameterKey=AMI,ParameterValue=ami-c80b0aa2 \
    ParameterKey=KeyName,ParameterValue=federico
```

Once the stack has been created, list the stack resources to find out the IP address and DNS names of the two instances:

```
$ aws cloudformation describe-stacks --stack-name puppet-stack | jq
'.Stacks[0].Outputs[]'
{
  "OutputValue": "54.166.216.73",
  "OutputKey": "PuppetMasterIP",
  "Description": "Public IP of the Puppet master instance"
}
{
  "OutputValue": "54.175.109.196",
  "OutputKey": "PuppetClientIP",
  "Description": "Public IP of the Puppet client instance"
}
{
  "OutputValue": "ip-172-31-50-245.ec2.internal",
  "OutputKey": "PuppetClientPrivateDNS",
  "Description": "Private DNS of the Puppet client instance"
}
{
  "OutputValue": "ip-172-31-62-113.ec2.internal",
  "OutputKey": "PuppetMasterPrivateDNS",
  "Description": "Private DNS of the Puppet master instance"
}
```

Now we need to install the Puppet master and create the manifests that will install Nginx on the client and make sure it is running.

Log in to the Puppet master host with SSH and install the Puppet master package—this will also install the Puppet client package:

```
sudo apt update
sudo apt install --yes puppetmaster
sudo service puppetmaster status
```

With the Puppet master installed, we can begin configuration. The following is a simple example of a *site.pp* file, which should be saved to */etc/puppet/manifests/site.pp*.

```
node default {
  package { 'nginx':
  ensure => installed
  }
  service { 'nginx':
  ensure => running,
  require=> Package['nginx']
  }
  file { '/tmp/hello_world':
  ensure => present,
  content=> 'Hello, World!'
  }
}
```

This *site.pp* file uses the default node definition, so it will be applied to any client that connects to the server. It will install Nginx and create a text file.

Now we can move on to the client. Connect to the client instance with SSH and install the Puppet client package:

```
sudo apt update
sudo apt install --yes puppet
```

Once installed, Puppet will run every 30 minutes by default. Unfortunately, this will not work immediately—usually your Puppet master will have a more friendly DNS name such as *puppet.example.com*. Because we have not yet set up DNS for the Puppet master, we must use its AWS-supplied DNS name. We further restrict access to the Puppet master by limiting client access exclusively to its private DNS and corresponding IP address. Security concerns dictates this policy, which we implement through dedicated security groups.

Puppet uses a key-based system for security. This provides two levels of protection: it ensures that communications between the master and clients are encrypted, and it also makes sure that only authorized clients can connect to the Puppet master and retrieve the configuration.

When a Puppet client first connects to the master, it will create a *key signing request* on the Puppet master. An administrator must authorize this request by running `puppet sign <hostname>`, which signs the key and confirms that the client is allowed to connect and retrieve its manifests file.

On the client, we configure the Puppet client by editing the file */etc/puppet/puppet.cong* to point to the master. Append the following line to the `[main]` section:

```
server = internal DNS name of master
```

Now intialize the client by executing the following command:

```
$ sudo puppet agent --waitforcert 120 --test
Info: Creating a new SSL key for ip-172-31-50-245.ec2.internal
Info: Caching certificate for ca
Info: csr_attributes file loading from /home/ubuntu/.puppet/csr_attributes.yaml
Info: Creating a new SSL certificate request for ip-172-31-50-245.ec2.internal
Info: Certificate Request fingerprint (SHA256):
85:50:1D:FB:94:0F:50:0B:8B:3B:5E:20:70:B9:7C:62:87:D9:89:76:85:90:70:79:AA:42:99
:A1:CA:E9:19:77
Info: Caching certificate for ca
```

This command tells Puppet that it should wait up to 120 seconds for the key to be signed on the master.

On the master, immediately list the waiting requests with this command:

```
$ sudo puppet cert list
  "ip-172-31-50-245.ec2.internal" (SHA256)
85:50:1D:FB:94:0F:50:0B:8B:3B:5E:20:70:B9:7C:62:87:D9:89:76:85:90:70:79:AA:42:99
:A1:CA:E9:19:77
```

Sign the request, taking care to update the client’s hostname to match that listed:

```
$ sudo puppet cert sign ip-172-31-50-245.ec2.internal
Notice: Signed certificate request for ip-172-31-50-245.ec2.internal
Notice: Removing file Puppet::SSL::CertificateRequest ip-172-31-50-
245.ec2.internal at '/var/lib/puppet/ssl/ca/requests/ip-172-31-50-
245.ec2.internal.pem'
```

Once you sign the request on the master, Puppet is nearly ready to do its job. Enable the client and make it spring into action and begin applying the configuration:

```
$ sudo puppet agent --enable
$ sudo puppet agent --test
Info: Caching certificate_revocation_list for ca
Info: Retrieving plugin
Info: Caching catalog for ip-172-31-50-245.ec2.internal
Info: Applying configuration version '1471818168'
Notice: /Stage[main]/Main/Node[default]/Package[nginx]/ensure: ensure changed
'purged' to 'present'
Notice: /Stage[main]/Main/Node[default]/File[/tmp/hello_world]/ensure: created
Info: Creating state file /var/lib/puppet/state/state.yaml
Notice: Finished catalog run in 6.77 seconds
```

Once Puppet finishes, the client instance will have installed and started Nginx, which can be verified by checking that the Nginx service is running:

```
service nginx status
```

The text file will also have been created:

```
cat /tmp/hello_world
```

Auto Scaling and Autosign: Disabling Certificate Security

Puppet’s key-signing system is great when clients have a certain level of permanence, but when you are constantly creating and destroying hosts, it can become an impediment. Manually signing key requests is clearly not an option when combined with AWS Auto Scaling, which automatically launches instances in response to changes in required capacity.

Aware that this method of signing keys might not be suitable for all situations, Puppet makes it easy to disable it with a feature known as *autosigning*. This is done by populating the */etc/puppet/autosign.conf* file with a list of hostnames for which autosigning is enabled—when these hosts connect to the Puppet master, key checking will be bypassed. Autosign can be enabled globally by using a wildcard (`*`) as the hostname.

Disabling security measures always involves a trade-off. It is our view that, as long as your Puppet master is sufficiently protected by security groups or firewalls, enabling autosigning is an acceptable risk. This is the only practical way of using Puppet in conjunction with Auto Scaling, and to a lesser extent EC2 as a whole.

This example uses only two types of Puppet resources: a service and a file. The Puppet Documentation site maintains a list of all available resource types on its Type Reference page.

Puppet is a very broad subject, covered in full detail in the recent Learning Puppet 4 by Jo Rhett (O'Reilly). Jo's book is the most up to date tome covering Puppet, and the first to cover version 4 in depth.

Puppet and CloudFormation

The previous example shows how Puppet can make it easy to manage the configuration of EC2 instances. Previous chapters have shown how CloudFormation provides a similar function for provisioning EC2 resources. What about the combination of the two?

Amazon has built some aspects of configuration directly into CloudFormation. In addition to creating EC2 instances, it can automatically install packages and run services on those instances after they have launched. This means there is quite a lot of overlap between Puppet and CloudFormation, which can sometimes lead to questions over which should be responsible for particular tasks. If CloudFormation can handle many of the tasks usually handled by Puppet, do you even need to use Puppet?

CloudFormation's configuration management system works by embedding configuration data into the stack template, via the AWS::CloudFormation::Init metadata attribute, which can be specified when declaring EC2 resources in stack templates. For example, this snippet would install the `puppet-server` package, using the Yum package manager:

