Infrastructure as Code

by Justin Mitchel

Speed. Agility. Control.

A step-by-step IaC guide for Terraform, Ansible, Puppet, Chef and Salt

linode

EBOOK

IaC

Infrastructure as Code

by

Justin Mitchel

Akamai Technologies

Infrastructure as Code

Justin Mitchel

Published by:
Akamai Technologies
249 Arch Street
Philadelphia, PA 19106

Typesetting: Reba Cooke, Jill McCoach
Cover design: Mitch Donaberger
Special thanks: Andy Stevens, Timothy Ryan, Blair Lyon, Rob Yoegel, Hillary Wilmoth, Maddie Presland, Justin Cobbett, Nathan Melehan and many, many more.

Dedicated to

To my wife, Emilee – Thank you for the love, support,
and encouragement all these years.
You are my guiding light and my everything.

To my kids, McKenna, Dakota, & Emerson – Thank you for the unconditional love and joy you bring to my life. I am proud of the girls you are and the women you will become.

I love you all and am so grateful to live this life with you.
I also can't wait for our next adventure!

Table of Contents

Chapter 1 Introduction

Chapter 1
Introduction

Welcome to Try Infrastructure as Code

Modern Infrastructure as Code (or IaC for short) tools provide a reliable way to maintain the state that you need your infrastructure to be in; it's a document-based accounting for:

- What you need: the number of virtual machines, load balancers, object storage buckets, etc.
- How you need them: Python 3.9, Nginx, Ubuntu 20.04 LTS installed with ACL controls, SSH keys, etc.
- How to scale them (up or down) reliably and predictably. The best part is many of them are cloud provider agnostic and can include on-premise services.

Document-based: most software applications are written with a bunch of documents, this includes collections of Python (`.py`), JavaScript (`.js`), C++ (`.cpp`), Swift (`.swift`), Java (`.java`) or other files. Most programming languages are imperative - this means you write the logic for each step needed to get a result. IaC tools, on the other hand, are declarative - so you write the results you want, regardless of the logic to get there.

Document-based tools allow you to leverage *version control* (as in `git`). Version control is a tool that enables you to track changes in a document over time.

At a glance, you should be able to tell exactly what a document does:

main.yaml

```
---
- hosts: all
  becomte: yes
  tasks:
    - name: Install Nginx
      apt:
        name: nginx
        state: present
        update_cache: yes
```

Above is an example `ansible` document. Can you guess what it does?

main.tf

```
resource “linode_instance” “cfe-pyapp” {
    count = 3
    image = “linode/ubuntu20.04”
    label = “app-${count.index + 1}”
    group = “project”
    region = “us-east”
    type = “g6-nanode-1”
    authorized_keys = [ var.authorized_key ]
    root_pass = var.root_user_pw
    tags = []

}
```

Above is an example `terraform` document. Can you guess what it does?

The above examples help illustrate a point: learning about IaC and IaC tools is important.

Everything as Version Control

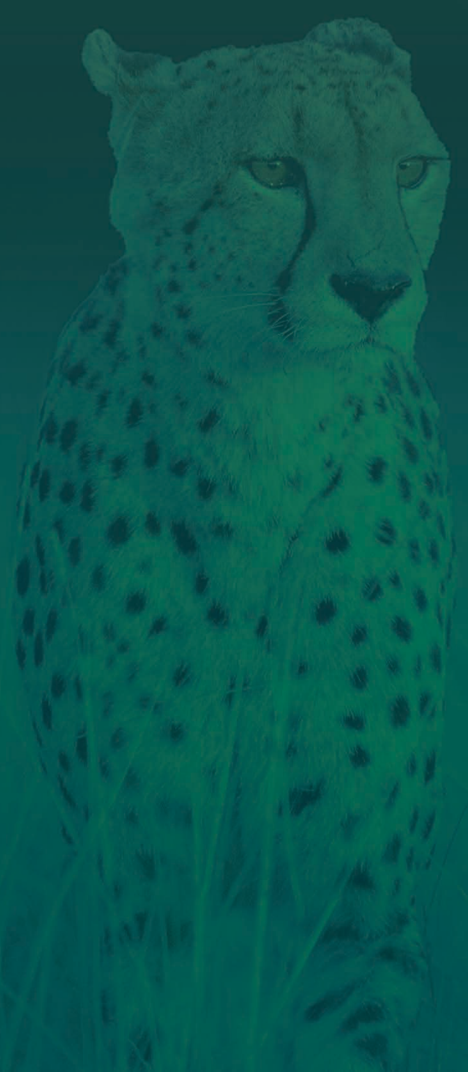
Version control helps address several major issues:

- Accidental code deletion
- Secure & safe sharing of code and secrets
- Secure & safe contribution from people everywhere (internal *and* external)
- Computer(s) damaged / lost / stolen
- Key team member (or employee) leaves the team
- Key team members lack technical skills
- Key services shut down, fail to perform well, or can no longer be used

Version control, in my opinion, is what catapults IaC tools from a niche activity to a mainstream requirement for organizations: managers have granular control of provisioned resources while maintaining strong integrity in the system itself. IaC tools require only a minimal background in software engineering and DevOps (to manage and even build).

Chapter 2

How to Use This Book



Chapter 2

How to use this Book

This book is a step-by-step guide for you to learn how to use some of the most in-demand IaC tools that exist. As of this writing, those tools are:

- Terraform
- Ansible
- Puppet (and Puppet Bolt)
- Chef
- Salt (aka SaltStack)

I recommend you go through the entire book to get a sense of what each technology is all about and how it might fit into what you do. If you're impatient, you can always choose a single tool to use.

Each tool is covered as a mostly stand-alone project that focuses on deploying a simple Docker-based Python web application from Github.

Chapter 3

References

Chapter 3

References

Each IaC tool has official documentation, and so does the code we use in this book. The GitHub repo will have the most up-to-date code for each section unless a major re-write occurs.

Terraform

- GitHub repo: https://github.com/codingforentrepreneurs/iac-terraform
- Official documentation: https://www.terraform.io/docs

Ansible

- GitHub repo: https://github.com/codingforentrepreneurs/iac-ansible
- Official documentation: https://docs.ansible.com/ansible/latest/index.html

Puppet (Puppet Bolt)

- GitHub repo: https://github.com/codingforentrepreneurs/iac-puppet
- Official documentation: https://puppet.com/docs/bolt/latest/bolt.html

Chef

- GitHub repo: https://github.com/codingforentrepreneurs/iac-chef
- Official documentation:
  - Chef Infra Server: https://docs.chef.io/server/
  - Chef Infra: https://docs.chef.io/chef_overview/
  - Chef Workstation: https://docs.chef.io/workstation/

Salt (aka: SaltStack)

- GitHub repo: https://github.com/codingforentrepreneurs/iac-salt
- Official documentation: https://docs.saltproject.io/en/latest/contents.html

Chapter 4

Terraform

Chapter 4
Terraform

Before we jump in, let's look at a fairly straightforward scenario for an app you're working on:
- Three web servers (i.e Gunicorn & Django)
- A load balancer (i.e NGINX)
- A database server (i.e Postgres)

Wouldn't it be nice if you could say "apply" and your cloud provider made that happen for you? Now, let's say a few months go by and you want to add:
- A datastore server (i.e Redis)
- Two microservice servers (i.e FastAPI & Flask)

Again, let's say "apply" and make it happen. Now, after some more time and even more traffic, we decide we want:
- Four web servers (i.e Gunicorn & Django)
- A load balancer (i.e NGINX)
- A database server (i.e Postgres)
- A datastore server (i.e Redis)
- A microservice server (i.e FastAPI)

If you have managed virtual machines from a cloud service provider, you know how common the above scenario is.

Handling these kinds of changes can be done in two ways:
- Manually through the console
- Automatically through Terraform

First, let's pose a few questions about the manual option:
- What if the UI changes and a simple fix takes 5-25 minutes to figure out?
- What if you have several people needing different kinds of compute resources on your team? Do they all log in and provision resources themselves?
- What if while your DevOps guy is unreachable on a 48-hour flight around the earth, you urgently need to reconfigure your environment?
- What if your CEO accidentally turns off all your virtual machines because they assumed they had logged in to a personal account?
- How do you track what resource(s) are you currently using? Through invoices alone? How do you audit said invoices?

We can add a long list of what-ifs here, but the fact of the matter is this:
- Everything that can be code, should be code.
- Everything that you can track through version control, aka Git, should be tracked through version control.
- Everything that you can automate, should be automated.

Terraform has won me over in a big way, not because of Terraform per se, but because the notion of provisioning required cloud resources through a document (like `.yaml`, `.hcl`, `.json`, etc.) is absolutely a win for us all; especially teams that lack true DevOps or Ops people.

Let's look at Terraform and see why.

Install Terraform

All official installation options are here.

macOS via Homebrew

```
brew tap hashicorp/tap
brew install hashicorp/tap/terraform
```

Windows via Chocolatey

```
choco install terraform
```

Linux (Ubuntu / Debian)

```
sudo apt-get update && sudo apt-get install -y gnupg software-properties-common curl
curl -fsSL https://apt.releases.hashicorp.com/gpg | sudo apt-key add -
sudo apt-add-repository “deb [arch=amd64] https://apt.releases.hashicorp.com
$(lsb_release -cs) main”
sudo apt-get update && sudo apt-get install terraform
```

Verify Installation

```
terraform -help
```

Clone Sample Python Web App

```
cd /path/to/your/project/folder/
```

We need to clone the following project:

```
git clone https://github.com/codingforentrepreneurs/iac-python
```

Remove the cloned `.git` repo:

```
rm -rf .git
```

Re-initialize this project as your Git project:

```
git init
git add --all
git commit -m “Initial Project”
```

The key to this sample code is that it is a Docker project.

Create Terraform Root

Create the root directory for our Terraform files (aka `hcl` files)

```
mkdir -p ./devops/tf/
cd ./devops/tf
```

File names are *very important* with Terraform. Here’s what we’ll cover (but not in this order):

- `main.tf`
- `output.tf`
- `terraform.tfvars`
- `variables.tf`
- `backend` (without `.tf`)

Current project structure

```
cd /path/to/your/project/folder/
tree .
.
    .gitignore
    Dockerfile
    README.md
    devops
        tf
    docker-compose.yaml
    entrypoint.sh
    nginx
        Dockerfile
        nginx.conf
    pytest.ini
    pyvenv.cfg
    requirements.txt
    runtime.txt
    src
        __init__.py
        main.py
        test_views.py
```

To install the command `tree`, review Appendix G.

Initialize Terraform for Linode

We need to map our iteration of Terraform with the Linode Provider Docs.
First, create the file `main.tf`:

```
touch main.tf
```

Within, `main.tf` update it to:

```
terraform {
    required_version = “>= 0.15”
    required_providers {
```

```
        linode = {
            source = "linode/linode"
            version = "1.25.0"
        }
    }
}
```

As of now, we have just one file: main.tf. This file lays the foundation for the required providers, which is Linode in this case. There are a lot of providers listed on the Terraform Registry. Some providers are officially supported by HashiCorp (the Terraform maintainers), while other providers are supported directly by the developers who built them.

Linode maintains the Linode Provider for Terraform, which you can see in the registry docs for the Linode Provider.

After you have the above module created, run:

```
terraform init
```

Did you see the message Terraform has been successfully initialized! ? If so, we can move on. If not, do not proceed until you do. My example may use versions that are no longer supported.

Linode API Token and terraform.tfvars

terraform.tfvars is a file to store any secrets Terraform may need, like your API tokens. For a detailed reference on creating a Linode Personal Access Token (aka _API Token_) review Appendix E.

Step 1

Add .tfvars to your .gitignore file (or just copy this reference). If you don't have a .gitignore file, add one to your project now.

Step 2

Login or sign up on linode

Step 3

Navigate to API Tokens

Step 4

Create a `Personal Access Token`

- Label: `PyTerra` (or call it what you want)
- Expiry: `In 6 months` (Choosing `never` is rarely recommended)
- Access:
    - - *Domains*: `Read/Write`
    - - *Events*: `Read/Write`
    - - *Images*: `Read/Write`
    - - *IPs*: `Read/Write`
    - - *Linodes*: `Read/Write`
    - - *Node Balancers*: `Read/Write`
    - - *Object Storage*: `Read/Write`
    - - *Volumes*: `Read/Write`

Step 5

Copy personal access token → if you lose it, generate a new one using steps 1-3. Do ***not*** share this code with anyone.

Step 6

Add this token to a terraform-specific variables file.

```
touch terraform.tfvars
```

In `terraform.tfvars` add:

```
linode_pat_token="<your-personal-access-token-from-step-4>"
```

Be sure to use quotes, like `linode_pat_token="dfa2fa4sfda3377cd25a6054cae10c5dbce33be7c6573829c4602bb-0c78d4be4"`

Step 7

Copy your local SSH Public Key:

View Local SSH Key

```
cat ~/.ssh/id_rsa.pub
```

Don't have an ssh key yet? Use the command `ssh-keygen` or follow Appendix B.

SSH Key Copy Shortcuts

macOS/Linux

```
cat ~/.ssh/id_rsa.pub | pbcopy
```

Windows

```
type ~\.ssh\id_rsa.pub | clip
```

Now in `terraform.tfvars` add:

```
authorized_key = “ssh-rsa your-public-rsa-key value -it should be long”
```

Step 8

Create a `root_user_pw` password for the `terraform.tfvars`

For a detailed guide on generating a password, review Appendix D.

```
#python3
import secrets
print(secrets.token_urlsafe(32))
```

```
root_user_pw=”<your-new-root-user-password>”
```

Your `terraform.tfvars` should be in the directory `devops/tf/terraform.tfvars`, and look something like:

```
linode_pat_token=”dfa2fa4sfda3377cd25a6054cae10c5dbce33be7c6573829c4602bb0c78d4be4”
authorized_key=”ssh-rsa ....truncated.... j@cfe.lan”
root_user_pw=”Er-WROP0OdRa0Aa23ZNJXRPW3t3hLdHA7oYsHqIaqB8”
```

Prepare for a Linode Virtual Machine

Without doing anything else with Terraform, let's see it in action.

Under the terraform {} declaration in main.tf add the following:

```
provider "linode" {
    token = var.linode_pat_token
}

resource "linode_instance" "cfe-pyapp" {
        count = "1"
        image = "linode/ubuntu20.04"
        label = "pyapp-${count.index + 1}"
        group = "CFE-Learner"
        region = "us-east"
        type = "g6-nanode-1"
        authorized_keys = [ var.authorized_key ]
        root_pass = var.root_user_pw
        tags = [ "python", "cfe" ]
}
```

Let's unpack this:

- provider "linode" {token = var.linode_pat_token} : This configures the Linode provider to use the var. linode_pat_token item (we'll discuss variables below).
- resource "linode_instance" : this comes directly from the linode provider we added above
- resource "linode_instance" "cfe-pyapp" : the cfe-pyapp of this statement must be unique across your entire terraform project.
- count = "1" : this will create only 1 instance of this resource. In this case, the resource is a linode_instance with the name cfe-pyapp
- image = "linode/ubuntu20.04" : this Linode image uses ubuntu20.04 . There's a lot of options here that are directly from Linode.
    - *To get the offered Linode image distributions, you can go to this link or run curl https://api.linode.com/v4/images | python3 -m json.tool*
- label = "pyapp-${count.index + 1}" : this is a unique label for each instance of this resource. If count="1" was not on this resource, you would not have access to ${count.index} like we do here.
- group = "CFE-Learner" : Label the group as you wish, CFE-Learner` is arbitrary.
- region = "us-east" : us-east is an official region from Linode.
- type = "g6-nanode-1" : This is the ID of the type of CPU plan you which to provision. g6-nanode-1 is the least expensive to test (at the time of this writing, it's about $5/mo if it runs 24/7 for the whole month).
    - *To get Linode instance type(s) ids, you can go to this link or run curl https://api.linode.com/v4/linode/types | python3 -m json.tool*
- authorized_keys = [var.authorized_key] - authorized_keys refers to actual authorized ssh keys that you want to provision with this. You can set multiple values for this. In our case, we used just var.authorized_

key but you could have [var.authorized_key1, var.authorized_key2, authorized_key3] and so on.
- var.authorized_key : var is a way to access the key/value pairs we stored in terraform.tfvars . So var.linode_pat_token is the personal access token we set for Linode. var.root_user_pw is the admin user password (aka root).
- root_pass = var.root_user_pw : here we set the admin user password. The admin username is typically root .
- tags = ["python", "cfe"] : this is an arbitrary list of tags you can add to this image. These tags are useful when you have a lot of instances running at any time.

Using Variables in Terraform

Create a variables.tf file:

```
touch variables.tf
```

Update variables.tf with:

```
variable “linode_pat_token” {
  sensitive   = true
}

variable “authorized_key” {
  sensitive   = true
}


variable “root_user_pw” {
  sensitive   = true
}
```

To use var.root_user_pw and var.authorized_key , we must create the file variables.tf. While it may seem similar to terraform.tfvars , variables.tf has more configuration options including the ability to set a default value. Adding variables.tf to your repo is necessary, but remember, you should *never* add terraform.tfvars to your Git repo.

Terraform Plan

Now that we have the following setup:

- `main.tf`
- `terraform.tfvars`
- `variables.tf`

We can run:

```
terraform plan
```

If you've set everything up correctly, you should see what Terraform wants to do given our configuration files (all the `.tf` files).

If the results in `terraform plan` look like what you intended, let's run:

Terraform Apply & Destroy

The two simple commands to spin up our infrastructure and take it down are below. I recommend running these commands several times to see how simple it is to control resources with Terraform. We're going to be building on this concept a lot so I want to emphasize, when in doubt just run `terraform destroy` and start over -- it's the best way to learn.

Terraform Apply

Let's create our instance!

```
terraform apply
```

Be sure to type `yes` when prompted.

Terraform Destroy

Let's destroy our instance!

```
terraform destroy
```

Be sure to type `yes` when prompted.

Auto Approve

To avoid typing *yes* each time you run `terraform apply` or `terraform destroy` use these commands:

```
terraform apply -auto-approve
```

or

```
terraform apply -auto-approve -destroy
```

Notice that `terraform destroy` is just a shortcut to writing `terraform apply -destroy`.

Important note: resources you provision accrue costs while they are running. It's a good idea as you learn to *always* run `terraform destroy` on your project.

Return Values with output.tf

How easy was that? You just added and removed an instance with two lines of code.

But something was missing -- what is the IP address of this instance? What if I created more instances? What are those IP addresses? What are the labels? That's where the `output.tf` file comes in:

```
touch output.tf
```

```
output "webapp_first_host" {
  value = "${linode_instance.cfe-pyapp.0.label} : ${linode_instance.cfe-pyapp.0.ip_ad-
dress}"
}
```

Let's break down what we see here:

- `output "webapp_first_host"` : For this module, we specify the ouput name `webapp_first_host` . It must be unique in the module.
- `value = ...` this is setting the value of this output.
- `"${somevar} ${someothervar}"` This is how you can do string substitution using variables. It works a little like `bash` string substitution.
- `linode_instance.cfe-pyapp.0.label` and `linode_instance.cfe-pyapp.0.ip_address` take a bit more explanation:
  - `linode_instance` is the `resource` we defined in `main.tf`
  - `cfe-pyapp` is the name of the `linode_instance` resource we defined in `main.tf`
  - `cfe-pyapp.0` in our `cfe-pyapp` resource, we set `count="1"` . Once you do this, the resource becomes an iterable. In this case, the `0` refers to the zeroth element (aka first iteration) of resources.
  - `cfe-pyapp.0.label` and `cfe-pyapp.0.ip_address` are referring to fields that are generated by the `linode_instance`.

Above we only declared one value. We can also include a loop of *all* the instances created as a result of the `linode_instance` set to anything other than `count="1"` :

```
output "webapp_hosts" {
  value = [for host in linode_instance.cfe-pyapp.*: "${host.label} : ${host.ip_address}"]
}
```

Now let's apply this Terraform configuration:

```
terraform plan
```

You'll see something like:

```
Plan: 1 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + webapp_first_host = (known after apply)
  + webapp_hosts      = [
```

```
      + (known after apply),
    ]
```

Now let's apply this:

```
terraform apply
```

You should see the same results from `terraform plan` before you write `yes`.

After `apply` finishes, you should see the outputs:

```
Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Outputs:

webapp_first_host = "pyapp-1 : 172.104.214.89"
webapp_hosts = [
  "pyapp-1 : 172.104.214.89",
]
```

How cool is this?

Introducing Terraform State

"And then you get on the plane. The pilot, of course, has to always come on the P.A. system. This guy is so excited about being a pilot. He cannot even stand himself, 'Well, I am going to take it up to about 20,000. And then I'm going to make a left by Pittsburgh. And then I'm gonna make a right by Chicago. And then I'm gonna bring down 15,000.' He gives you the whole route and all his moves. We are in the back going, 'Yeah, fine. That's all... You know. Do whatever the hell you gotta do. I don't know. Just end up where it says on the ticket, really."

Jerry Seinfeld

To me, this is a great metaphor for declarative programming like Terraform. You buy the ticket to go to someplace, and the actual logistics of how the plane gets there doesn't matter much to you so long you get to your destination. The pilot, like Terraform, needs to know all the details about how to get there, and how to reverse course if needed.

The same pilot, however, does not care if you're opening a bag of peanuts, or if you are watching a nearly 30-year-old TV show. The pilot does have instructions to pass out the snacks but, the pilot has no idea if you ate them. Once you arrive at your destination, the pilot does not care what you do. When you come back with a new destination and a new ticket (even if it's back home), the pilot's work can begin again.

Terraform provisions your infrastructure:

- How many servers do you need?
- What kind of image?
- How big should it be?
- What labels do you want?
- What tags are you using?
- Do you need to update memory or CPUs?
- Do you need a different OS image?
- Do you need object storage?
- Do you need another service from another cloud provider?
- Do you need to include coworkers' SSH keys?

Terraform gets you to your destination, including installing things *along the way*. Terraform will not install things *after* you reach your destination *unless* you make significant changes to any given resource. In this case, the original resource will be *replaced* with a new one; the old resource will be removed completely.

Here's another way to frame it.

- I need a server. Great, here's Terraform.
- I need a server with Docker installed. Great, here's Terraform.
- I need fifty servers with Docker installed. Great, here's Terraform. Six months later, on those same servers, I need to uninstall Docker without taking down those servers. Oops, that's not Terraform.

Terraform is declarative. You tell it what you want the infrastructure state to be and the tool gets the job done. It's about the result, not the steps to get there.

Python is imperative. You tell the computer what you want it to do when something happens; it's all step-by-step, *you* design what the result should be.

Terraform is declarative, so it needs to manage the current infrastructure state before it knows if changes need to be made. The command `terraform plan` shows you the results of tracking the current state. Let's be clear, Terraform tracks the current state so long you use Terraform to *modify* the state. If you destroy a resource outside Terraform, the state will not be updated automatically. You can update the state in Terraform with `terraform apply -refresh-only -auto-approve`, but we'll leave that for another time.

This idea also extends to what happens within any given terraform provisioned resource. As far as Terraform is concerned, once the infrastructure has been applied successfully (`terraform apply`), its job is done.

Without additional configuration, `terraform apply` will create a `terraform.tfstate` file in your local project. We want to use a cloud-based state file so `terraform apply` runs nearly anywhere.

Create a Linode Object Storage Bucket

For a more detailed look, review Appendix F.

1. Log in to the Linode console
2. Create a new bucket in Object Storage
3. Create Access Keys with read/write access to your created bucket

Create Terraform Backend for Cloud-Based Terraform State

To manage our state file through Linode Object Storage, add a new file called `backend` (no extension):

```
touch backend
```

In our `backend` file we'll add the following:

```
skip_credentials_validation = true
skip_region_validation = true
bucket=”yourbucket”
key=”your-terraform.tfstate”
region=”us-southeast-1”
endpoint=”us-southeast-1.linodeobjects.com”
access_key=”your_access_key”
secret_key=”your_secret_key”
```

Let's break this down:

- `skip_credentials_validation` and `skip_region_validation` are fundamentally for a service different than Linode; that's why we will skip them.
- `bucket` is the name of the bucket you created in Step 12.
- `key` is how you want to store your state file. I have a bucket just for my Terraform project.
- `region` is the region your bucket was created in Step 12.
- `endpoint` is the endpoint for your bucket; it's typically the region id and `linodeobjects.com` (on Linode, you can remove the `bucket` name from this endpoint.
- `access_key` is the public key to access your bucket
- `secret_key` is the secret key to access your bucket. Keep this safe.

Update gitignore

```
echo “backend” >> .gitignore
```

Important note: It's critical to keep secret keys out of version control. The `backend` file is essentially a file we need to keep hidden.

Update main.tf to Include:

```
terraform {
    required_version = “>= 0.15”
    required_providers {
        linode = {
            source = “linode/linode”
            version = “1.22.0”
        }
    }
    backend “s3” {
        skip_credentials_validation = true
        skip_region_validation = true
    }
}
```

Initialize our New Backend

```
terraform init -backend-config=backend
```

Pro-tip

You can also run `terraform -chdir=./devops/tf init -backend-config=backend` where `-chdir=./devops/tf` allows you to declare where the root of your terraform project is.

After we run this command, terraform will use a cloud-based state file. This is great because we can now share this terraform project or use it in a CI/CD pipeline or across multiple machines while retaining the correct state of the entire Terraform project.

Provisioning with Terraform for Docker

Terraform can perform remote execution as well as push files into your resources. In our case here, we'll use Terraform to push our code, including docker files, into production as well as run all commands needed to get the machine fully running.

Terraform can perform these tasks but it will *not* do it as well as other tools. You'd probably use Terraform with something like Ansible or SaltStack to ensure your infrastructure is fully provisioned.

Terraform's docs state plainly that **Provisioners are a Last Resort**. That said, we're going to use them.

No, we're not using them to *stick it to the man*, no we're using them because, in our case, we're mostly bootstrapping our resource so we need to provision it accordingly.

What does this look like exactly?

- Update our Ubuntu system packages (`apt-get update`)
- Install Docker & docker-compose dependencies
- Upload our non-public facing code (ie not a public repo on GitHub)
- Do the above *only* on creation AND prepare for integration with Ansible for long term management (not in this guide)

First, let's do a simple command to update our system:

In `main.tf` let's update the following resource:

```
resource “linode_instance” “cfe-pyapp” {
        count = “1”
        image = “linode/ubuntu20.04”
        label = “pyapp-${count.index + 1}”
        group = “CFE-Learner”
        region = “us-east”
        type = “g6-nanode-1”
        authorized_keys = [ var.authorized_key ]
        root_pass = var.root_user_pw
        tags = [ “python”, “cfe” ]

        provisioner “remote-exec” {
             connection {
                host     = “${self.ip_address}”
                type     = “ssh”
                user     = “root”
                password = “${var.root_user_pw}”
             }

             inline = [
                 “sudo apt-get update”,
                 “curl -fsSL https://get.docker.com -o get-docker.sh”,
                 “sudo sh get-docker.sh”
             ]
        }
}
```

Let's breakdown the provisioner block:

- provisioner "remote-exec" this is a way to execute a command on a remote host
- connection : how should this provisioner execute this command?
  - host : using ${self.ip_address} will be autoset for us when we run terraform apply
  - type = "ssh" : we use ssh (aka secure shell). We set the SSH key on authorized_keys = [var.authorized_key]
  - user = "root" : Which use do we want to SSH in with? ie ssh@myip
  - password = "${var.root_user_pw}" We set the orginal key on root_pass = var.root_user_pw
- inline = [] : This is a list of commands you can write, called in order, that you want this remote-exec provisioner to call.

Now run:

```
terraform apply
```

Review the plan and type yes

Did Docker Install?

Now use a secure shell to see if Docker was installed successfully. This book is not about Docker, but luckily for those of us who aren't familiar with Docker, a simple way to check if its installed is by running:

```
docker ps
```

The result should be:

```
CONTAINER ID    IMAGE     COMMAND    CREATED    STATUS     PORTS     NAMES
```

And not command not found: docker or something similar.

Provision with Scripts

The `inlines` from above are *fine* but often not very reusable or testable. That's why it's a good idea to use a `bash` script instead.

Create a file called `bootstrap-docker.sh` next to `main.tf` with the contents:

```
#!/bin/bash

sudo apt-get update
sudo apt-get install git curl -y
curl -fsSL https://get.docker.com -o get-docker.sh
sudo sh get-docker.sh
```

If you're using a Linux workstation (like in Appendix C), you should be able to run this script with: `chmod +x bootstrap-docker.sh && sudo sh bootstrap-docker.sh`

Now we can use this script instead of the `inlines` we had above. Update `main.tf` with:

```
resource “linode_instance” “cfe-pyapp” {
        count = “1”
        image = “linode/ubuntu20.04”
        label = “pyapp-${count.index + 1}”
        group = “CFE-Learner”
        region = “us-east”
        type = “g6-nanode-1”
        authorized_keys = [ var.authorized_key ]
        root_pass = var.root_user_pw
        tags = [ “python”, “cfe” ]


        provisioner “file” {
            connection {
                host = “${self.ip_address}”
                type = “ssh”
                user = “root”
                password = “${var.root_user_pw}”
            }
```

```
            source = "bootstrap-docker.sh"
            destination = "/tmp/bootstrap-docker.sh"
        }


        provisioner "remote-exec" {
             connection {
                host     = "${self.ip_address}"
                type     = "ssh"
                user     = "root"
                password = "${var.root_user_pw}"
             }
             inline = [
                 "chmod +x /tmp/bootstrap-docker.sh",
                 "sudo sh /tmp/bootstrap-docker.sh",
             ]
        }
}
```

This introduces the `file provisioner` and is a simple way to copy a local file or many local files into your instance(s) that you're provisioning with Terraform.

After using `file provisioner` we use `remote-exec` to execute our copied file. Pretty neat right?

Terraform Locals

Before we continue, let's consider our current project structure:

```
tree
    .gitignore
    Dockerfile
    README.md
    devops
        tf
            backend
            bootstrap-docker.sh
            main.tf
            output.tf
            terraform.tfstate
            terraform.tfvars
            variables.tf
```

```
    docker-compose.yaml
    entrypoint.sh
    nginx
        Dockerfile
        nginx.conf
    pytest.ini
    pyvenv.cfg
    requirements.txt
    runtime.txt
    src
     __init__.py
        main.py
        test_views.py
```

What we want to do here is include the following to each of our terraformed instance(s):

- `src/` (entire directory)
- `requirements.txt`
- `Dockerfile`
- `entrypoint.sh`

However, `main.tf` exists in `devops/tf/`, not in the root of our project (like next to `requirements.txt`).

