Terraform it!

A brief guide to the Infrastructure-as-a-Code world.

Introduction

Do you like how easy we can add a new virtual host to a web server configuration? Just few lines of nginx or apache configuration and we have a new website up and running. Let's recall, how it goes sometimes...

- How to set up logging? We have to use a custom location for logs.
- Add this line.
- Nice. Now we must prevent an access to the folder A.
- Add another line. That's an example from a documentation.
- Aha, it's not accessible anymore. Good. A next point in the list is to configure caching.
- A little bit more work, but not too much. Take this as an example.
- Oooops, it isn't working now! Something is broken!
- Don't worry, take a look at the previous version of the configuration and compare differences. git diff, bla bla bla. You know.
- Oh! ...Ok, now it works. Just a stupid typo. That's amazing how *Version Control Systems* made our live easier.
- For sure. So, we have adjusted the configuration as it was required, haven't we?
- Yes, but... Now we have to set up our 42 production web servers. In exactly the same way like the staging one. Not like it was the last time when Sam forgot to reconfigure one of machines and we spent a week trying to hunt down the bug!
- Of course. That's why we introduced *Configuration Management Tool* last month. Now we can watch next series of the big bang theory while puppet delivers latest changes to the servers. Notice, much faster than human and without any "oh-i-forgot-about-it" errors.
- Yeah, that is cool. Ok, thanks for your help!
- No problem.

What are the two main concepts which stays behind of this dialogue? There are the Version Control Systems and the Configuration Management Tools. Together they have easily solved really complex issues. The VC systems like *git* or *mercurial* allow us to keep control over a source code and configuration. The CM tools like *puppet* or *chef* allow us to deliver the changes and manage with hundreds of servers runnings in a fast and consistent way. It's really amazing, how these two ideas simplified our job.

Actually, there are many powerful tools are under the hood of the modern development workflow. We develop with always hungry but convenient *IDEs*, handle with code versions with *VCS*, store projects with *remote repositories* like github, build and test everything with *Continuous Integration servers,* deliver changes and updates with *Configuration Management tools* and *Deployment Systems*. Later, in production environment we protect user data with *Backup Systems.*

But one highly important thing is missing. Which one?

Let's discuss a system components first. What is a typical web application consists of?

- The first component is obviously the *source code*. You may have a stockpile of servers but without an application source code they are just a set of expensive bricks devouring electricity.
- The second component is the *configuration*. The configuration includes a lot of different things defining an application's behaviour, from environment variables to configuration files. The main difference from the first component is that the source code will be the same on a developer's laptop and on a production server, but the configuration definitely will be different.
- The third component is the *Data.* How much will cost Facebook if they lost all user data, messages and pictures of kittens?
- And the fourth one is the *Infrastructure.* We need a place to run the *source code* considering the *configuration* and storing the user's *data.* For simple and small application it may be just a tiny server somewhere deep in a datacenter, but for a complex highly-loaded system we need a complex solution: load balancers and elastic IPs, routers and networks, dozens and hundreds of servers.

For the first three components we have a lot of great tools simplifying our job but things aren't so shiny when we talk about the infrastructure.

A long time ago virtualisation has come, and now adding a new server instance may be just a few mouse clicks, not a real installation of a hardware server into a rack. Until you have enough resources, you should not wait for a month for a new hardware to arrive. Based on a virtualisation concept, clouds were introduced, and a lot of companies, even so well-known as Netflix, completely migrated to the computing clouds.

Even with so powerful ideas like virtualisation and virtual clouds, life isn't easy enough for server administration and operations guys. Anytime when we need to change something in the infrastructure, we have to go to a cloud control panel and manually create a new instance or remove an old one, adjust something in an autoscaling group, update security rules or whatever. That's inconsistent and unreliable. Non-versionable. Non-transparent. Unclear and puzzling. It may be much better, if we may consider Infrastructure as a code or a configuration and keep it under git, but we can't...

Now we can.

The Terraform is a new tool by Hashicorp company, which fills this last gap between Version Control and Infrastructure. Its principle is extremely impressive though simple. We describe our infrastructure in a special configuration format and Terraform converts it to API calls based on your cloud provider API.