```
"Resources": {
  "MyInstance": {
    "Type": "AWS::EC2::Instance",
    "Metadata" : {
      "AWS::CloudFormation::Init" : {
        "config" : {
          "packages" : {
            "yum": {
              "puppet-server": []
            }
          },
[ truncated ]
```

When the instance is launched, this metadata is parsed, and the required packages are installed. The metadata is parsed by the cfn-init script, which also performs the actual package installation. This script, developed by Amazon, is preinstalled on all Amazon Linux AMIs and is also available for installation on other operating systems.

Note

The `cfn-init` script is preinstalled on Amazon Linux AMIs, but can also be installed manually on most Linux operating systems. Amazon provides RPM packages (for RedHat-based systems, and source code for others).

`cfn-init` is short for *CloudFormation initialization*, which is a hint as to its purpose. It is executed during the instance's boot process, at a similar point in time to when */etc/rc.local-like* scripts are executed, by passing a shell script as user data to the instance.

The script is responsible for performing post-boot configuration of the instance. It queries the EC2 API to find out information about the CloudFormation stack it is part of—for example, it looks at the `Metadata` attribute for the instance, to see which software packages should be installed.

Remember that accessing the EC2 API requires the use of IAM credentials with permissions to query the API interface. Data such as tags and CloudFormation metadata is not directly available to the instance in the same way that user data is.

Because configuration information is contained within CloudFormation stack templates, it must be valid JSON. This means certain characters must be escaped, and strings can consist of only a single line (multiline strings can be created with the `Fn::Join` function).

Stack templates have a maximum size of 450 KB (50 KB when not using S3 to pass the template), which acts as a hard limit to how much configuration info S3 to pass the template rmation can be contained in a single stack. Nested stacks, though often less than convenient, may be a way to work around this limit.

Working around these limitations may make a dedicated configuration management tool like Puppet easier to work with, but that does not mean CloudFormation's tools are redundant.

Note

Amazon maintains and supports its own Linux distribution, Amazon Linux, a derivative of Red Hat Enterprise Linux. Amazon Linux AMIs come preinstalled with a number of AWS tools, such as the latest versions of all command-line tools, and the `cfn-init` package.

More information is available on the Linux AMI page.

Combining CloudFormation and Puppet accomplishes the majority of the aims set out in "Bootstrapping an Infrastructure." CloudFormation can create the AWS resources that make up your infrastructure, and Puppet can configure the operating system and application.

Because both use text-based template files, everything can be stored in a revision-control system. By checking out a copy of your repository, you have absolutely everything you need to bootstrap your infrastructure to a running state.

To demonstrate a few of the CloudFormation configuration management features and see how they interact with Puppet, we will create an example stack that automatically installs and configures a Puppet master.

This can provide the base of an entirely bootstrapped infrastructure. Information describing both resources (EC2 instances) and application-level configuration is contained within a single stack template.

In this example we will be using the Amazon Linux AMI, which does not have Puppet *baked in*—that is, Puppet has not previously been installed when the instance launches. Instead, it will be installed by `cfn-init` when the instance has finished booting.

Example 4-2 shows a CloudFormation stack template that declares an EC2 instance resource and uses its metadata to install, configure, and run the Puppet master service. The template also includes some supplementary resources—the security groups required to access the SSH and Puppet services running on the instance.

The `PuppetMasterGroup` is a security group that will contain the Puppet master instance. It allows Puppet clients to access the master, and also allows SSH from anywhere so we can administer the instance.

Example 4-2. Puppet master CloudFormation stack

```
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Description": "Example Puppet master stack",
    "Parameters" : {
        "KeyName": {
            "Description" : "EC2 KeyPair name",
            "Type": "String",
            "MinLength": "1",
            "MaxLength": "255",
            "AllowedPattern" : "[\\x20-\\x7E]*",
            "ConstraintDescription" : "can contain only ASCII characters."
        },
        "AMI" : {
            "Description" : "AMI ID",
            "Type": "String"
        }
    },
    "Resources": {
        "CFNKeys": {❶
            "Type": "AWS::IAM::AccessKey",
            "Properties": {
                "UserName": {
                    "Ref": "CFNInitUser"
                }
            }
        },
        "CFNInitUser": {
            "Type": "AWS::IAM::User",
            "Properties": {
                "Policies": [
                   {
                    "PolicyName": "AccessForCFNInit",
                    "PolicyDocument": {
                       "Statement": [
                           {
                            "Action": "cloudformation:DescribeStackResource",
                            "Resource": "*",
                            "Effect": "Allow"
                           }
                       ]
                    }
                  }
                ]
            }
        },
        "PuppetClientSecurityGroup": {❷
            "Type": "AWS::EC2::SecurityGroup",
            "Properties": {
             "SecurityGroupIngress": [
                 {
                  "ToPort": "22",
                  "IpProtocol": "tcp",
                  "CidrIp": "0.0.0.0/0",
                  "FromPort": "22"
                 }
             ],
             "GroupDescription": "Group for SSH access to Puppet clients"
            }
        },
        "PuppetMasterSecurityGroup": {❸
            "Type": "AWS::EC2::SecurityGroup",
            "Properties": {
             "SecurityGroupIngress": [
               {
                "ToPort": "8140",
                "IpProtocol": "tcp",
                "SourceSecurityGroupName": { "Ref":
"PuppetClientSecurityGroup"},
                "FromPort": "8140"
               },
               {
                "ToPort": "22",
                "IpProtocol": "tcp",
                "CidrIp": "0.0.0.0/0",
                "FromPort": "22"
               }
             ],
             "GroupDescription": "Group for Puppet client to master
communication"
            }
        },
        "PuppetMasterInstance": {
            "Type": "AWS::EC2::Instance",
            "Properties": {
                "UserData": {❹
                    "Fn::Base64": {
                        "Fn::Join": [
                            "",
                            [
       "#!/bin/bash\n",
       "/opt/aws/bin/cfn-init --region ", { "Ref": "AWS::Region" },  " -s ",
       { "Ref": "AWS::StackName" }, " -r PuppetMasterInstance ",
       " --access-key ", { "Ref": "CFNKeys" },
       " --secret-key ", { "Fn::GetAtt": [ "CFNKeys", "SecretAccessKey" ] },
"\n"
                            ]
                        ]
                    }
                },
                "KeyName": { "Ref" : "KeyName" },
                "SecurityGroups": [
                    {
                        "Ref": "PuppetMasterSecurityGroup"
                    }
                ],
                "InstanceType": "t2.micro",
                "ImageId": { "Ref" : "AMI" }❺
            },
            "Metadata": {
                "AWS::CloudFormation::Init": {
                    "config": {
                        "files": {❻
                            "/etc/puppet/autosign.conf": {
                                "content": "*.internal\n",
                                "owner": "root",
                                "group": "wheel",
                                "mode": "100644"
                            },
                            "/etc/puppet/manifests/site.pp": {
                                "content": "import \"nodes\"\n",
                                "owner": "root",
                                "group": "wheel",
                                "mode": "100644"
                            },
                            "/etc/puppet/manifests/nodes.pp": {
                                "content": {
                                    "Fn::Join": [
                                     "",
                                     [
                                      "node basenode {\n",
                                      "  include cfn\n",
                                      "  package { 'nginx':\n",
                                      "    ensure => installed\n",
                                      "  }\n",
                                      "  service { 'nginx':\n",
                                      "    ensure => running,\n",
                                      "    require=> Package['nginx']\n",
                                      "  }\n",
                                      "}\n",
                                      "node /^.*internal$/ inherits basenode
{\n",
                                      "}\n"
                                     ]
                                    ]
                                },
                                "owner": "root",
                                "group": "wheel",
                                "mode": "100644"
                            },
                            "/etc/puppet/modules/cfn/lib/facter/cfn.rb": {
                                "owner": "root",
                                "source":
"https://s3.amazonaws.com/cloudformation-examples/cfn-facter-plugin.rb",
                                "group": "wheel",
                                "mode": "100644"
                            },
                            "/etc/yum.repos.d/epel.repo": {
                                "owner": "root",
                                "source":
"https://s3.amazonaws.com/cloudformation-examples/enable-epel-on-amazon-linux-
ami",
                                "group": "root",
                                "mode": "000644"
                            },
                            "/etc/puppet/fileserver.conf": {
                                "content": "[modules]\n allow *.internal\n",
                                "owner": "root",
                                "group": "wheel",
                                "mode": "100644"
                            },
                            "/etc/puppet/puppet.conf": {
                                "content": {❼ ❽
                                    "Fn::Join": [
                                        "",
                                        [
                                            "[main]\n",
                                            " logdir=/var/log/puppet\n",
                                            " rundir=/var/run/puppet\n",
                                            " ssldir=$vardir/ssl\n",
                                            " pluginsync=true\n",
                                            "[agent]\n",
                                            " classfile=$vardir/classes.txt\n",
                                            " localconfig=$vardir/localconfig\n"
                                        ]
                                    ]
                                },
                                "owner": "root",
                                "group": "root",
                                "mode": "000644"
                            },
                            "/etc/puppet/modules/cfn/manifests/init.pp": {
                                "content": "class cfn {}",
                                "owner": "root",
                                "group": "wheel",
                                "mode": "100644"
                            }
                        },
                        "packages": {❾
                            "rubygems": {
                                "json": []
                            },
                            "yum": {
                                "gcc": [],
                                "rubygems": [],
                                "ruby-devel": [],
                                "make": [],
                                "puppet-server": [],
                                "puppet": []
                            }
                        },
                        "services": {❿
                            "sysvinit": {
                                "puppetmaster": {
                                    "ensureRunning": "true",
                                    "enabled": "true"
                                }
                            }
                        }
                    }
                }
            }
        }
    },
    "Outputs": {⓫
```