We can use `locals` to help solve this issue. So in `devops/tf/` create a new file called `locals.tf` with the contents:

```
locals {
    root_dir = "${abspath(path.root)}"
    devops_dir = "${dirname(local.root_dir)}"
    project_dir = "${dirname(local.devops_dir)}"
    templates_dir = "${local.root_dir}/templates/"
}
```

Let's break this down:

- `path.root` is the relative path to your Terraform project based on where this module exists (ie where `locals.tf` exists in this case).
- `root_dir` is an example variable name (ie you can name it what you want) but it's meant to represent the absolute path (`abspath()`) to `root` of the Terraform directory (ie `/Users/cfe/dev/try-terraform/devops/tf` in my case and *not* just `devops/tf`). This path will be set correctly even if we move our project, as long as your `locals.tf` module is next to `main.tf`.
- `devops_dir` is the parent directory name for the `root_dir` path variable we set above. `dirname()` can be chained together too just like what we see in `project_dir`
- `project_dir` is the root of the entire project. Another way to write this would be `${dirname(dirname(abspath(path.root)))}`

- templates_dir is something we have yet to implement but it's just another path that considers the above steps.

If you're a Python developer, abspath and dirname are very similar to os.path.abspath and os.path.dirname .

Now, anywhere in your terraform files you can reference:

```
local.root_dir
```

or

```
local.project_dir
```

To get the respective directories. Keep in mind that a trailing slash will be absent.

Built-in Terraform Functions

Above we used the built-in terraform functions for abspath and dirname. These built-in functions allow us to limit the amount of hard-coding as much as possible. Instead of having a bunch of variables in terraform.tfvars we can use a number of the built-in functions.

Copy Directories to Instances

Now that we have locals.tf let's update main.tf to handle directories relative to our terraform files.

```
resource “linode_instance” “cfe-pyapp” {
        ...


        provisioner “file” {
            ...
            source = “${local.root_dir}/bootstrap-docker.sh”
            ...
        }



        provisioner “remote-exec” {
```

```
            ...
        }
}
```

Going forward, I'll use `...` to signify lines in `main.tf` that have remained unchanged.

Now, using `file` provisioners we can also upload a directory like:

```
resource "linode_instance" "cfe-pyapp" {
        ...


        provisioner "file" {
            ...
        }

        provisioner "remote-exec" {
             ...
             inline = [

                 "chmod +x /tmp/bootstrap-docker.sh",
                 "sudo sh /tmp/bootstrap-docker.sh",
                 "mkdir -p /var/www/src/",
             ]
        }
        provisioner "file" {
            connection {
                host = "${self.ip_address}"
                type = "ssh"
                user = "root"
                password = "${var.root_user_pw}"
            }
            source = "${local.project_dir}/src/"
            destination = "/var/www/src/"
        }
}
```

So uploading directories is as simple as just providing a path to it. The `destination` must exist (at least the parent directory) or it *may* not copy files correctly.

Now let's finish adding all of the required files:

```
resource “linode_instance” “cfe-pyapp” {
        ...

        provisioner “file” {
            ...
        }

        provisioner “remote-exec” {
             ...
        }
        provisioner “file” {
            ...
        }
        provisioner “file” {
            connection {
                host = “${self.ip_address}”
                type = “ssh”
                user = “root”
                password = “${var.root_user_pw}”
            }
            source = “${local.project_dir}/Dockerfile”

            destination = “/var/www/Dockerfile”
        }
        provisioner “file” {
            connection {
                host = “${self.ip_address}”
                type = “ssh”
                user = “root”
                password = “${var.root_user_pw}”
            }
            source = “${local.project_dir}/entrypoint.sh”
            destination = “/var/www/entrypoint.sh”
        }
        provisioner “file” {
            connection {
                host = “${self.ip_address}”
                type = “ssh”
                user = “root”
                password = “${var.root_user_pw}”
            }
```

```
            source = "${local.project_dir}/requirements.txt"
            destination = "/var/www/requirements.txt"
        }
}
```

The Biggest Terraform Flaw (in my opinion)

Terraform is great at provisioning the resources you need and configuring their initial state (i.e. we added a bunch of files, executed a few scripts, and installed required packages).

This is great, but it leaves a little something to be desired: instance configuration.

What I mean by this is if our web application code changes and the configuration requirements for any given virtual machine instance change, it would be great if `terraform apply` could handle those changes... but it doesn't. This, to me, is one of Terraform's biggest flaws but it can be solved *very* simply: Ansible. In the Ansible Chapter I'll show you exactly how to make Terraform & Ansible work together. In the meantime, we'll use Terraform to continue to handle the configuration of our resources.

This is also why I believe the Terraform docs mention that **Provisioners are a Last Resort**.

Docker & Terraform

In this section, we're going to build a Docker container and run it - all with Terraform. Before we do, I want to mention that it would be better practice to use a Docker Container Registry of some kind and use a CI/CD tool to build our container images. I am not doing this so I can keep this project self-contained and limit the complexity that *can* go into a project like this.

The reason we use Docker in the first place is so that we can deploy nearly *any* Docker container which means we can deploy nearly *any* web application on *any* tech stack (such as Python, JavaScript/Node, Ruby, Nginx, etc) with few exceptions.

Let's update `main.tf` to build and run our container image. For more details on how this works check out Appendix H.

```
resource "linode_instance" "cfe-pyapp" {
        ...
```

```
        provisioner “file” {
            ...
        }

        provisioner “remote-exec” {
             ...
        }
        provisioner “file” {

            ...
        }
        provisioner “file” {
            ...
        }
        provisioner “file” {
            ...

        }
        provisioner “file” {
            ...
        }
        provisioner “remote-exec” {
            connection {
                host = “${self.ip_address}”
                type = “ssh”
                user = “root”
                password = “${var.root_user_pw}”
            }
            inline = [
                “cd /var/www/”,
                “docker build -f Dockerfile -t pyapp-via-git . “,
                “docker run --restart always -p 80:8001 -e PORT=8001 -d pyapp-via-git”
            ]
        }
}
```

Adding this final `remote-exec` provisioner will add a significant amount of time since we have a `docker build` command within, but it should work. If you’re having trouble with this step, you should *omit it* and attempt it manually first. That is what I did, and I recommend you do the same.

Terraform Apply

Let's first destroy all previously created resources with:

```
terraform apply -destroy -auto-approve
```

This will automatically take down everything we have done up until this point (assuming you run the command in the `devops/tf` directory).

Now, let's run;

```
terraform apply -auto-approve
```

You should see something like:

```
Plan: 1 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + webapp_hosts = [
      + (known after apply),
    ]
linode_instance.cfe-pyapp[0]: Creating...
```

Then you should see:

```
linode_instance.cfe-pyapp[0]: Creating...
linode_instance.cfe-pyapp[0]: Still creating... [10s elapsed]
linode_instance.cfe-pyapp[0]: Still creating... [20s elapsed]
linode_instance.cfe-pyapp[0]: Still creating... [30s elapsed]
linode_instance.cfe-pyapp[0]: Still creating... [40s elapsed]
linode_instance.cfe-pyapp[0]: Provisioning with 'file'...
linode_instance.cfe-pyapp[0]: Still creating... [50s elapsed]
```

You'll also see things like:

```
linode_instance.cfe-pyapp[0] (remote-exec): + sh -c curl -fsSL “https://download.docker.
com/linux/ubuntu/gpg” | gpg --dearmor --yes -o /usr/share/keyrings/docker-archive-keyring.
gpg
linode_instance.cfe-pyapp[0] (remote-exec): + sh -c echo “deb [arch=amd64 signed-by=/usr/
share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu focal
stable” > /etc/apt/sources.list.d/docker.list
linode_instance.cfe-pyapp[0] (remote-exec): + sh -c apt-get update -qq >/dev/null
linode_instance.cfe-pyapp[0] (remote-exec): + sh -c DEBIAN_FRONTEND=noninteractive apt-get
install -y -qq --no-install-recommends  docker-ce-cli docker-scan-plugin docker-ce >/dev/
null
```

Which shows the installation process for our Terraform instance provisioners. All part of the normal process. The whole process should take 5 to 10 minutes. After it's done, you should have a newly created Linode instance running a Docker-based python web application.

After it's all said and done, we should see:

```
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

webapp_first_host = “pyapp-1 : 173.255.226.85”
webapp_hosts = [
  “pyapp-1 : 173.255.226.85”,
]
```

Naturally, `pyapp-1` will have a different IP address for you.

Open your browser and navigate to your IP address listed in `pyapp-1`. Mine results in:

Congratulations! You have successfully deployed a Docker-based Python web application using Terraform and Linode! Ready for some more?

Adding Instances

Adding instances could not be easier. We just up our instance count in `main.tf` from

```
resource "linode_instance" "cfe-pyapp" {
        count = "1"
        ...
}
```

to

```
resource "linode_instance" "cfe-pyapp" {
        count = "5"
        ...
}
```

I put `count = "5"` to illustrate an example. You can put however many you would like. After you update count, go ahead and run:

```
terraform apply -auto-approve
```

After you do, you should see:

```
Changes to Outputs:
  ~ webapp_hosts = [
        "pyapp-1 : 173.255.226.85",
      + (known after apply),
      + (known after apply),
      + (known after apply),
      + (known after apply),
    ]
linode_instance.cfe-pyapp[3]: Creating...
```

How amazing is this? No matter how many times you change the count, Terraform will easily be able to provision instances for you. Each instance is provisioned asynchronously to save time and speed up deployment but, there are at least two things to consider to improve build/deployment speed:

- External Docker Container Builder & Registry
- Create a reusable image in Linode (much like how we are using the pre-built image `"linode/ubuntu20.04"`) that only needs to be spun up.

When it's done, you should have the following (or something like it):

```
Apply complete! Resources: 4 added, 0 changed, 0 destroyed.

Outputs:

webapp_first_host = "pyapp-1 : 173.255.226.85"
webapp_hosts = [
  "pyapp-1 : 173.255.226.85",
  "pyapp-2 : 172.104.211.236",
  "pyapp-3 : 97.107.128.102",
  "pyapp-4 : 97.107.129.113",
  "pyapp-5 : 173.255.236.216",
]
```

To handle five instances, we'll implement a Node Balancer in a future step.

Should Instance Count be in .tfvars?

Before we answer this question, let's see how we would update files for instance to count as a variable.

First, we'd update `terraform.tfvars`:

```
...
py_app_count=3
```

Then we'd update `variables.tf` perhaps even with a default value:

```
...
variable "py_app_count" {
  default   = 1
}
```

And now finally update `main.tf` with:

```
resource "linode_instance" "cfe-pyapp" {
        count = "${var.py_app_count}"
        ...
}
```

After we make these changes, we can just run:

```
terraform plan
```

It should be worth mentioning that you can save the output of `terraform plan` and then run `terraform apply` on that saved output. We'll save doing so for another time.

Let's see if our expectation matches what we changed. In my case, I see:

```
Plan: 0 to add, 0 to change, 2 to destroy.

Changes to Outputs:
  ~ webapp_hosts = [
        # (2 unchanged elements hidden)
        "pyapp-3 : 97.107.128.102",
      - "pyapp-4 : 97.107.129.113",

      - "pyapp-5 : 173.255.236.216",
    ]
```

Because I went from a count of five to three.

Now we can apply this plan:

```
terraform apply -auto-approve
```

After a short duration I should see:

```
Apply complete! Resources: 0 added, 0 changed, 2 destroyed.

Outputs:

webapp_first_host = "pyapp-1 : 173.255.226.85"
webapp_hosts = [
  "pyapp-1 : 173.255.226.85",
  "pyapp-2 : 172.104.211.236",
  "pyapp-3 : 97.107.128.102",
]
```

It's probably not surprising that deleting resources is significantly faster than provisioning them, especially in our case.

Provision Node Balancers with Terraform

Since we added three new instances in the last step, we'll now add a load balancing service by Linode called a Node Balancer.

First, add the `linode_nodebalancer` resource.

```
resource “linode_nodebalancer” “pycfe_nb” {

    label = “pycfe-nodebalancer”
    region = “us-east”
    client_conn_throttle = 20

    depends_on = [
        linode_instance.cfe-pyapp
    ]
}
```

Before we go any further, note that the `region` is the same for the `linode_nodebalancer` and the `linode_instance` resource we set above; the region must be the same. Do you think it would be a good idea to turn this region into a reusable variable? Try it now.

Now that we have the `linode_nodebalancer` resource, we need to add the default configuration for this node balancer with:

```
resource “linode_nodebalancer_config” “pycfe_nb_config” {
    nodebalancer_id = linode_nodebalancer.pycfe_nb.id
    port = 80
    protocol = “http”
    check = “http”
    check_path = “/”
    check_interval = 35
    check_attempts = 15
    check_timeout = 30
    stickiness = “http_cookie”
    algorithm = “source”
}
```

Let's break this down:

- nodebalancer_id this references the linode_nodebalancer resource we defined above.
- port = 80 PORT 80 is the standard port for HTTP access (standard web traffic).
- protocol - the options are HTTP, HTTPS and TCP. If you set HTTPS you must include TLS/SSL certificates (which is outside the scope of this chapter).
- check, check_interval, check_attempts, and check_timeout are all active health checks of this node_balancer service.
- check_path is the path health checks will occur.
- stickiness I choose http_cookie for web applications (read more on Linode's docs here.
- algorithm you have three options for this attribute on how your node balancer will route traffic:
  - roundrobin: this will rotate connections between nodes one by one.
  - leastconn: this assigns connections to the backend with the least connections.
  - source: this uses the client's IPv4 address.

Now, our node balancer needs to be configured for the actual instance(s) it will use. Before we can configure it, we need to ensure that our linode_instance (s) have a private IP address.

Let's revisit our linode_instance configuration:

```
resource "linode_instance"  "cfe-pyapp" {
        count = "${var.py_app_count}"
        image = "linode/ubuntu20.04"
        label = "pyapp-${count.index + 1}"
        group = "CFE-Learner"
        region = "us-east"
        type = "g6-nanode-1"
        authorized_keys = [ var.authorized_key ]
        root_pass = var.root_user_pw
        tags = [ "python", "cfe" ]
}
```

Now just add private_ip = true as a new argument:

```
resource "linode_instance" "cfe-pyapp" {
    ...
    private_ip = true
    ...
}
```

Now run terraform apply before continuing any further. I had to run terraform destroy to start fresh with these nodes.

If our linode_instance (s) are missing private_ip = true we will be unable to attach them to a node_balancer . private_ip is set to false by default.

Now, let's add our linode_nodebalancer node configuration to main.tf :

```
resource "linode_nodebalancer_node" "pycfe_nb_node" {
    count = var.py_app_count
    nodebalancer_id = linode_nodebalancer.pycfe_nb.id
    config_id = linode_nodebalancer_config.pycfe_nb_config.id
    label = "pycfe_node_pyapp_${count.index + 1}"
    address = "${element(linode_instance.cfe-pyapp.*.private_ip_address, count.index)}:80"
    weight = 50
    mode = "accept"
}
```

Before we go line by line, we'll look at the address argument.

Start with 1 linode_instance and assume it has the following attributes:

- Terraform resource name is cfe-pyapp
- Resource label is pyapp-1
- private_id_address is available and set to 192.168.156.59
- Running application exposed at PORT 80

A hardcoded look at this instance's linode_nodebalancer_node address argument would be:

```
address = "192.168.156.59:80"
```

This helps us understand how ${element() } works. ${element()} is a simple way to combine Terraform resource information at run time at a specific index value.
Take a closer look at ${element(linode_instance.cfe-pyapp.*.private_ip_address, count.index)}

- linode_instance.cfe-pyapp.*.private_ip_address does the following:
    - linode_instance : references this resource value
    - cfe-pyapp : references the resource with this name
    - * is a wildcard value; we want to replace this with an index value
    - private_ip_address is now available from this resource because of private_ip = true being set before
- count.index : since we have count = var.py_app_count terraform will automatically iterate over this resource configuration and count.index will be set at runtime.
- element will replace the wildcard value * with the count.index value.

After the configuration is complete, you can run terraform plan to see exactly how this renders out.

Now, we break down the other arguments to `linode_nodebalancer_node`:

- `count` -- this is the number of instances you'd like to make as nodes. In my case, I am matching the number of `linode_instance` `count` with the `linode_nodebalancer_node` `count` by using the variable we added in previous steps.
- `nodebalancer_id` - this configuration references the `linode_nodebalancer` resource we defined above and used in `linode_nodebalancer_config` as well.
- `config_id` is a reference to the `linode_nodebalancer_config` we created before.
- `label` - this is how we label this node. It's recommended to use `${count.index}` as a part of this label to ensure each label is unique. Just like with the `linode_instance`, `${count.index}` is only available when you declare the count argument (like we did with `count = var.py_app_count`.
- `weight = 50` Nodes with a higher weight will receive more traffic; values are between 1-255.
- `mode = "accept"` this means the node can accept connections. Other options include `reject`, `drain`, and `backup`.

Using Templates with Terraform

Terraform has a straightforward way of using templates. It's as simple as:

```
templatefile("path/to/template.tpl", { context="value" })
```

The `templatefile` function takes in a path to the `.tpl` file along with a context dictionary (key/value pairs).

Here's a look at a simple template replacement:

templates/sample-template.tpl

```
Hello ${name},

Here are the items you ordered:

%{ for item in items ~}
- ${item}
%{ endfor ~}
```

And the template function:

```
templatefile("templates/sample-template.tpl", { name = "Justin", items=[ "Camera", "Smart-
phone", "Coffee" ] })
```

The template renders the following:

```
<<EOT
Hello Justin,

Here are the items you ordered:
- Camera
- Smartphone
- Coffee

EOT
```

You can try this command out with:

```
terraform console
```

Just make sure that you store your sample template in a template directory named `templates` and a template file name as `sample-template.tpl`.

We should be able to update our command (because of `locals.tf`) to:

```
templatefile("${local.templates_dir}/sample-template.tpl", { name = "Justin", items=[
"Camera", "Smartphone", "Coffee" ] })
```

Terraform Console

If you set up your project correctly, running `terraform console` brings up an interactive console where you can practice all of the commands rendered by functions like `templatefile`. You can also run loops like `[for host in linode_instance.cfe-pyapp.*: "${host.label} : ${host.ip_address}"]`

Just navigate to the root of your Terraform project and run:

```
terraform console
```

This will give you access to your local Terraform project where you can run:

```
[for host in linode_instance.cfe-pyapp.*: "${host.label} : ${host.ip_address}"]
```

or

```
[for host in linode_instance.cfe-pyapp.*: “${host.ip_address}”]
```

And other versatile commands.

Create a local_file resource using Templates

In this example, we create an Ansible inventory file. Terraform is about provisioning resources, Ansible configures them going forward (this includes accounting for various state changes that may occur). More on Ansible in the Ansible Chapter.

First, we design the template. We'll put it in the same location as `local.templates_dir` from `locals.tf` so:

templates/ansible-inventory.tpl

```
%{ for host in hosts ~}
${host}
%{ endfor ~}
```

Now that we have this template file let's implement a `local_file` resource in `main.tf`

```
resource “local_file” “ansible_inventory” {
    content = templatefile(“${local.templates_dir}/ansible-inventory.tpl”, { hosts=[for host in linode_instance.cfe-pyapp.*: “${host.ip_address}”] })
    filename = “${local.devops_dir}/ansible/inventory.ini”
}
```

Enter the `terraform console` and test the new `templatefile` argument we created for this new file. If you haven't fully configured your file, you should see an error.

When we run `terraform plan` we should now see:

```
  Error: Could not load plugin


  Plugin reinitialization required. Please run “terraform init”.
...
```

We're seeing this because we added a new resource that we haven't used yet: `local_file`. You'll see this every time you use a resource for the first time.

Re-initialize the project with your backend using:

```
terraform init -backend-config=backend
```

Run `terraform plan` and you should see:

```
Terraform will perform the following actions:

  # local_file.ansible_inventory will be created
  + resource "local_file" "ansible_inventory" {
      + content              = <<-EOT
            173.255.226.85
            45.33.64.242
            45.79.128.237
        EOT
      + directory_permission = "0777"
      + file_permission      = "0777"
      + filename             = "/Users/myuser/Dev/myproject/devops/ansible/inventory.ini"
      + id                   = (known after apply)

    }

Plan: 1 to add, 0 to change, 0 to destroy.
```

The `content` section should match what you tested in `terraform console` above and the `filename` will be an absolute path on your local system.

Now run `terraform apply`.

Once you do this, it will create the file for you in `devops/ansible/inventory.ini`. Using your machine **delete** `devops/ansible/inventory.ini` manually. Run `terraform apply` again. Now, add some random data within the `devops/ansible/inventory.ini` file. Run `terraform apply` again.

Another huge advantage of using Terraform is that it ensures the state of your resource(s) matches what you declare in your Terraform project, including local files.

If you need to change the `devops/ansible/inventory.ini` update `templates/ansible-inventory.tpl` and/or the `resource "local_file" "ansible_inventory"` configuration.

Using templates removes any ambiguity that arises from creating local files with Terraform.

In the Ansible section, you'll learn more about how to use this inventory file.

Terraform & GitHub Actions

After getting some practice with Terraform, you should try implementing the workflow automation tool GitHub Actions.

"GitHub Actions makes it easy to automate all your software workflows, now with world-class CI/CD. Build, test, and deploy your code right from GitHub."

So how do we use Terraform with GitHub Actions?

These instructions assume that your project is already available on GitHub and that you are using the Terraform `gitignore`. Learning how to use Git is outside the context of this book.

Step 1: Backend & Repo Secrets

To use GitHub Actions, we **must** have a cloud-based Terraform backend setup (possibly through Linode Object Storage) or the *state* of our Terraform project will be unusable.

Remember the following files should *never* be checked into Git or publicly exposed:

- `.tfstate`
- `backend`
- `terraform.tfvars`

Here's what our `backend` file contains:

```
skip_credentials_validation = true
skip_region_validation = true
bucket = "your-bucket"
key = "try-iac-book.tfstate"
region = "us-southeast-1"
endpoint = "us-southeast-1.linodeobjects.com"
access_key = "your-key"
secret_key = "your-secret"
```

The configuration values to keep hidden are:

- `bucket`
- `key`
- `access_key`
- `secret_key`

Add these to your repo's secrets:

- Navigate to your repo on GitHub
- Click on `Settings` (your URL should look similar to `https://github.com/codingforentrepreneurs/iac-terraform/settings`)
- Click on `Secrets`
- For each `configuration value`, click `New repository secret` and store them like:
  - `TERRAFORM_BUCKET_NAME` (for `bucket`)
  - `TERRAFORM_STATE_KEY` (for `key`)
  - `LINODE_OBJECT_STORAGE_ACCESS_KEY` (for `access_key`)
  - `LINODE_OBJECT_STORAGE_SECRET_KEY` (for `secret_key`)

Step 2: `terraform.tfvars` & Repo Secrets

Just like `backend` we're going to add the values from `terraform.tfvars` to your repo secrets.

Here's our `terraform.tfvars`

```
linode_pat_token = "your_personal_access_otken"
authorized_key = "your_ssh_pub_key"
root_user_pw = "your_default_root_user_pw"
py_app_count = 3
```

We'll put each one of these values into our GitHub repo secrets as:

- `LINODE_PA_TOKEN` (for `linode_pat_token`)
- `SSH_PUB_KEY` (for `authorized_key`)
- `ROOT_USER_PW` (for `root_user_pw`)
- `PYAPP_NODE_COUNT` (for `py_app_count`)

Step 3: Your Terraform Workflow

At this point, we should have the following values stored in our GitHub repo secrets:

- `TERRAFORM_BUCKET_NAME`
- `TERRAFORM_STATE_KEY`
- `LINODE_OBJECT_STORAGE_ACCESS_KEY`
- `LINODE_OBJECT_STORAGE_SECRET_KEY`
- `LINODE_PA_TOKEN`
- `SSH_PUB_KEY`
- `ROOT_USER_PW`
- `PYAPP_NODE_COUNT`

To use these secrets, we'll do `${{ secrets.TERRAFORM_BUCKET_NAME }}` in our workflow file(s) as you'll see below.

Start by creating the necessary GitHub Actions folders at the root of your project:

```
mkdir -p .github
mkdir -p .github/workflows/
```

Now, create your workflow file:

.github/workflows/apply-terraform.yaml

```
name: Apply Infrastructure via Terraform

on:
  workflow_dispatch:
  push:
     branches: [main]

jobs:
  terraform:
    runs-on: ubuntu-latest
    steps:

      - name: Checkout
        uses: actions/checkout@v2

      - name: Setup Terraform
        uses: hashicorp/setup-terraform@v1
        with:
          terraform_version: 1.1.4
      - name: Add Terraform Backend for S3
        run: |
          cat << EOF > devops/tf/backend
          skip_credentials_validation = true
          skip_region_validation = true
          bucket = "${{ secrets.TERRAFORM_BUCKET_NAME }}"
          key = "${{ secrets.TERRAFORM_STATE_KEY }}"
          region = "us-southeast-1"
          endpoint = "us-southeast-1.linodeobjects.com"
          access_key = "${{ secrets.LINODE_OBJECT_STORAGE_ACCESS_KEY }}"
          secret_key = "${{ secrets.LINODE_OBJECT_STORAGE_SECRET_KEY }}"
          EOF
      - name: Add Terraform TFVars
        run: |
```

```
        cat << EOF > devops/tf/terraform.tfvars
        linode_pa_token = "${{ secrets.LINODE_PA_TOKEN }}"
        authorized_key = "${{ secrets.SSH_PUB_KEY }}"
        root_user_pw = "${{ secrets.ROOT_USER_PW }}"
        py_app_count = ${{ secrets.PYAPP_NODE_COUNT }}"
        EOF
    - name: Terraform Init
      run: terraform -chdir=./devops/tf init -backend-config=backend

    - name: Terraform Validate
      run: terraform -chdir=./devops/tf validate -no-color
    - name: Terraform Apply Changes
      run: terraform -chdir=./devops/tf apply -auto-approve
```

Let's break this down:

- `name` is the name of the workflow, make it unique or it can get confusing.
- `on` when do we want this workflow to run?
- `workflow_dispatch` allows us to trigger this workflow manually on GitHub as well as call this workflow via the GitHub API.
- `push` with `branches: [main]` means that this workflow will run automatically every time we push this code onto the `main` branch.
- `jobs` is the declaration for the job(s) we want to run.
- `jobs:terraform:` The first and only job is named `terraform` but it could be named anything you like.
- `runs-on: ubuntu-latest` is the docker container image type this workflow runs on. `ubuntu-latest` is the most common.
- `steps` contains each command (or step) we want to run in the order we want to run them.
- `name: Checkout` and `uses: actions/checkout@v2` gets the code for us.
- `name: Setup Terraform` , `uses: hashicorp/setup-terraform@v1` and `with: terraform_version: 1.1.4` will install the Terraform CLI to our workflow.
- `name: Add Terraform Backend for S3` and `run: |` this step allows us to create our `backend` file based on the store secrets. Doing this will automatically hide the `backend` values.
- `name: Add Terraform TFVars` and `run: |` will create our `terraform.tfvar` file just as the `backend` file.
- Finally, we run `terraform validate` and `terraform apply` to automatically apply all changes. `terraform validate` is run before `terraform apply` to ensure the Terraform files are valid before making changes. If they are invalid, the workflow will fail.

That's it. Once you push this file to GitHub, the workflow will automatically run.

Pretty neat, right? Remove the lines that contain `push:branches:[main]` if you want to run this workflow manually.

Clean Up

As mentioned we we learned about `terraform destroy`, I recommend that you destroy any provisioned resource(s) unless you intend to use them:

```
terraform apply -destroy -auto-approve
```

or

```
terraform -chdir=./devops/tf apply -destroy -auto-approve
```

This is a good habit to get into. You also may consider rolling (or deleting) your **Linode Personal Access** (`LINODE_PA_TOKEN`) token and your **Linode Object Storage Secret Key** (`LINODE_OBJECT_STORAGE_SECRET_KEY`) so you don't accidentally provision resource(s) you didn't intend to.

Challenge

Create a workflow that handles `terraform -chdir=./devops/tf apply -destroy -auto-approve` so you can remove this configuration as needed.

Chapter 5

Ansible

Chapter 5
Ansible

Ansible has many built-in features; we'll focus primarily on Ansible Playbooks. Ansible Playbooks "record and execute Ansible's configuration, deployment, and orchestration functions" which essentially means we tell Ansible what we want it to achieve, and Ansible does it. For example:

Us: Hey Ansible, please install NGINX
Ansible: No problem

Us: Hey Ansible, on these 3 servers install NGINX. Also, on this 1 server install Apache.
Ansible: Done. All machines have been updated.

Us: Hey Ansible, on all 4 servers, please install Docker. Purge NGINX and Apache from all systems.
Ansible: Done.

This scenario is common for tools like Ansible because that's what they are designed for. How we instruct Ansible to run is not conversational like above (maybe someday), it's in a structured format using `YAML` files. If you're not familiar with `YAML`, you'll get a lot more familiar with it as you use tools like Ansible.

Here's a basic example of an Ansible Playbook file in YAML (we'll implement this one exactly in a few sections):

```
---
hosts: all
become: yes
tasks:
   - name: Install Nginx
     apt:
       name: nginx
       state: present
       update_cache: yes
```

For those of you who are familiar with installing NGINX on Debian machines, you may know of the command:

```
sudo apt update && sudo apt install nginx -y
```

Both of these examples aim to achieve the same result: install NGINX.

But why do we *need* Ansible if we can just write a bunch of installation commands?

The answer boils down to 4 aspects of Ansible (as well as many other IaC tools): *DRY*, *Repeatable*, *Reversible*, & *Version Control*

1. DRY: Don't Repeat Yourself
A core tenet of software development is to not repeat yourself. Using Ansible correctly unlocks DRY across your deployment.

2. Repeatable
We're focusing on Ansible Playbooks, which are a bunch of YAML files. Assuming Ansible is installed, these YAML files can run anywhere anytime.

The result of running it is *also* repeatable. That means running a playbook on 10,000 servers is as easy as running it on 1. Naturally provisioning 10,000 servers will take longer than provisioning 1, but using playbooks is easier than manually provisioning servers or writing custom scripts to provision them (Ansible can also run custom scripts for us).

3. Reversible
If you configure everything through Ansible (or other IaC tools) reversing changes becomes significantly easier simply because the changes are well documented. I tend to think of Ansible as code-based documentation that configures what you need.

4. Version Control
Version control tracks the history of changes for every file within a project that uses version control correctly. Git, the type of version control we recommend, is one of the best ways for all your team members to collaborate on vital code and configuration your team needs. Having version control means we can leverage additional automation tools that make Ansible run only after meeting certain conditions, this limits who can make major changes.

I could write books about the importance of these 4 points and the thousands of other points I did not address.

I recommend trying tools to see if they make a good fit for your workflow, so let's get started with Ansible.

Getting Started & Core Installations

This chapter requires a few core technologies before we get started. If you have any issues installing these items on your local machine, I highly recommend creating a remote workspace as we do in Appendix C.

Python

To run Ansible and Ansible Playbooks, we need to install Python version 3 (ideally 3.7 or greater). For most systems, installing it directly from python.org/downloads is the easiest way to get started.

Virtual Environments (venv)

To provide some isolation between Python projects, we use virtual environments (`venv`). We'll use the built-in module venv. If you're familiar with another virtual environment manager, feel free to use it. Here are a few others worth considering

- Poetry
- Pipenv
- Virtualenv

I use the built-in `venv` because I believe it's the easiest way for most of us to get started, and it's also been around the longest. I'll show you how to create one of these later in this chapter.

