Extending user-data in Terraform modules

Ash Berlin-Taylor | Mon 28 August 2017 | In terraform

A very common task when launching an instance in AWS is wanting to configure the instance in some way - maybe we want to install a few packages, download and launch the latest release of our project, or perhaps register with Chef or Puppet and continue configuring that way.

One option might be to build an AMI with your desired behaviour baked in as a startup script, but often the parameters to these commands can subtly change from environment-to-environment/region-to-region, or we just don't want to go to the effort of building and maintaining a custom AMI.

When you launch an instance on AWS you can give it a blob of "user-data", up to 16kB of text that can be fetched from on the instance itself via the AWS metadata service at `http://169.254.169.254/latest/user-data` (this URL will only work on an EC2 instance, 169.254.0.0/16 is an IPv4 "link-local" address, meaning it will never be routed over the public internet. Give or take.)

Putting the commands in user data is a good way to not have to 1) log in run them manually (which I try to avoid in almost every circumstan 2) build a custom AMI for each and every combo, or 3) pay AMI storage costs for an open source Terraform module.

If we wanted we could have our own startup task that examines the user-data and performs the actions/commands we specified. Luckily we don't have to write it as it already exists.

It's called **cloud-init**, and it comes pre-installed on almost every AMI. (This is the case for every AMI I've ever run or extended, including Debian, Ubuntu, Amazon Linux, and CentOS. I've yet to find an AMI it didn't come pre-installed on in 6 years of using AWS.)

(cloud-init also works on Azure and Google Cloud (or at least some bits of it? I haven't used GC* myself yet), and many, many more hosting platforms.)

cloud-init understands a few formats of user-data. If the user-data starts with `#!` it is treated as a simple command to run – i.e. put a shell script in your user-data and it will be at boot. For simple cases we could stop there, but we're talking about extensible Terraform modules here.

Another format it understands is `#cloud-config` which is a YAML document that cloud-init processes through various modules – it can write files, run commands (on first boot only or on every boot) and lots more. Nothing that couldn't be done via a shell script, but it's just a slightly more "declarative" syntax and with better error handling.

I'm not going to go into too much detail into what cloud-init can do, as to cover any of it in enough detail would warrant its own post. (And I'm not sure it's interesting to enough people.) If you want a non-trivial example of what cloud-init can do then check out `nat-user-data.conf.tmpl` from tf_aws_nat – this uses cloud-init too install and update packages, write files, and run commands.

Specifying user-data in Terraform

Giving an EC2 instance user-data in Terraform is quite easy. If you want a simple value you can give the `user_data` argument a string literal, but in most cases it's complex enough that you either want to use the `file()` function, or the template_file data source if you need to interpolate values.

For example this (simplified) example taken from the `tf_aws_nat` module will launch an instance with the user-data populated with the contents of the "nat-user-data.conf" file found along side the other .tf files:

```
data "template_file" "user_data" {
  template = "${file("${path.module}/nat-user-data.conf")}"
  vars {
    name = "${var.name}"
  }
}

resource "aws_instance" "nat" {
 # ...
 user_data = "${data.template_file.user_data.rendered}"
}
```

The cloud-config provided in the module is quite powerful, but what if we want to allow the user of the module to also be able to customize the instance on boot? (say they want to install a monitoring agent, or configure user's ssh keys etc.)

Luckily cloud-init can read from multiple user-data sections

Multiple sections of user-data

One of the formats that cloud-init understands is a multi-part MIME archive, (yes MIME as in email. It's the same format used when sending attachments via email) and since Terraform v0.6.9 (released January 8, 2016) it has been able to produce them using the template_cloudinit_config data source:

```
data "template_cloudinit_config" "x" {
  part {
    content = "#cloud-config\n---\nruncmd:\n - date"
  }
  part {
    filename     = "init.sh"
    content_type = "text/x-shellscript"
    content      = "echo Hi\ntouch /root/There\n"
  }
}
```

The “rendered” property of the data source produces something that looks like this:

```
Content-Type: multipart/mixed; boundary="MIMEBOUNDARY"
MIME-Version: 1.0

--MIMEBOUNDARY
Content-Transfer-Encoding: 7bit
Content-Type: text/plain
Mime-Version: 1.0

#cloud-config
---
runcmd:
 - date
--MIMEBOUNDARY
Content-Disposition: attachment; filename="init.sh"
Content-Transfer-Encoding: 7bit
Content-Type: text/part-handler
Mime-Version: 1.0

echo Hi
touch /root/There

--MIMEBOUNDARY--
```

As we can see, we have two parts each with a bit of metadata. (If you are wondering I don’t think the filename has any bearing, it’s purely for our informational purposes.) cloud-init will process each part according it a predefined ordering – certain types first, then for each type the order in which it is defined.