Take a look at this simple example:

Point	State	Description
1.	Initial state	We have: • An empty account on AWS as a cloud provider (for example) • A Terraform agent is installed locally We need: • A server to run an application • An elastic IP to make this app reachable • The elastic IP has to be allocated for the server instance • A security group to allow an access from web • The security group has to be assigned to the instance
2.	Initialisation	We create a configuration file called example.tf and add some essential details there: • The provider access info, e.g. access key etc. • The instance configuration • The elastic IP configuration All together it looks like that:

```
$ cat example.tf
provider "aws" {
  access_key = "XXXXXXXXXXXXXXXXXX"
  secret_key = "YYYYYYYYYYYYYYYYYY"
  region     = "eu-central-1"
}
resource "aws_instance" "web_server" {
  ami             = "ami-016e8c6e"
  instance_type   = "t2.micro"
  security_groups = ["${aws_security_group.allow_http.name}"]
}

resource "aws_eip" "web_server_ip" {
  instance = "${aws_instance.web_server.id}"
}

resource "aws_security_group" "allow_http" {
  name = "allow_http"
  ingress {
      from_port = 80
      to_port = 80
      protocol = "tcp"
      cidr_blocks = ["0.0.0.0/0"]
  }
}
```

<table>
<tr>
<td>3.</td>
<td>Planning</td>
<td>

We ask terraform agent to plan required actions:

```
$ terraform plan
Refreshing Terraform state prior to plan...

The Terraform execution plan has been generated and is shown below.
Resources are shown in alphabetical order for quick scanning. Green reso
will be created (or destroyed and then created if an existing resource
exists), yellow resources are being changed in-place, and red resources
will be destroyed.

+ aws_eip.web_server_ip
    allocation_id:     "" => "<computed>"
    association_id:    "" => "<computed>"
    domain:            "" => "<computed>"
    instance:          "" => "${aws_instance.web_server.id}"
    network_interface: "" => "<computed>"
    private_ip:        "" => "<computed>"
    public_ip:         "" => "<computed>"

+ aws_instance.web_server
    ami:                      "" => "ami-d3c022bc"
    availability_zone:        "" => "<computed>"
    ebs_block_device.#:       "" => "<computed>"
    ephemeral_block_device.#: "" => "<computed>"
    instance_state:           "" => "<computed>"
    instance_type:            "" => "t2.micro"
    key_name:                 "" => "<computed>"
    placement_group:          "" => "<computed>"
    private_dns:              "" => "<computed>"
    private_ip:               "" => "<computed>"
    public_dns:               "" => "<computed>"
    public_ip:                "" => "<computed>"
    ...

+ aws_security_group.allow_http
    description:                             "" => "Managed by Terraform"
    egress.#:                                "" => "<computed>"
    ingress.#:                               "" => "1"
    ingress.2214680975.cidr_blocks.#:        "" => "1"
    ingress.2214680975.cidr_blocks.0:        "" => "0.0.0.0/0"
    ingress.2214680975.from_port:            "" => "80"
    ingress.2214680975.protocol:             "" => "tcp"
    ingress.2214680975.security_groups.#:    "" => "0"
    ingress.2214680975.self:                 "" => "false"
    ingress.2214680975.to_port:              "" => "80"
    name:                                    "" => "allow_http"
    owner_id:                                "" => "<computed>"
    vpc_id:                                  "" => "<computed>"

Plan: 3 to add, 0 to change, 0 to destroy.
```

Terraform considered our empty cloud and defined that we have to create one elastic IP, one security group and one planning step.