Git

Git is an open-source version control tool. Using Git is recommended for all your code projects. We'll be using Git to copy example code in this case.

Download Git for your system.

VS Code

Visual Studio Code, or VS Code, is one of the most widely used tools for writing code because it's free, is based on open-source, has a plugin marketplace, and Microsoft develops it.

Download it here.

Alternatives:

- Atom
- Sublime Text
- PyCharm

Clone the Sample Python Web App

We're starting with a Python web application (like we do for the other IaC tools in the Try IaC series) as a way to deploy a functioning web application into production. This gives us a practical look at how tools like Ansible work.

In this case, we'll be deploying a Docker-based Python web application. Using Docker makes this even more practical because once you know how to deploy an app with Docker, you can deploy nearly *any* type of open-source application. Now, this book is not about Docker and all its amazing features but it does leverage some of them as a means for the most practical way to bring an app, any app, into production.

```
cd /path/to/your/project/folder/
```

I used `cd ~/dev/iac-ansible`
We need to clone the following project:

```
git clone https://github.com/codingforentrepreneurs/iac-python .
```

Remove the cloned `.git` repo:

```
rm -rf .git
```

Re-initialize this project as your git project:

```
git init
git add --all
git commit -m “Initial Project”
```

Create a Python Virtual Environment & Install Ansible

With Python applications, it’s recommended that you create a virtual environment. We’re going to use Python’s built-in Virtual Environment manager: `venv`

*_Using `venv` (recommended)_*

```
python3 -m venv .
```

*_Activate on macOS / Linux_*

```
source bin/activate
```

*_Activate on Windows_*

```
.\Scripts\activate
```

*_Install_*

```
(iac-ansible) $ python -m pip install -r requirements.txt
(iac-ansible) $ python -m pip install ansible
```

You can also run `./bin/python -m pip install -r requirements.txt`

Using `pipenv`

```
pipenv install -r requirements.txt
pipenv install ansible
```

Using `virtualenv`
This is almost identical to Python's built-in `venv` tool.

```
virtualenv .
```

Activate on macOS / Linux

```
source bin/activate
```

Activate on Windows

```
.\Scripts\activate
```

Install

```
(iac-ansible) $ python -m pip install -r requirements.txt
(iac-ansible) $ python -m pip install ansible
```

You can also run `./bin/python -m pip install -r requirements.txt`

Using `poetry`

```
cat requirements.txt|xargs poetry add
poetry add ansible
```

Inventory & Provision Instances on Linode

We'll start by provisioning 1 Linode instance that Ansible will configure.

1. Login to Linode Console
2. Click Create > Linode and use the following

- Image type: `ubuntu 20.04`
- Region: (near you or your users)
- Linode Plan: Shared CPU → Nanode 1GB (just $5/mo)
- Linode Label: `ansible-1` (or whatever you choose)
- Add tags: (optional)
- Root password: set a secure password (Consider using Appendix D).
- SSH keys: you *must* set an SSH key to your local machine and save it in the Linode Console. Use Appendix A to do so.
- Those are all the required options. Click *Create Linode* and let it provision.

3. Repeat these steps for future instances

Once you have provisioned your virtual machine, add the public IP Address to `inventory.ini` like:

```
192.168.86.41
```

As a side note, you can automate the provisioning instance(s) step by leveraging Terraform.

Now that you have an instance provisioned and Ansible installed locally, let's create your first playbook.

Your First Playbook

Ansible Playbooks are the heart of what Ansible is all about. You write playbooks as a way to use Ansible to automate configuration, deployment, and orchestration.

devops/ansible/main.yaml

```
---
- hosts: all
  become: yes
  tasks:
    - name: Install Nginx
      apt:
        name: nginx
        state: present
        update_cache: yes
```

Now run this playbook with:

```
ansible-playbook main.yaml -i inventory.ini
```

Let's break this down:

- ansible-playbook is the command-line command to execute a playbook.
- main.yaml is the playbook file we created above.
- inventory.ini is the inventory file that includes our IP Address(es) from the previous step.

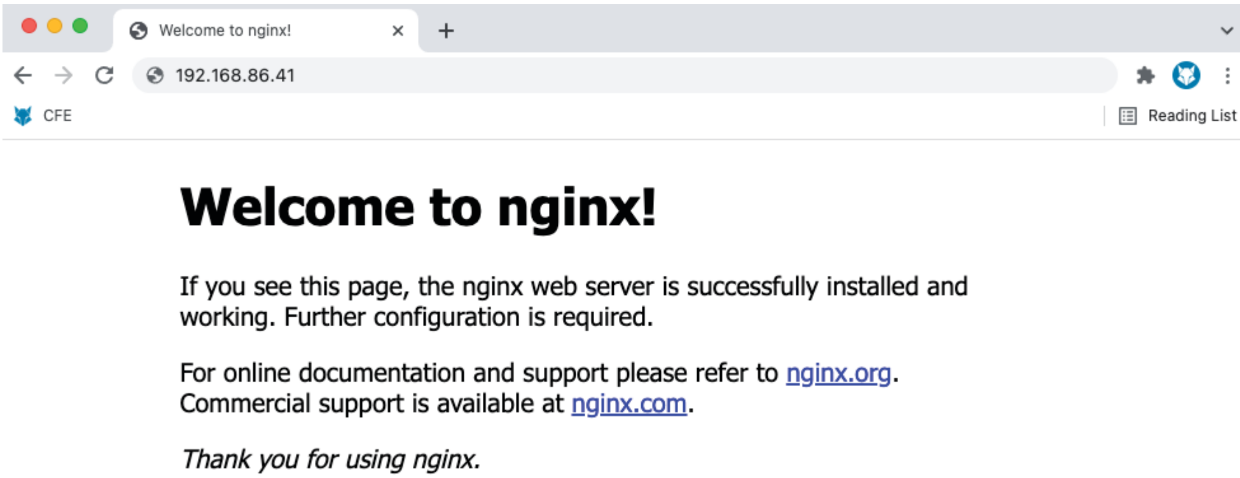
After you run this command, you should see something like:

```
PLAY [all] *******************************************************************************
*******************

TASK [Gathering Facts] *******************************************************************
*******************
ok: [192.168.86.41]

TASK [Install Nginx] *********************************************************************
*******************
changed: [192.168.86.41]

PLAY RECAP *******************************************************************************
*******************
192.168.86.41             : ok=2    changed=1    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
```

This shows what Ansible did based on our playbook.

Keep in mind that if you have just provisioned your instance, Ansible may not be able to make changes yet. When in doubt, SSH into your instance if Ansible responds with fatal: [192.168.86.41]: UNREACHABLE!

If you open the IP address in your web browser, you should see:



Default Ansible Configuration - ansible.cfg

We want to *always* use our `inventory.ini file` . So let's set up the default Ansible configuration that `ansible-playbook` looks for. It's a file called `ansible.cfg`:

devops/ansible/ansible.cfg

```
[defaults]
ansible_python_interpreter="/usr/bin/python3"
deprecation_warnings=False
inventory=inventory.ini
remote_user="root"
retries=2
```

Let's break this down:

- `ansible_python_interpreter` is the version of Python you want your hosts to use.
- `deprecation_warnings` ignore deprecation warnings when using `ansible`
- `inventory` this is the path to our local inventory file (from above)
- `remote_user` What user do you want to use for `ansible`?
- `retries` The number of times you want `ansible` to attempt to run a playbook.

When you run a playbook for the first time on a remote host (a virtual machine) you have never used SSH on, you will likely see something to the effect `The authenticity of host '96.126.104.175 (96.126.104.175) can't be established...Are you sure you want to continue connecting (yes/no/[fingerprint])?` If you have just provisioned this IP address, you will want to write `yes` and `enter/return` to continue. You can also add `host_key_checking=False` in your `ansible.cfg` if you want to skip this step.

Now, we can just run `ansible-playbook main.yaml`, and we'll see:

```
PLAY [all] ********************************************************************************
*******************

TASK [Gathering Facts] ********************************************************************
*******************
ok: [192.168.86.41]

TASK [Install Nginx] **********************************************************************
*******************
ok: [192.168.86.41]

PLAY RECAP ********************************************************************************
*******************

192.168.86.41             : ok=2    changed=0    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
```

Notice that the result gives us `changed=0`. This means that Ansible did not need to change our instances simply because the `main.yaml` playbook has already run correctly.

This simple example gives us insight into what Ansible does well: configuring systems when they *need* to be changed and *only* when they need to be changed.

Let's expand on this some more.

Replace Remote Files with Ansible

devops/ansible/main.yaml

```
---
- hosts: all
  become: yes
  tasks:
    - name: Install Nginx
      apt:
       name: nginx
       state: present
       update_cache: yes
```

```
    - name: Update Nginx Default Homepage
      copy:
       dest: /var/www/html/index.nginx-debian.html
       content: |
          <h1>Hello World</h1>
          <p>This is awesome</p>
```

Let's examine what has changed:

```
    - name: Update Nginx Default Homepage
      copy:
       dest: /var/www/html/index.nginx-debian.html
       content: |
          <h1>Hello World</h1>
          <p>This is awesome</p>
```

This creates a file on your instance(s) at the location written in `dest` with the text in `content`. We're going to make this better soon, but let's see the result now.

Now run

```
ansible-playbook main.yaml
```

After adding this new task, we see:

```
TASK [Update Nginx Default Homepage] *****************************************************
*******************
changed: [96.126.104.175]

PLAY RECAP *******************************************************************************
*******************
96.126.104.175             : ok=3    changed=1    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
```

What we see here is the built-in plugin `copy` , but this case has a fundamental issue: *writing inline code*.

We solve this by using templates.

Using Templates with Playbooks

Now that we've seen two built-in plugins to Ansible Playbooks - `apt` and `copy` - it's time to examine how to use external files with any given task and/or plugin.

We do this by using templates. Templates in Ansible are written in Jinja, which is essentially string substitution taken to a whole new level. We'll discuss these templates more later. For now, let's start with a standard HTML page:

devops/ansible/templates/nginx.default.html

```
<!DOCTYPE html>
<html>
    <body>
        <h1>Hello World</h1>
        <p>This is awesome</p>
    </body>
</html>
```

What we see here is just a standard HTML file, but we can choose any file type. Since we're using this file to change the default NGINX page, we need to use HTML.

Let's update our playbook to reflect this newly created template:

devops/ansible/main.yaml

```
---
hosts: all
become: yes
tasks:
   - name: Install Nginx
     apt:
       name: nginx
       state: present
       update_cache: yes
   - name: Update Nginx Default Homepage
     copy:
       dest: /var/www/html/index.nginx-debian.html
       src: ./templates/nginx.default.html
```

Notice that `content` has become `src` in the `copy` built-in. `src` is merely the source path local to this Ansible Playbook.

Run `ansible-playbook main.yaml` to verify your changes were successful.

Templates are even more useful when we sprinkle in variables so that our templates can change whenever our inventory/hosts need them to. Let's see how we can use variables in templates.

Using Variables in Templates

Before we jump into variables, I want you to consider the following Python string:

```
print("Hello world, {name} is a great tool.".format(name="Ansible"))
```

The result of this command is:

```
Hello world, Ansible is a great tool.
```

Above is a simple string substitution example (a somewhat outdated example).

Below is what made this substitution work:

```
- `{name}`
- `.format(name="Ansible")`
```

In this case, the `name` is the key, and `Ansible` is the value of that key. Wherever the key shows up, it will be replaced automatically. This concept is simple in terms of programming and it's certainly not unique to Python.

I showed you the above example to illustrate how Jinja works. Here's a basic Jinja example:

In the template:

```
Hello {{ my_variable }}
```

In the playbook:

```
vars:
    my_variable: Whatever I choose
```

The result:

```
Hello Whatever I choose.
```

Jinja does get more complex than this (allowing for loops and conditions). For now, we'll take a look at a practical example with this data:

devops/ansible/main.yaml

```
---
- hosts: all
  become: yes
  vars:
    title: Hello there
    description: Some more news!


  tasks:
    - name: Install Nginx
      apt:
        name: nginx
        state: present
        update_cache: yes
    - name: Update Nginx Default Homepage
      copy:
        dest: /var/www/html/index.nginx-debian.html
        src: ./templates/nginx.default.html
```

Let's break down what we added. The `vars` parameter is what we set to use custom inline variables in our playbook. Within the `vars` block, we add `key`/`value` pairs such as `title` (the key) and `Hello there` the value of `title` (same is true for `description`).

Now let's update our template:

templates/nginx.default.html

```
html
<!DOCTYPE html>
<html>
```

```
    <body>
         <h1>{{ title }} - {{ inventory_hostname }}</h1>


        <p>{{ description }}</p>
    </body>
</html>
```

But where did {{ inventory_hostname }} come from? Before we answer that run:

```
ansible-playbook main.yaml
```

Open up your IP Address and take a look at the result. inventory_hostname was set! How did {{ inventory_hostname }} work despite being absent in the vars declaration in our playbook? That's because it's called a special variable and is built right into Ansible. Pretty neat huh?

Configure Multiple Hosts

Now it's time to provision 3 more instances on Linode. As a refresh you'll need to:

1. Login to Linode Console
2. Click Create > Linode and use the following
- Image type: ubuntu 20.04
- Region: (near you or your users)
- Linode Plan: Shared CPU → Nanode 1GB (just $5/mo)
- Linode Label: ansible-1 (or whatever you choose)
- Add tags: (optional)
- Root Password: set a secure password. (Consider using Appendix D)
- SSH Keys: you *must* set an ssh key to your local machine and save it in the Linode Console. Use Appendix A to do so.
- That's all the required options. Click *Create Linode* and let it provision"
- Update inventory.ini with your new IP address (like below)

3. Repeat these steps for each new instance

Update inventory.ini:

```
45.33.115.4
96.126.118.76
198.58.105.179
23.239.25.5
```

Now that you have 4 instances, let's use Ansible to configure them:

```
ansible-playbook main.yaml
```

Here's my result:

```
PLAY [all] ******************************************************************************
*******************

TASK [Gathering Facts] ******************************************************************
*******************
ok: [66.175.209.101]
ok: [96.126.104.175]
ok: [45.33.64.242]
ok: [23.239.11.23]

TASK [Install Nginx] ********************************************************************
*******************
ok: [96.126.104.175]
changed: [66.175.209.101]
changed: [45.33.64.242]
changed: [23.239.11.23]

TASK [Update Nginx Default Homepage] ****************************************************
*******************
ok: [96.126.104.175]
changed: [66.175.209.101]
changed: [45.33.64.242]
changed: [23.239.11.23]

PLAY RECAP ******************************************************************************
*******************
23.239.11.23               : ok=3    changed=2    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
45.33.64.242               : ok=3    changed=2    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
66.175.209.101             : ok=3    changed=2    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
96.126.104.175             : ok=3    changed=0    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
```

As we see, it shows that the 3 new instances have changes while our previous one stays the same. From here on out I am not going to show these results but I did want to highlight how simple it is to review exactly what occurred doing any given playbook action.

Inventory Groups & Load Balancing

Now let's see how simple it is to implement load balancing with Ansible and NGINX. As you may know, load balancing allow us to handle more traffic by adding more machines with minimal specs (horizontal scaling) instead of bumping up the specs of each machine (vertical scaling).

To create inventory groups in ansible we just proceed the IP address or addresses with the `[mygroup]` designation. `mygroup` can be any name you decide.

Here's an example:

Update `inventory.ini`:

```
[loadbalancer]
23.239.25.5



[webapps]
45.33.115.4
96.126.118.76
198.58.105.179
```

We now have two groups `webapps` and `loadbalancer`. We can target these groups within our playbooks by using `hosts: webapps` and `hosts: loadbalancer` repsecitvely. Up until this point we've used `hosts: all` which automatically targets every host listed in `inventory.ini`.

Before we update our playbook, let's add the load balancer NGINX configuration template:

devops/ansible/templates/nginx.conf

```
upstream myproxy {
     {% for host in groups[‘webapps’] %}
        server {{ host }};
    {% endfor %}
}
```

```
server {
    listen 80;
    server_name {{ inventory_hostname }};
    root /var/www/html/;

    location / {
        proxy_pass http://myproxy;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $host;
        proxy_redirect off;
    }

}
```

This is as simple of configuration as it gets for a load balancer. As you see this Ansible/Jinja template includes a For Loop:

```
{% for host in groups['webapps'] %}
        server {{ host }};
    {% endfor %}
```

For loops in Ansible/Jinja templates are essentially:

```
{% for my_var in my_list %}{{ my_var }}{% endfor %}
```

Assuming that `my_list` is an iterable variable, you can use the for loop template tag. In this case, `my_var` is the looping variable and is arbitrary.

In our template, however, we use the `groups['webapps']` as our iterable. How is this possible?

- `webapps` is coming directly from `inventory.ini`. You may recall it's simpy an inventory group
- `groups` is a builtin special variable for Ansible templates. (Just like `inventory_hostname`
- Inventory `groups` are iterable by default (even if there's only 1 member).
- In place of `{{ inventory_hostname }}`, we could use `{{ groups['loadbalancer'][0] }}` -- pretty neat huh?

Now let's update our playbook.

devops/ansible/main.yaml

```
---
- hosts: webapps
  become: yes
  vars:
     title: Hello there
     description: Some more news!
  tasks:
    - name: Install Nginx
      apt:
        name: nginx
        state: present
        update_cache: yes
    - name: Update Nginx Default Homepage
      template:
        dest: /var/www/html/index.nginx-debian.html
        src: ./templates/nginx.default.html



- hosts: loadbalancer
  become: yes
  tasks:
    - name: Install Nginx
      apt:
        name: nginx
        state: present
        update_cache: yes
    - name: Add Nginx Config
      template:
        dest: /etc/nginx/sites-available/default
        src: ./templates/nginx.conf
    - name: Enable New Nginx Config
      file:
        dest: /etc/nginx/sites-enabled/default
        src: /etc/nginx/sites-available/default
        state: link
    - name: Reload Nginx
      service:
        name: nginx
        state: reloaded
```

Let's hone in on the new aspects here

- `template:` - Notice that I replaced all instances of `copy` for `template`. These two modules do almost the exact same thing but `template` will work more consistently when using Jinja template replacement (like we've been doing). You'll see projects using these two interchangeably.
- `file:` This module is made to move (or link) files on your remote machine. In our case we link newly created (from a template) `/etc/nginx/sites-available/default` to `/etc/nginx/sites-available/default` so `nginx` knows what configuration to use (our load balancer)
- `service:` Linux machines have a lot of services that we can use. In the case of `nginx`, if we make configuration changes we want to ensure nginx is reloaded (You can also use `restarted` but `reloaded` does not turn off nginx and start it back up it just refreshes it's configuration).

Optionally, let's update the HTML file for our `webapps` and the default `nginx` web page. This will just verify our load balancer is working correctly by showing a different `{{ inventory_hostname }}` every time the load balancer is accessed (ie the IP Address directly).

templates/nginx.default.html

```
<!DOCTYPE html>
<html>
    <body>
         <h1>{{ title }} - {{ inventory_hostname }}</h1>
        <p>{{ description }}</p>
        <div>
            {% for host in groups['webapps'] %}
                <p>{{ host }}</p>
            {% endfor %}
            {{ groups['loadbalancer'] }}
        </div>
    </body>
</html>
```

Import Playbooks

Now it's time to break apart our main.yaml into smaller chunks. This will allow our code to stay DRY (don't repeat yourself) as well as simplify the requirements each host may have.

Let's start by splitting up main.yaml . First we'll create a new module called install-nginx.yaml (notice that I created a new folder called playbooks

devops/ansible/playbooks/install-nginx.yaml

```
---
- hosts: all
  become: yes
  tasks:
    - name: Install Nginx
      apt:
        name: nginx
        state: present
        update_cache: yes
```

The only real update here is we made hosts be all.

Next up, our load balancer

devops/ansible/playbooks/loadbalancer.yaml

```
---
- hosts: loadbalancer
  become: yes
  tasks:
    - name: Add Nginx Config
      template:
        src: ../templates/nginx.conf
        dest: /etc/nginx/sites-available/default
    - name: Enable New Nginx Config
      file:
        src: /etc/nginx/sites-available/default
        dest: /etc/nginx/sites-enabled/default
        state: link
    - name: Reload Nginx
      service:
         name: nginx
         state: reloaded
```

Now our webapps config:

devops/ansible/playbooks/webapps.yaml

```
---
- hosts: webapps
  become: yes
  vars:
      title: Hello there
      description: Some more news!
  tasks:
    - name: Update Nginx Default Homepage
      template:
        dest: /var/www/html/index.nginx-debian.html
        src: ../templates/nginx.default.html
```

Notice that I had to update the `template:src` from `templates/` to `../templates/` because we moved these playbooks into a new subdirectory.

If you omit `templates/` from a template module, Ansible will automatically look in the relative `templates` folder. This means inside of the directory `playbooks/` you can add a folder called `templates/` and store all your templates then reference them just with the name of the file. Personally, I like writing full relative paths to make it easier to debug later if needed.

And finally, we can now import each one of these playbooks into `main.yaml`. Remember, the order you declare items in Ansible Playbooks is the order it will execute. The same is true for importing playbooks:

devops/ansible/main.yaml

```
---
- name: Install Nginx
  import_playbook: ./playbooks/install-nginx.yaml

- name: Update Webapps
  import_playbook: ./playbooks/webapps.yaml

- name: Configure LoadBalancer
  import_playbook: ./playbooks/loadbalancer.yaml
```

Now run:

```
ansible-playbook main.yaml
```

Every time you run this now you should see that the task `Reload Nginx` will always run so you'll see `changed=1` for at least the `loadbalancer` group.

Ansible Role Basics

Now it's time to move on to Ansible roles. In the previous section we installed `nginx` on all hosts. This would be great if we needed NGINX on all hosts but, as we will find later in this chapter, that is not always true.

Roles help solve this issue. Let's create an `nginx` role.

We start by creating the `roles` and `nginx` folder. The `roles` folder is required, the `nginx` folder is just the role name.

```
mkdir -p roles/nginx
```

Now that we have a role named `nginx` we're going to make `tasks` that are associated to this role. To do this, we'll create a new folder called `tasks` with a file called `main.yaml`.

So here's the format we must follow:

```
<project_dir>/roles/<role_name>/tasks/main.yaml
```

We'll see more roles later. For now, let's create the task we need.

devops/ansible/roles/nginx/tasks/main.yaml

```
- name: Install Nginx
  apt:
    name: nginx
    state: present
    update_cache: yes
```

The key to this file is it's no longer a playbook, it's just a list of tasks that you want this role to perform. Now, in playbooks, to use this role's task we do this:

devops/ansible/playbooks/example.yaml

```
- hosts: all
  become: yes
  roles:
     - ./../roles/nginx
```

We'll see how this works once we use it practically. Let's do that now:

devops/ansible/playbooks/loadbalancer.yaml

```
- hosts: loadbalancer
  become: yes
  roles:
    - ./../roles/nginx
  tasks:
    - name: Add Nginx Config
      template:
        src: ./../templates/nginx.conf
        dest: /etc/nginx/sites-available/default
    - name: Enable New Nginx Config
      file:
        src: /etc/nginx/sites-available/default
        dest: /etc/nginx/sites-enabled/default
        state: link
    - name: Reload Nginx
      service:
         name: nginx
         state: reloaded
```

devops/ansible/playbooks/webapps.yaml

```
---
- hosts: webapps
  become: yes
  vars:
```

```
      title: Hello there
      description: Some more news!
  roles:
    - ./../roles/nginx
  tasks:
    - name: Update Nginx Default Homepage
      template:
        dest: /var/www/html/index.nginx-debian.html
        src: ./../templates/nginx.default.html
```

devops/ansible/main.yaml

```
---
- name: Update Webapps
  import_playbook: ./playbooks/webapps.yaml

- name: Configure LoadBalancer
  import_playbook: ./playbooks/loadbalancer.yaml
```

Let’s run this:

```
ansible-playbook main.yaml
```

Now we’ll see that our `nginx` role runs prior to the other tasks in each playbook. How cool is that?

Ansible Handlers

When tasks are complete, we have the option to notify another task. We do this with the `notify` module coupled with an Ansible handler.

Let’s implement it:

devops/ansible/playbooks/loadbalancer.yaml

```
- hosts: loadbalancer
  become: yes
  roles:
```

```
    - ./../roles/nginx
  tasks:
    - name: Add Nginx Config
      template:
        src: ./../templates/nginx.conf
        dest: /etc/nginx/sites-available/default
      notify: reload nginx
    - name: Enable New Nginx Config
      file:
        src: /etc/nginx/sites-available/default
        dest: /etc/nginx/sites-enabled/default
        state: link
  handlers:
    - name: reload nginx
      service:
         name: nginx
         state: reloaded
```

The format is:

```
notify: <some name>

handlers:
    - name: <some name>
```

If `<some name>` is not identical, the handler will not execute. `notify` will only trigger if something in that task changes. In this case, the handler will only be triggered if `template:src:` changes. This ensures we're not reloading NGINX, or running any other task unnecessarily.

Lastly, the handler will run *after* all other tasks are completed. When one task is complete, you can run another one immediately after. If you need a task to run right after another task, make it a task itself.

Handlers in Roles

Here's a situation where we *definitely* want to move our handlers into the NGINX role. This is pretty simple to do. Let's start with creating the role handler:

The format is almost identical to role tasks but we use the `handlers` folder instead like:

```
<project_dir>/roles/<role_name>/handlers/main.yaml
```

In our case, we'll create the following handlers:

devops/ansible/roles/nginx/handlers/main.yaml

```
- name: reload nginx
  service:
      name: nginx
      state: reloaded

- name: restart nginx
  service:
      name: nginx
      state: restarted
```

Notice that I have both the `nginx` service `reload` and `restart` ability. They are the same as doing `sudo service nginx reload` and `sudo service nginx restart` respectively.

This means I can run `notify: reload nginx` or `notify: restart nginx` in any task that has a playbook that implements the `nginx` role.

Here's what that looks like:

devops/ansible/playbooks/loadbalancer.yaml

```
- hosts: loadbalancer
  become: yes
  roles:
    - ./../roles/nginx
  tasks:
    - name: Add Nginx Config
```

```
      template:
        src: ./../templates/nginx.conf
        dest: /etc/nginx/sites-available/default
      notify: reload nginx
    - name: Enable New Nginx Config
      file:
        src: /etc/nginx/sites-available/default
        dest: /etc/nginx/sites-enabled/default
        state: link
```

devops/ansible/playbooks/webapps.yaml

```
---
- hosts: webapps
  become: yes
  vars:
      title: Hello there
      description: Some more news!
  roles:
    - ./../roles/nginx
  tasks:
    - name: Update Nginx Default Homepage
      template:
        dest: /var/www/html/index.nginx-debian.html
        src: ./../templates/nginx.default.html
      notify: reload nginx
```

Notice that now both playbooks reload NGINX while only 1 did before.

Install Docker via Role

Now that we understand a few Ansible fundamentals, it's time to install Docker using a role.

We'll start by creating our Docker role folders:

```
mkdir -p roles/docker/tasks
mkdir -p roles/docker/handlers
```

Before we create our tasks, let's talk about what needs to happen.

- Install Docker via the Docker install script on https://get.docker.com. This is my preferred method to install Docker on Linux machines. You cannot use `apt` to install Docker at this time.
- Each time we run the Docker role in Ansible, we want to check that Docker is installed/running. To do this, I will run `command -v Docker` as a shell command. This shell command will let us know if `docker` is an executable on our remote Linux machines.
- If `command -v docker` fails, we want to have all tasks continue to run. As you may have experienced already, if a task fails in Ansible, other tasks are skipped. To ensure the tasks continue, we'll use the shell command `command -v docker >/dev/null 2>&1`. To try it yourself, SSH into one of your virtual machines and run `command -v some_fake_command_that_doe_not_exist >/dev/null 2>&1`.

Let's see how the above works within our role's tasks:

devops/ansible/roles/docker/tasks/main.yaml

```
- name: Grab Docker Install Script
  get_url:
    url: https://get.docker.com
    dest: /tmp/get-docker.sh
    mode: 0755
  notify: exec docker script

- name: Verify Docker Command
  shell: command -v docker >/dev/null 2>&1
  ignore_errors: yes
  register: docker_exists

- debug: msg="{{ docker_exists.rc }}"

- name: Trigger docker install script if docker not running
  shell: echo "Docker command"
  when: docker_exists.rc != 0
  notify: exec docker script
```

Let's break down the new items:

- `get_url:` This can download a file from a URL. `mode: 0755` gives this downloaded file executable permission
- `shell:` This is how you can write shell commands. In this case, we're just verifying that the command exists.
- `ignore_errors: yes` Ensures future tasks run. In this case, if we do have errors from our `shell:` command, there's a good chance that Docker was not installed correctly from the previous task.
- `register` Stores the result of the task in a variable we can reference elsewhere. In this case, we use the variable `docker_exists`, but you can choose any variable you'd like.

- debug: msg="{{ docker_exists.rc }}" Conveniently highlights what the registered variable docker_exists is set to while you run the playbook(s) that reference this role.
- when: docker_exists.rc != 0 Allows us to run a task based on a condition. This is similar to notify, but in this case, we only want to notify if this condition is met.

Now let's create the handler for the above Docker task(s).

devops/ansible/roles/docker/handlers/main.yaml

```
- name: exec docker script
  shell: /tmp/get-docker.sh
```

This handler will run the output of our get_url tasks from above.

Update the following files:

- devops/ansible/playbooks/loadbalancer.yaml
- devops/ansible/playbooks/webapps.yaml

To include these roles:

```
  roles:
    - ./../roles/nginx
    - ./../roles/docker
```

All we did was add - ./../roles/docker to our roles for each playbook.

Now run ansible-playbook main.yaml .

You should see this block repeated a few times:

```
TASK [./../roles/docker : Verify Docker Command] *******************************
fatal: [69.164.221.67]: FAILED! => {"changed": true, "cmd": "command -v docker >/dev/null 2>&1", "delta": "0:00:00.004127", "end": "2022-01-27 17:24:06.202568", "msg": "non-zero return code", "rc": 127, "start": "2022-01-27 17:24:06.198441", "stderr": "", "stderr_lines": [], "stdout": "", "stdout_lines": []}
...ignoring
```

The key is that it says ignoring which is exactly what we want to happen when we verify the Docker command. Note that "rc" is 127 in the error above.

Later we'll see:

```
TASK [./../roles/docker : debug] ***********************************************
ok: [69.164.222.142] => {
    "msg": "127"
}
ok: [96.126.104.175] => {
    "msg": "127"
}
ok: [69.164.221.67] => {
    "msg": "127"
}
```

This debug message is 127, just like the "rc" value in the error message before.
We can see that `docker_exists.rc != 0`, so this condition will be triggered. The `Verify Docker Command` sets the variable `docker_exists` to the output you see in the error itself. Pretty cool huh?