```
        "PuppetMasterPrivateDNS": {
            "Description": "Private DNS Name of PuppetMaster",
            "Value": {
                "Fn::GetAtt": [ "PuppetMasterInstance", "PrivateDnsName" ]
            }
        },
        "PuppetMasterPublicDNS": {
            "Description": "Public DNS Name of PuppetMaster",
            "Value": {
                "Fn::GetAtt": [ "PuppetMasterInstance", "PublicDnsName" ]
            }
        },
        "PuppetClientSecurityGroup": {
            "Description": "Name of the Puppet client Security Group",
            "Value": { "Ref" : "PuppetClientSecurityGroup" }
        }
    }
}
```

❶

Remember that `cfn-init` requires an IAM user with access to the EC2 API. The `CFNKeys` and `CFNInitUser` resources declare an IAM user with permissions to describe all CloudFormation stack resources, and also an IAM access key and secret. These credentials are passed to the Puppet master instance via user data. The same result could be achieved by using IAM roles.

❷

The `PuppetClientGroup` is a security group that will be populated with Puppet client instances. Any members of this group will be allowed to contact the Puppet master on TCP port 8140 (the default Puppet master port).

❸

The `PuppetMasterGroup` is a security group that will contain the Puppet master instance. It allows Puppet clients to access the master, and also allows SSH from anywhere so we can administer the instance.

❹

The `User Data` attribute for the instance is a Base64-encoded shell script. This script runs the `cfn-init` program, passing it some information about the stack that it is part of. This includes the EC2 region, the stack name, and the IAM access key and secret that will be used to query the EC2 API.

Because JSON does not support multiline strings, the `Fn::Join` function is used to create a multiline shell script.

❺

Find the latest ID for the Amazon Linux AMI-ami-6869aa05 at the time of writing. This value is passed as a parameter when creating the stack.

❻

Here is where the interesting part begins. Remember that the `cfn-init` script will retrieve this metadata after the instance has launched. The file's `Metadata` attribute lists a number of files that will be created on the instance, along with information about the file's user and group permissions.

❼

Files can be retrieved from remote sources such as web servers or S3. They will be downloaded when the instance launches. Basic HTTP authorization is supported.

❽

Alternatively, the content can be explicitly set, using the `Fn::Join` function to create multiline files.

❾

Software packages can be installed from multiple sources—this example shows some RubyGems being installed, along with some packages from the Yum package manager.

❿

Finally, we specify that the Puppet master service should be enabled (i.e., it should start automatically when the instance boots) and running.

⓫

The `Outputs` section makes it easier to retrieve information about the resources in your stack. Here, we are specifying that we want to access the private DNS name of the Puppet master instance and the name of the Puppet client security group.

Note

This example is based on Amazon's documentation, which you can access for further information and additional examples of what can be done with Puppet and CloudFormation.

Create the stack with the command-line tools:

```
aws cloudformation create-stack --stack-name puppet-master --template-body \
    file://puppet_master_cloudformation.json \
    --region us-east-1 --capabilities CAPABILITY_IAM \
    --parameters ParameterKey=AMI,ParameterValue=ami-6869aa05 \
    ParameterKey=KeyName,ParameterValue=federico
```

Because this stack will create an IAM user, we need to add the `--capabilities CAPABILITY_IAM` option to the command. Without this setting, CloudFormation would refuse to create the IAM user. Capabilities are used to prevent some forms of privilege escalation, such as a malicious user creating an IAM policy granting access to resources that the user would not otherwise have access to.

Once the stack has been created, we can find out the Puppet master's DNS name and the security group by querying the stack's outputs:

```
aws cloudformation describe-stacks --stack-name puppet-master | jq \
'.Stacks[0].Outputs[]'
```

To verify that everything is working as expected, log in to the instance with SSH and check whether the `puppetmaster` service is running.

Warning

`cfn-init` halts client-side execution upon encountering the first error, but the default behavior of a stack is to rollback a failed initialization, decommissioning all stack resources. In order to more easily troubleshoot a problem, set `DisableRollback` in your `create-stack` calls (in the console, this is called *Rollback on failure*). This makes debugging a very linear process, but also a straightforward one. All support binaries are found under `/opt/aws/bin/`, and initialization is logged at `/var/log/cfn-init.log` and `/var/log/cfn-init-cmd.log`.

Now that we know the Puppet master is up and running, we can bring up a client to demonstrate it in action.

Example 4-3 shows a CloudFormation stack that declares a Puppet client instance. This instance, when launched, will connect to our Puppet master and retrieve its configuration—in this case, it will install Nginx on the client.

In Example 4-3, we will use two parameters—one for the Puppet master's DNS name and one for the Puppet client security group. The parameters to this stack are the output of the previous stack.

Example 4-3. Puppet client CloudFormation stack