</td>
</tr>
</table>

4.	Applying	With this plan we ask Terraform to apply planned changes.

```
$ terraform apply

aws_security_group.allow_http: Creating...
  description:                          "" => "Managed by Terraform"
  egress.#:                             "" => "<computed>"
  ingress.#:                            "" => "1"
  ingress.2214680975.cidr_blocks.#:     "" => "1"
  ingress.2214680975.cidr_blocks.0:     "" => "0.0.0.0/0"
  ingress.2214680975.from_port:         "" => "80"
  ingress.2214680975.protocol:          "" => "tcp"
  ...
aws_security_group.allow_http: Creation complete
aws_instance.web_server: Creating...
  ami:                          "" => "ami-016e8c6e"
  availability_zone:            "" => "<computed>"
  ebs_block_device.#:           "" => "<computed>"
  ephemeral_block_device.#:     "" => "<computed>"
  instance_state:               "" => "<computed>"
  instance_type:                "" => "t2.micro"
  key_name:                     "" => "<computed>"
  placement_group:              "" => "<computed>"
  private_dns:                  "" => "<computed>"
  private_ip:                   "" => "<computed>"
  public_dns:                   "" => "<computed>"
  public_ip:                    "" => "<computed>"
  security_groups.2200183879: "" => "allow_http"
  ...
aws_instance.web_server: Still creating... (10s elapsed)
aws_instance.web_server: Still creating... (20s elapsed)
aws_instance.web_server: Creation complete
aws_eip.web_server_ip: Creating...
  allocation_id:     "" => "<computed>"
  association_id:    "" => "<computed>"
  domain:            "" => "<computed>"
  instance:          "" => "i-0cd491b0"
  network_interface: "" => "<computed>"
  private_ip:        "" => "<computed>"
  public_ip:         "" => "<computed>"
aws_eip.web_server_ip: Creation complete

Apply complete! Resources: 3 added, 0 changed, 0 destroyed.

The state of your infrastructure has been saved to the path
below. This state is required to modify and destroy your
infrastructure, so keep it safe. To inspect the complete state
use the `terraform show` command.
State path: terraform.tfstate
```

Behind the stage terraform has made few API calls kindly asking AWS to create appropriate resources.

5. Final State

The requirements are met, we have the server, the group & the IP was allocated.

To check:

1. *terraform show*

```
$ terraform show

aws_eip.web_server_ip:
  id = eipalloc-7a02b713
  association_id = eipassoc-e770d58f
  domain = vpc
  instance = i-cb5c1977
  network_interface = eni-8ac4d3f7
  private_ip = 172.31.24.135
  public_ip = 52.58.172.226
aws_instance.web_server:
  id = i-cb5c1977
  ami = ami-d3c022bc
  availability_zone = eu-central-1b
  disable_api_termination = false
  ...
aws_security_group.allow_http:
  id = sg-bdfe55d5
  description = Managed by Terraform
  egress.# = 0
  ingress.# = 1
  ingress.2214680975.cidr_blocks.# = 1
  ingress.2214680975.cidr_blocks.0 = 0.0.0.0/0
  ingress.2214680975.from_port = 80
  ingress.2214680975.protocol = TCP
  ...
```

2. AWS control panel

Instance ID	Instance Type	Availability Zone	Instance State
i-cb5c1977	t2.micro	eu-central-1b	running

3. and even website!

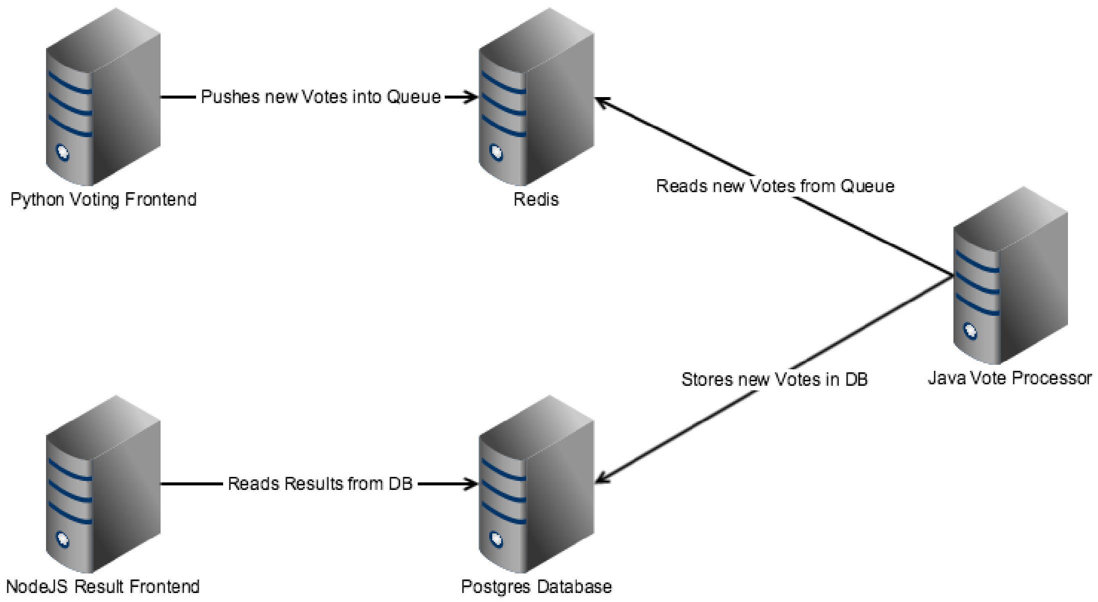
Wasn't it impressive? But Terraform can do much more! Would you like to try? We prepared a real hands-on laboratory so move on, install the terraform agent and create your own Infrastructure (as a Code, as we promised!)