The play recap should look something like this:

```
69.164.221.67              : ok=8    changed=4    unreachable=0    failed=0    skipped=0
rescued=0    ignored=1
69.164.222.142             : ok=8    changed=4    unreachable=0    failed=0    skipped=0
rescued=0    ignored=1
69.164.222.227             : ok=9    changed=4    unreachable=0    failed=0    skipped=0
rescued=0    ignored=1
96.126.104.175             : ok=8    changed=4    unreachable=0    failed=0    skipped=0
rescued=0    ignored=1
```

If we run `ansible-playbook main.yaml` again, we should have a play recap like this:

```
69.164.221.67              : ok=6    changed=1    unreachable=0    failed=0    skipped=1
rescued=0    ignored=0
69.164.222.142             : ok=6    changed=1    unreachable=0    failed=0    skipped=1
rescued=0    ignored=0
69.164.222.227             : ok=7    changed=1    unreachable=0    failed=0    skipped=1
rescued=0    ignored=0
96.126.104.175             : ok=6    changed=1    unreachable=0    failed=0    skipped=1
rescued=0    ignored=0
```

Further, if we SSH into any of the hosts that used the Docker role, we should be able to run `docker ps` and see:

```
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
```

This section might take some time to digest. I recommend that you do it a few times if you need to. Using `debug` can be pretty helpful, so can attempting all of these tasks manually on a freshly provisioned Linode instance.

Purging Packages with Roles

Undoing installations is another thing you'll need to do from time to time. Honestly, it might be easier to provision a new instance than to create a bunch of purging rules.

Nevertheless, we'll look at how to purge `nginx` from our systems since we're going to be using Docker (and the Docker-based `nginx`) going forward.

Make the role `nginx_purge` by doing the following:

```
mkdir -p roles/nginx_purge/tasks
echo “” > roles/nginx_purge/tasks/main.yaml
```

And create the role:

devops/ansible/roles/nginx_purge/tasks/main.yaml

```
- name: Remove Nginx
  apt:
    name: “{{ item }}”
    state: absent
    purge: yes
  with_items:
    - nginx
- name: Stop Nginx Services
  service:
    name: “{{ item }}”
    state: stopped
  with_items:
    - nginx
  ignore_errors: yes
```

So we now see a new way to write tasks, the `with_items:` method. Think of `with_items` as a for loop that will iterate through the list of items you designate in that block.

Here's a simple way to understand `with_items`

```
- debug: msg="{{ item }}"
  with_items:
    - abc
    - 123
    - easy peasy
```

This shows us that `with_items` can be used nearly anywhere in Ansible. When I see this, I think of it in terms of Python:

```
with_items = ['abc', 123, 'easy peasy']
for item in with_items:
    print(f"msg={item}")
```

The reason I think of it this way reminds me that `item` is *always* the iterated variable when you use `with_items`

Whenever you iterate in this way, you can limit the amount of redundancy in your steps. Designing our `nginx_purge` role in this way allows us to modify the rule to remove/purge any other dependencies we may want to remove.

Let's implement this role just on our load balancer playbook for now:

devops/ansible/playbooks/loadbalancer.yaml

```
- hosts: loadbalancer
  become: yes
  roles:
    - ./../roles/nginx_purge
    - ./../roles/docker
...
```

The `...` just represents we made no changes to that portion of the document.

Now run:

```
ansible-playbook main.yaml
```

Now our load balancer is down completely. If we were to provision a new instance for our load balancer, the only new item added would be Docker. In other words, it will not purge something from our system that isn't already there.

Docker-based Nginx Load Balancer

Now we're going to modify our load balancer to leverage Docker's system. We are *not* using the community-managed version of a Docker plugin at this time because it's not officially supported by the core Ansible development team.

Further, if you are new to Docker, this will be a great reference for you to use as you learn more about Docker and running various commands.

Before we jump into the new `loadbalancer.yaml` playbook, let's make a minor change to a task in the Docker role:

In `devops/ansible/roles/docker/tasks/main.yaml` update:

```
- name: Trigger docker install script if docker not running
  shell: echo "Docker command"
  when: docker_exists.rc != 0
  notify: exec docker script
```

to

```
- name: Run docker install script if docker not running
  shell: /tmp/get-docker.sh
  when: docker_exists.rc != 0
```

This update ensures this task is run immediately instead of running as a handler. In the playbook below, you can hopefully see why (hint: we use Docker commands):

devops/ansible/playbooks/loadbalancer.yaml

```
- hosts: loadbalancer
  become: yes
  roles:
    - ./../roles/nginx_purge
    - ./../roles/docker
  tasks:
    - name: Verify /var/www/ exists
      file:
        path: /var/www
        state: directory
        mode: 0755
    - name: Add Nginx Config
      template:
        src: ./../templates/nginx.conf
        dest: /var/www/nginx.conf
    - name: Has Running Docker Images
      shell: docker ps -aq >/dev/null 2>&1
      register: containers_running
      ignore_errors: yes
    - debug: msg="{{ containers_running.rc }}"
    - name: Docker Stop Running Containers
      shell: docker stop $(docker ps -aq)
      when: containers_running.rc != 0
    - name: Docker Remove Previous Containers
      shell: docker rm $(docker ps -aq)
      when: containers_running.rc != 0
    - name: Run Docker-based Nginx
      shell: |
        docker run \
        --restart always \
        -v /var/www/nginx.conf:/etc/nginx/conf.d/default.conf:ro \
        -p 80:80 \
        -d nginx
```

Let's break this down:

- `roles:` At this point, we should be able to remove `./../roles/nginx_purge`, but I am leaving it there for now.
- Task `name: Verify /var/www/ exists` This ensures the folder exists so we can use it to store our NGINX configuration. What might not be clear is that when you install `nginx` using `apt` this folder is generated for you. Since we never intend to install `nginx` on this machine, we must create the directory.
- ask `name: Add nginx Config` This has a destination to the previous `file:` module's directory.
- Task `name: Has Running Docker Images`. The command `docker ps -aq` will show any containers that are currently running. As of our first run, we have none.

- Task `name: Docker Stop Running Containers` If we do have running containers, we want to stop all of them as we prep to replace the configuration.
- Task `name: Docker Remove Previous Containers` once we stop running containers, we'll remove it so we can rebuild a fresh NGINX container.
- `name: Run Docker-based Nginx` here's where we run our Docker-based NGINX configuration. the keys for this are:
- `--restart always` this will ensure that the Docker image will restart if our remote host is rebooted.
- `-v /var/www/nginx.conf:/etc/nginx/conf.d/default.conf:ro` This will attach the `nginx.conf` file we added in the task `name: Add Nginx Config`. Keep in mind that the Docker container pulls files from our remote machine (our ansible host) and never our local machine. The Docker-related items are only happening on the remote host.
- `-p 80:80` This will map the Docker port `80` to the remote host port `80` so external HTTP web traffic can access the contents within the Docker container.
- `-d` This is called detached mode and allows for the container to run as a background service
- `nginx` In this case, it's the official Docker nginx image. Docker is smart enough to know that.

Now run:

```
ansible-playbook main.yaml
```

If you open up the IP address for your load balancer service, you will see that the Docker-based NGINX load balancer is now running!

When it comes to NGINX, I often opt for the pure NGINX implementation since it's much easier to update/reload NGINX configuration changes. That said, Docker-based NGINX is not difficult and becomes pretty useful when you start moving into other areas of Docker like Docker Compose, Docker Swarm, and Kubernetes.

Using Facts & Variables

Whenever we set a variable using `register:` we can use that variable throughout the playbook. `set_fact` is another way to set key/value pairs that allow us to use a variable across our entire `ansible-playbook` run.

First, let's take a look at a register block:

```
- name: Add Nginx Config
  template:
    src: ./../templates/nginx.conf
    dest: /var/www/nginx.conf
    register: nginx_conf_dict
```

Using register here will provide future tasks with a dictionary of values from the nginx_conf_dict variable. In the case of template, we'll have access to nginx_conf_dict.path which is the destination path for the NGINX configuration file on the host system.

Using set_fact we can take the nginx_conf_dict.path and set it to a new variable:

```
- set_fact:
    nginx_lb_conf_path:  "{{ nginx_conf_dict.path }}"
```

After we do both of these, we can debug the results with:

```
- debug: msg="nginx_conf_dict: {{ nginx_conf_dict }}"
- debug: msg="nginx_lb_conf_path: {{ nginx_lb_conf_path }}"
```

We can also use variables as a backup to nginx_lb_conf_path with:

```
- name: Debug Docker-based Nginx Conf
  vars:
    - _lb_backup_path: /etc/nginx/conf.d/default.conf
  debug: msg="{{ nginx_lb_conf_path | default(_lb_backup_path) }}"
```

The debug block shows us how to use a fallback variable if our set_fact fails to set correctly.

Let's take a look at this in our load balancer playbook:

devops/ansible/playbooks/loadbalancer.yaml

```
- hosts: loadbalancer
  become: yes
  roles:
    - ./../roles/docker
  tasks:
    - name: Verify /var/www/ exists
      file:
        path: /var/www
        state: directory
        mode: 0755
```

```
    - name: Add Nginx Config
      template:
        src: ./../templates/nginx.conf
        dest: /var/www/nginx.conf
      register: nginx_conf_dict
    - debug: msg="{{ nginx_conf_dict }}"
    - set_fact:
        nginx_lb_conf_path:  "{{ nginx_conf_dict.path }}"
    - debug: msg="{{ nginx_lb_conf_path }}"
    - name: Debug Docker-based Nginx Conf
      vars:
        - _nginx_lb_path: /etc/nginx/conf.d/default.conf
      debug: msg="{{ nginx_lb_conf_path | default(_nginx_lb_path) }}"
    - name: Has Running Docker Images
      shell: docker ps -aq >/dev/null 2>&1
      register: containers_running
      ignore_errors: yes
    - name: Docker Stop Running Containers
      shell: docker stop $(docker ps -aq)
      when: containers_running.rc == 0
    - name: Docker Remove Previous Containers
      shell: docker rm $(docker ps -aq)
      when: containers_running.rc == 0
    - name: Run Docker-based Nginx
      vars:
        - _nginx_lb_path: /etc/nginx/conf.d/default.conf
      shell: |
        docker run \
        -v {{ nginx_lb_conf_path | default(_nginx_lb_path) }}:/etc/nginx/conf.d/default.
conf:ro \
        -p 80:80 \
        -d nginx
```

Docker Container Roles

We're going to create a role that's specific to Docker containers. The current Docker role is primarily to ensure Docker is installed and running. Next, we'll implement a solution to manage Docker containers.

```
mkdir -p roles/docker_containers/tasks
mkdir -p roles/docker_containers/handlers
```

We need a task that checks if we have Docker containers running. We'll do that with:

devops/ansible/roles/docker_containers/tasks/main.yaml

```
- name: Has Running Docker Images
  shell: docker ps -aq >/dev/null 2>&1
  register: containers_running
  ignore_errors: yes
```

Now we're going to implement handlers based on these tasks. As you'll notice, one of the handlers will even run our load balancer:

devops/ansible/roles/docker_containers/handlers/main.yaml

```
- name: docker stop containers
  shell: docker stop $(docker ps -aq)
  when: containers_running.rc == 0
  ignore_errors: yes
- name: docker remove containers
  shell: docker rm $(docker ps -aq)
  when: containers_running.rc == 0
  ignore_errors: yes


- name: docker run nginx lb
  vars:
    - _nginx_lb_path: /etc/nginx/conf.d/default.conf
  shell: |
    docker run \
    --restart always \
    -v {{ nginx_lb_conf_path | default(_nginx_lb_path) }}:/etc/nginx/conf.d/default.con-
f:ro \
    -p 80:80 \
    -d nginx
```

Nothing about the above should be new here; we're just highlighting Ansible's features while also making our load balancer playbook more concise. Let's take a look:

devops/ansible/playbooks/loadbalancer.yaml

```
---
- hosts: loadbalancer
  become: yes
```

```
roles:
  - ./../roles/docker
  - ./../roles/docker_containers
vars:
  - nginx_config_dest: /var/www/nginx.conf
tasks:
  - name: Verify /var/www/ exists
    file:
      path: /var/www
      state: directory
      mode: 0755
  - name: Add Nginx Config
    template:
      src: ./../templates/nginx.conf
      dest: “{{ nginx_config_dest }}”
    register: nginx_conf_dict
  - debug: msg=”{{ nginx_conf_dict }}”
  - set_fact:
      nginx_lb_conf_path:  “{{ nginx_config_dest }}”
  - debug: msg=”{{ nginx_lb_conf_path }}”
  - name: Debug Docker-based Nginx Conf
    vars:
      - _nginx_lb_path: /etc/nginx/conf.d/default.conf


    debug: msg=”{{ nginx_lb_conf_path | default(_nginx_lb_path) }}”
  - name: Trigger Docker Container Changes
    shell: echo “Triggering docker changes”
    notify:
      - docker stop containers
      - docker remove containers
      - docker run nginx lb
    when: nginx_conf_dict.changed == true
```

A couple of things to note:

- The `set_fact` for `nginx_lb_conf_path` matches the `nginx_lb_conf_path` that’s in the handlers in the `docker_containers` role.
- In the last task on this playbook, we notify each handler in the order we want them to run. These notifications will only trigger when the NGINX configuration has changed. A load balancer will only need to change when new instance(s) are added to the mix of web app hosts.

Copy Web App Project

We're going to bring our web application into our `webapps` group and the virtual machines listed there. In the _*Clone Sample Python Web App*_ *section*, we cloned a repo called `iac-python`. It contains a Dockerfile that we intend to use.

Where you cloned that app will determine our `root_dir` variable below. In my case it's `/Users/cfe/Dev/iac-ansible`.

devops/ansible/playbooks/webapps.yaml

```
---
- hosts: webapps
  become: yes
  vars:
    root_dir: “~/Dev/iac-ansible”
    dest_dir: /var/www
  roles:
    - ./../roles/nginx_purge
    - ./../roles/docker
  tasks:
    - name: Setup /var/www/src
      file:
        path: “{{ dest_dir }}/src”
        state: directory
        mode: 0755
    - name: Copy Src folder
      copy:
        src: “{{ root_dir }}/src/”
        dest: “{{ dest_dir }}/src/”
    - copy:
        src: “{{ root_dir }}/{{ item }}”
        dest: “{{ dest_dir }}”
      with_items:
        - Dockerfile
        - requirements.txt
        - entrypoint.sh
```

Let's break this down:

- `root_dir`: Using `”~/Dev/iac-ansible”` Is a shortcut to `/Users/cfe/Dev/iac-ansible/`. This value pretty much only works on my local machine. If we want to make this project truly reusable we need a relative path to the project root such as `”./../../../”` instead. Try this out on your machine until it works as intended.
- Notice how I am reusing the `nginx_purge` role? How cool is that?

- copy : You might be tempted to copy your entire project folder -- this is not ideal because you might have a lot of files that your web app does not need to run. What's more, you might accidentally copy files that should remain secret (such as .env or inventory.ini or other sensitive files.).

There are a few important things to note about what I am attempting to accomplish with this playbook:
- A practical example using Ansible to deploy a Docker-based app with as little complexity as possible.
- This particular playbook would be better suited to run on GitHub Actions and/or GitLab CI/CD which adds complexity but, in general, a lot of files are never checked in (or sent to) repos on GitHub/GitLab limiting the possibility that you accidentally expose sensitive files.
- You can build a Docker image in many ways, this is the foundation for using Ansible to build a Docker image.

Build & Run our Web Apps

We need to update our docker_containers role to account for the Docker image we want to build and run for our webapps.

We'll add the following to devops/ansible/roles/docker_containers/handlers/main.yaml:

```
...

- name: docker build
  vars:
    _docker_app_name: app
  shell:
    cmd: docker build -f Dockerfile -t "{{ docker_app_name | default(_docker_app_name) }}"
.
    chdir: "{{ dest_dir }}"


- name: docker run app
  vars:
    _docker_app_name: app
  shell: |
    docker run \
    --restart always \
    -p 80:8001 \
    -e PORT=8001 \
    -d "{{ docker_app_name | default(_docker_app_name) }}"
```

Let's break this down:

- `docker build` This handler builds a Docker container.
- `shell:cmd:` This is the command we need to run to build our container. Notice that I am adding in `docker_app_name` so we know what tag to run.
- `shell:chdir` This shows us how we can change directories for this shell command. Remember that when we copied the files we set the destination to `/var/www`.
- `docker run app` This handler will run the app that we specify.

Here's a breakdown of the Docker commands:

- `docker run` The default Docker command to run a container image.
- `--restart always` This will ensure the container restarts if the virtual machine restarts.
- `-e PORT=8001` This is the environment variable for the Docker container to tell Python to run on port `8001`
- `-p 80:8001` Maps the external port `80` to the internal port `8001`. `8001` is the port our Python application will run on within the Docker container. Port `80` is used to expose any given host to standard HTTP traffic. (Port 80 also allows us to use `server {{ host }};` in our `nginx.conf`. If you wanted to use a different port for your load balancer configuration, such as 8312, you'd need to update `nginx.conf` to `server {{ host }}:8312;` and the above Docker port mapping from `-p 80:8001` to `-p 8312:8001`. You may also need to update your firewall settings if you have them. Sticking with port `80` simplifies things for us here.
- `-d` runs this Docker container in the background (so it can keep running after Ansible completes and so it doesn't keep Ansible running either).
- `"{{ docker_app_name | default(_docker_app_name) }}"` this is a reference to the tag we're going to use. In theory, we could use the Docker load balancer in this way too but there's no need.

Finally, let's update our `webapps` playbook:

devops/ansible/playbooks/webapps.yaml

```
---
- hosts: webapps
  become: yes
  vars:
    root_dir: "~/iac-ansible"
    dest_dir: /var/www/app
    docker_app_name: app
  roles:
    - ./../roles/docker
    - ./../roles/docker_containers
  tasks:
    - name: Setup /var/www/src
      file:
        path: "{{ dest_dir }}/src"
        state: directory
        mode: 0755
    - name: Copy Src folder
```

```
      copy:
        src: "{{ root_dir }}/src/"
        dest: "{{ dest_dir }}/src/"
      register: app_folder
    - copy:
        src: "{{ root_dir }}/{{ item }}"
        dest: "{{ dest_dir }}"
      register: app_files
      with_items:
        - Dockerfile
        - requirements.txt
        - entrypoint.sh
    - name: Trigger Build & Run
      shell: echo "Running build"
      # when: (app_files.changed) or (app_folder.changed)
      notify:
        - docker build
        - docker stop containers
        - docker remove containers
        - docker run app
```

Now let's run this:

```
ansible-playbook main.yaml
```

What you should notice is it takes *significantly* longer to run this time around. That's because we're now building the Docker container inline on *each* one of our virtual machines.

Pros & Cons of Building Docker Images on each Host

Pros

- Less complexity.
- The `build` happens on the same machine as the `run` ensuring the built image will almost certainly run.
- Less dependence on third-party services to build the image.
- Less dependence on third-party services to store/host the built image.

Cons

- Takes a long time; our machines are not optimized for building images, and we build `n` number of images for `n` number of web servers (ugh, this is not great).
- Pulls resources away from currently running application servers (`docker build` is not trivial on resources).
- As you add more features to the web app (our Python app), the likelihood of copying files that should remain hidden grows significantly.
- Does not account for best practices for building Docker images (or CI/CD pipelines)

Okay, so why build Docker images in this way? It came down to *less complexity*. Remember, this entire chapter covers Ansible and how to use it practically as an IaC tool.

Bonus: Automate with GitHub Actions

Below is a workflow to automate running Ansible Playbooks within GitHub. One of the primary things you'll need to do is create new `ssh` keys and set `ANSIBLE_PRIVATE_KEY` in your repo's secrets.

To make this workflow work:

- Your entire project must exist in a GitHub repo you own. You can import my repo.
- You must set up the following repo secrets:
  - `ANSIBLE_PRIVATE_KEY` (this is an SSH private key; you must have a corresponding SSH public key installed on your instances).
  - `LOAD_BALANCER_IP` : provision a Linode instance (with the public key from above) and store the IP address for it.
  - `WEB_APP_1_IP` , `WEB_APP_2_IP` , and `WEB_APP_3_IP` , Create 3 instance(s) for your web apps. (if you need less just update the `Create inventory file` step below.

.github/workflows/main.yaml

```
# This is a basic workflow to help you get started with GitHub Actions

name: Ansible CICD via Repo Inventory

# Controls when the workflow will run
on:
  # Allows you to run this workflow manually from the Actions tab
  workflow_dispatch:
  # Uncomment below to trigger the workflow on push or pull request events but only for
the main branch
  # push:
  #   branches: [ main ]
  # pull_request:
  #   branches: [ main ]


# A workflow run is made up of one or more jobs that can run sequentially or in parallel
jobs:
  # This workflow contains a single job called "build"
  build:
   # The type of runner that the job will run on

    runs-on: ubuntu-latest
```

```
steps:
  - uses: actions/checkout@v2
  - uses: actions/setup-python@v2
    with:
      python-version: '3.8'
  - name: Install Ansible
    run: |
      pip install ansible
  - name: Create PEM Key
    run: |
      cat << EOF > devops/ansible/private.pem
      ${{ secrets.ANSIBLE_PRIVATE_KEY }}
      EOF
  - name: Update key permissions
    run: |
      chmod 400 devops/ansible/private.pem
  - name: Create inventory file
    run: |
      cat << EOF > devops/ansible/inventory.ini
      [loadbalancer]
      ${{ secrets.LOAD_BALANCER_IP }}

      [webapps]
      ${{ secrets.WEB_APP_1_IP }}
      ${{ secrets.WEB_APP_2_IP }}
      ${{ secrets.WEB_APP_3_IP }}
      EOF
  - name: Add PEM Key Path to Ansible Config
    run: |
      cat << EOF > devops/ansible/ansible.cfg
      [defaults]
      ansible_python_interpreter='/usr/bin/python3'
      deprecation_warnings=False
      inventory=./inventory.ini
      remote_user="root"
      retries=2
      private_key_file = ./private.pem
      EOF
  - name: Run main playbook
    run: |
      cd devops/ansible
      ansible-playbook main.yaml
```

If you're interested in learning more about GitHub workflows please let me know @justinmitchel on Twitter.

Bonus 2: Integrating Ansible & Terraform

I like the combination of Ansible and Terraform managed through GitHub Actions. Assuming you did the Terraform Section here's how you'd update a few files:

devops/tf/main.tf:

```
terraform {
    required_version = ">= 0.15"
    required_providers {
        linode = {
            source = "linode/linode"
            version = "1.25.0"
        }
    }
}


provider "linode" {
    token = var.linode_pat_token
}


resource "linode_instance" "cfe-loadbalancer" {
    image = "linode/ubuntu18.04"
    label = "loadbalancer"
    group = "CFE_Terrafrom_PROJECT"
    region = var.region
    type = "g6-nanode-1"
    authorized_keys = [ var.authorized_key ]
    root_pass = var.root_user_pw
    private_ip = true
    tags = ["loadbalancer"]
}

resource "linode_instance" "cfe-pyapp" {
    count = var.linode_instance_count
    image = "linode/ubuntu18.04"
    label = "pyapp-${count.index + 1}"
    group = "CFE_Terrafrom_PROJECT"
    region = var.region
    type = "g6-nanode-1"

    authorized_keys = [ var.authorized_key ]
```

```
    root_pass = var.root_user_pw
    private_ip = true
    tags = ["webapps"]
}

resource "local_file" "ansible_inventory" {
    content = templatefile("${local.templates_dir}/ansible-inventory.tpl", { webapps=[for
host in linode_instance.cfe-pyapp.*: "${host.ip_address}"], loadbalancer="${linode_in-
stance.cfe-loadbalancer.ip_address}" })
    filename = "${local.devops_dir}/ansible/inventory.ini"
}
```

This should be *all* that you need in `main.tf` if you followed the Terraform Section exactly.

Notice that the `ansible_inventory` resource references to `filename = "${local.devops_dir}/ansible/inventory.ini"` ? This will change the inventory.ini file to match exactly what Terraform has.

devops/tf/templates/ansible-inventory.tpl

```
[webapps]
%{ for host in webapps ~}
${host}
%{ endfor ~}

[loadbalancer]
${loadbalancer}
```

How cool is this? Now you'll run:

```
cd devops/terraform
terraform apply -auto-approve
cd ../ansible
ansible-playbook main.yaml
```

You can update this so it's a GitHub Action workflow as well but that's not something we're going to cover at this time.

Chapter 6

Chef

Chapter 6
Chef

In this one, we're going to examine how to automate configuration using Chef to deploy a Python app that leverages a Docker container runtime.

One of the biggest differences between Chef and a few other IaC tools we cover in this book, is that Chef has an agent running at all times. This means that in order to make changes to your infrastructure using Chef, your Chef Infra Server must be running in addition to any nodes (aka virtual machines) you may need to update.

Chef also makes use of Ruby as a means for configuration. If you are familiar with Ruby, this is a welcome feature. If you're new to Ruby, it will take some getting used to.

A key advantage to Chef is the Chef Supermarket, which not only gives many pre-built examples that help you provision infrastructure, but will continue to give you insights into best practices in crafting your Ruby configuration files.

Linode Configurations

To get started we need a minimum of 3 Linode Instances provisioned. Login to the console and provision using the following settings:

Chef Infra Server
- Image: Ubuntu 18.04 (required)
- Min Plan: Linode 8GB

Chef Workstation
- Image: Ubuntu 18.04 (required)
- Min Plan: Linode 1GB

Chef Node
- Image: Ubuntu 18.04 (required)
- Min Plan: Linode 1GB

I highly recommend adding your SSH Keys to each instance in order to ensure you can make configuration changes.

Custom Domain

Chef **requires a custom domain** for your configuration. You can purchase a domain at sites like Name.com or GoDaddy.com but the idea is that you purchase them from a reputable source.

Once you purchase a domain, be sure to add it to Linode Domains and update your nameservers on your domain registrar.

The nameservers are:
- ns1.linode.com
- ns2.linode.com
- ns3.linode.com
- ns4.linode.com
- ns5.linode.com

The domain I used is:

```
tryiac.com
```

After I provisioned my instances above, I have the following IP addresses:
- 69.164.222.22 (for the infra-server)
- 69.164.221.67 (for the workstation)
- 69.164.222.142 (for the node)

From this, I'll update my domain's A records:
- Hostname: chef, IP Address: 69.164.222.22, TTL: 2 minutes
- Hostname: workstation.chef, IP Address: 69.164.221.67 , TTL: 2 minutes
- Hostname: node1.chef, IP Address: 69.164.222.142, TTL: 2 minutes

Configure Each Linode Instance

Now that we have domain mappings and provision instances, we need to update hostnames for each virtual machine.

Infra Server

```
ssh root@69.164.222.22
```

If you setup your SSH Keys correctly, you should be able to just login without a password.

Then update /etc/hostname

```
sudo nano /etc/hostname
```

Change it to:

```
chef.tryiac.com
```

You can also just run `sudo hostnamectl set-hostname chef.tryiac.com`. Or, to do it manually, edit the hosts file:

```
sudo nano /etc/hosts
```

Then add in:

```
127.0.0.1       chef.tryiac.com
```

After these are complete, run:

```
sudo reboot
```

Workstation

```
ssh root@69.164.221.67
```

If you setup your SSH Keys correctly, you should be able to just login without a password.

Then update `/etc/hostname`

```
sudo nano /etc/hostname
```

Change it to:

```
workstation.chef.tryiac.com
```

You can also just run `sudo hostnamectl set-hostname workstation.chef.tryiac.com`

Now update `/etc/hosts`

```
sudo nano /etc/hosts
```

```
127.0.0.1       localhost
127.0.0.1       workstation.chef.tryiac.com

69.164.222.227  chef.tryiac.com
69.164.221.67   workstation.chef.tryiac.com
69.164.222.142  node1.chef.tryiac.com
```

Be sure to include the IP address and domain of each instance for Chef.

Node 1

```
ssh root@69.164.222.142
```

If you setup your SSH Keys correctly, you should be able to just login without a password.

Then update `/etc/hostname`

```
sudo nano /etc/hostname
```

Change it to:

```
node1.chef.tryiac.com
```

You can also just run `sudo hostnamectl set-hostname node1.chef.tryiac.com`

Now update `/etc/hosts`

```
sudo nano /etc/hosts
```

```
127.0.0.1       localhost
127.0.0.1       node1.chef.tryiac.com
```

```
69.164.222.227  chef.tryiac.com
69.164.221.67   workstation.chef.tryiac.com
69.164.222.142  node1.chef.tryiac.com
```

Repeat the above steps for however many node(s) you need. In our case, a Chef node is going to run our Docker-based web application.

Install Chef Infra Server

At this point, we have the following complete:

- Provisioned Linode with Image: Ubuntu 18.04 (required) & Plan: Linode 8GB
- Mapped A Name chef.tryiac.com to 69.164.222.22 (or your IP Address)
- Host and Hostnames have been updated

SSH in

```
ssh root@chef.tryiac.com
```

You can also ssh into the IP address directly: `ssh root@69.164.222.227`

Update & Upgrade

```
sudo apt update && sudo apt upgrade
```

Download Chef Infra Server

Reference: https://downloads.chef.io/tools/infra-server

```
shell
export CHEF_SERVER_VERSION=”14.12.21”
wget “https://packages.chef.io/files/stable/chef-server/$CHEF_SERVER_VERSION/ubuntu/18.04/
chef-server-core_$CHEF_SERVER_VERSION-1_amd64.deb”

sudo dpkg -i chef-server-core_*.deb
rm chef-server-core_*.deb
```

Upgrade & Restart

```
chef-server-ctl upgrade
chef-server-ctl start
chef-server-ctl cleanup
```

You must accept the licenses to continue

Configure server

```
sudo chef-server-ctl reconfigure --accept-license
```

You must accept the licenses to continue

Default settings /etc/opscode/chef-server.rb ref

Generate Password

```
python3 -c “import secrets;print(secrets.token_urlsafe(32))”
```

Make ~/.chef Directory

```
mkdir -p ~/.chef
```

Create Chef User ref

```
chef-server-ctl user-create username_lowercased FIRST_NAME [MIDDLE_NAME] LAST_NAME EMAIL
‘PASSWORD’ (options)
```

Example:

```
chef-server-ctl user-create jmitch JUSTIN MITCHEL hello@teamcfe.com ‘opUsWaBgwSwb6NHZ7E-
MouFgqfBhwsd-3_dZurwn_Nzw’ --filename ~/.chef/user.pem
```

Make a mistake? Just run `sudo chef-server-ctl user-delete jmitch` to delete the user.

Create Chef Organization ref

```
chef-server-ctl org-create orgname_lowercased “ORG_FULL_NAME” (options) --association_user
username_lowercased --filename ~/.chef/ORG_NAME
```

Example:

```
sudo chef-server-ctl org-create cfe “Team CFE” --association_user jmitch --filename
~/.chef/org.pem
```

Make a mistake? Just run `sudo chef-server-ctl org-delete cfe` to delete the organization.