```
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Description": "Example Puppet client stack",
    "Parameters" : {❶
      "KeyName": {
        "Description" : "EC2 KeyPair name",
        "Type": "String",
        "MinLength": "1",
        "MaxLength": "255",
        "AllowedPattern" : "[\\x20-\\x7E]*",
        "ConstraintDescription" : "can contain only ASCII characters."
      },
      "AMI" : {
        "Description" : "AMI ID",
        "Type": "String"
      },
      "PuppetMasterDNS" : {
        "Description" : "Private DNS name of the Puppet master instance",
        "Type": "String"
      },
      "PuppetClientSecurityGroup" : {
        "Description" : "Name of the Puppet client Security Group",
        "Type": "String"
      }
    },

    "Resources": {
        "CFNKeys": {
            "Type": "AWS::IAM::AccessKey",
            "Properties": {
                "UserName": {
                    "Ref": "CFNInitUser"
                }
            }
        },
        "CFNInitUser": {
            "Type": "AWS::IAM::User",
            "Properties": {
                "Policies": [
                    {
                     "PolicyName": "AccessForCFNInit",
                     "PolicyDocument": {
                         "Statement": [
                             {
                              "Action": "cloudformation:DescribeStackResource",
                              "Resource": "*",
                              "Effect": "Allow"
                              }
                         ]
                     }
                    }
                ]
            }
        },
        "PuppetClientInstance": {
            "Type": "AWS::EC2::Instance",
            "Properties": {
                "UserData": {❷
                    "Fn::Base64": {
                        "Fn::Join": [
                            "",
                            [
"#!/bin/bash\n",
"/opt/aws/bin/cfn-init --region ", { "Ref": "AWS::Region" },  " -s ",
{ "Ref": "AWS::StackName" }, " -r PuppetClientInstance ",
" --access-key ", { "Ref": "CFNKeys" },
" --secret-key ", { "Fn::GetAtt": [ "CFNKeys", "SecretAccessKey" ] }
                            ]
                        ]
                    }
                },
                "KeyName": { "Ref" : "KeyName" },
                "SecurityGroups": [
                    {
                        "Ref": "PuppetClientSecurityGroup"
                    }
                ],
                "InstanceType": "t2.micro",
                "ImageId": { "Ref" : "AMI" }
            },
            "Metadata": {
                "AWS::CloudFormation::Init": {
                    "config": {
                        "files": {
                            "/etc/puppet/puppet.conf": {
                                "content": {
                                    "Fn::Join": [
                                        "",
                                        [
                        "[main]\n",
                        " server=", { "Ref": "PuppetMasterDNS" }, "\n",
                        " logdir=/var/log/puppet\n",
                        " rundir=/var/run/puppet\n",
                        " ssldir=$vardir/ssl\n",
                        " pluginsync=true\n",
                        "[agent]\n",
                        " classfile=$vardir/classes.txt\n",
                        " localconfig=$vardir/localconfig\n"
```

```
                                        ]
                                    ]
                                },
                                "owner": "root",
                                "group": "root",
                                "mode": "000644"
                            }
                        },
                        "packages": {❸
                            "rubygems": {
                                "json": []
                            },
                            "yum": {
                                "gcc": [],
                                "rubygems": [],
                                "ruby-devel": [],
                                "make": [],
                                "puppet": []
                            }
                        },
                        "services": {❹
                            "sysvinit": {
                                "puppet": {
                                    "ensureRunning": "true",
                                    "enabled": "true"
                                }
                            }
                        }
                    }
                }
            }
        }
    },
    "Outputs" : {
        "PuppetClientIP" : {
          "Description" : "Public IP of the Puppet client instance",
          "Value" : { "Fn::GetAtt" : [ "PuppetClientInstance", "PublicIp" ] }
        },
        "PuppetClientPrivateDNS" : {
          "Description" : "Private DNS of the Puppet client instance",
          "Value" : { "Fn::GetAtt" : [ "PuppetMasterInstance",
"PrivateDnsName" ] }
        }
    }
}
```

❶

The parameters define values that can be specified by the user when the stack is launched. If a value is not specified, the default is used. If no default is set, attempting to create the stack without specifying a value will result in an error message.

❷

The user data for this instance is the same as in the previous example: run `cfn-init`, and let it configure the instance according to the `Metadata` attributes.

❸

We need to install only the Puppet package, as Puppet master is not required on clients.

4

> Finally, we start the Puppet agent, which will periodically connect to the master and retrieve its configuration.

Because this stack uses parameters, creating it requires a slightly different invocation of the command-line tools. We need to pass the output of the previous stack as parameters, like so:

```
aws cloudformation create-stack --stack-name puppet-client --template-body \
    file://puppet_client_cloudformation.json \
    --region us-east-1 --capabilities CAPABILITY_IAM \
    --parameters ParameterKey=AMI,ParameterValue=ami-6869aa05 \
    ParameterKey=KeyName,ParameterValue=federico \
    ParameterKey=PuppetMasterDNS,ParameterValue=ip-172-31-48-10.ec2.internal \
    ParameterKey=PuppetClientSecurityGroup,ParameterValue=puppet-master-
PuppetClientSecurityGroup-YI5PIW673C1N
```

Once the stack has been created, we can log in to the client instance and check its status. Remember, it takes a few minutes for Puppet to run, so if you log in to the instance immediately after it finishes booting, the Puppet run might still be in progress. Puppet will write its logs to */var/log/puppet/*, so check this location first if you need to troubleshoot.

Consider what this example achieves, in the context of “Bootstrapping an Infrastructure.” We have deployed a fully automated Puppet master, which did not require any manual configuration whatsoever. Once that was running, we used it to bootstrap the rest of the environment—again, without any manual configuration.

If your CloudFormation stacks are stored in (for example) GitHub, your disaster recovery plan can be summarized in four steps:

1. Check out your GitHub repository.
2. Install the AWS command-line tools.
3. Launch the Puppet master stack.
4. Launch the Puppet client stack.

This is incredibly easy to document, and the steps can be followed without requiring expertise in Puppet (or indeed, Nginx).

There are a few deficiencies in this workflow that make it somewhat inconvenient to use in production. For example, if your Puppet master’s DNS name changes—which will happen if the instance is stopped and restarted, or terminated and replaced with a new instance—you will need to delete and re-create your Puppet client stack to pick up the new changes. This is clearly suboptimal—as we improve on this stack through the rest of the book, we will look at ways to make this easier to use in production. To solve the “changing DNS name” problem, we will use Route 53 to automatically assign DNS names to our instances, so our Puppet master will always be reachable at *puppet.example.com*.

Another potential improvement is in how files are created on the instance. The Puppet master stack demonstrates two methods: pulling files from HTTP sites and explicitly defining the content in the stack template. There are other options available, such as retrieving files from an S3 bucket.

It is worth noting that there is nothing specific to Puppet in this example. The same concept could be used to bootstrap any configuration management software, such as Chef or Ansible, or indeed any software package that can be installed with Apt, Yum, RubyGems, pip, or any other package manager supported by CloudFormation.

There is a lot of overlap between CloudFormation and configuration management tools. While it would technically be possible to replace most of Puppet's core features with CloudFormation, it would be a lot less convenient than using a dedicated configuration management tool. If your instance configuration is very simple, CloudFormation might suffice on its own-we find it simpler to put configuration management in place from the beginning.

User Data and Tags

AWS provides two built-in mechanisms to provide data to your EC2 instances: user data and tags. User data is supplied to the instance at launch time and cannot be changed without restarting the instance. Tags are more flexible—these key/value pairs can be changed at any point during the instance's life cyle.

Both of these methods can be used to provide data to your EC2 instances, which can then be used by your scripts and applications. These building blocks enable several useful features. For example, you can create an AMI that can perform two roles (e.g., running a web server or a database, but not both). When launching an instance from this AMI, you could set a `role=web` or `role=dbms` tag. The launch scripts on the AMI would read this tag and know whether it should start Nginx or PostgreSQL.

Before the introduction of tags, it was necessary to build your own inventory storage system if you wanted to keep track of particular details about your instances. With tags, EC2 is itself its own inventory system. While user data is available only within an individual EC2 instance, tags can be queried externally by any party with authorized API access.

Tags and user data—and the differences between them—are described in more detail in "Mapping Instances to Roles". For now, it is worth knowing that we can use tags both within EC2 instances, and from applications running outside AWS, a kind of inventory management system that stores metadata for each instance.

CloudFormation also uses tags. For example, when an EC2 instance is created as part of a CloudFormation stack, it is automatically tagged with the name and ID of the CloudFormation stack to which it belongs.

In relation to configuration management tools, tags and user data are both extremely useful features. Through the use of Facter plug-ins (which gather additional information about your systems), Puppet is able to access user data and tags and use them as standard variables in its configuration manifests.

Note

CloudFormation is most commonly used as a boot-time configuration tool-something reflected in its general design. However, the availability of the `cfn-hup` helper daemon overturns this assumption: `cfn-hup` monitors resource metadata for changes, and applies any new client-side configuration when necessary. The stack template of a running stack can be updated with the update-stack CLI command.

Typically, Puppet uses the hostname of the instance to decide which configuration should be applied. Because EC2 instances have autogenerated hostnames, this is not immediately useful. One way to work around this problem is to use user data to control which configuration should be applied. We will do this by providing JSON-formatted user data that can be read by Puppet.

To begin, launch a new Ubuntu 14.04 “Trusty” instance with the following user data:

```
{"role": "web"}
```

This is a JSON-formatted string simply containing a `role=web` key/value pair.

```
$ aws ec2 run-instances --image-id ami-c80b0aa2 --region us-east-1 \
  --key federico --security-groups ssh --instance-type t2.micro \
  --user-data '{"role": "web"}' --output text
740376006796    r-453484fb
INSTANCES       0       x86_64          False   xen     ami-c80b0aa2    i-
4f5f827e      t2.micro        federico        2016-08-29T11:37:54.000Z
ip-172-31-54-180.ec2.internal   172.31.54.180           /dev/sda1       ebs
True            subnet-2a45b400 hvm     vpc-934935f7
[output truncated]
```

Once the instance has launched, you can verify the user data has reached the instance by logging in and running the `ec2metadata` command:

```
$ ec2metadata
ami-id: ami-c80b0aa2
ami-launch-index: 0
ami-manifest-path: (unknown)
ancestor-ami-ids: unavailable
availability-zone: us-east-1a
block-device-mapping: ami
ephemeral0
ephemeral1
root
instance-action: none
instance-id: i-4f5f827e
instance-type: t2.micro
local-hostname: ip-172-31-54-180.ec2.internal
local-ipv4: 172.31.54.180
kernel-id: unavailable
mac: unavailable
profile: default-hvm
product-codes: unavailable
public-hostname: ec2-54-208-246-122.compute-1.amazonaws.com
public-ipv4: 54.208.246.122
public-keys: ['ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAQEA83H2O96JIchWxmFMITAfQ4mgfgP4CgF2mZteBdwHnWVgiMlzwnL/
zfAoAUeKCgFZ+H5L2qxv3aERoipnFwUVI1Y0Ym7IjWs+CgadDMsfJr1MsitFdLhRTu8D8kYg4E32FeKn
4ZNJN/QxANj15bNDZ2XYTE1v/0QWSorao0NQv7bK/anN7IuPtfPjbhXwTLVVbHSG5SErIMSqVbksj0r1
pzjnBxAmmeXdHHmQV889oMsHEpvWFroGIRsTopYmVe7H8d2+P6lZgkz3WrDCOdoGwTWvbnfNHonYvE0w
SQro/nEa1b7OB3i23tLzque7Z0PdUPKvg48JaLSEBiL6ydLCyw== federico']
ramdisk-id: unavailable
reserveration-id: unavailable
security-groups: ssh
user-data: {"role": "web"}
```

We install Puppet and its stdlib library from the upstream Puppet repositories, as we need a newer version of Facter than what is included in Trusty for our code to work correctly:

Example 4-4. Installing Puppet's upstream release

```
wget --quiet http://apt.puppet.com/puppetlabs-release-trusty.deb
sudo dpkg -i puppetlabs-release-trusty.deb
sudo apt update
# Should not be installed, but just in case we caught you using an old
instance...
sudo apt remove --yes puppet puppet-common
# Install latest version of puppet from PuppetLabs repo
sudo apt install --yes puppet facter -t trusty
#install the stdlib module
sudo puppet module install puppetlabs-stdlib
rm puppetlabs-release-trusty.deb
```

Now create a new file */etc/puppet/manifests/site.pp* with the contents shown in Example 4-5.

Example 4-5. Puppet and user data

```
require stdlib

node default {

    $userdata = parsejson($ec2_userdata)

    $role = $userdata['role']

    case $role {
        "web": {
            require my_web_module
        }
        "db": {
            require my_database_module
        }
        default: { fail("Unrecognised role: $role") }
    }

}
```

This file is responsible for changing the behavior of Puppet so that it ignores the hostname of the instance and instead looks at the user data. `node default` indicates that the following configuration should be applied to all nodes, regardless of their hostname. We then use the `parsejson` function to read the EC2 user data string into a Puppet variable. Finally, we include a different module depending on the value of the `$role` variable.

You could proceed with running this example by executing `puppet apply /etc/puppet/manifests/site.pp`. Because we have not yet created a `my_web_module`, Puppet will fail. However, it will fail with an error that `my_web_module` could not be found, demonstrating that our underlying theory is indeed working as planned.