Hands-on Lab

Plan

What we are going to do? That's a good question. We are going to create an infrastructure for a complex distributed application and deploy it in two different clouds at the same time, to provide a disaster-proof solution.

What is the application? It's a demo voting application, where you basically can vote for one of two options and see results. Though it doesn't look like a big deal, it has a complex solution under the hood. The application consists of 5 different services, uses queues and ready for scaling. To run the application we will use docker images.



Limitations

Due to introduction purpose of the article, we have two limitations:

- We will not set up a database replication, but a single database instead. The question of Postrgres replication is too complex and completely out of the scope for the article.
- We will deliver an application to cloud instances via Terraform's provisioning tool. In general it's not recommended way because it is designed to perform simplest operations and not suitable for continuous delivery. In real life we should use Configuration Management Tools and Deployment Tools.

Preparations

To set everything up, we will need four things:

1. An active account on AWS
2. An active account on Digital Ocean
3. An installed Terraform agent.
4. A folder to keep laboratory's files.

Terraform works with dozens of different providers, but we will use AWS and Digital Ocean just for an example.

> Notice that AWS and Digital Ocean are commercial companies and this practice may produce some costs, but since this is just an experiment and cloud resources will be consumed only for a short time, the bill should be really small. During my preparations I've spent approximately one euro total.

Providers

AWS

First, let's configure our cloud providers. Create a file 'aws_config.tf' and add details. Terraform uses a special DSL to define *resources*. In this case we are defining a cloud provider, a keypair for ssh connection (we will use it later) and five security groups.

- *provider aws* allows to define connection details.
- resource aws_key_payr defines ssh key. We will not copy-paste your key but load it from filesystem instead.
- resource aws_security_group defines an access security group to control an access to the instances.

aws_config.tf

```
provider "aws" {
  access_key = "AWC_ACCESS_KEY"
  secret_key = "AWS_SECRET_KEY"
  region     = "eu-central-1"
}

resource "aws_key_pair" "Terraform" {
  key_name = "Terraform"
  public_key = "${file("/Users/developer/.ssh/id_rsa.pub")}"
}

resource "aws_security_group" "ssh" {
  ingress {
      from_port = 22
      to_port = 22
      protocol = "tcp"
      cidr_blocks = ["0.0.0.0/0"]
  }
}

resource "aws_security_group" "http" {
  ingress {
      from_port = 80
      to_port = 80
      protocol = "tcp"
      cidr_blocks = ["0.0.0.0/0"]
  }
}

resource "aws_security_group" "postgres" {
  ingress {
      from_port = 5432
      to_port = 5432
      protocol = "tcp"
      cidr_blocks = ["0.0.0.0/0"]
  }
}

resource "aws_security_group" "redis" {
  name = "allow_redis"
  ingress {
      from_port = 6379
      to_port = 6379
      protocol = "tcp"
      cidr_blocks = ["0.0.0.0/0"]
  }
}

resource "aws_security_group" "outgoing" {
  name = "allow_outgoing"
  egress {
      from_port = 0
      to_port = 0
      protocol = "-1"
      cidr_blocks = ["0.0.0.0/0"]
  }
}
```