Configure Chef Workstation

Bootstrap Command

```
ssh root@workstation.chef.tryiac.com
```

```
sudo apt update && sudo apt install git
```

Reference: https://downloads.chef.io/tools/workstation

```
CHEF_WORKSTATION_VERSION=”22.1.745”
wget https://packages.chef.io/files/stable/chef-workstation/$CHEF_WORKSTATION_VERSION/
ubuntu/18.04/chef-workstation_$CHEF_WORKSTATION_VERSION-1_amd64.deb


sudo dpkg -i chef-workstation_*.deb
rm chef-workstation_*.deb
```

Setup Workstation Keys on SSH Server

Generate an SSH key on your workstation:

```
ssh-keygen -b 4096
```

Accept all the defaults with no passphrase (unless you need it)

```
ssh-copy-id root@chef.tryiac.com
```

Did you forget the password to `root@chef.tryiac.com` ? Then do this:

```
cat ~/.ssh/id_rsa.pub
```

Copy the result of `cat ~/.ssh/id_rsa.pub`

SSH into Chef Server

```
ssh root@chef.tryiac.com
```

Edit authorized keys

```
sudo nano ~/.ssh/authorized_keys
```

Add a new line and paste the results from your **workstation** `cat` command above (`cat ~/.ssh/id_rsa.pub`)

SSH back into Workstation

```
ssh root@workstation_ip
```

Generate Chef Repo on Workstation

```
cd ~/
chef generate repo chef-repo
```

You must accept the licenses to continue

This command creates:

- `~/chef-repo` containing `chefignore cookbooks data_bags environments LICENSE README.md roles`
- `~/chef-repo/.git` which means it's already driven by `git`
- You can replace `chef-repo` with a custom value or `.`. Such as `chef generate repo my-chef` or `chef generate repo .`. We are not going to change this name at this time to keep us all on the same page.

Copy Chef Server PEM Files to Workstation

```
mkdir -p ~/chef-repo/.chef/
scp root@chef.tryiac.com:~/.chef/*pem ~/chef-repo/.chef/
```

Remember that `chef-repo` is the name of the repo we created above.

Configure Knife on our Workstation

Create `~/chef-repo/.chef/config.rb`:

```
current_dir = File.dirname(__FILE__)
log_level                :info
log_location             STDOUT
node_name                '<username_lowercased>'
client_key               "user.pem"
validation_client_name   '<orgname_lowercased>-validator'
validation_key           "org.pem"
chef_server_url          'https://chef.tryiac.com/organizations/<orgname_lowercased>'
cache_type               'BasicFile'
cache_options( :path => "#{ENV['HOME']}/.chef/checksums" )
cookbook_path            ["#{current_dir}/../cookbooks"]
```

Working Example:

~/chef-repo/.chef/config.rb

```
current_dir = File.dirname(__FILE__)
log_level                :info
log_location             STDOUT
node_name                'jmitch'
```

```
client_key               “user.pem”
validation_client_name   ‘cfe-validator’
validation_key           “org.pem”
chef_server_url          ‘https://chef.tryiac.com/organizations/cfe’
cache_type               ‘BasicFile’
cache_options( :path => “#{ENV[‘HOME’]}/.chef/checksums” )
cookbook_path            [“#{current_dir}/../cookbooks”]
```

Fetch Chef-Server Certs

```
cd ~/chef-repo
knife ssl fetch
```

This will result in:

```
WARNING: Certificates from chef.tryiac.com will be fetched and placed in your trusted_cert
       directory (/root/chef-repo/.chef/trusted_certs).

       Knife has no means to verify these are the correct certificates. You should
       verify the authenticity of these certificates after downloading.
Adding certificate for chef_tryiac_com in /root/chef-repo/.chef/trusted_certs/chef_try-
iac_com.crt
```

The warning is to let you know the certificates from our Chef Infra Server will be trusted on this workstation.

Verify config.rb

```
knife client list
```

You should see `<orgname_lowercased>-validator`, in my case I saw `cfe-validator`

Configure Chef Node from your Chef Workstation

As of now, the Chef Node (linode virtual machine) has not been configured to Chef. In order to configure this node or any future nodes, we use the *Chef Workstation*.

In your *Chef Workstation* run:

```
cat /etc/hosts
```

You have at least:

```
127.0.0.1       localhost
127.0.0.1       workstation.chef.tryiac.com

69.164.222.227  chef.tryiac.com
69.164.221.67   workstation.chef.tryiac.com
69.164.222.142  node1.chef.tryiac.com
```

Notice that `69.164.222.142  node1.chef.tryiac.com` is directly tied to the IP Address for the *Chef Node*.

On your workstation:

```
knife bootstrap 69.164.222.142 -x root -P password --node-name node1.chef.tryiac.com
```

Change `password` to the one you set while provisioning this server in the Linode console.

You should see:

```
The authenticity of host ‘69.164.222.142 ()’ can’t be established.
fingerprint is SHA256:RA3y0RArhs6Z9PU3HTdGHVQvXTQZL4lE9+3/B0VVVwA.

Are you sure you want to continue connecting
? (Y/N) Y
```

Be sure to type `Y` as I have above.

This command `knife bootstrap` will configure our node.

If you see `Node node1 exists, overwrite it? (Y/N)` , that means it's already a Chef-managed node. If not, your node will be configured now.

After you configure your node, it's time to have chef automate installations for us.

Creating Cookbooks & Recipes

A cookbook is, not surprisingly, a collection of recipes. When it comes to managing our Chef project, we'll almost only use the Workstation server.

To understand the process of using cookbooks, we'll use the name `my_awesome_cookbook` . You do not have to run these commands just yet. We'll do that when we create the Docker Cookbook.

On your *Chef Workstation* run:

```
cd ~/chef-repo/cookbooks
chef generate cookbook my_awesome_cookbook
```

After you generate a cookbook, you can upload it to your _*Chef Infra Server*_ with:

```
knife cookbook upload my_awesome_cookbook
```

After you upload the cookbook to your _*Chef Infra Server*_, you add the cookbook recipe(s) to your node(s)

```
knife node run_list add node1.chef.tryiac.com ‘recipe[my_awesome_cookbook]’
```

This command does a few things:

- Targets the _*Chef Node*_ `node1.chef.tryiac.com`
- Adds `my_awesome_cookbook/recipes/default.rb` to a `run_list`
- The `run_list` has an order and it's base on when you added the recipe to it (ie the above command)

To view our current run list we can:

```
knife node show node1.chef.tryiac.com
```

From there you'll see something like:

```
Node Name:   node1.chef.tryiac.com
Environment: _default
FQDN:        node1.chef.tryiac.com
IP:          192.168.208.78
Run List:    recipe[my_awesome_cookbook]
Roles:
Recipes:
Platform:    ubuntu 18.04
Tags:
```

We can remove items from the run list using

```
knife node run_list remove node1.chef.tryiac.com 'recipe[my_awesome_cookbook]'
```

this will give you:

```
node1.chef.tryiac.com:
  run_list:
```

Finally, to actually *execute* the `Run List` on our node(s) we use the `knife ssh` ref & other examples command:

On Workstation

```
sudo knife ssh 'name:node1.chef.tryiac.com' 'sudo chef-client'
```

*To run this on **all*** nodes, you can simply use `"name:*"` instead.* like:

```
sudo knife ssh 'name:*' 'sudo chef-client'
```

If you see `root@node1.chef.tryiac.com's password:`, be sure to add your SSH keys to your node from your workstation with:

```
ssh-copy-id root@node1.chef.tryiac.com
```

We did the same thing before with the *Chef Infra Server*

On node directly

```
sudo chef-client
```

In other words, if you have an SSH session in your node(s) you can run the chef client with sudo chef-client

To summarize

1. Create a cookbook with chef generate cookbook <yourcookbook>
2. Update the recipe on cookbooks/<yourcookbook>/recipes/default.rb (we'll do this below)
3. Upload the cookbook to _*Chef Infra Server*_ after every recipe change knife cookbook upload <yourcookbook>
4. Add recipe/cookbook to the node(s) you want: knife node run_list add <nodename> "recipe[<yourcookbook>]
5. Execute the recipe with sudo knife ssh name:node1 sudo chef-client

Let's Get Practical

Now we're going to implement the process to deploy a Python web application through Docker using Chef. At this point, we've configured our environments so it's time to put it to use.

What we're doing only scratches the surface of what's possible with Chef but it's still a great way to get started down the path of learning it more.

Personally, I prefer other tools because I am not a huge fan of using Ruby. Chef recipes are written in Ruby. Luck for us they are not that complicated. Let's take a look.

Create a Docker Cookbook

In this section, we're going to use our *Chef Workstation* to create a Docker installation workbook, upload it to our *Chef Infra Server*, then add it to our *Chef Node*run list, then execute the *Chef Node* run list.

SSH into your workstation

```
ssh root@workstation.chef.tryiac.com
```

Create the docker_init Cookbook

Create the cookbook

```
cd ~/chef-repo/cookbooks
chef generate cookbook docker_init
```

Now we need to create our first configuration recipe. Each block here is called a resource docs and will help us define how we want our environment.

To me, the fundamental recipe resource goes like this

```
execute 'my_cmd_name' do
    command 'echo "hello world"'
end
```

Let's break it down:

- `execute` is a type of resource that allows us to run a command
- `my_cmd_name` just gives this block a name we decide (this isn't true for all resources)
- `do` and `end` allow us to pass configuration between
- `command`, in the `execute` resource `command` is a configuration option. In this case, `command` will run this command within a `node` (assuming this recipe is in the `run_list` for that node).

Before we use this resource block, let's think about what I want to do with raw shell commands:

```
sudo apt-get update
sudo apt-get install -y apt-transport-https ca-certificates curl software-properties-com-
mon
curl https://get.docker.com/ -o docker-bootstrap.sh
sudo sh docker-bootstrap.sh
```

The above script will install docker for us. Once it's done, we can run `docker ps` and see that docker is running.

Let's see what this looks like as a Chef Recipe:

```
sudo nano ~/chef-repo/cookbooks/docker_init/recipes/default.rb
```

```
apt_update 'Run apt-update' do
  frequency 86400
end

package 'apt-transport-https'
package 'ca-certificates'
package 'software-properties-common'

package 'Install Curl' do
  package_name "curl"
  action :install
end

execute 'Download Docker Bootstrap Script' do
    command 'curl https://get.docker.com/ -o docker-bootstrap.sh'
end

execute 'Run Docker Bootstrap Script' do
    command 'sudo sh docker-bootstrap.sh'
end

service 'docker' do
    action [:start, :enable]
end
```

Let's beak this down:

- `apt_update` is a built-in resource since it's required so often `apt_update` docs
- `package 'apt-transport-https`` is a way to ensure this package is installed. This is the shorthand way to write it. `package` docs
- `package 'Install Curl' do...` this version is the longhand way to write how to install a package.
- `execute 'Download Docker Bootstrap Script'` execute the command to download the needed script `execute` docs
- `execute 'Run Docker Bootstrap Script'` this will run downloaded script
- `service 'docker'` this just ensures that our docker service is running and is enabled. service docs

We'll do this again in a future section but here's how we would implement this cookbook:

```
knife cookbook upload docker_init
```

Now let's add this to our run list:

```
knife node run_list add node1.chef.tryiac.com 'recipe[docker_init]'
```

Now let's execute our `run_list`:

```
sudo knife ssh 'name:node1.chef.tryiac.com' 'sudo chef-client'
```

Create the webapp Cookbook

For the webapp we're going to use the public repository for the IaC Python FastAPI App. As you can see in the repo, the app has a Dockerfile already.

Since this `Dockerfile` exists, here's the raw scripting I need to do:

```
sudo apt-get update && sudo apt-get install -y git
mkdir -p /var/www/app/
cd /var/www/app/
git clone http://github.com/codingforentrepreneurs/iac-python.git .
docker build -t py_web_app -f DockerFile .
```

The next step in this script would be to use `docker run`. Before I do, I want to implement a condition that checks if *any* docker container is running with:

```
if [ "$(docker ps -q)" ]; then
    docker stop $(docker ps -a -q)
    docker rm $(docker ps -a -q)
fi
```

The reason I do this is to shutdown and remove any background services that may be running.
Now, let's run our service:

```
docker run --restart always -e PORT=8001 -p 80:8001 -d py_web_app
```

Let's break down this docker command:

- `docker run` will run a container, we could do `docker run my_container_tag` but we need more configuration
- `--restart always` is a configuration item that will cause this container to start running again if the *Chef Node* restarts (or the container fails for some reason).
- `-e PORT=8001` This adds an environment variable `PORT` set to `8001`. In this project, that environment variable is where the web server (via `gunicorn`) will run on within docker.
- `-p 80:8001` this maps the external port `80` so that our node's IP address can be mapped to port the running docker container port `8001`.
- `-d` this means run this container in detached mode. This is *very* important especially when running in a *Chef Node* (or IaC) environment.

Example Script

```
sudo apt-get update && sudo apt-get install -y git
mkdir -p /var/www/app/
cd /var/www/app/
git clone http://github.com/codingforentrepreneurs/iac-python.git .
docker build -t py_web_app -f DockerFile .
if [ “$(docker ps -q)” ]; then
    docker stop $(docker ps -a -q)
    docker rm $(docker ps -a -q)
fi
docker run --restart always -e PORT=8001 -p 80:8001 -d py_web_app
```

SSH into your workstation

```
ssh root@workstation.chef.tryiac.com
```

Create webapp cookbook

```
cd ~/chef-repo/cookbooks
chef generate cookbook webapp
```

Update the default recipe on this cookbook:

```
sudo nano ~/chef-repo/cookbooks/webapp/recipes/default.rb
```

Then add in:

```
apt_update
package "git"


directory 'Create Project Directory' do
    owner 'root'
    group 'root'
    path '/var/www/app/'
    recursive true
    mode '0755'
    action :create
end

git "Sync Git Repository" do
    repository "git://github.com/codingforentrepreneurs/iac-python.git"
    destination "/var/www/app"
    checkout_branch "main"
    action :sync
end

execute "Build App via Docker" do
    command  "docker build -t py_app -f Dockerfile ."
    cwd "/var/www/app/"
    live_stream true
end

bash 'Docker stop & Remove' do
    code <<-EOH
        if [ "$(docker ps -q)" ]; then
            docker stop $(docker ps -a -q)
            docker rm $(docker ps -a -q)
        fi
    EOH
end

execute "Run App in Background" do
    command  "docker run --restart always -p 80:8001 -e PORT=8001 -d py_app"
end
```

Let's break this down

- `apt_update` & `package "git"` we saw these both in the `docker_init` cookbook
- `directory` this is a nice resource to ensure a directory exists and has the right permissions (`directory` docs)
- `git "Sync Git Repository"` this is a built-in resource that makes syncing our github repo easy. Using `action :sync` means it will always replace the current code with what is in the repo (ignoring *Chef Node* changes) (`git` docs)
- `execute "Build App via Docker"` The new parts are `cwd` and `live_stream`. `cwd` means what directory to run this command on. We set this directory in the `git "Sync"` portion. `live_stream` means the output of this command will be shown. (`execute` docs)
- `bash 'Docker stop & Remove'` this block allows us to run a few lines of commands. It's true we could technically put all commands in here, it's not good practice. (`bash` docs)
- `execute "Run App in Background"` final execution command to run our app.

Run git commit

When we created the `chef-repo`, we also initialized a `git` repositiory for version control. Whenever you make changes, you should consider running a git commit like:

```
cd ~/chef-repo
git add --all
git commit -m "Added docker_init and webapp cookbooks"
```

What's more is you may want to add the entire `chef-repo` into a GitHub or GitLab account. That's out side the context of the scope of this book but it's something worth doing.

Docker - Pros & Cons in our Recipes

Docker is a great way to run applications since it's so flexible. This flexibilty comes at a cost. In our case, we have several pros and cons (as also mentioned in our Ansible section in relation to building docker containers on any given Chef node.

Pros

- Less complexity
- The `build` happens on the same machine as the `run` ensuring the built image will almost certainly run
- Less dependence on third party services to build the image.
- Less dependence on third party services to store/host the built image.

Cons

- Takes a long time; not only our machines not optimize for building images but we build `n` number of images for `n` number of web servers (ugh this is not great)
- Pulls resources away from currently running application servers. (`docker build` is not trivial on resources)
- As you add more features to the web app (our python app), the likelihood of copying files that should remain hidden grows significantly.
- Does not account for best practices for building docker images (or CI/CD pipelines)

In the long run, I would prefer to build my docker containers using CI/CD tools like Github Actions or Gitlab CI/CD then host my container images on either a private docker container host on Linode or utilizing Docker's official hub.docker.com.

In the short run, I think using recipes (as well as docker) in this way highlight some of the great things that Chef has to offer. Let's go ahead and update our node(s) now to see if our work paid off.

Update Nodes

Whenever we make changes to our cookbooks and the respective recipes, we should be updating the *Chef Infra Server* with these changes.

On our *Chef Workstation* we should now have two cookbooks:

- `~/chef-repo/cookbooks/docker_init`
- `~/chef-repo/cookbooks/webapp`

And 2 corresponding recipes:

- `~/chef-repo/cookbooks/docker_init/recipes/default.rb`
- `~/chef-repo/cookbooks/webapp/recipes/default.rb`

Upload Cookbooks

Before we update our nodes, we need to ensure our cookbooks are uploaded to our _*Chef Infra Server*_:

Enter your workstation (if you haven't already) with:

```
ssh root@workstation.chef.tryiac.com
```

Now

```
cd ~/chef-repo
knife cookbook upload docker_init
knife cookbook upload webapp
```

Uploading your workbooks is common. You should also consider updating the `metadata.rb` within the cookbook to manage the meta data for this cookbook (such as Version and the maintainer email and so on).

Add Recipes to 'run_list'

Our `run_list` works based on cookbooks that exist on our *Chef Infra Server* regardless of what's on our *Chef Workstation*

```
knife node run_list add node1.chef.tryiac.com "recipe[docker_init]"
knife node run_list add node1.chef.tryiac.com "recipe[webapp]"
```

Reminder: removing recipes from your node(s) you simply do: `knife node run_list remove node1.chef.tryiac.com "recipe[webapp]"`

Once you run the above commands, you should see:

```
node1.chef.tryiac.com:
  run_list:
    recipe[docker_init]
    recipe[webapp]
```

Execute chef-client on our *Chef Node(s)*

`chef-client` will run the recipes within the `run_list` in our node(s).

From the *Chef Workstation*:

```
knife ssh 'name:*' 'sudo chef-client'
```

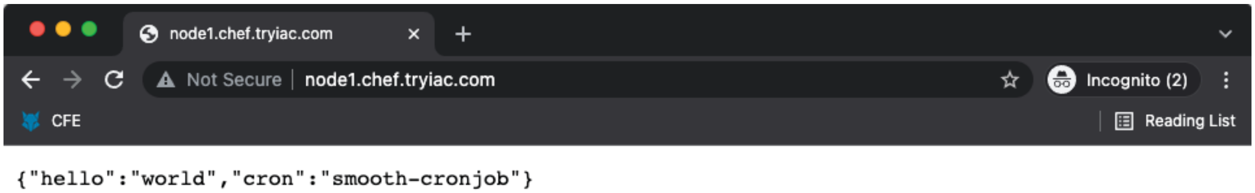
From the *Chef Node:*

```
ssh root@node1.chef.tryiac.com
sudo chef-client
```

This execution will take a good amount of time due to the fact that we're building a Docker container in the `webapp` recipe.

Review our Node

In a web browser, open up http://node1.chef.tryiac.com or whatever your domain is. What you should see is something like:



Chef Supermarket

Above we created our own cookbooks and recipes. Chef Supermarket allows you to use what other people have made. This can certainly unlock your projects in a big way as well as learn how to improve your own cookbooks. Let's look at a simple example:

```
knife cookbook site search cron-delvalidate
```

This example is also used on the the very help how-to Chef guide right on Linode.

After we search for a cookbook, we can download it to our workstation: (I assume you ran a `git` commit above).

```
cd ~/chef-repo/cookbooks
knife cookbook site download cron-delvalidate
```

After this command runs, you will see a new folder called `cron-devalidate`. This contains the following recipe:

~/chef-repo/cookbooks/cron-devalidate/recipes/default.rb

```
#
# Cookbook Name:: cron-delvalidate
# Recipe:: Chef-Client Cron & Delete Validation.pem
#
#

cron "clientrun" do
  minute '0'
  hour '*/1'
```

```
  command "/usr/bin/chef-client"
  action :create
end

file "/etc/chef/validation.pem" do
  action :delete
end
```

Notice there is a `cron` resource? Cron is a way to run tasks on a schedule. The cool think about this one is it will run `"/usr/bin/chef-client"` every 30 minutes (that's what the `hour '*/1` does).

What's more, when `chef-client` is run on our node, our entire `run_list` is also executed. This means our `webapp` cookbook/recipe can be ran every 30 minutes. In other words, our webapp would be updated every 30 minutes no matter what.

How cool is that?

Let's upload this cookbook:

```
knife cookbook upload cron-delvalidate
```

If you want to add this, update your node `run_list`

```
knife node run_list add node1 'recipe[cron-delvalidate]'
```

Another way to search the Chef Supermarket is to go to https://supermarket.chef.io/.

I did a quick search for `docker` and found https://supermarket.chef.io/cookbooks/docker. This cookbook gives me a lot of options to use with Docker; many of which this project does not need. That said, it would be an excellent cookbook to explore to see if we can get rid of our `docker_init` cookbook all together -- I'd say it's not only possible but likely. That's a challenge I'll leave up to you.

Next Steps

Now that you have Chef fully configured, I suggest you try to do the following:

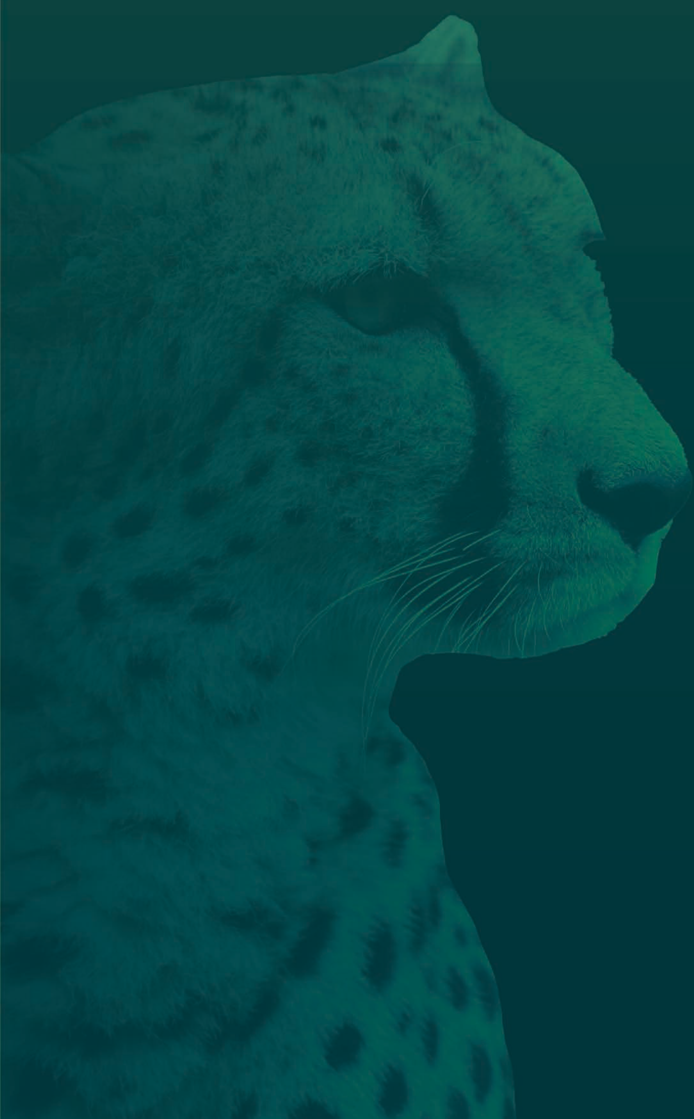
- Provision 3 more *Chef Nodes.*
- Implement 2 additional nodes using our Docker-based Python project.
- For 1 node, using `docker run --restart always -p 80:80 -d nginx` instead of the webapps one.

Clean Up

Be sure to shutdown or remove instance(s) that you have provisioned on Linode if you do not intend to use Chef going forward as they will accrue expenses for as long as you run them.

Chapter 7

Puppet Bolt



Chapter 7
Puppet Bolt

Provision Linode Instances

To get started we need a minimum of 3 Linode Instances provisioned. Login to the console and provision using the following settings:

Puppet Workstation
- Image: Ubuntu 20.04 (recommended)
- Min Plan: Linode 1GB
- Example IP Address: `45.79.174.248`

Puppet Node
- Count: 2
- Image: Ubuntu 20.04 (recommended)
- Min Plan: Linode 1GB
- Example IP Addresses: `45.79.174.212`, `45.79.174.219`

While you provision these instances consider:
- Ading your SSH Keys to each instance
- Generate User passwords with Python

Install Puppet on your Workstation

Install on Ubuntu ref

Install Puppet Bold on Ubuntu 20.04 (default choice)

Login to your workstation machine

```
ssh root@45.79.174.248
```

Declare BOLT version

```
export BOLT_VERSION=”focal”
```

Run bolt installs

```
wget "https://apt.puppet.com/puppet-tools-release-$BOLT_VERSION.deb"
sudo dpkg -i puppet-tools-release-*.deb
sudo apt-get update
sudo apt-get install puppet-bolt
rm puppet-tools-release-*.deb
```

Verify install:

```
bolt --version
```

At the time of this writing, mine responds with:

```
3.21.0
```

The command line tool `bolt` is the agentless version of Puppet. The workspace we have here is optional but it's recommended as you learn how to use *Puppet Bolt* before you move into using tools like Github Actions or Gitlab CI/CD.

Other linux system installs (optional)
If you decided to not use `Ubuntu 20.04` then you can use the following or review the installation docs:

For `Ubuntu 16.04`:

```
export BOLT_VERSION="xenial"
```

For `Ubuntu 18.04`:

```
export BOLT_VERSION="bionic"
```

For `Debian 9`:

```
export BOLT_VERSION="stretch"
```

For `Debian 10`:

```
export BOLT_VERSION=”buster”
```

For `Debian 11`:

```
export BOLT_VERSION=”bullseye”
```

After that, run:

```
wget “https://apt.puppet.com/puppet-tools-release-$BOLT_VERSION.deb”
sudo dpkg -i puppet-tools-release-*.deb
sudo apt-get update
sudo apt-get install puppet-bolt
rm puppet-tools-release-*.deb
```

Create Puppet Bolt Project

```
ssh root@45.79.174.248
```

Reminder that `45.79.174.248` is the IP Address of our workstation. Update yours as needed.

```
mkdir -p ~/iac-puppet
```

```
cd ~/iac-puppet
```

```
bolt project init iac_puppet
```

The above will generate the following:

```
iac-puppet/
    .gitignore
    bolt-project.yaml
    inventory.yaml
```

It's great that `.gitignore` is added by default so we can utilize `git` (version control) from the start of using our project.

Add our Inventory

inventory.yaml

```
groups:
  - name: webapps
    targets:
      - 45.79.174.212
      - 45.79.174.219
    config:
      transport: ssh
      ssh:
        user: root
        password: Er-WROP0OdRa0Aa23ZNJXRPW3t3hLdHA7oYsHqIaqB8
        host-key-check: false
```

Let's break this down:

- `groups` : we can leverage multiple groups of instances using puppet, for this chapter, we'll just use 1 group.
- `name` : this is the name we'll use to reference this group
- `targets` this is a list of IP addresses we provisioned for our `Puppet Node` s
- `config:transport:ssh` This means that puppet `bolt` will use an `ssh` connection (secure shell) to handle all configuration.
- `configuration:ssh:user:root` this is the default user when you provision an Linode instance
- `configuration:ssh:user:password` this is the password you set while provisioning a Linode instance.
- `host-key-check: false` this will not verify your SSH pub key against the `allowed_hosts` file.

Now that we have inventory, let's see how we can use it with a simple command:

```
bolt command run “echo ‘Hello World’” --targets webapps
```

Your result should be something like:

```
Started on 45.79.174.212...
Started on 45.79.174.219...
Finished on 45.79.174.212:
  Hello World
Finished on 45.79.174.219:
  Hello World
Successful on 2 targets: 45.79.174.212,45.79.174.219
Ran on 2 targets in 2.24 sec
```

Pretty neat huh?

For those of you that know SSH well, you probably relealise this is almost like running:

```
ssh root@45.79.174.212 echo 'Hello World'
ssh root@45.79.174.219 echo 'Hello World'
```

But with just 1 command. It's pretty neat huh?

New SSH Keys

Hard coding passwords isn't a great idea. Let's change how bolt accesses each instance by generating SSH keys.

On your Bolt manager, run:

```
ssh-keygen
```

Accept all the defaults for example:

```
root@localhost:~/iac-puppet# ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/root/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /root/.ssh/id_rsa
Your public key has been saved in /root/.ssh/id_rsa.pub
The key fingerprint is:
```

```
SHA256:6I+z8yVN71MNlnItCsP7S7417YEL5OfXv/sVIXav+O0 root@localhost
The key's randomart image is:
+---[RSA 3072]----+
|                 |
|                 |
|         .   o = |
|       .  + o O +|
|      . S .= = =.|
|     .   o+...ooo|
|      . . o++++.+|
|      oo o +*o.=+|
|      o=o   =*o+E|
+----[SHA256]-----+
```

Now your `ssh` public key is located at `~/.ssh/id_rsa.pub`. We want to copy the value of this public key to the `~/.ssh/authorized_keys` on each one of our instances.

First let's set a variable on our workstation:

```
export SSH_PUB_KEY=$(cat ~/.ssh/id_rsa.pub)
```

This command will store the value of the command `cat ~/.ssh/id_rsa.pub` to the variable `SSH_PUB_KEY`.

Now let's ensure that `~/.ssh` exists on our webapp instances:

```
bolt command run "mkdir -p ~/.ssh" --targets webapps
```

Now let's add our `SSH_PUB_KEY` to each instance at `~/.ssh/authorized_keys`

```
bolt command run "echo $SSH_PUB_KEY >> ~/.ssh/authorized_keys" --targets webapps
```

Update Inventory

Now that each instance has our workstation's public SSH key, let's update our `inventory.yaml` file.

inventory.yaml

```
groups:
  - name: webapps
    targets:
      - 45.79.16.224
      - 66.228.52.37
    config:
      transport: ssh
      ssh:
        user: root
        private-key: ~/.ssh/id_rsa
        host-key-check: false
```

The `private-key` / `public-key` authentication method for SSH is much preferred as it's more secure and also easier to move this project to different workspaces (or into Github Actions or Gitlab CI/CD).

Verifying Hosts (optional)

Above, we see `host-key-check: false`. This is so we don't see an error when we try to use bolt. Why would we see this error? We haven't approved the target hosts yet. Now that we have used bolt to update our `~/.ssh/authorized_keys`, we can easily verify our hosts:

```
ssh root@45.79.16.224
```

Accept the fingerprint

```
ssh root@66.228.52.37
```

Accept the fingerprint

Note: you can remove `host-key-check: false` if you'd like and repeat this process for future hosts. The reason this step is optional is because removing `host-key-check: false` can cause headaches when you add new host targets in the future.