```
$ facter --version
2.4.6
$ puppet apply /etc/puppet/manifests/site.pp
Error: Could not find class my_web_module for ip-172-31-54-180.ec2.internal on
node ip-172-31-54-180.ec2.internal
Error: Could not find class my_web_module for ip-172-31-54-180.ec2.internal on
node ip-172-31-54-180.ec2.internal
```

In the following section, we will use tags to look up instances based on the role they are tagged with and then execute shell scripts on the returned instances.

Executing Tasks with Fabric

The standard way to run Puppet is to allow the clients to contact the master according to a regular schedule, either using Puppet's internal scheduling (when running as a daemon) or a tool such as cron. This lets you make changes on the central Puppet server, knowing that they will eventually propagate out to your instances.

Sometimes, it can be useful to take more control over the process and run Puppet only when you know there are changes you would like to apply. To take a common example, let's say you have a cluster of Nginx web servers behind an Elastic Load Balancer, and you would like to deploy a configuration change that will cause Nginx to restart. Allowing Puppet to run on all instances at the same time would restart all of your Nginx servers at the same time, leaving you with zero functional web servers for a brief period.

In this case, it would be better to run Puppet on a few instances at a time, so that there are always enough running Nginx instances to service incoming requests.

In some cases, it is necessary to do additional work either before or after a Puppet run. Continuing with the example of web servers behind an Elastic Load Balancer—if you just need to restart Nginx, it is sufficient to leave the machine *in service* (active) while Puppet runs, as Nginx will not take a long time to restart. But what if you need to perform an operation that might take a few minutes? In this rolling-update scenario, you will need to remove the instance from the ELB, update it, and then return it to service—not something easily achieved with Puppet alone.

Several tools are dedicated to simplifying the task of running commands on groups of servers. Fabric is particularly flexible when it comes to working with EC2 instances and traditional hardware alike, and we will use it for the following examples.

Fabric is a Python tool used to automate system administration tasks. It provides a basic set of operations (such as executing commands and transferring files) that can be combined with some custom logic to build powerful and flexible deployment systems, or simply make it easier to perform routine tasks on groups of servers or EC2 instances. Because Boto is also Python-based, we can use it to quickly integrate with AWS services.

Tasks are defined by writing Python functions, which are usually stored in a file named *fabfile.py*. These functions use Fabric's Python API to perform actions on remote hosts. Here is a simple example of a Fabric file supplied by *fabfile.org*:

```
from fabric.api import run

def host_type():
    run('uname -s')
```

The `host_type` task can be executed on numerous servers, for example:

```
$ fab -H localhost,linuxbox host_type
[localhost] run: uname -s
[localhost] out: Darwin
```

```
[linuxbox] run: uname -s
[linuxbox] out: Linux

Done.
Disconnecting from localhost... done.
Disconnecting from linuxbox... done.
```

Fabric understands the concept of *roles*—collections of servers, grouped by the role they perform (or some other factor). Using roles, we can easily do things like running TaskA on all web servers, followed by TaskB on all database servers.

Roles are typically defined in your Fabric file as a static collection of roles and the hostnames of their members. However, they can also be created dynamically by executing Python functions, which means roles can be populated by querying the AWS API with Boto. This means we can execute TaskA on all EC2 instances tagged with `role=webserver`, without needing to keep track of a list of instances.

To demonstrate this, we will launch an instance and then use Fabric to execute commands on that host.

Mike has written a small Python package containing a helper function that makes it easy to look up EC2 instances using tags. It is used in the following example and can be downloaded from GitHub.

Begin by installing Fabric and the helper library as follows:

```
pip install fabric
sudo apt install git --yes
pip install git+git://github.com/mikery/fabric-ec2.git
```

Using the Management Console or command-line tools launch a `t2.micro` EC2 instance and provide it with some EC2 Tags. For this example, we will use two tags: *staging:true* and *role:web*:

```
aws ec2 create-tags --resources i-3b20870a --tags Key=role,Value=web \
  Key=staging,Value=true
```

While the instance is launching, create a file named *fabfile.py*, which will store our Fabric tasks and configuration. You could also give it another name, but you will need to pass this as an option when executing Fabric—for example, `fab --fabfile=/some/file.py`. The file should contain the following contents:

```
from fabric.api import run, sudo, env
from fabric_ec2 import EC2TagManager

def configure_roles():
    """ Set up the Fabric env.roledefs, using the correct roles for the given
environment
    """
    tags = EC2TagManager(common_tags={'staging': 'true'})

    roles = {}
    for role in ['web', 'db']:
        roles[role] = tags.get_instances(role=role)

    return roles

env.roledefs = configure_roles()
```

```
def hostname():
        run('hostname')
```

Once the instance has launched, the Fabric task can be executed thus:

```
$ fab -u ubuntu hostname --roles web
[ec2-54-161-199-160.compute-1.amazonaws.com] Executing task 'hostname'
[ec2-54-161-199-160.compute-1.amazonaws.com] run: hostname
[ec2-54-161-199-160.compute-1.amazonaws.com] out: ip-172-31-62-71
[ec2-54-161-199-160.compute-1.amazonaws.com] out:


Done.
Disconnecting from ec2-54-161-199-160.compute-1.amazonaws.com... done.
```

Fabric will use the EC2 API to find the hostname of any instances that match the tag query and then perform the requested task on each of them. In our example case, it will have found only our single test instance.

With this in place, you have a simple way of running tasks—including applying Puppet manifests—selectively across your infrastructure. Fabric provides an excellent base for automating your deployment and orchestration processes.

Because tasks are just Python functions, they can be used to do helpful things. Before deploying to a web server instance, Fabric can first remove it from an ELB and wait for live traffic to stop hitting the instance before it is updated.

Master-less Puppet

So far, we have been working with Puppet in the typical master/client topology. This is the most common way to run Puppet, especially when working with physical hardware, outside of a cloud environment.

Puppet has another mode of operation, which does not require a master server: in *local mode*, where the client is responsible for applying its own configuration rather than relying on a master server.

There are two key reasons that make this useful when working with AWS.

The first is availability. When Puppet runs during the instance boot process, it becomes a core part of your infrastructure. If the Puppet master is unavailable, Auto Scaling will not work properly, and your application might become unavailable. Although you can deploy a cluster of Puppet masters to increase resilience to failures and load, it can be simply easier to remove it from the equation.

The second reason relates to Auto Scaling. Given that AWS can launch new instances in response to changing factors such as traffic levels, we cannot predict when new instances will be launched. In environments where large numbers of instances are launched simultaneously, it is possible to overwhelm the Puppet master, leading to delays in autoscaling or even instances that fail to launch properly as Puppet is never able to complete its configuration run.

When operating in local mode, an instance is more self-contained: it does not rely on an operational Puppet master in order to become operational itself. As a result, you have one less single point of failure within your infrastructure.

As always, there is a trade-off to be considered—there's no such thing as a free lunch, after all. A number of useful features rely on a master/client setup, so moving to a master-less topology means these features are no longer available. These include things like Puppet Dashboard, which provides an automatically populated inventory based on data supplied by clients, and exported resources.

In practical terms, applying a configuration without a Puppet master is simply a case of executing the following:

```
puppet apply /etc/puppet/manifests/site.pp
```

This will trigger the usual Puppet logic, where the node's hostname is used to control which configuration will be applied. It is also possible to apply a specific manifest:

```
puppet apply /etc/puppet/modules/mymodule/manifests/site.pp
```

This does mean that the Puppet manifests must be stored on every instance, which can be achieved in various ways. They can be baked in to the AMI if they do not change frequently, or deployed alongside your application code if they do.

We can use a modified version of Example 4-3 to create a CloudFormation stack with embedded Puppet manifests. Example 4-6 shows the updated stack template.

Example 4-6. Masterless Puppet CloudFormation stack

```
{
    "AWSTemplateFormatVersion" : "2010-09-09",
    "Description": "Example Puppet masterless stack",
    "Parameters" : {
        "KeyName": {
            "Description" : "EC2 KeyPair name",
            "Type": "String",
            "MinLength": "1",
            "MaxLength": "255",
            "AllowedPattern" : "[\\x20-\\x7E]*",
            "ConstraintDescription" : "can contain only ASCII characters."
        },
        "AMI" : {
            "Description" : "AMI ID",
            "Type": "String"
        }
    },
    "Resources" : {
        "CFNKeys": {
            "Type": "AWS::IAM::AccessKey",
            "Properties": {
                "UserName": {
                    "Ref": "CFNInitUser"
                }
            }
        },
        "CFNInitUser": {
            "Type": "AWS::IAM::User",
            "Properties": {
                "Policies": [
                    {
                        "PolicyName": "AccessForCFNInit",
                        "PolicyDocument": {
                            "Statement": [
                             {
                              "Action": "cloudformation:DescribeStackResource",
                              "Resource": "*",
                              "Effect": "Allow"
                             }
                            ]
                        }
                    }
                ]
            }
        },
        "NginxInstance" : {
            "Type" : "AWS::EC2::Instance",
            "Metadata" : {
                "AWS::CloudFormation::Init" : {
                    "config" : {
                        "packages" : {
                            "yum" : {
                                "puppet" : [],
                                "ruby-devel" : [],
                                "gcc" : [],
                                "make" : [],
                                "rubygems" : []
                            },
                            "rubygems" : {
                                "json" : []
                            }
                    },
                    "files" : {
                        "/etc/yum.repos.d/epel.repo" : {
                        "source" : "https://s3.amazonaws.com/cloudformation-
examples/enable-epel-on-amazon-linux-ami",
                        "mode" : "000644",
                        "owner" : "root",
                        "group" : "root"
                        },
                        "/etc/puppet/autosign.conf" : {
                            "content" : "*.internal\n",
                            "mode" : "100644",
                            "owner" : "root",
                            "group" : "wheel"
                        },
                        "/etc/puppet/puppet.conf" : {
                            "content" : { "Fn::Join" : ["", [
                                "[main]\n",
                                " logdir=/var/log/puppet\n",
                                " rundir=/var/run/puppet\n",
                                " ssldir=$vardir/ssl\n",
                                " pluginsync=true\n",
                                "[agent]\n",
                                " classfile=$vardir/classes.txt\n",
                                " localconfig=$vardir/localconfig\n"]] },
                            "mode" : "000644",
                            "owner" : "root",
                            "group" : "root"
                        },
                        "/etc/puppet/manifests/site.pp" : {
                            "content" : { "Fn::Join" : ["", [
                                "node basenode {\n",
                                "  package { 'nginx':\n",
                                "    ensure => present\n",
                                "  }\n\n",
                                "  service { 'nginx':\n",
                                "    ensure => running,\n",
                                "    require=> Package['nginx']\n",
                                "  }\n",
                                "}\n",
                                "node /^.*internal$/ inherits basenode {\n",
                                "}\n"
                                ]]
                             }
                            },
                            "mode" : "100644",
                            "owner" : "root",
                            "group" : "wheel"
                        }
                    }
                }
            },
            "Properties" : {
                "InstanceType" : "t2.micro",
                "SecurityGroups" : [ { "Ref" : "NginxGroup" } ],
                "KeyName": { "Ref" : "KeyName" },
                "ImageId": { "Ref" : "AMI" },
                "UserData" : { "Fn::Base64" : { "Fn::Join" : ["", [
"#!/bin/bash\n",
"/opt/aws/bin/cfn-init --region ", { "Ref" : "AWS::Region" },
" -s ", { "Ref" : "AWS::StackName" }, " -r NginxInstance ",
" --access-key ", { "Ref" : "CFNKeys" },
" --secret-key ", { "Fn::GetAtt" : ["CFNKeys", "SecretAccessKey"]}, "\n",
"/usr/bin/puppet apply /etc/puppet/manifests/site.pp", "\n"]]}}
            }
        },
        "NginxGroup" : {
            "Type" : "AWS::EC2::SecurityGroup",
            "Properties" : {
                "SecurityGroupIngress": [
                    {
                        "ToPort": "22",
                        "IpProtocol": "tcp",
                        "CidrIp": "0.0.0.0/0",
                        "FromPort": "22"
                    }
                ],
                "GroupDescription" : "Security Group for managed Nginx"
            }
        }
    },
    "Outputs" : {
     "NginxDNSName" : {
      "Value" : { "Fn::GetAtt" : [ "NginxInstance", "PublicDnsName" ] },
      "Description" : "DNS Name of Nginx managed instance"
     }
    }
}
```

The CloudFormation metadata on the `NginxInstance` resource ensures that the Puppet client package is installed, but that the Puppet agent service is not running. If it were, it would be regularly trying to connect to a nonexistent Puppet master.

The metadata also declares the Puppet configuration files that will be created on the instance. In this example, */etc/puppet/manifests/sites.pp* contains a basic manifest that installs the Nginx web server package and ensures it is running.

One of the Properties, UserData, contains a script that runs puppet apply when the instance launches, applying the configuration manifest stored in *site.pp*.

Creating this stack will launch an EC2 instance that automatically installs and runs Nginx, without any reliance on a Puppet master. Although the Puppet manifests used in the example were basic, they can be expanded upon to build a more complicated infrastructure.

For more information about running Puppet in standalone mode, and other ways of scaling Puppet, see Puppet's documentation.

Building AMIs with Packer

AMI creation is a tedious process that should be automated as soon as possible. Making AMIs manually is slow and error prone, and installing the same packages over and over will soon get tiresome, making some type of configuration management tool a necessity.

This section presents some ways to automate the process of developing and building AMIs.

When starting out with AWS, a lot of people use a simple workflow for creating AMIs: launch an instance, manually install packages and edit configuration files, and then create an AMI (version 1).

To change the AMI, the same process is followed: launch the current version of the AMI, make some configuration changes, and then create a new AMI (version 2).

This is all well and good for getting up and running quickly, but by the time you reach version 10, you have a problem on your hands. Unless you have been meticulously documenting the changes you have made, repeating this process will be difficult. You will have no changelog describing what was changed when, why, and by whom.

This trap should be avoided as soon as possible. You should always use a configuration management tool to help create AMIs so that the process can be easily automated. Automating this process means you can release changes faster, without wasting time manually creating AMIs.

HashiCorp's Packer is a tool for automating the process of creating machine images. It can use various configuration management tools—including Chef, Puppet, and Salt—to create images for several platforms, including AWS. Where AMIs are concerned, we already discussed as the interactive tool of choice, while Packer is by far the most widely preferred automation option.

Packer automates the following processes:

- Launching a new EC2 instance
- Applying a configuration
- Creating an AMI
- Adding tags to the AMI
- Terminating the instance

Once configured, these processes can be performed with a single command. Integrating Packer with continuous integration tools such as Jenkins means you can completely automate the process of creating new AMIs and perhaps release newly created AMIs into your staging environment automatically.

We will use Packer to build an AMI with Nginx installed. Nginx will be installed by Puppet, demonstrating how to use configuration management tools in combination with Packer.

Begin by installing Packer according to its installation instructions. Packer 0.10.1 ships as a single compressed binary on its developer's site:

```
wget https://releases.hashicorp.com/packer/0.10.1/packer_0.10.1_linux_amd64.zip
unzip packer_0.10.1_linux_amd64.zip
```

Once Packer has been installed in your path, create a new directory to work in, containing the subdirectories required for the Puppet manifest files:

```
mkdir packer_example
cd packer_example
mkdir -p puppet/{manifests,modules/nginx/manifests}
```

Create a file named *puppet/manifests/site.pp* with the following contents:

```
node default {
    require nginx
}
```

This will instruct Puppet to apply the Nginx class to any node using this configuration. Next, create a file named *puppet/modules/nginx/manifests/init.pp* with the following contents:

```
class nginx {
    package { 'nginx':
        ensure => present
    }

    service { 'nginx':
        ensure => running
    }
}
```

This is a riff on the theme of our previous manifests, accomplishing the same outcome but using Puppet classes for variety. There are no virtual hosts configured, so Nginx will just display the default welcome page as configured on your operating system.

In this example, we will be using Ubuntu 14.04 per our usual. We alredy demonstrated the use of newer builds than those available in Trusty's repositories in the Ubuntu Archive to sidestep some bugs in Facter's EC2 user data parsing, and we want to maintain this capability. We will use a shell script to add Puppet Lab's *apt* repository and install the most recent version of Puppet.

Create a file named *install_puppet.sh* in our example's directory, containing the same text used in Example 4-4.

We can now move on to the Packer configuration.

It is important to understand two Packer concepts for this section:

Provisioners

> These control which tools will be used to configure the image—for example, Puppet or Chef. Multiple provisioners can be specified in the Packer configuration, and they will each be run sequentially.

Builders

> These are the outputs of Packer. In our case, we are building an AMI. You could also build images for VMware, OpenStack, VirtualBox, and other platforms. By using multiple builders with the same provisioner configuration, it is possible to create identical virtual machines across multiple cloud environments.

Example 4-7 shows the configuration file we will use to build our example AMI.

Example 4-7. Packer example

```
{
  "variables": {
    "aws_access_key": "",
    "aws_secret_key": ""
  },
  "provisioners": [
    {
      "type": "shell",
      "script": "install_puppet.sh"
    },
    { "type": "puppet-masterless",
      "manifest_file": "puppet/manifests/site.pp",
      "module_paths": ["puppet/modules"]
    }
  ],
  "builders": [{
    "type": "amazon-ebs",
    "access_key": "",
    "secret_key": "",
    "region": "us-east-1",
    "source_ami": "ami-c80b0aa2",
    "instance_type": "t2.small",
    "ssh_username": "ubuntu",
    "ami_name": "my-packer-example-",
    "associate_public_ip_address": true
  }]
}
```

In the `provisioners` section, we have our two provisioners: first Packer will run the shell script to install Puppet, and then it will use the `puppet-masterless` provisioner to apply the Puppet manifests.

Next, we have the `builders` section, which contains a single builder object. This contains the information Packer needs to launch an EC2 instance and begin configuring it using the provisioners.

Applying the Puppet configuration to a vanilla installation of the operating system ensures that the instance can be re-created from scratch if necessary. It also ensures that everything required to configure the instance is contained within Puppet's configuration, as any manual changes will be removed the next time the image is made.

The `variables` section provides your AWS credentials to Packer. If these are not hard-coded in the configuration file, Packer will attempt to retrieve the credentials from the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environment variables or the local `~/.aws/credentials` file.

Save the Packer configuration file to *packer_image.json*, after changing the `region` and `source_ami` parameters if desired.

First, we check and validate the configuration file, as Packer runs are too time consuming to be needlessly used as a debugging tool:

```
$ packer validate packer_image.json
Template validated successfully.
```

To create the AMI, execute Packer:

```
$ packer build packer_image.json
amazon-ebs output will be in this color.

==> amazon-ebs: Prevalidating AMI Name...
==> amazon-ebs: Inspecting the source AMI...
==> amazon-ebs: Creating temporary keypair: packer 57cb368f-1381-3d49-e634-
764b35662582
==> amazon-ebs: Creating temporary security group for this instance...
==> amazon-ebs: Authorizing access to port 22 the temporary security group...
==> amazon-ebs: Launching a source AWS instance...
    amazon-ebs: Instance ID: i-a6844bbe
==> amazon-ebs: Waiting for instance (i-a6844bbe) to become ready...
==> amazon-ebs: Waiting for SSH to become available...
==> amazon-ebs: Connected to SSH!
==> amazon-ebs: Provisioning with shell script: install_puppet.sh
...
[output truncated]
...
    amazon-ebs: Notice: Finished catalog run in 8.91 seconds
==> amazon-ebs: Stopping the source instance...
==> amazon-ebs: Waiting for the instance to stop...
==> amazon-ebs: Creating the AMI: my-packer-example-1472935567
    amazon-ebs: AMI: ami-6a5d367d
==> amazon-ebs: Waiting for AMI to become ready...
==> amazon-ebs: Terminating the source AWS instance...
==> amazon-ebs: Cleaning up any extra volumes...
==> amazon-ebs: No volumes to clean up, skipping
==> amazon-ebs: Deleting temporary security group...
==> amazon-ebs: Deleting temporary keypair...
Build 'amazon-ebs' finished.

==> Builds finished. The artifacts of successful builds are:
--> amazon-ebs: AMIs were created:

us-east-1: ami-6a5d367d
```

Packer is quite verbose in its output, so you can follow through as it launches the EC2 instance, runs the shell script, runs Puppet, and finally creates the AMI. In a successful run, the ID of the newly minted AMI will be the closing result of Packer’s output.

Once Packer has finished, you can use the AWS Management Console to verify the existence of your new AMI. If you launch an instance of this image and connect to its public DNS name in your web browser, you will be greeted with the default Nginx welcome page.

This simple example demonstrates how Packer can be used to build AMIs. Packer can significantly reduce the amount of time you spend involved in the AMI building process, and just like our other automation examples, it enables you to entirely rebuild from scratch another key infrastructure component by using just a few files checked out of your trusted version control system.

Automate All the Things

The Chef team successfully co-opted the “all the things” Internet meme to summarize the philosophy of automation we subscribe to. Even if you are not already using configuration management tools, it takes little time to implement an automation strategy that readily pays for itself in saved time and mistakes avoided, to say nothing of cloud-native tasks that are simply ill-suited to manual, step-by-step execution. The AWS console is there to make us comfortable with the cloud environment and help us discover functionality, but it is really not the way AWS was meant to be used.