- The filename may vary, it is completely up to you, that is not important. Terraform will analyse all *.tf files in the folder, so it's up to you to organise the structure somehow.
- Normally these highly sensitive details like access tokens should not be kept in config files, but we will use this approach to simplify the example. The best practices recommend to keep them in environment vars or provide as command-line options.
- To get AWS tokens, visit the https://console.aws.amazon.com/iam/home?region=eu-central-1#security_credential and open "Access Keys" tab.
- Don't forget to adjust a path to a ssh key in *aws_key_pair* resource definition.

Digital Ocean

Now let's define a configuration for Digital Ocean cloud provider.

do_config.tf

```
provider "digitalocean" {
  token = "DO_TOKEN"
}

resource "digitalocean_ssh_key" "default" {
  name = "Terraform"
  public_key = "${file("/Users/developer/.ssh/id_rsa.pub")}"
}
```

As you see, DO provides much less options to configure. You can generate a DO token here: https://cloud.digitalocean.com/settings/api/tokens. Adjust a path to ssh key!

Instances

Now let's define our infrastructure!

AWS

aws_instances.tf

```
# Elastic IPs

resource "aws_eip" "voting_ip" {
  instance = "${aws_instance.voting.id}"
}
resource "aws_eip" "db_ip" {
  instance = "${aws_instance.db.id}"
}
resource "aws_eip" "result_ip" {
  instance = "${aws_instance.result.id}"
}

# Instances

resource "aws_instance" "redis" {
  ami           = "ami-cfca25a0"
  instance_type = "t2.micro"
  security_groups = ["${aws_security_group.ssh.name}",
"${aws_security_group.outgoing.name}", "${aws_security_group.redis.name}"]
  key_name        = "${aws_key_pair.deployer.key_name}"
  connection {
    user = "core"
  }
  provisioner "remote-exec" {
    inline = [
```

```
      "docker run -dp 6379:6379 redis",
    ]
  }
}

resource "aws_instance" "db" {
  ami           = "ami-cfca25a0"
  instance_type = "t2.micro"
  security_groups = ["${aws_security_group.ssh.name}",
"${aws_security_group.postgres.name}", "${aws_security_group.outgoing.name}"]
  key_name        = "${aws_key_pair.deployer.key_name}"
  connection {
    user = "core"
  }
  provisioner "remote-exec" {
    inline = [
       "docker run -dp 5432:5432 postgres",
    ]
  }
}

resource "aws_instance" "voting" {
  ami           = "ami-cfca25a0"
  instance_type = "t2.micro"
  security_groups = ["${aws_security_group.ssh.name}",
"${aws_security_group.http.name}", "${aws_security_group.outgoing.name}"]
  key_name        = "${aws_key_pair.deployer.key_name}"
  connection {
    user = "core"
  }
  provisioner "remote-exec" {
    inline = [
      "docker run -dp 80:80 -e REDIS_HOST=${aws_instance.redis.private_ip}
ditmc/voting",
    ]
  }
}

resource "aws_instance" "worker" {
  ami           = "ami-cfca25a0"
  instance_type = "t2.micro"
  security_groups = ["${aws_security_group.ssh.name}",
"${aws_security_group.outgoing.name}"]
  key_name        = "${aws_key_pair.deployer.key_name}"
  connection {
    user = "core"
  }
  provisioner "remote-exec" {
    inline = [
        "docker run -d -e REDIS_HOST=${aws_instance.redis.private_ip} -e
DB_HOST=${aws_instance.db.private_ip} ditmc/worker"
    ]
  }
}

resource "aws_instance" "result" {
  ami           = "ami-cfca25a0"
  instance_type = "t2.micro"
  security_groups = ["${aws_security_group.ssh.name}",
"${aws_security_group.http.name}", "${aws_security_group.outgoing.name}"]
  key_name        = "${aws_key_pair.deployer.key_name}"
  connection {
    user = "core"
```

```
  }
  provisioner "remote-exec" {
    inline = [
        "docker run -dp 80:80 -e DB_HOST=${aws_instance.db.private_ip}
ditmc/result"
```

```
    ]
  }
}
```