Your First Bolt Module

Modules are a collection of steps that we need our group instances to run. These steps run in order and to the targets we designate (targets are typically groups that are named in `inventory.yaml`)

For our first module, we'll install `nginx` to our `webapps` group.

```
ssh root@workstation_ip
```

Replace `workstation_ip` with the IP address for your workstation. Mine is `45.79.174.248`

```
cd ~/iac-puppet
mkdir -p modules/nginx/plans
touch modules/nginx/plans/install.yaml
```

Update `modules/nginx/plans/install.yaml` to:

```
parameters:
  targets:
    type: TargetSpec

steps:
  - name: update_apt
    command: sudo apt-get update
    targets: $targets
  - name: install_nginx
    task: package
    targets: $targets
    parameters:
      action: install
      name: nginx
    description: “Install Nginx”
  - resources:
    - type: service
      title: nginx
      parameters:
        ensure: running
    targets: $targets
    description: “Set up nginx on the webservers”
```

Let's break this down:

- The format goes `modules/<your-module-name>/plans/<your-plan-name>.yaml`, this is required as we'll see shortly
- `parameters:targets:type:TargetSpec` means that we have to include a target (or targets) when we run this plan
- In `steps` we have a series of items we need bolt to execute. These are executed in order.
- `command` this is how we run an arbitary command much like `bolt command run "echo 'Hello World'" --targets webapps`
- `targets: $targets` is a reference to the `parameters` `targets`
- `task: package` is a built-in method for installing apt packages (similar to the command `sudo apt install nginx`)
- `resources:task:type:service`: this block will ensure that `nginx` is running (similar to the command `sudo service nginx start`)

Run our module

Bolt plan run `NGINX::install -t webapps`

Let's break this command down:

- `nginx::install` maps to `modules/nginx/plans/install.yaml`
- `nginx` is the name of the module
- `install` is the name of the plan
- The directory `plans` is inferred
- `-t webapps` declares the `targets` spec which is inventory of the webapps group.

After your run the above command you should see something like:

```
Starting: plan nginx::install
Starting: command 'sudo apt-get update' on 45.79.174.212, 45.79.174.219
Finished: command 'sudo apt-get update' with 0 failures in 17.43 sec
Starting: Install Nginx on 45.79.174.212, 45.79.174.219
Finished: Install Nginx with 0 failures in 51.45 sec
Starting: install puppet and gather facts on 45.79.174.212, 45.79.174.219
Finished: install puppet and gather facts with 0 failures in 27.44 sec
Starting: Set up nginx on the webservers on 45.79.174.212, 45.79.174.219
Finished: Set up nginx on the webservers with 0 failures in 13.72 sec
Finished: plan nginx::install in 1 min, 50 sec
Plan completed successfully with no result
```

Remove NGINX

The above example was meant to show how simple and effective Puppet is and can be. We do not need NGINX going forward, so now we'll create a plan to purge it.

Create modules/nginx/plans/purge.yaml :

```
sudo nano modules/nginx/plans/purge.yaml
```

Add in:

```
parameters:
  targets:
    type: TargetSpec

steps:
  - resources:
    - type: service
      title: nginx
      parameters:
        ensure: stopped
    - package: nginx
      parameters:
        ensure: absent
    targets: $targets
    description: “Stop nginx service and remove it.”
```

Then run the plan:

```
bolt plan run nginx::purge -t webapps
```

Another way to purge would be with a little more of a manual approach.

Create modules/nginx/plans/purge_alt.yaml:

```
sudo nano modules/nginx/plans/purge_alt.yaml
```

Add in:

```
yaml
parameters:
  targets:
```

```
    type: TargetSpec

steps:
  - name: stop_nginx
    command: sudo systemctl stop nginx
    targets: $targets
  - name: purge_nginx
    task: package
    targets: $targets
    parameters:
      action: uninstall
      name: nginx
    description: "Stop & remove nginx the hard Way"
```

Run this with:

```
bolt plan run nginx::purge-alt -t webapps
```

Getting the hang of it? Let's get Docker setup for us.

Docker Module

Now we're going to create our Docker module by making use of Bolt's `files` and `plans`.

Start by creating the following folders:

```
mkdir -p ~/iac-puppet/modules/docker/files
mkdir -p ~/iac-puppet/modules/docker/plans
```

In `modules/docker/files`, we're going to have the following scripts:

- `docker_init.sh`
- `docker_build.sh`
- `dokcer_run.sh`

If you want to learn more about Bash script arguments, review Appendix I

Install Docker Script

Docker has an official install script on https://get.docker.com that makes is very convenient to install the latest version of docker.

We only want to download this script if the docker command does not exist on our instance.

Create modules/docker/files/docker_init.sh:

```
sudo nano ~/iac-puppet/modules/docker/files/docker_init.sh
```

Add in:

```
#!/bin/bash

if [ ~ “$(command -v docker )”]; then
    curl https://get.docker.com -o /tmp/get-docker.sh
    sudo sh /tmp/get-docker.sh
fi
```

Again, to manually run this we would use:

```
cd ~/iac-puppet/modules/docker/files/
sudo sh docker_init.sh
```

Docker Build Container Script

This script will build our Docker container on our instances. Naturally, it requires that we run git_clone_pull.sh and docker_init.sh prior to running this one.

Create modules/docker/files/docker_build.sh:

```
sudo nano ~/iac-puppet/modules/docker/files/docker_build.sh
```

Add in:

```
#!/bin/bash

DEST=${1:-"/var/www/proj"}
TAG=${2:-"proj"}
mkdir -p $DEST
cd $DEST
docker build -t $TAG -f Dockerfile .
```

This script will attempt to build our Docker container based on two positional arguments:

- Positional argument 1 (ie `$1`) mapped to `DEST` (defaults to `/var/www/proj`)
- Positional argument 2 (ie `$2`) mapped to `TAG` (defaults to `proj`)

Again, to manually run this we would use:

```
cd ~/iac-puppet/modules/docker/files/
sudo sh docker_build.sh /var/www/proj app
```

Docker Build - Pros & Cons within Modules

Docker is a great way to run applications since it's so flexible. This flexibility comes at a cost. In our case, we have several pros and cons, in relation to building Docker containers on any given Puppet node.

Pros

- Less complexity
- The `build` happens on the same machine as the `run` ensuring the built image will almost certainly run
- Less dependence on third party services to build the image.
- Less dependence on third party services to store/host the built image.

Cons

- Takes a long time; not only our machines not optimize for building images but we build `n` number of images for `n` number of web servers (ugh this is not great)
- Pulls resources away from currently running application servers. (`docker build` is not trivial on resources)
- As you add more features to the web app (our python app), the likelihood of copying files that should remain hidden grows significantly.
- Does not account for best practices for building docker images (or CI/CD pipelines)

In the long run, I would prefer to build my docker containers using CI/CD tools like Github Actions or Gitlab CI/CD then host my container images on either a private docker container host on Linode or utilizing Docker's official hub.docker.com.

In the short run, I think using recipes (as well as Docker) in this way highlight some of the great things that Chef has to offer. Let's go ahead and update our node(s) now to see if our work paid off.

Docker Run Container Script

Now it's finally time to create our run script. This is the last script we need in order to run our container. It's true that we *could* combine each one of these scripts but I prefer to have them concise and separate so they are easier to test and to update.

Create `modules/docker/files/docker_run.sh`:

```
sudo nano ~/iac-puppet/modules/docker/files/docker_run.sh
```

Add in:

```
#!/bin/bash

TAG=${1:-"proj"}
if [ "$(docker ps -aq )" ]; then
    docker stop $(docker ps -aq)
    docker rm $(docker ps -aq)
fi

docker run --restart always -p 80:8001 -e PORT=8001 -d $TAG
```

This script will:

- Stop all other running containers
- Remove all previous running containers
- Run our container based on the tag argument

Inherent in this script is downtime for our app. The downtime should be minimal but it's definitely going to happen because we stop old running containers and then restart. In my tests, the downtime can be as small as 2 seconds but as long as 2 minutes.

This downtime is acceptable for a couple reasons:

- We're learning
- Most applications can tolerate a certain level of downtime to a point
- If we implement a NodeBalancer on Linode, we can separate our inventory into two groups to (such as `webapps-1` and `webapps-2`) and then run our `bolt plan run` for each group that needs to be upgraded.

Let's break down the `docker run` command:

- `docker run` is the default command to run a docker container
- `--restart always` is ideal so that our docker container runs if the instance restarts (for some reason)

- `-e PORT=8001` this sets the environment variable PORT to 8001 so that our Python web app within our Docker container runs at port 8001.
- `-p 80:8001` This flag maps port 80 on our linode instance (aka virtual machine) to the PORT 8001 within our Docker container (notice how it matches exactly to the environment variables)
- `-d` runs this docker container in `detach` mode which is background mode; it essentially turns this Docker application into a service that will run in the background and restart always (thanks to the --restart always flag).
- `$TAG` this will help us run a specific docker image that we built in a previous script.

Docker Install Plan

Create `modules/docker/plans/install.yaml`:

```
sudo nano ~/iac-puppet/modules/docker/plans/install.yaml
```

Add in:

```
parameters:
  targets:
    type: TargetSpec

steps:
  - name: update_apt
    command: sudo apt-get update

    targets: $targets
  - name: run_docker_init
    targets: $targets
    script: docker/docker_init.sh
```

The only real new thing in this plan is `script`; use the format `<module_name>/<file_name>` based on `~/<project_name>/modules/<module_name>/files/<file_name>`. Bolt is smart enough to find the `docker_init.sh` file within the `modules/docker/files/` directory. Pretty cool right?

Now we need to implement this plan within *another plan*. Let's have a look.

Creating our the pyapp Module

This module is made to implement our Docker-based python web application by means of Git.

Let's create our module directories:

```
mkdir -p modules/pyapp/files
mkdir -p modules/pyapp/plans
```

In `modules/pyapp/files` we're going to add the following scripts:

- `git_clone_pull.sh`

First, modules/pyapp/files/git_clone_pull.sh:

This script is designed to clone or pull a repo to a specific destination. It can be very useful in future projects as well.

```
sudo nano ~/iac-puppet/modules/pyapp/files/git_clone_pull.sh
```

Add in:

```
#!/bin/bash

if [ $# != 2 ]; then
    echo “You must use 2 arguments for DEST & REPO”
    exit 2
fi


export DEST=$1
export REPO=$2

if [ -z “$DEST” ]; then
    echo “Destination dir missing. Please add it as the first argument”
    exit 2
fi
if [ -z “$REPO” ]; then
    echo “Repo missing. Please add it as the second argument”
    exit 2
fi
```

```
mkdir -p $DEST
cd $DEST

if [ -d .git ]; then
    echo “Pulling repo in $DEST”
    git reset --hard && git pull origin main
else
    echo “Cloning $REPO to $DEST”
    git clone $REPO .
fi
```

To manually run this script on a Linux machine, you’d do something like:

```
cd ~/iac-puppet/modules/pyapp/files/
sudo sh git_clone_pull.sh /var/www/proj https://github.com/codingforentrepreneurs/iac-py-
thon
```

Or better yet, using variables like:

```
cd ~/iac-puppet/modules//files/
export DEST_FOLDER=/var/www/proj
export GIT_REPO=https://github.com/codingforentrepreneurs/iac-python
sudo sh git_clone_pull.sh $DEST_FOLDER $GIT_REPO
```

Create our pyapp plan:

In `modules/pyapp/plans` we’re going to add the following plans:

- `install.yaml`
- `run.yaml`

First, let’s start with `~/iac-puppet/modules/pyapp/plans/install.yaml`:

```
sudo nano ~/iac-puppet/modules/pyapp/plans/install.yaml
```

Add in:

```
parameters:
  targets:
    type: TargetSpec
  repo:
    type: String
    default: https://github.com/codingforentrepreneurs/iac-python
  dest:
    type: String
    default: /var/www/app/
  tag:
    type: String
    default: pyapp

steps:
  - name: install_docker
    plan: docker::install
    targets: $targets
  - name: install_git
    task: package
    targets: $targets
    parameters:
      action: install
      name: git
  - name: make_dest_dir
    command: mkdir -p /var/www/app/
    targets: $targets
  - name: git_clone_pull
    targets: $targets
    script: pyapp/git_clone_pull.sh
    arguments:
        - $dest
        - $repo
  - name: docker_build_container
    targets: $targets
    script: docker/docker_build.sh
    arguments:
        - $dest
        - $tag
  - name: docker_run_webapps
    targets: $targets
    script: docker/docker_run.sh
    arguments:
        - $tag
```

Let's break this one down:

- `parameters` listed `repo`, `dest`, and `tag` can be used within our steps just like we did with `targets`. The primary difference here is we set defaults and the type is `String` for each one.
- `plan: docker::install` Here's one of the best features of bolt - calling other modules within modules. This step will execute the `modules/docker/plans/install.yaml` plan just as if we were going to run `bolt plan run docker::install -t webapps`
- `script: pyapp/git_clone_pull.sh` just like we did in the Docker portion, we can execute a local script right here. In this case, we use the `parameters` `dest` and `repo` based on the positional arguments that `git_clone_pull.sh` requires. (this step maps to `sh git_clone_pull.sh /var/www/app/ https://github.com/codingforentrepreneurs/iac-python`)
- `script: docker/docker_build.sh` this continues with the docker module script `docker_build.sh` along with our `dest` and `tag` parameters (this step maps to `sh docker_build.sh /var/www/app/ pyapp`)
- `script: docker/docker_run.sh` yet another Docker module script. (This step maps to `sh docker_run.sh pyapp`)

Now, we can finally run these modules with:

```
bolt plan run pyapp::install -t webapps
```

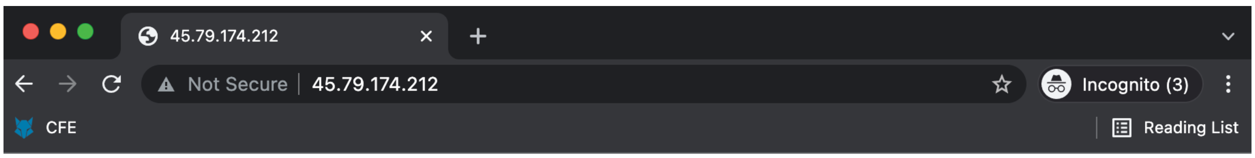
Here's the result:

```
Starting: plan pyapp::install
Starting: plan docker::install
Starting: command 'sudo apt-get update' on 45.79.174.212, 45.79.174.219
Finished: command 'sudo apt-get update' with 0 failures in 4.05 sec
Starting: script /root/iac-puppet/modules/docker/files/docker_init.sh on 45.79.174.212,
45.79.174.219
Finished: script /root/iac-puppet/modules/docker/files/docker_init.sh with 0 failures in
1.03 sec
Finished: plan docker::install in 5.12 sec
Starting: task package on 45.79.174.212, 45.79.174.219
Finished: task package with 0 failures in 2.51 sec
Starting: command 'mkdir -p /var/www/app/' on 45.79.174.212, 45.79.174.219
Finished: command 'mkdir -p /var/www/app/' with 0 failures in 0.81 sec
Starting: script /root/iac-puppet/modules/pyapp/files/git_clone_pull.sh on 45.79.174.212,
45.79.174.219
Finished: script /root/iac-puppet/modules/pyapp/files/git_clone_pull.sh with 0 failures in
1.3 sec
Starting: script /root/iac-puppet/modules/docker/files/docker_build.sh on 45.79.174.212,
45.79.174.219
Finished: script /root/iac-puppet/modules/docker/files/docker_build.sh with 0 failures in
74.55 sec
```

```
Starting: script /root/iac-puppet/modules/docker/files/docker_run.sh on 45.79.174.212,
45.79.174.219
Finished: script /root/iac-puppet/modules/docker/files/docker_run.sh with 0 failures in
2.01 sec
Finished: plan pyapp::install in 1 min, 26 sec
Plan completed successfully with no result
```

Easy enough eh?

Let's take a look at our IP address in our browser:



{"hello":"world","cron":"smooth-cronjob"}

Clean Up

Be sure to shutdown or remove instance(s) that you have provisioned on Linode if you do not intend to use your Puppet workstation and/or nodes going forward as they will accrue expenses for as long as you run them.

Chapter 7

Salt and the Salt Stack

Chapter 8
Salt & the SaltStack

In this one, we're going to examine how to automate configuration using Salt to deploy a Python app that leverages a Docker container runtime.

Provision Linode Instances

To get started we need a minimum of 3 Linode Instances provisioned. Login to the console and provision using the following settings:

Salt Master
- Image: Ubuntu 20.04 (recommended)
- Min Plan: Linode 1GB
- Example IP Address: `45.79.174.248`

Salt Minion
- Count: 2
- Image: Ubuntu 20.04 (recommended)
- Min Plan: Linode 1GB
- Example IP Addresses: `45.79.174.212` , `45.79.174.219`

While you provision these instances consider:
- Ading your SSH Keys to each instance
- Generate User passwords with Python

Step 1: Create a Master Virtual Machine

Salt uses a single master machine to control minion machines. Each minion machine can perform different actions (such as being a web app server or a load balancer or a database server and so on).

The master will orchestrate all of the minion machines with various Salt commands (as we'll see).

1. Provision & SSH
Login to Linode, provision your master virtual machine. After complete, SSH in such as:

```
ssh root@yourmasterip
```

`yourmasterip` should be the IP address that linode gives you when your provision a virtual machine.

2. Change Hostname

We're going to set our hostname to `gru` . This is an arbitrary name but since **Gru** is the master of all the **Minions** I figured it's silly enough to remember the concepts.

```
sudo hostnamectl set-hostname gru
```

An alternative method would be to change the value in `/etc/hostname` with `sudo nano /etc/hostname`

After you set your hostname be sure to change `localhost` to `gru` in `/etc/hosts`:

From

```
127.0.0.1       localhost
```

To

```
127.0.0.1       gru
```

Now reboot:

```
sudo reboot
```

After reboot finishes, login:

```
ssh root@yourmasterip
```

Verify hostname:

```
cat /etc/hostname
```

or

```
echo $HOSTNAME
```

3: Install Salt via Boostrap Script

```
curl https://bootstrap.saltproject.io/ -o bootstrap_salt.sh
sudo sh bootstrap_salt.sh -M -N
```

- `-M` creates the master
- `-N` removes the master from being a minion

4. Verify Master is Running

```
sudo service salt-master status
```

You should see or something similar (with more data too):

```
salt-master.service - The Salt Master Server
   Loaded: loaded (/lib/systemd/system/salt-master.service; enabled; vendor preset: en-
abled)
   Active: active (running) since Wed 2021-10-13 15:17:12 UTC; 5min ago
```

Create Your First Minion Virtual Machine

Each minion will be controlled by the master (as stated above) but, when needed, we can login to an individual minion to ensure the state has been applied correctly (more on state later). Minions having access to what their state should be is a great feature of Salt.

1. Provision & SSH

Login to Linode, provision your master virtual machine. After complete, ssh in such as:

```
ssh root@your_minion_1_ip
```

`your_minion_1_ip` should be the IP address that linode gives you when your provision a virtual machine.

2. Add Salt Master to your minion /etc/hosts file:

```
export SALT_MASTER_IP=yourmasterip
echo "$SALT_MASTER_IP salt" >> /etc/hosts
```

This results in something like:

```
echo "104.200.17.101 salt" >> /etc/hosts
```

Using `salt` as the master hostname is required in order for the minion to communicate with the master.

Now ping the master:

```
ping salt
```

Do you see `64 bytes from 104.200.17.101` included in the results? Good keep going.

3. Install Salt via Boostrap Script

```
curl https://bootstrap.saltproject.io/ -o bootstrap_salt.sh
sudo sh bootstrap_salt.sh
```

Notice that we do not add any parameters to our `bootstrap_salt.sh` as we did with the master.

4. (Optional) Create a Linode Image for the minion machine.

Since we're going to be adding more than 1 minion to our stack, I want to make an image to shortcut the above 3 steps.

In a larger project, I would use `terraform` with `salt` to ensure this step is tracked through version control. To keep things as simple as possible for the Salt section, we're going to just use Linode images manually.

5. Change Minion Hostname

For each minion, it's a good idea to have a unique hostname. For this one, we'll use the hostname `web1`. Check the reference above if you need alternative ways of setting the hostname.

```
sudo hostnamectl set-hostname web1
```

After you set your hostname be sure to change `localhost` to `web1` in `/etc/hosts`:

From

```
127.0.0.1       localhost
```

To

```
127.0.0.1       web1
```

Now reboot:

```
sudo reboot
```

After reboot finishes, login:

```
ssh root@your_minion_1_ip
```

Verify hostname:

```
cat /etc/hostname
```

or

```
echo $HOSTNAME
```

6. Restart Minion

```
sudo service salt-minion restart
```

7. Verify Minion is Running

```
sudo service salt-minion status
```

You should see or something similar (with more data too):

```
salt-minion.service - The Salt Minion
   Loaded: loaded (/lib/systemd/system/salt-minion.service; enabled; vendor preset: en-
abled)
   Active: active (running) since Tue 2021-10-12 04:53:41 UTC; 1 day 10h ago
```

8. Update *Master* Hosts:

SSH into Master (aka `gru`):

```
ssh root@yourmasterip
```

Update *Master* Hosts with your Minion's hostname & IP:

```
export MINION_IP=45.33.31.220
export MINION_HOSTNAME=web1
echo "$MINION_IP $MINION_HOSTNAME" >> /etc/hosts
```

Is this step required for Salt to work? Absolutely not. This is a very nice convenience to keep our minion IP addresses in a common location (`/etc/hosts`) instead of having it in many different places.

9. Accept salt-key for minion in Master

Still in your Master (aka `gru`) we need to verify the minion is valid:

```
salt-key
```

The response should look like this:

```
Accepted Keys:
Denied Keys:
Unaccepted Keys:
web1
Rejected Keys:
```

Notice that `web1` is currently in `Unaccepted Keys`. Let's accept it as a minion:

```
salt-key --accept=web1
```

Are you seeing an IP address in these keys? That means the previous step was not setup correctly for this minion. Go back and try again or even run `sudo reboot` on your master.

10. Test Total Installation

Again, in your Master. Let's ping our minion(s):

```
salt “*” test.ping
```

Here's what you should get back:

```
web1:
    True
```

The Basics of Managing State with Salt

Now that we have a master (`gru`) and a minion (`web1`) it's time to start managing state. In this case, state means the desired configuration for our minion virtual machine(s).

Like many Infrastructure as Code tools, Salt is declarative. This means we tell salt how we *want* our machine to be, and it will do all the steps to make it that way. In other words, we do not care *how* Salt arrives at the destination we just care that it does. In contrast, writing a web application is usually imperative which means you decide all the steps to arrive at a destination -- ie you care how it gets done each step of the way.

If you have been writing a bunch of imperative code (like Python, JavaScript, Ruby, etc), you might find declara-

tive code a bit like magic or downright frustrating. I have experienced both of these feelings.
In this section, I'll show you have to provision your single minion to run a single web application. It's amazing how easy it is.

First, we need to make a directory called /srv/salt on our master machine. This is the *default* name and location for these state files (aka [file roots](https://docs.saltproject.io/en/latest/ref/configuration/master.html#file-roots)). It can be changed but we wont.

Let's start with something that is super visual:

/srv/salt/nginx.sls

```
nginx:
  pkg.installed: []
  service.running:
    - require:
      - pkg: nginx
```

Each state file needs the extension sls but the file itself is a yaml file. To run this file, we have:

Now, let's update our minion state:

```
salt "*" state.apply nginx
```

Let's break down this command:

- salt using the salt cli
- "*" means all minions (more on this later)
- state.apply means we're going to applying state
- nginx refers directly to /srv/salt/nginx.sls.

After this command finishes, let's see the state of the NGINX service:

```
salt "*" cmd.run "systemctl status nginx.service"
```

Let's break this down:

- salt using the salt cli
- "*" means all minions (more on this later)
- cmd.run is how we can run any command on our minion
- "systemctl status nginx.service" is a simple command to see if the nginx service is running on our minion.

You should be able to open your IP address in a browser too and see the NGINX working html page.
Let's remove NGINX:

/srv/salt/nginx-remove.sls

```
nginx_service:
  service.dead:
    - name: nginx

nginx_removed:
  pkg.purged:
    - name: nginx
```

After you create that, run:

```
salt "*" state.apply nginx-remove
```

Better understanding state.apply

From above, if we changed `/srv/salt/nginx.sls` to `/srv/salt/nginx-start.sls` our command would be:

```
salt "*" state.apply nginx-start
```

If we wanted to just apply this to our 1 single minion we'd run:

```
salt "web1" state.apply nginx-start
```

If we wanted to just apply this to a few minions with a matching pattern we can:

```
salt "web[0-9]" state.apply nginx-start
```

In this case, the block `[0-9]` matches all numbers that are appended to `web`.

Docker & Salt

My favorite way to run any application in production is using Docker. The reason? If Docker is running, your app will run. Period.

It's true there *may* be exceptions to this but generally speaking, those exceptions are outliers and are often solved by (1) spinning up a new virtual machine with more CPUS/RAM/Storage or (2) rebooting your virtual machine.

The other thing about using Docker is the best `Dockerfile` s will give us the exact commands we need to provision the non-Docker environment. In other words, `Dockerfile` s are recipes we can follow even if we don't want to use docker.

Let's create a few files to get Docker working:

/srv/salt/docker/install.sls

```
docker_script:
   cmd.run:
     - name: curl https://get.docker.com/ -o docker.sh
     - cwd: /var/www

docker_install:
   cmd.run:
     - name: sh docker.sh
     - cwd: /var/www
     - require:
       - docker_script
```

To run this state file, we would run:

```
salt "*" state.apply docker.install
```

Take note that we did not call `salt "*" state.apply docker/install`. This matches how Python works (which Salt was written in) when accessing modules in sub folders.

Now, let's verify our Docker installation:

```
salt "*" cmd.run "docker ps"
```

Do you see something like:

```
web1:
    CONTAINER ID   IMAGE         COMMAND                  CREATED        STATUS         PORTS
NAMES
```

If so, your minions now have docker installed. Was that easy or what?

Git & Salt

Now we're going to clone a project to our minion(s). In other words, using Salt to run Git commands.

/srv/salt/git/install.sls

```
git_pkg:
  pkg.installed:
    - name: git
```

Let's break this down:

- `git_pkg` is the name I have given this block
- `pkg.installed` tells Salt install packages
- `name: git` is one of the packages to install. We can have a whole list of them here too.

Now run:

```
salt "*" state.apply git.install
```

This will ensure that `git` is installed on our minions.

After we have `git installed` we can clone our project from:

```
https://github.com/codingforentrepreneurs/iac-python.git
```

In this case, we're going to be using a public repository. To use a private repository, please review Appendix J.

/srv/salt/app/pull.sls

```
python_app_repo_clone:
  file.directory:
    - name: /var/www
  cmd.run:
    - name: git clone https://github.com/codingforentrepreneurs/iac-python.git app
    - cwd: /var/www/
    - runas: root
    - creates: /var/www/app/
```

Let's break this one down:

- `python_app_repo_clone` is the name I have given this block
- `file.directory` is a way to ensure your directory/directories are created.
- `- name: /var/www/` is the root destination for our app. If you installed nginx, this directory will already exist. Using `file.directory` just ensures that it does exist.
- `cmd.run` : This is how we run commands that we need to run.
- `- name: git clone https://github.com/codingforentrepreneurs/iac-python.git app` this is how we declare a command we want to run
- `- cwd` this is the working directory we want to run the command
- `- runas` gives you the option to use a different user to run this command. We are using root as to simplify learning Salt.
- `- creates` You should declare this if your command will end up creating a least 1 directory

Now, I can run both of these commands one after another like:

```
salt "*" state.apply git.install
salt "*" state.apply app.pull
```

Or I can create a Salt module to use both of these modules:

/srv/salt/webapp.sls

```
include:
  - git.install
  - app.pull
  - docker.install
```

Each one of these is referenced like this:

- `git.install` → `/srv/salt/git/install.sls`
- `app.pull` → `/srv/salt/app/pull.sls`
- `docker.install` → `/srv/salt/docker/install.sls`

Now I can simply run:

```
salt “*” state.apply webapp
```

How cool is that?

Templates & Salt

At this point, running `salt “*” state.apply webapp` gives us many solid advantages couple with at least 2 major flaws:

- `docker.install` will *always* attempt to install docker
- `app.pull` will *always* attempt to clone our app.

What if we could run parts of these modules based on the current state? That’s what templates allow for us to do.

Remember this command:

```
salt “*” cmd.run “echo ‘hello world’”
```

For example, let’s verify docker is installed:

```
salt “*” cmd.run ‘command -v docker’
```

The `cmd.run` allows us to execute code on our minions at will. We can use something similar within an state module (`.sls`):

/srv/salt/docker/install.sls

```
{% set has_docker = salt[‘cmd.shell’](‘command -v docker’) %}

{% if not has_docker %}
```

```
docker_script:
   cmd.run:
     - name: curl https://get.docker.com/ -o docker.sh
     - cwd: /var/www

docker_install:
   cmd.run:
     - name: sh docker.sh
     - cwd: /var/www
     - require:
       - docker_script

{% endif %}
```

Let’s dig a bit deeper

```
{% set has_docker = salt[‘cmd.shell’](‘command -v docker’) %}
```

The above line is Jinja template context item. Jinja is built-in to salt as it’s a popular Python template rendering system (fun fact, it’s inspired by the Django Template Engine but made to be used in any Python project not just Django).

- `{%` and `%}` declare a jinja-managed item. This string combination is rarely used for anything besides templates.
- `set has_docker` allows us to set the variable `has_docker` so we can use it throughout our `sls` file.
- `salt[‘cmd.run’]` is how we can run `salt` commands _within_ an `sls` file.
- `command -v` is a way to check if any given command exists on a system `command -v docker` is merely checking if the `docker` command exists.
- Now, this entire block uses salt to check if docker exists on any given minion and sets that result to the variable `has_docker`.

The block starting with `{% if not has_docker %}` and ending with `{% if endif %}` includes Salt configuration that only be executed if the minion does not have Docker).

To Clone or not to Clone?

There are two ways to think about how we manage our web application code: replacement or updating. With replacement, we would simply remove the folder `rm -rf /var/www/app/` and run `git clone ..` again.

With updating, we would use `git` to update the current state of the code. Personally, I think running `git pull` is a better method as it leverages the built-in features of version control. Further, it can let us know if any of the code was changed on a minion (something we don't want).