Using this technique we could add a variable to our module so that we can have a complex cloud-config user data defined in the module, and still let

the caller add an extra shell script to be run:

```
variable "instance_boot_script" {
  default = ""
}

data "template_cloudinit_config" "userdata" {
  part {
    content = <<EOF
#cloud-config
---
runcmd:
- [ wget, "http://example.org", -O, /tmp/index.html ]
EOF
  }

  part {
    filename     = "extra.sh"
    content_type = "text/x-shellscript"
    content      = "${var.instance_boot_script}"
  }
}

resource "aws_instance" "instance" {
  # ...
  user_data = "${data.template_cloudinit_config.userdata.rendered}"
}
```

This works, and probably gives enough control as the module caller could do anything they needed via a shell script, but it would be nice if they could also use a cloud-config format. With one extra option to the cloudinit_config data source this is possible.

Merging cloud-init sources

If cloud-init encounters two cloud-config parts it will merge them. Lets say we had the two cloud-config parts, this one in our module

```
run_cmd:
  - bash1
  - bash2
```

and this one provided by the user

```
run_cmd:
  - bash3
  - bash4
```

The intent here is probably to run `bash1`, `bash2`, `bash3`, `bash4` but by default cloud-init will just run `bash3` and `bash4`. This is because cloud-init follows a set of "merge rules", one of which is when it finds a list/array it will replace the content. A few years ago this used to be the only option, and although it is still the default it is now possible to customize the merge behaviour.

There is a page in cloud-init's docs about merging user-data sections manages to give lots of detail whilst at the same time failing to mention what the possible merge settings are, or what results they give. After a bit of searching and trial and error I found a merge behaviour I'm happy with:

```
list(append)+dict(recurse_array)+str()
```

This tells cloud-init that when it encounters a list it should append the new one to the old, when it encounters a list inside a dictionary it should use the previous list rule, and when it encounters a string it should use the default behaviour (replace the old value).

There are a couple of ways cloud-init can be configured, but the easiest for us is to specify this as a MIME header on the extra parts, which Terraform supports.

Putting it all together

Taking everything we've mentioned so far we can turn it into a mostly-complete Terraform module:

```
provider "aws" {}

data "aws_ami" "ami" {
  most_recent = true
  owners      = "099720109477" # Canonical

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-xenial-16.04-amd64-server-*"]
  }

  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }
}

variable "extra_userdata" {
  default     = ""
  description = "Extra user-data to add to the default built-in"
}
variable "extra_userdata_type" {
  default     = "text/cloud-config"
  description = "What format is extra_userdata in - eg 'text/cloud-config' or
}
variable "extra_userdata_merge" {
  default     = "list(append)+dict(recurse_array)+str()"
  description = "Control how cloud-init merges user-data sections"
}

data "template_cloudinit_config" "userdata" {
  part {
    content = "${file("${path.module}/cloud-init.yml")}"
  }

  part {
    filename     = "extra.sh"
    content_type = "${var.extra_userdata_type}"
    content      = "${var.extra_userdata}"
    merge_type   = "${var.extra_userdata_merge}"
  }
}

resource "aws_instance" "instance" {
  ami           = "${data.aws_ami.ami.id}"
  instance_type = "t2.micro"
  user_data     = "${data.template_cloudinit_config.userdata.rendered}"
}
```

I've highlighted where we use the new input variables. This module would default have just the built-in user data, but the caller can provide a va to the `extra_userdata` variable to add their own instance customization, like this:

```
data "aws_region" "current" {
  current = true
}

data "template_file" "extra-userdata" {
  vars {
    region = "${data.aws_region.current.name}"
  }
  # Generally you shouldn't use a heredoc as a template, but it's easier to s
  template = <<EOF
---
runcmd:
- [ 'sh', '-c', 'echo export AWS_DEFAULT_REGION=$${region} >> ~ubuntu/.bashr
EOF
}

module "mymod" {
  source = ...

  extra_userdata = "${data.template_file.extra-userdata.rendered}"
}
```

Nifty.

This post is part 2 of the "Extensible Terraform Modules" series:

1. Patterns for extensible Terraform modules - AMI IDs
2. Extending user-data in Terraform modules (this post)