Digital Ocean

do_instances.tf

```
# instances

resource "digitalocean_droplet" "redis" {
  image    = "coreos-stable"
  region   = "ams3"
  size     = "512mb"
  name     = "redis"
  ssh_keys = ["${digitalocean_ssh_key.default.fingerprint}"]
  connection {
    user = "core"
  }
  provisioner "remote-exec" {
    inline = [
      "docker run -dp 6379:6379 redis",
    ]
  }
}

resource "digitalocean_droplet" "voting" {
  image    = "coreos-stable"
  region   = "ams3"
  size     = "512mb"
  ssh_keys = ["${digitalocean_ssh_key.default.fingerprint}"]
  name     = "voting"
  connection {
    user = "core"
  }
  provisioner "remote-exec" {
    inline = [
      "docker run -dp 80:80 -e
REDIS_HOST=${digitalocean_droplet.redis.ipv4_address} ditmc/voting",
    ]
  }
}

resource "digitalocean_droplet" "worker" {
  image    = "coreos-stable"
  region   = "ams3"
  size     = "512mb"
  ssh_keys = ["${digitalocean_ssh_key.default.fingerprint}"]
  name     = "redis"
  connection {
    user = "core"
  }
  provisioner "remote-exec" {
    inline = [
      "docker run -dp 80:80 -e
REDIS_HOST=${digitalocean_droplet.redis.ipv4_address} -e
DB_HOST=${aws_eip.db_ip.public_ip} ditmc/worker"
    ]
  }
```

```
}

resource "digitalocean_droplet" "result" {
  image    = "coreos-stable"
  region   = "ams3"
  size     = "512mb"
  ssh_keys = ["${digitalocean_ssh_key.default.fingerprint}"]
  name     = "result"
  connection {
    user = "core"
  }
  provisioner "remote-exec" {
    inline = [
      "docker run -dp 80:80 -e DB_HOST=${aws_eip.db_ip.public_ip} ditmc/result"
```

```
    ]
  }
}
```

Output

Though this part is completely optional, it makes sense to configure "output", or define which information Terraform should print when the job will be finished. Normally we are interested in the IP addresses of newly created systems. The application has two frontends, one for voting and one for viewing results, and we deployed it into two clouds, so we have four entry points to use.

output.tf

```
output "AWS Voting IP" {
  value = "${aws_eip.voting_ip.public_ip}"
}
output "DO Voting IP" {
  value = "${digitalocean_droplet.voting.ipv4_address}"
}
output "AWS Result IP" {
  value = "${aws_eip.result_ip.public_ip}"
}
output "DO Result IP" {
  value = "${digitalocean_droplet.result.ipv4_address}"
}
```

Fire it up!