Now that we understand templates inside an `sls` let's update ours:

/srv/salt/app/pull.sls

```
sls
{% if not salt['file.directory_exists' ]('/var/www/app/') %}
python_app_repo:
  file.directory:
    - name: /var/www
  cmd.run:
    - name: git clone https://github.com/codingforentrepreneurs/iac-python.git app
    - cwd: /var/www/
    - runas: root
    - creates: /var/www/app/
{% else %}
python_app_repo_reset:
  cmd.run:
    - name: git reset --hard HEAD
    - cwd: /var/www/app/
    - runas: root
python_app_repo:
  cmd.run:
    - name: git pull origin main
    - cwd: /var/www/app/
    - runas: root
{% endif %}
```

Now, let's break this down:

- `salt['file.directory_exists' ]('/var/www/app/')` this command will tell us if this directory exists or not, if not, it will clone the repo into the directory `/var/www/app/`
- `python_app_repo_clone` this block is the same as before
- `python_app_repo_reset` this is where we will force a code reset. The purpose is to ensure that if our code as changed on our mininion, all of those changes would be reset. In this step, you could do additional `git` configuration to see what changes may have occurred. For example, you could make a new `branch`, `commit` the files, and `push` those changes onto your repo with this new branch.
- `python_app_repo_pull` this block merely pulls a new version of our code.

Is this simple or what?

Now when we run:

```
salt “*” state.apply app.pull
```

Salt will automatically pull the latest code OR it will clone it; whichever the minion needs.

This sets us up perfectly to start building our web app’s container with Docker.

Build Docker Image with Salt

In the IaC Python repo, there’s a Dockerfile that contains:

```
FROM python:3.8-slim

COPY . /app
WORKDIR /app

RUN apt-get update && \
    apt-get install -y \
    build-essential \
    python3-dev \
    python3-setuptools \
    gcc \
    make

# Create a virtual environment in /opt
RUN python3 -m venv /opt/venv

# Install requirments to new virtual environment
RUN /opt/venv/bin/pip install -r requirements.txt

# purge unused
RUN apt-get remove -y --purge make gcc build-essential \
    && apt-get autoremove -y \
    && rm -rf /var/lib/apt/lists/*

# make entrypoint.sh executable
RUN chmod +x entrypoint.sh

CMD [ “./entrypoint.sh” ]
```

Check https://github.com/codingforentrepreneurs/iac-python for the most up to date version of this Dockerfile and related code.

This `Dockerfile` is the basis for our container image. In order to run this container image, we need to have a built one on our system (this is certainly not the only way but it is what we'll do).

To build this container image, we would run:

```
docker build -t py_web_app -f /var/www/app/Dockerfile /var/www/app/
```

Or simply:

```
cd /var/www/app/
docker build -t py_web_app -f Dockerfile .
```

To invoke this command on your master just run:

```
salt "*" cmd.run "cd /var/www/app/;docker build -t py_web_app -f Dockerfile ."
```

Naturally, we want command to be automated (and remembered) so we'll create another state file:

/srv/salt/docker/build.sls

```
{% set has_docker  = salt['cmd.shell']('command -v docker') %}
{% if has_docker %}
docker_build:
   cmd.run:
     - name: docker build -t py_web_app -f Dockerfile .
     - cwd: /var/www/app/
{% endif %}
```

Now we can just include the template declarations we need.

So, why don't we just combine `/srv/salt/docker/install.sls` and `/srv/salt/docker/build.sls`?

You absolutely can but I think of it this way:

```
salt “*” state.apply docker.build
```

This one line makes it clear that I need docker to build my webapp. Using just, `salt “*” state.apply docker.install` seems to imply that I am merely installing Docker and not building anything.

Before we continue, let’s ask ourselves, is `/srv/salt/docker/build.sls` really the right name for this? My answer: no. Over time you’ll start to develop an intuition for where files should live. It’s true the `/srv/salt/docker/build.sls` command builds a docker container, but *which* docker container? Our project is simple right now so we know exactly which container. But as our project grows, we do not want the situation where we’re writing:

- `/srv/salt/docker/build.sls`
- `/srv/salt/docker/build2.sls`
- `/srv/salt/docker/build3.sls`
- `/srv/salt/docker/build4.sls`

or even

- `/srv/salt/docker/build-webapp.sls`
- `/srv/salt/docker/build-database.sls`

And so on. No, instead I am going to change the file to the following:

/srv/salt/app/docker/build.sls

```
{% set has_docker  = salt[‘cmd.shell’](‘command -v docker’) %}
{% if has_docker %}
docker_build:
   cmd.run:
     - name: docker build -t py_web_app -f Dockerfile .
     - cwd: /var/www/app/
{% endif %}
```

So Now, when we need to update our state we run:

```
salt “*” state.apply app.docker.build
```

Although this seems to add complexity, following this format will ensure that as our services grow, we can always build them like this:

- `salt “*” state.apply app.docker.build`
- `salt “*” state.apply db.docker.build`
- `salt “*” state.apply load_balancer.docker.build`
- `salt “*” state.apply redis.docker.build`

At a glance we can see the above commands will build our `app`, our `database`, our `load_balancer`, and `redis`, all using Docker.

The Salt Top File

This whole time we have seen commands like:

```
salt “*” state.apply docker.install
salt “*” state.apply app.pull
salt “*” state.apply app.docker.build
```

We know that `*` will run each state module against *all* minions. For learning purposes, this is fine. In practice, we want to use the Salt Top file. Before we do, let’s remember that you can run the above commands against the specific minion itself

```
salt “web1” state.apply docker.install
salt “web1” state.apply app.pull
salt “web1” state.apply app.docker.build
```

Or better yet, we can use a pattern to match to:

```
salt “web[0-9]” state.apply docker.install
salt “web[0-9]” state.apply app.pull
salt “web[0-9]” state.apply app.docker.build
```

But what if we could just run

```
salt “*” state.apply
```

And it just work? Introducing the Salt Top File:

/srv/salt/top.sls

```
base:
  '*':
    - git.install
    - docker.install
  'web[0-9]':
    - app.pull
    - app.build
```

Let's break this down:

- base . this is related to different environments that Salt can manage. We're skipping this unneeded complexity for this series but the idea is you can have different Salt environments (Dev, Prod, etc).
- '*' will apply git.install and docker.install in that order to *all* minions.
- 'web[0-9]' will match the pattern for all minions with the names web1 , web2 , and so on. This is nice for scaling horizontally.
- 'web[0-9]' will also run app.pull and app.docker.build on each minion that matches this pattern.

After we have this *Top File* (always at /srv/salt/top.sls), we can just simply run:

```
salt "*" state.apply
```

And all of our minions will fall inline with what their state should be (assuming we have correct sls state modules in the first place).

But, the Docker Run Command!

Yes, it's true we have yet to implement the docker run command for our project. This was done on purpose to bring everything together.

```
`/srv/salt/app/run.sls`
sls
{% set has_running_containers  = salt['cmd.shell']('docker ps -a -q') %}

{% if has_running_containers %}

docker_stop_all:
   cmd.run:
```

```
    - name: docker stop $(docker ps -aq)

docker_remove_all:
  cmd.run:
    - name: docker rm $(docker ps -aq)

{% endif %}

docker_run:
  cmd.run:
    - name: docker run --restart always -e PORT=8001 -p 80:8001 -d py_web_app
```

Let's break down what's happening here:

- `salt['cmd.shell']('docker ps -a -q')` this is checking if *any* containers are running on my webapp minions
- `{% if has_running_containers %}` if containers are running, stop them (the `docker_stop_all` block), and remove them (the `docker_remove_all` block).
- `docker_run` this block will run our container.

Let's break down the docker command `docker run --restart always -e PORT=8001 -p 80:8001 -d py_web_app` further:

- `docker run` will run a container, we could do `docker run my_container_tag` but we need more configuration
- `--restart always` is a configuration item that will cause this container to start running again if the minion restarts (or the container fails for some reason).
- `-e PORT=8001` This adds an environment variable `PORT` set to `8001`. In this project, that environment variable is where the web server (via `gunicorn`) will run on within docker.
- `-p 80:8001` this maps the external port `80` so that our minion's IP address can be mapped to port the running docker container port `8001`.
- `-d` this means run this container in detached mode. This is *very* important especially when running in a Salt (or IaC) environment.

Now we just need to update our top file:

/srv/salt/top.sls

```
base:
  '*':
    - git.install
    - docker.install
  'web[0-9]':
    - app.pull
    - app.build
    - app.run
```

Before we move on, let’s update this block:

```
  ‘web[0-9]’:
    - app.pull
    - app.build
    - app.run
```

to

```
  ‘web[0-9]’:
    - app.init
```

Then in `/srv/salt/app/init.sls`:

```
include:
  - pull
  - build
  - run
```

So finally ending up with the following top file:

/srv/salt/top.sls

```
base:
  ‘*’:
    - git.install
    - docker.install
  ‘web[0-9]’:
    - app
```

When you have `init.sls` within a folder, you can just use the folder name. So `/srv/salt/app/init.sls` is `app` instead of `app.init`

Using Pillars

Pillars are great way to share data across minions and they “allow confidential, targeted data to be securely sent only to the relevant minion.” Docs

Let’s create our first pillar so we can move around our webapp repo more easily:

Fisrt create the following:

```
mkdir /srv/pillar
```

/srv/pillar/data.sls

```
git_repo: https://github.com/codingforentrepreneurs/iac-python.git
docker_tag_name: iac_app
docker_port: 8002
```

Now, the *Pillar Top File*:

/srv/pillar/top.sls

```
base:
  ‘*’:
    - data
```

Now refresh this new pillar data:

```
salt ‘*’ saltutil.refresh_pillar
```

We can verify this data using:

```
salt ‘*’ pillar.items
```

In `/srv/pillar/data.sls` we have the value `git_repo` we can use this value in our state template. Let’s update our app pull template:

/srv/salt/app/pull.sls

```
{% if not salt['file.directory_exists' ]('/var/www/app/') %}
python_app_repo:
  file.directory:
    - name: /var/www
  cmd.run:
    - name: git clone {{ pillar['git_repo'] }} app
    - cwd: /var/www/
    - runas: root
    - creates: /var/www/app/
{% else %}
python_app_repo_reset:
  cmd.run:
    - name: git reset --hard HEAD
    - cwd: /var/www/app/
    - runas: root
python_app_repo:
  cmd.run:
    - name: git pull origin main
    - cwd: /var/www/app/
    - runas: root
{% endif %}
```

Notice how all we added was `{{ pillar['git_repo'] }}`? Now updating our git repo is as simple as updating the value in `/srv/pillar/data.sls`

Let's do the same thing for the `docker_tag_name` and `docker_port` value by updating the following:

- `/srv/salt/app/docker/build.sls`
- `/srv/salt/app/docker/run.sls`

First `/srv/salt/app/docker/build.sls`:

```
{% set has_docker  = salt['cmd.shell']('command -v docker') %}
{% if has_docker %}
docker_build:
   cmd.run:
     - name: docker build -t {{ pillar['docker_tag_name'] }} -f Dockerfile .
     - cwd: /var/www/app/
{% endif %}
```

Now /srv/salt/app/docker/run.sls

```
{% set has_running_containers  = salt['cmd.shell']('docker ps -a -q') %}

{% if has_running_containers %}

docker_stop_all:
   cmd.run:
     - name: docker stop $(docker ps -a -q)

docker_remove_all:
   cmd.run:
     - name: docker rm $(docker ps -a -q)

{% endif %}

docker_run:
   cmd.run:
     - name: docker run --restart always -e PORT={{ pillar['docker_port'] }} -p 80:{{ pil-
lar['docker_port'] }} -d {{ pillar['docker_tag_name'] }}
```

Thank you for reading Try IaC. It was a lot of fun putting this book together as well as making the related videos. Please contact us if you have ideas for future projects at hello@teamcfe.com.

Thank you!

Appendix A

Add SSH Keys to the Linode Console

Appendix A

Add SSH Keys to the Linode Console

To speed up nearly any project you work with on Linode, you'll need to add in your SSH keys. SSH Keys are essentially passwords between computers for secure access (SSH stands for Secure Shell). Adding them into the Linode Console makes it much easier for your workstation(s) to work with your virtual machines.

Keep in mind that this is a shortcut / helpful method to add SSH Keys with very little long term work. Sometimes, especially when you have a large team, you may need to manually add SSH keys to the instance(s) you want to give team members (or other virtual machines) access too.

Let's this section be your guide.

Step 1: Login to the Console

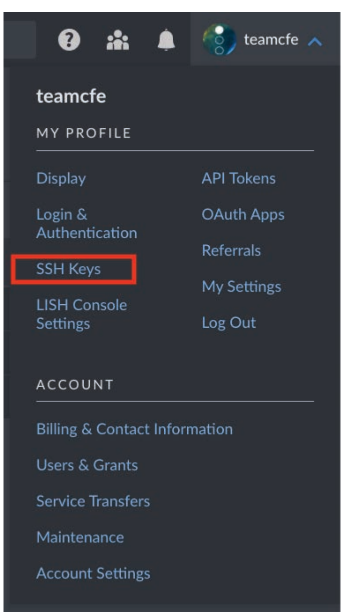
Step 2: Navigate to Linode-stored SSH Keys

- You can go to this link

or

- Click your profile dropdown.
- Click `SSH Keys` (highlighted in red)

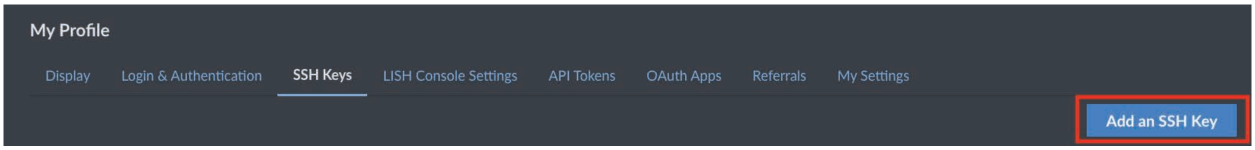
Profile Dropdown



Step 3: Add you SSH Public Key

- Click `Add an SSH Key` (highlighted in red)

Profile Navigation



- Open up `Terminal` (*macOS / Linux users*) or `PowerShell` (*Windows users*)
- Copy default SSH Public Key

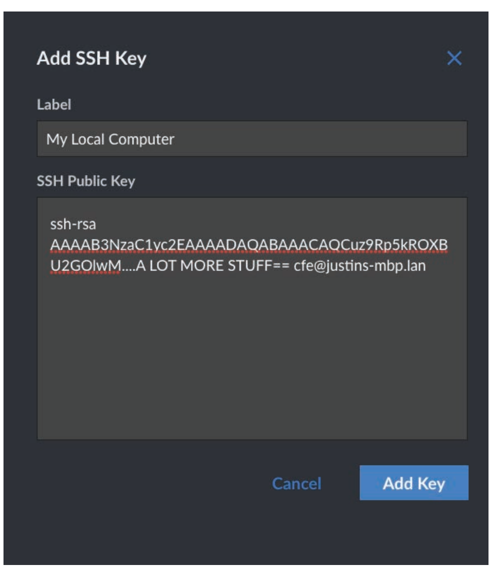
macOS / Linux users:

```
cat ~/.ssh/id_rsa.pub | pbcopy
```

Windows users

```
type ~\.ssh\id_rsa.pub | clip
```

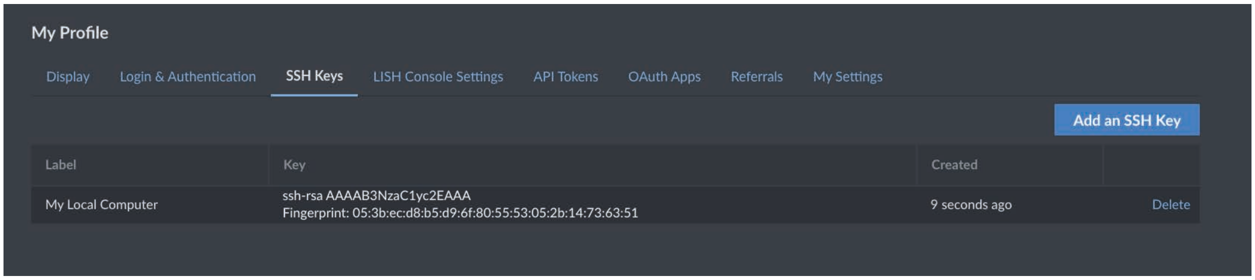
- Paste it into your Linode Console and add a Label like:



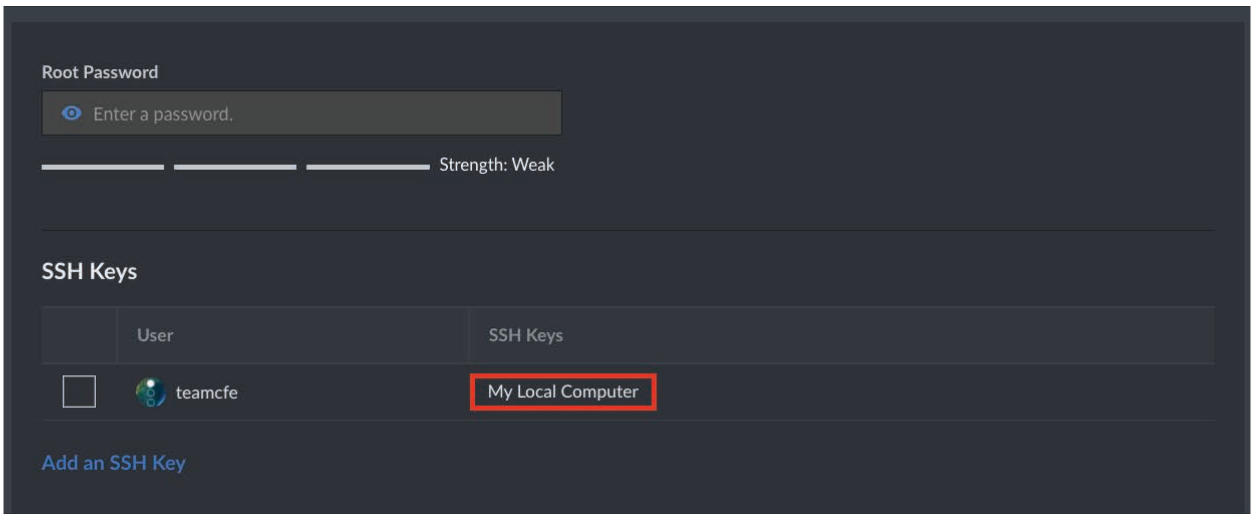
- Click `Add Key`

Step 4: Verify Key Added

In the SSH Keys section of your profile, you should see something like:



When you go to provision a Linode Instance you should see that same key on that page as well:



Where are my local SSH Keys?

macOS / Linux / Windows users:

Just run:

```
ls ~/.ssh
```

If you have keys, you'll see them here.

SSH Key Missing or Cannot Copy SSH Key

In `Step 3`, when you did:

macOS / Linux users:

```
cat ~/.ssh/id_rsa.pub | pbcopy
```

Windows users

```
type ~\.ssh\id_rsa.pub | clip
```

Did you get an error?

If so, there's a good chance you do not have an ssh key. Go to Appendix B to Generate a new key.

After your SSH Key generated, you can return to `Step 3`

I added my SSH key to a Linode Instance (or any Virtual Machine) and can no longer login, what to do?

This will require you to use a Secure Shell Session (ie `ssh` session) with the Linode Instance's root password something like:

```
ssh -o PreferredAuthentications=password -o PubkeyAuthentication=no your_root_user@your_
linode_ip
```

- Replace `your_root_user` with your root username (probably just `root`)
- Replace `your_linode_ip` with something like `45.79.58.61`

Appendix B

Generate SSH Keys

Appendix B
Generate SSH Keys

SSH Keys are great for seamless (and often password-less) access to virtual machines. We use SSH Keys a lot with IaC automation tools so it's important we know how to generate them:

Step 1: Open Command Line

Open:
- `Terminal` (*macOS / Linux users*)
- `PowerShell` (*Windows users*)
- You can also use the command line on VSCode and similar programs

Step 2: Use ssh-keygen

1. Generate a key with ssh-keygen

macOS / Linux / Windows users:

```
ssh-keygen
```

You should see:

```
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/cfe/.ssh/id_rsa):
```

Replace `/Users/cfe/.ssh/id_rsa` with your local system path(s) including your user instead of mine (`cfe`).

2. Hit return/enter to accept the default location.

3. Need to Overwrite? (Might not show up)
Is it asking you to overwrite the current value? If so, say `n` like:

```
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/cfe/.ssh/id_rsa):
/Users/cfe/.ssh/id_rsa already exists.
Overwrite (y/n)? n
```

This means your key has already been generated. You should only overwrite your key if you know your old one is not in use somewhere OR you want to make the old one invalid.

It's incredibly common to issue SSH keys so getting comfortable with adding or removing them is a great idea.

4. Generate Passphrase

```
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/cfe/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
```

That means you are creating a new SSH key. You can add a `passphrase` if you'd like but keep in mind that what you type will be hidden. Having a passphrase is a good practice but it's *okay* while you learn to leave it blank (or not have one). Later you can overwrite your ssh key (like right above here) with a passphrase.

5. Verify Key

After you finish the passphrase step, you should see:

```
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/cfe/.ssh/id_rsa): /
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/cfe/.ssh/id_rsa2
Your public key has been saved in /Users/cfe/.ssh/id_rsa2.pub
The key fingerprint is:
SHA256:70EPOoGDiobfHva0spmo/hhkoXILA+IhaevZAIe8UCA cfe@justins-
The key's randomart image is:
+---[RSA 3072]----+
|E..              |
|o+               |
|O=.              |
|X.=   . .        |
|=B. . o S o      |
|*=+o   . = o     |
|o=o.o . o o .    |
|..o+.* . o .     |
|o++o*oo   .      |
+----[SHA256]-----+
```

Now open up your command line (ie `Terminal`/`PowerShell` / etc), and run:

macOS / Linux users:

```
cat ~/.ssh/id_rsa.pub
```

Windows users

```
type ~\.ssh\id_rsa.pub
```

You should see something like:

```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQDEfREUfoaH3xkmWfJ27KU9BWmB7xlphHSNCHi+GY5PLOXn+wVGfcN
BSwxuAGhLW1Co8oJk+mG472mgnf0ub/SwjEgVxZ2I6SDhvrfS9oGoXul++3kFnRQcFBivi0/Wh1ucy8/3iK2Hmad+K0
q9EbCPGP7/GdWZHuOmgc+BfE43Xd76caC4WZFPnDxrwGr63szNGJmwqmOd/WasYXUPSA7/WMZ4KJvk1QoN7bc+wDwnK
hmjYkaveBPq4tgS5QladCqqUmMaZj+scur/2rxDlGbqgdkZeJpyPWr+tsjgWZnvjBqH3oq0wjq8GHVo/3ftmn9vWQXA
ZF8OjuECSFpC1dUHeDZD1sBLLy9fH/sA22cy3/UTk5h4w1rtSh42aL4QGrpnF9RVkaN9uh6Y2M3XZ9srnWCFuYkOFE0
hgWkLOtCbbeLfvCrmgGwxbkZHUI47h4ijPqc0q/yha5tiI2KyTSDZyoqebjv62Eg+I6HsW1K97fUvxRlBkp+Dliyy8I
uBhM8= cfe@justins-mbp.lan
```

This is a public key (hence the `.pub` in `id_rsa.pub`) and can be shared. The private key (`id_rsa` without `.pub`) should *never* be shared.

Appendix C

Create a Remote Workstation

Appendix C

Create a Remote Workstation

I highly recommend using a Remote Workstation while you're learning IaC tools. Remote workstations often remove issues when installing packages that you need (macOS & Windows run into issues installing many things that Linux does very easily).

Recommended Reading

- Add SSH Keys to the Linode Console
- Generate SSH Keys

We're going to be using VSCode as our text editor for a simple reason: the Remote-SSH Extension makes a remote virtual machine workstation feel like you're working on your local machine. VSCode works identically on macOS, Windows, & Linux.

Step 1: Login to the Console

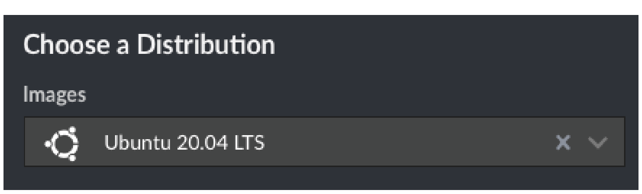
Step 2: Create a Linode Instance

Use the following configuration:

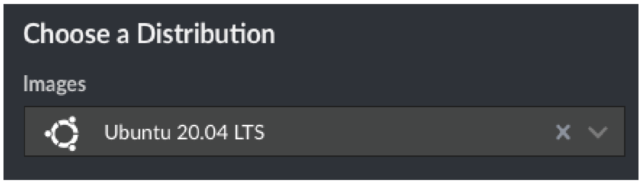
- Image Distribution: `Ubuntu 20.04`
- Region: `<pick closest to you>` (example, pick Dallas, Texas if you're in Texas like me or you can use Linode's speed test tool)
- Linode Plan: `Shared CPU` / `Nanode 1 GB` should suffice for the vast majority of Workstations
- Linode Label: `my-workstation`
- Add Tags: Such as `workstation` (Optional)
- Root Password: `<set a strong password>` (Use Appendix D to create one)
- SSH Keys: `<select a key you already added>` (Add keys with Appendix A)
- Attach a VLAN: **can skip**
- Add-ons: **can skip**
- After all options are selected, under **Linode Summary**, click `Create Linode`

Here are the screenshots:

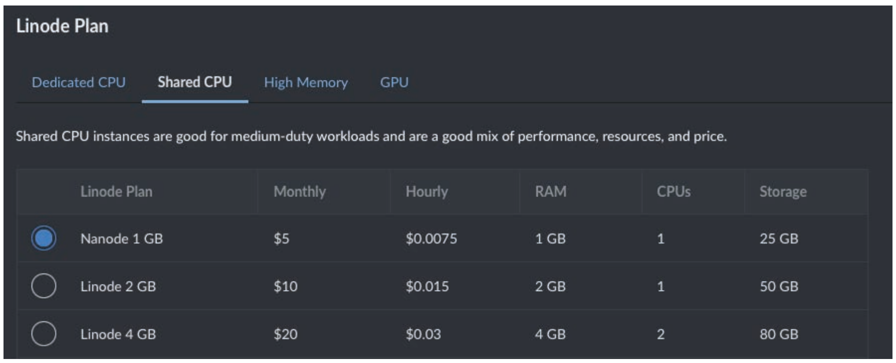
Image Distribution



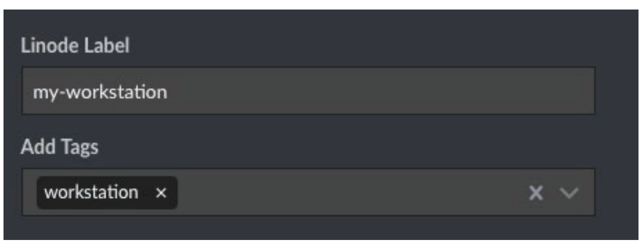
Region



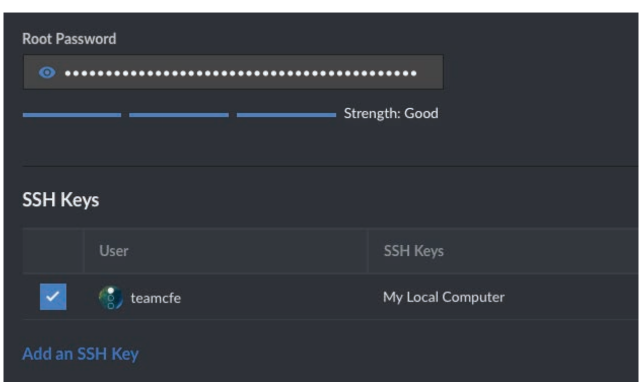
Linode Plan



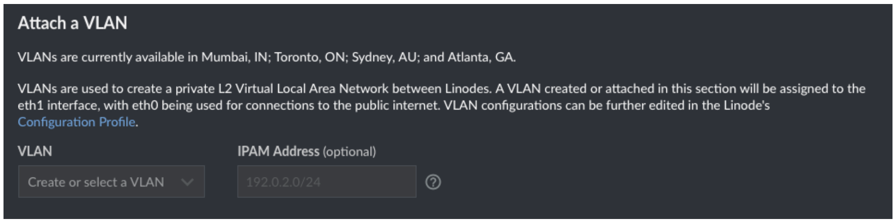
Linode Label & Add Tags



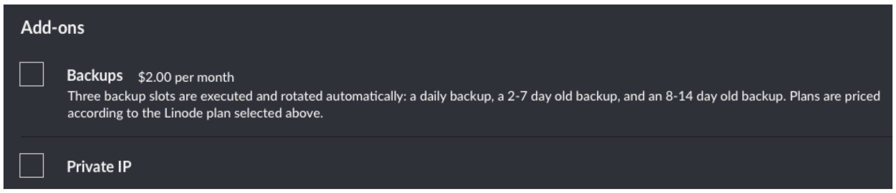
Root Password & SSH Keys



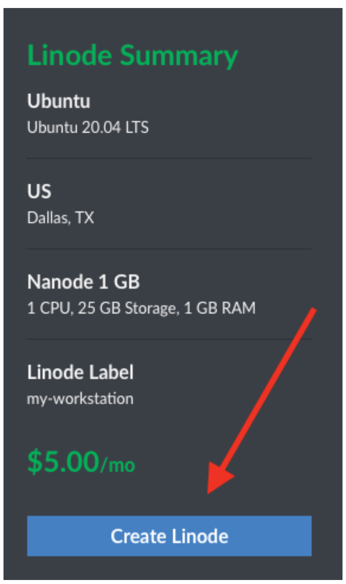
Attach a VLAN



Add Ons



Click Create Linode

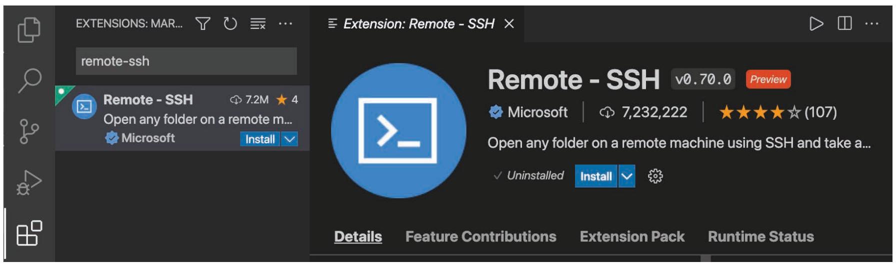


Step 3: VSCode & Extension

- Download VSCode and install it (it's free)
- Download & Install the Remote - SSH extension.