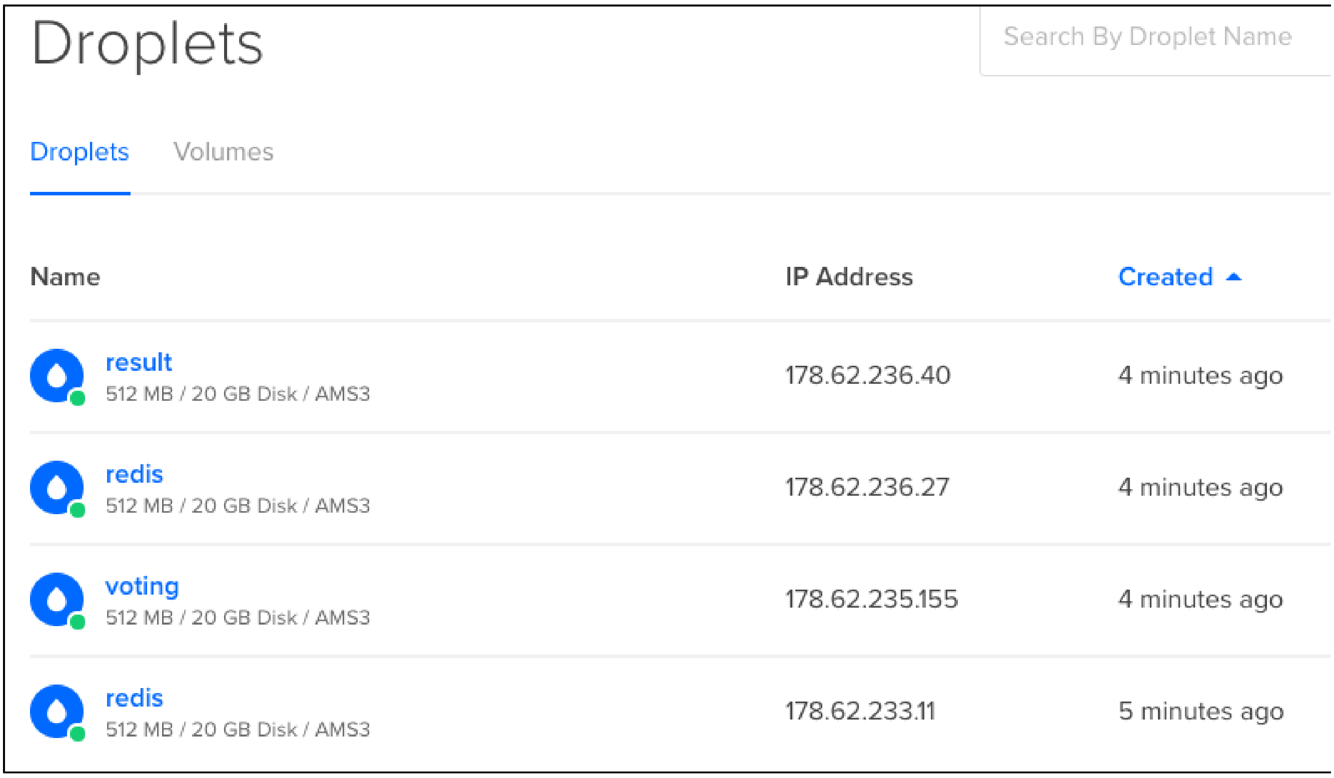
Terraform apply!

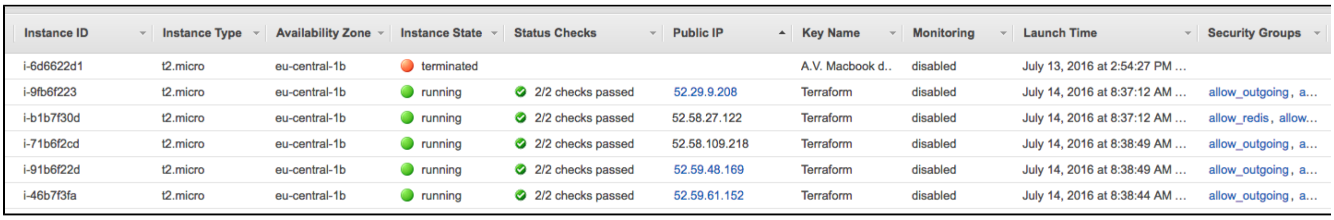
```
$ terraform apply

digitalocean_ssh_key.default: Creating...
...
Apply complete! Resources: 19 added, 0 changed, 0 destroyed.
The state of your infrastructure has been saved to the path
below. This state is required to modify and destroy your
infrastructure, so keep it safe. To inspect the complete state
use the `terraform show` command.
State path: terraform.tfstate
Outputs:
  AWS Result IP = 52.59.48.169
  AWS Voting IP = 52.59.61.152
  DO Result IP  = 178.62.236.40
  DO Voting IP  = 178.62.235.155
```