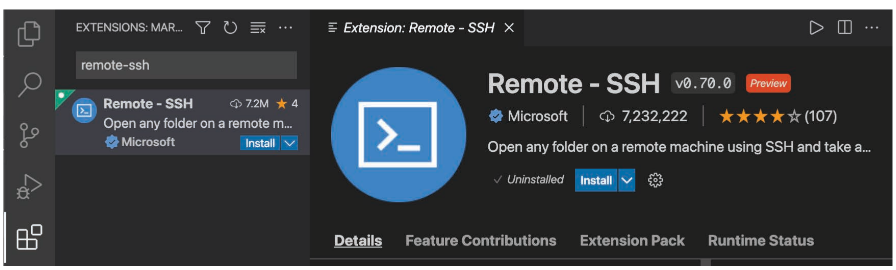
To find extensions:

- Open VSCode
- Navigate to the Extensions Marketplace
- Search for Remote-SSH
- You should see:



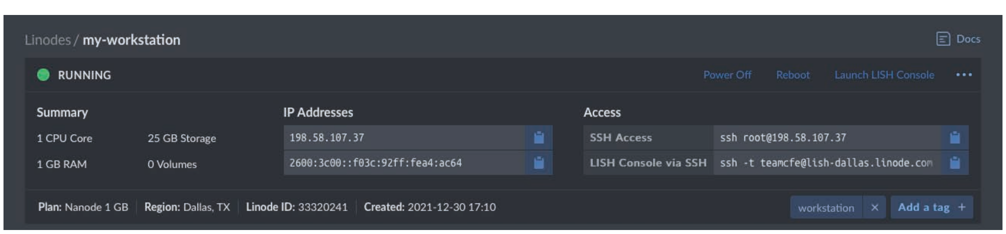
Note: this may require a VSCode restart

Remote SSH Installed

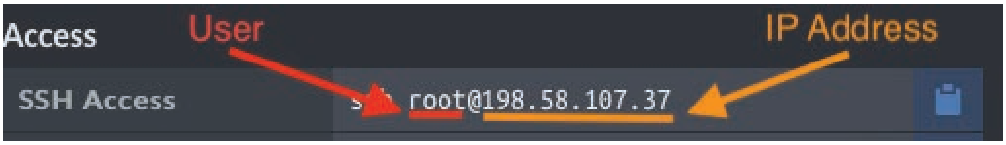


Step 4: Get Linode Instance Details

- Navigate to your list of Linode Instances
- Select `my-workstation` (or whatever you named it in Step 3)
- Copy your *SSH Access* string. Something like: `ssh root@198.58.107.37` . This includes your username `root` and your *IP Address* for this instance (`198.58.107.37`). Your IP Address will likely be different. You user will likely be `root` . Here's what it looks like:



And the SSH Info:

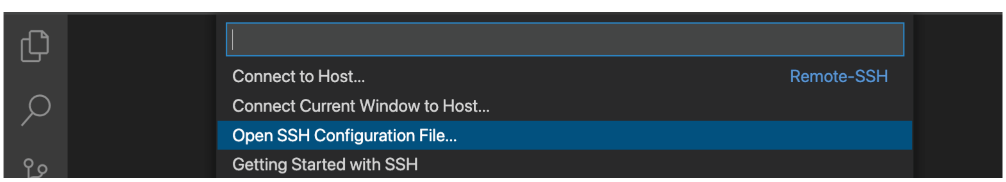


Step 5: Update SSH Config

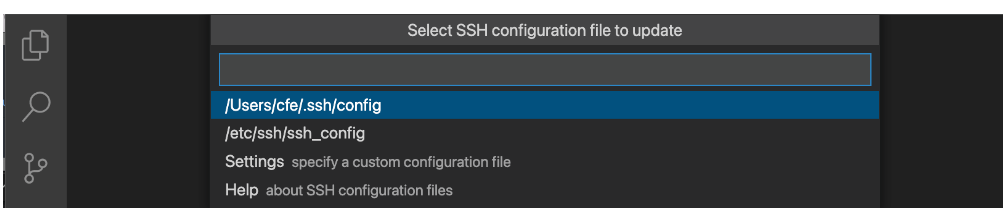
- Click the Remote button on VSCode

Remote Button

This will open the *Remote Menu*, select `Open SSH Configuration File`:



This will open the Select SSH configuration file to update, select your users' config file much like:

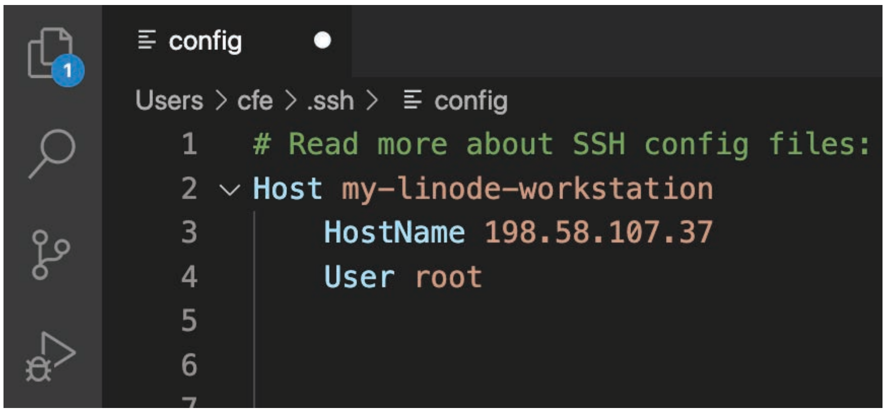


Update your config file with:

```
Host my-linode-workstation
    HostName 198.58.107.37
    User root
```

- Replace Host with any name of your choosing (without spaces)
- Replace *HostName* with your IP Address
- Replace *User* with your User
- Save the file and close

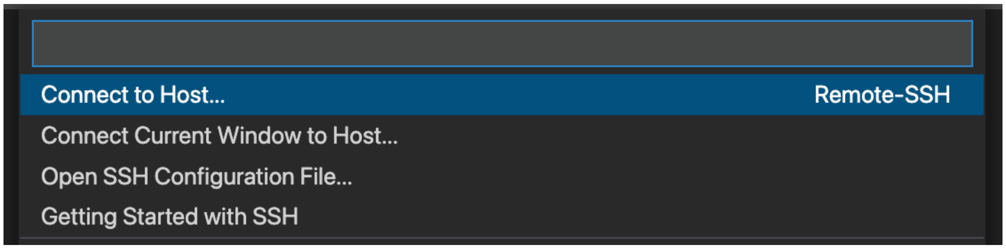
It will look like:



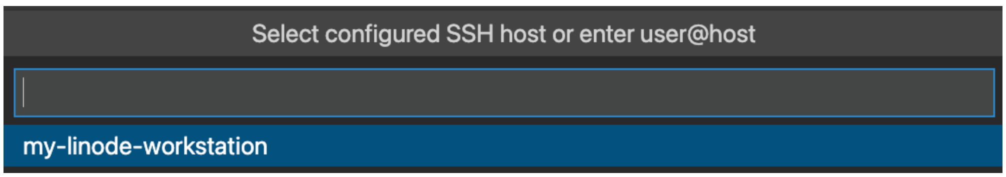
Step 6: Connect to Host

- Click the Remote button on VSCode

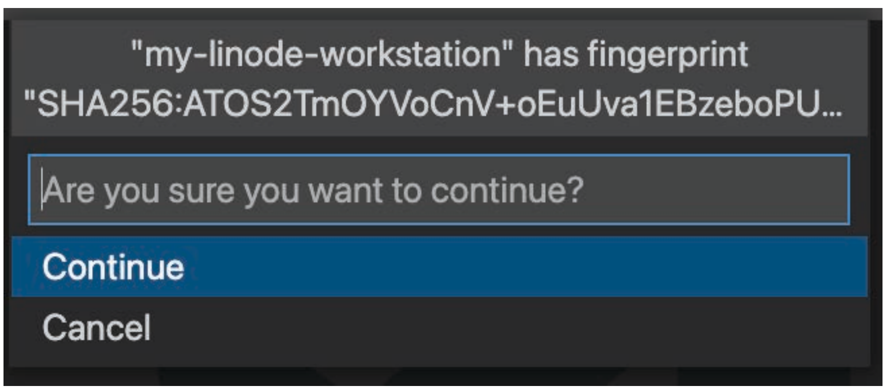
- Click `Connect to Host`



- Select `my-linode-workstation` (or what you named it in Step 3)



- If it's your first time connecting to this host, you'll see:



Select `Continue`

If you did not setup your SSH Key(s) correctly, you may be prompted for your root password. (Which is fine!)

Step 7: Connected!

Now you have a workstation! Congrats. To reconnect to your workstation just repeat Step 6!

Appendix D

Create a Password with Python

Appendix D

Create a Password with Python

Using Secrets

We'll use the built-in secrets package for Python 3 for this.

Via the command line:

Windows PowerShell

```
python -c “import secrets;print(secrets.token_urlsafe(32))”
```

macOS / Linux Terminal

```
python3 -c “import secrets;print(secrets.token_urlsafe(32))”
```

via Python3 directly:

```
# python3
import secrets
nbytes = 32
print(secrets.token_urlsafe(nbytes))
```

```
pIjVaxanJ0JaxE3AyaswNnWX_emV1C9fo5ng885dzQs
```

Using UUID

UUIDs are often used as secrets as well although I recommend the above method due to it's built-in complexity and that it's made for generating secrets (UUID is not).

Via the command line:

Windows PowerShell

```
import uuid

python -c “import uuid;print(str(uuid.uuid4()))”
```

macOS / Linux Terminal

```
python3 -c “import uuid;print(str(uuid.uuid4()))”
```

via Python3 directly:

```
#python 3
import uuid

uuid4_str = str(uuid.uuid4())
print(uuid4_str)
```

```
1750c27f-c672-4f2c-8b45-0b121b1e9e9e
```

Appendix E

Create a Linode API Token

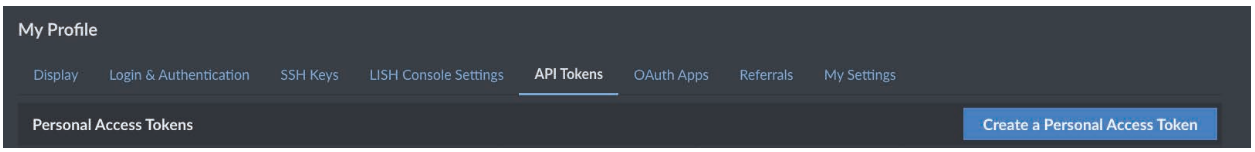
Appendix E

Create a Linode API Token

Step 1: Login to the Console

Step 2: Navigate to API Tokens

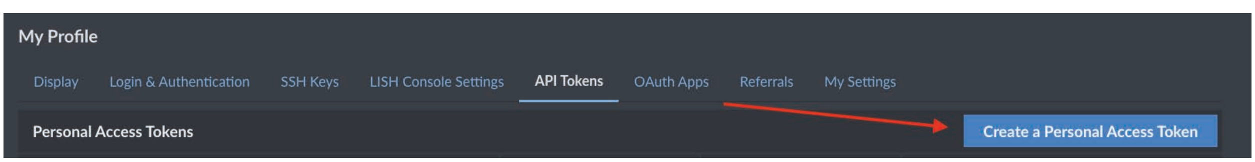
It should look like:



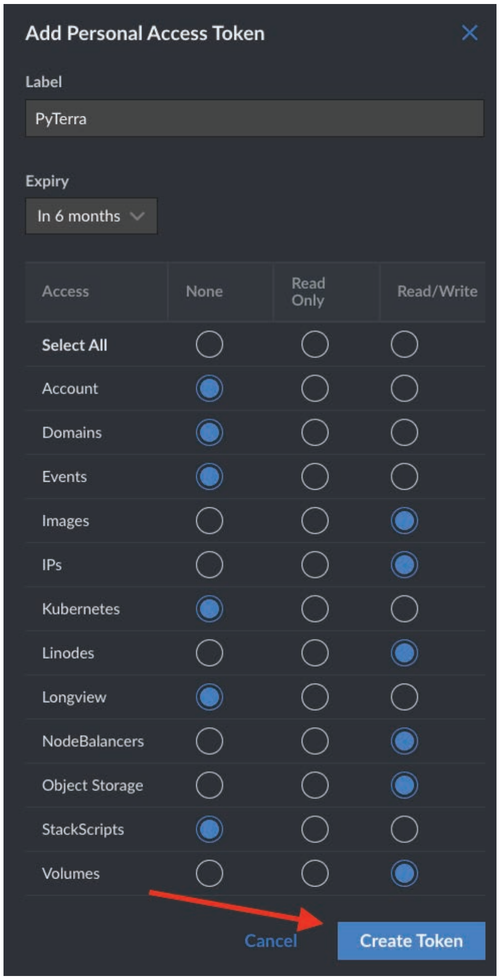
Step 3. Create a Personal Access Token

- Label: `PyTerra` (or call it what you want)
- Expiry: `In 6 months` (Choosing `never` is rarely recommended)
- Access:
  - **Images**: `Read/Write`
  - **IPs**: `Read/Write`
  - **Linodes**: `Read/Write`
  - **Node Balancers**: `Read/Write`
  - **Object Storage**: `Read/Write`
  - **Volumes**: `Read/Write`

Click Create a Personal Access Token



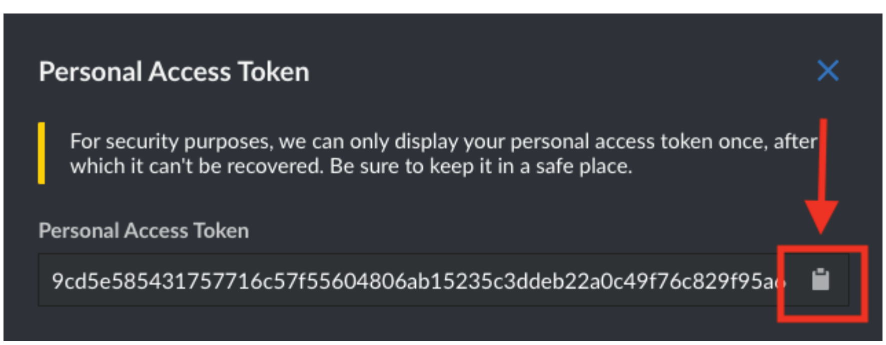
Add Personal Access Token



Step 4. Copy Token

Do **not** share this code with anyone unless you know what you're doing. When in doubt, generate a new key with steps 1-3.

Copy Personal Access Token



Appendix F

Create a Linode Object Storage Bucket

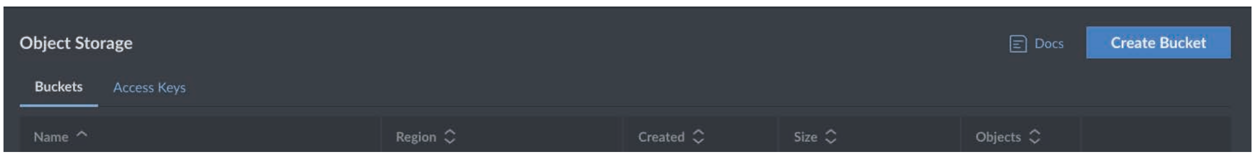
Appendix F

Create a Linode Object Storage Bucket

Step 1: Login to the Console

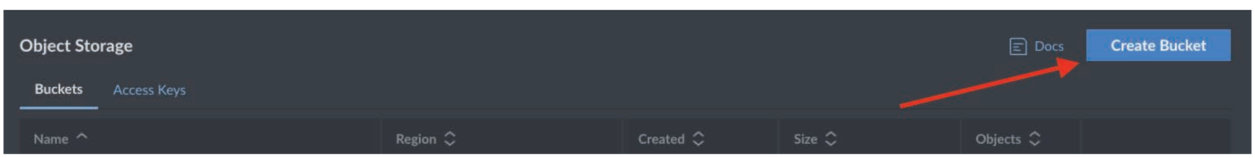
Step 2: Navigate to Object Storage

It should look like:



Step 3. Create a new bucket in Object Storage

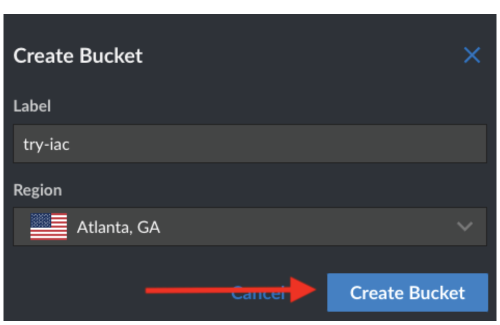
Click the create button



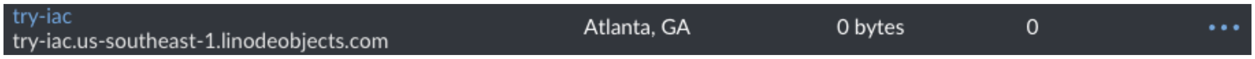
Configure the new bucket with:

- **Label**: `try-iac` (you'll have to create a unique one
- **Region**: `Alanta, GA` (or a region near you)
- Click `Create Bucket`

Configuration example:



Now in Object Storage, you should see something like:

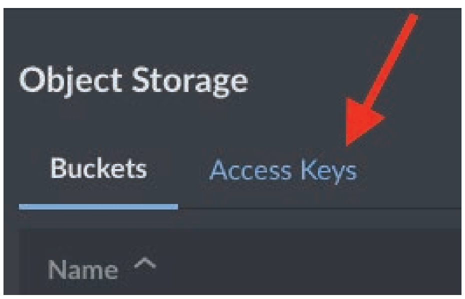


This image shows:

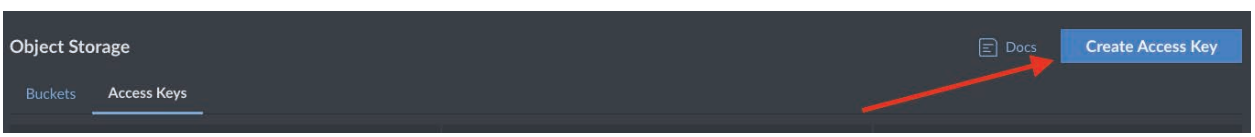
- Bucket Name as `try-iac`
- Endpoint as `try-iac.us-southeast-1.linodeobjects.com` (sometimes referred to as `AWS_S3_ENDPOINT_URL`)
- Region name as `Altanta, GA`
- Region ID as `us-souteast-1` (sometimes referred to as `AWS_S3_REGION_NAME`)

Step 4. Create Access Keys to your bucket(s).

Still in Object Storage, go to this link or click `Access Keys`



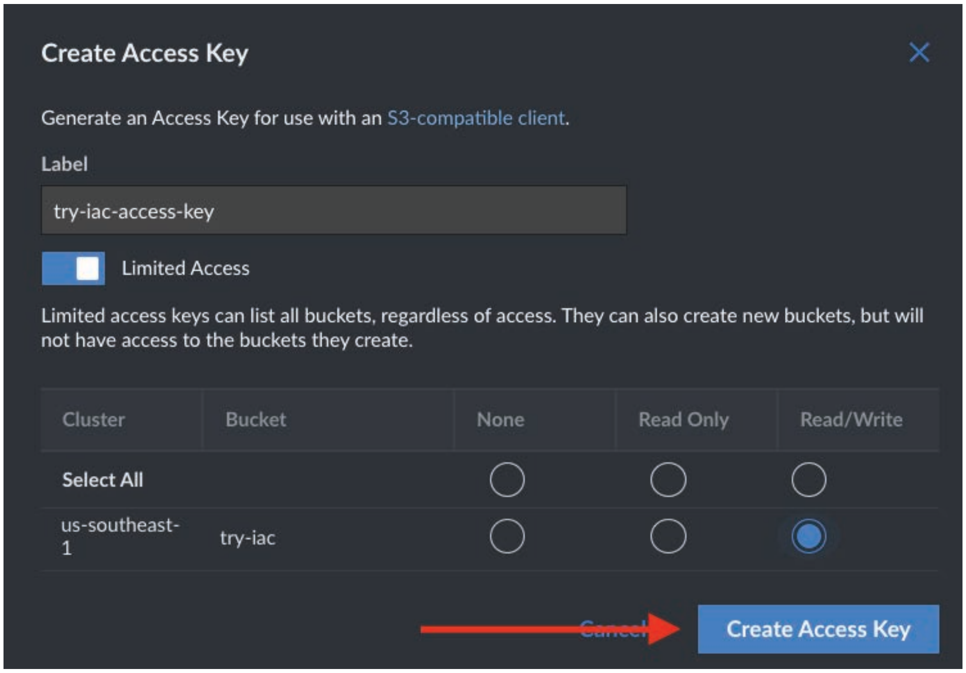
Now click the button `Create Access Key`



In v`Create Access Key`, add the following options:

- **Label**: `try-iac-access-key`
- **Limited Access**: Ensure Checked
- Under your bucket (mine is `try-iacv`) ensure `Read/Write`
- Click v`Create Access Key`

The whole thing should look like:



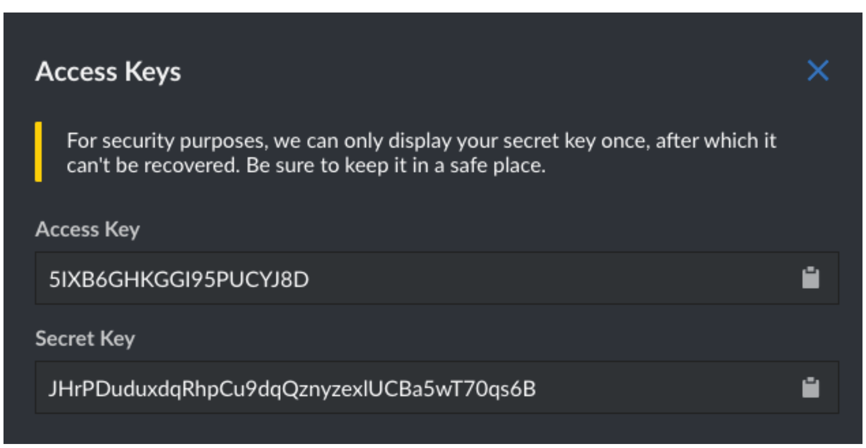
Step 5. Save Access Keys

You now have 2 keys that you can save:

- `Access Keyv` (sometimes called `Public Keyv`, `AWS_S3_ACCESS_KEY_ID`, and `AWS_ACCESS_KEY_ID`)
- `Secret Key` (sometimes called `Secret Access Key`, `AWS_S3_SECRET_ACCESS_KEY`, and `AWS_SECRET_ACCESS_KEY`)

Keep these safe. Regenerate keys as needed or every 3-6 months.

Here's what mine look like:



Appendix G

Minor Installations

Appendix G
Minor Installations

This appendix is for various installations you may need throughout the book.

1 homebrew (macOS Only)

Visit brew.sh and copy and paste the command into your terminal. At the time of this writing the command is:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/in-
stall.sh)"
```

This command is subject to change so *always* use brew.sh as your best installation source

2. tree to review folder structure

Tree is used to list the folder (directory) & file hierarchy

Linux Ubuntu:

```
sudo apt-get install tree
```

macOS

```
brew install tree
```

Windows

- No stable solution found

Appendix H

Docker & Python Web Apps

Appendix H
Docker & Python Web Apps

Throughout this book we will use Docker to run our web application. Docker is a great way to run nearly *any* web application regardless of the tech stack -- Python, Node, Java, Ruby, PHP, Nginx, Apache, etc.

It might feel more complex to use Docker, but it actually simplifies our environments significantly because if you can get Docker installed and running, your Docker-based app should run as well.

While it's not always that simple, it very often is. Here's how Docker can simply our life.

Without Docker:

- Machine 1: Python 3.7 web app
- Machine 2: Node.js 16.13 web app
- Machine 3: Python 3.9 web app
- Machine 4: Nginx
- Machine 5: Node.js 17.3 web app

With Docker:

- Machine 1: Docker
- Machine 2: Docker
- Machine 3: Docker
- Machine 4: Docker
- Machine 5: Docker
- Machine N: Docker

The idea here, if Docker is installed it can save us a lot of complexity in our infrastructure configuration. Let's take a look at a simple example:

```
git clone https://github.com/codingforentrepreneurs/iac-python
cd iac-python
docker build -t iac-python -f Dockerfile .
docker run --restart always -p 80:8001 -e PORT=8001 -d iac-python
```

Once you have Docker installed, it will be this simple for nearly *every* Docker-based project.

Let's break down what's going on above:

- `git clone ...` : this is clone (or copying) a pre-existing project that I created for this book. This project is dead-simple but has a number of system-based requirements. Review this project here on GitHub.
- `docker build ...` : This is how you build a “container image” which is essentially a mini isolated operating system.
  - `-f Dockerfile` : This tells `docker build` the location that a Dockerfile exists. A `Dockerfile` is essentially a set of instructions that docker needs to follow to create our container image.
  - `-t iac-python` : This is called `tagging` and it’s how we give a name to our Docker container image that we can use later. It’s a good idea to use unique tags that help identify your container image as you see I tagged mine with the exact same repository name I gave it on github. You can also append to the tag with `-t iac-python:some-other-tag` like `-t iac-python:v1` or v`-t iac-python:v2`
  - `.` At the end of the build command we use a period. This denotes to build this container image within the current directory. You can put any path you want here just so long the `Dockerfile` is setup to handle different paths correctly.

The `docker build` command takes time to run because it’s going to be download everything your system needs for the project to run as per what you put in the `Dockerfile`.

- `docker run ...` : This is how you run an already-built Docker image. I tend to think about this as “turning on” our Docker container image -- ie turning on a docker-based web application or turning on a docker-based database.
- `-e PORT=8001` . This sets an environment variable for our Docker container image. In our case, this will ensure our Python web app runs on port `8001` within the Docker container.
- `-p 80:8001` This maps port `80` to port `8001` . Port `80` is the default port for web traffic so we use this port to allow this pre-built Docker image that’s actually a Docker-based python web application be exposed to the world.
- `--restart always` If our machine running this container image restarts, this running container image will restart as long as is it’s running in detach mode (`-d`)
- `-d` means to run this Docker container in “detached” mode which is essentially turning the running image into a background service.
- `iac-python` is simply the name of the tagged image from the build phase.

I could write an entire book on Docker so this appendix was meant to help with context of how we use it throughout the book. If you want more, please shoot me a tweet @justinmitchel.

Appendix I

Basic Bash Scripts Arguments & Conditions

Appendix I
Basic Bash Scripts Arguments & Conditions

This is a quick guide on using positional arguments in Bash scripts.

Take the following script call:

```
sudo sh my_script.sh my_param my_other_param
```

`my_param` and `my_other_param` correspond to the `1` and `2` positional arguments. Let's take a look at how we can see this in the script `my_script.sh`:

```
#!/bin/bash

echo "Hello there param 1: $1"
echo "Hello there param 2: $2"
FALLBACK_ARG=${3:-"fallbackarg"}
echo "This script does not have a third argument, instead it will use: $FALLBACK_ARG"
```

Conditional Statements in Bash Scripts

If I wanted to stop this script if the `4` th positional argument is omitted, then we would update our script to:

```
#!/bin/bash

if [ $1 ]; then
    echo "Hello there param 1: $1"
fi

echo "Hello there param 2: $2"
FALLBACK_ARG=${3:-"fallbackarg"}
echo "This script does not have a third argument, instead it will use: $FALLBACK_ARG"

if [ -z "$4" ]; then
    echo "Argument 4 does not exist exiting."
    exit 22
```

```
fi

echo “run the rest!”
```

The `if []; then fi` block is a basic way to run conditional statements in a bash script. The `-z “$4”` is a simple condition to check if the 4 argument is empty or not.

There are many more ways to handle conditions in Bash scripts.

Appendix J

Cloning a Private Github Repo

Appendix J

Cloning a Private Github Repo

Version control through `git` is a modern marvel of programming excellence. Since `git` is a vast and fundamental tool, I'll keep this appendix highly focused on using your private code repositories through a managed service (ie Github).

If you do not know how to use `git` I recomend going to https://git-scm.com and going through some of the offocial getting started guides.

I use Github Private Repositories for the vast majority of my non open-sourced projects; Gitlab is an outstanding alternative but we'll leave that for another time.

Technically, a private repo works exactly as a public repo with the major exception: you need permission to *view/fork/clone* the code.

Before we get started I am going to assume that you have the following:

- git installed
- A github.com account
- A private repository (that your account has access to)
- A virtual machine, such as a Linode Instance, that currently lacks access to the private repo

Repository not found

When you have a public repo, cloning code is very easy:

```
git clone https://github.com/codingforentrepreneurs/iac-python.git
```

This will download all of the files in the repo for you to use.

A private repo, on the other hand will yield a different result:

```
git clone https://github.com/codingforentrepreneurs/iac-python-private.git
```

Yields:

```
git clone https://github.com/codingforentrepreneurs/iac-python-private.git
Cloning into 'iac-python-private'...
Username for 'https://github.com':
```

Hang on, it's asking for my Username and Password to GitHub? I don't want that on this machine.

or you'll see:

```
git clone github.com/codingforentrepreneurs/iac-python-private.git
fatal: repository 'github.com/codingforentrepreneurs/iac-python-private.git' does not ex-
ist
```

Notice that the first one uses `https` and the second does not. You *must* use https for personal access tokens.

There's two things to check right now:

- Does the repository actually exist?
- Does my current `git` user have permsion to the repository?

Does the repository actually exist?
Seeing `repository 'github.com/codingforentrepreneurs/iac-python-private.git does not exist` is misleading since the repo *does* exist I just did not include `https`. Sometimes, though, the repo just simply does not exist.

Does my current git user have permsion to the repository?
I can't tell you how many times I was logged in as a different Github user only to find that user was not a Collaborator on the repo itself.

The easiest way to check is my grabbing your local `git` user email:

```
git config --global user.email
```

Now you can navigate to your Github Email Settings to verify this email exists in your github account.

After you verify your email, be sure to check the private repo's settings as well by going to something like:

```
https://github.com/codingforentrepreneurs/iac-python-private/settings/access
```

The format is https://github.com/<GITHUB_USERNAME>/<REPO_NAME>/settings/access

Generate a Github Personal Access Token

The official docs are very detailed so I'll just summarize it here.

1. Login to Github
2. Navigate to Developer Settings
3. Select Personall access token
4. Click _*Generate new token*_
5. Setup:
    - *Note*: try_iac_book_token
    - *Expiration*: *30 Days*
    - *Selected Scopes*:
        - [x] Repo (all sub items too)
        - [x] Workflow
6. Click *Generate token*
7. Save result, something like `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` locally. This is your **Github Personal Access Token**.

Using a Github Personal Access Token to access a Private Repo

Now, we'll just run:

```
export TOKEN=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
git clone https://${TOKEN}:x-oauth-basic@github.com/codingforentrepreneurs/iac-python-pri-
vate.git
```

And that's it. How cool is that? Simple and doesn't require your github username and password. Plus, you can deactivate/revoke this token *at any time*.

About Akamai

Akamai powers and protects life online. Leading companies worldwide choose Akamai to build, deliver, and secure their digital experiences — helping billions of people live, work, and play every day. With the world's most distributed compute platform — from cloud to edge — we make it easy for customers to develop and run applications, while we keep experiences closer to users and threats farther away. Learn more about Akamai's security, compute, and delivery solutions at **akamai.com** and **akamai.com/blog**, or follow Akamai Technologies on **Twitter** and **LinkedIn**.

CLOUD COMPUTING SERVICES FROM

Cloud Computing Developers Trust

linode.com | Support: 855-4-LINODE | Sales: 844-869-6072
249 Arch St., Philadelphia, PA 19106 Philadelphia, PA 19106