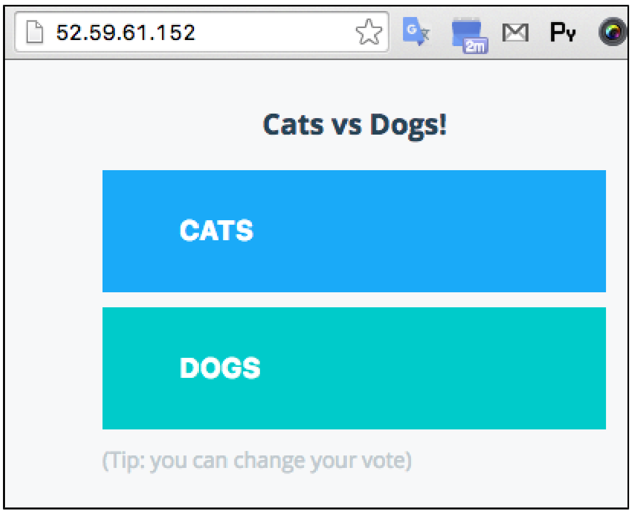
It will take few minutes, of course, so keep calm.

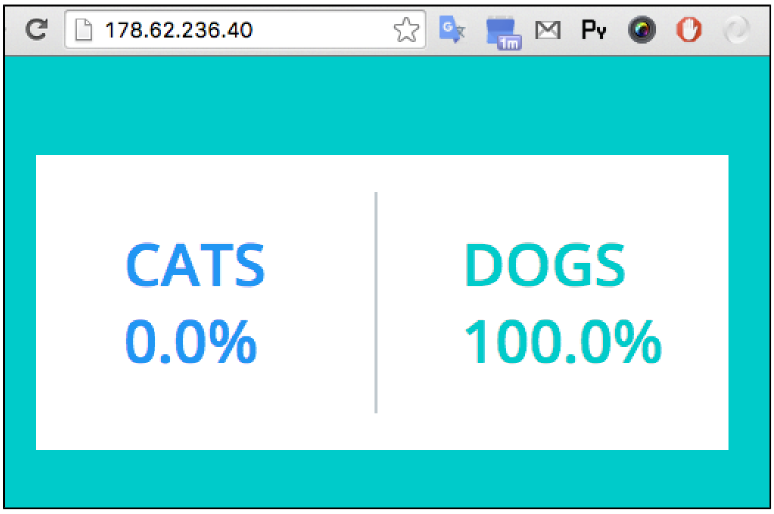
Now take a look to the results. Terraform printed the IP addresses of our entrypoints in both AWS and DO datacenters. Please visit them and assure that application works from both side. Then check your resources list in DO and AWS control panel:





Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Public IP	Key Name	Monitoring	Launch Time	Security Groups
i-6d6622d1	t2.micro	eu-central-1b	terminated			A.V. Macbook d..	disabled	July 13, 2016 at 2:54:27 PM ...	
i-9fb6f223	t2.micro	eu-central-1b	running	2/2 checks passed	52.29.9.208	Terraform	disabled	July 14, 2016 at 8:37:12 AM ...	allow_outgoing, a...
i-b1b7f30d	t2.micro	eu-central-1b	running	2/2 checks passed	52.58.27.122	Terraform	disabled	July 14, 2016 at 8:37:12 AM ...	allow_redis, allow...
i-71b6f2cd	t2.micro	eu-central-1b	running	2/2 checks passed	52.58.109.218	Terraform	disabled	July 14, 2016 at 8:38:49 AM ...	allow_outgoing, a...
i-91b6f22d	t2.micro	eu-central-1b	running	2/2 checks passed	52.59.48.169	Terraform	disabled	July 14, 2016 at 8:38:49 AM ...	allow_outgoing, a...
i-46b7f3fa	t2.micro	eu-central-1b	running	2/2 checks passed	52.59.61.152	Terraform	disabled	July 14, 2016 at 8:38:44 AM ...	allow_outgoing, a...





Connect to the instances with SSH. You may get IP adresses in control panels or with *terraform show*. Don't forget to login with *core* userna
me.

```
$ ssh core@52.59.48.169
The authenticity of host '52.59.48.169 (52.59.48.169)' can't be established.
ECDSA key fingerprint is SHA256:Q40RYkuLm+D8RCGpao7qozEcCpnRZoDEyS48Eh6YSdg.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '52.59.48.169' (ECDSA) to the list of known hosts.
Last login: Thu Jul 14 06:39:39 2016 from 31.13.171.253
CoreOS stable (1010.5.0)
core@ip-172-31-22-226 ~ $
```

Playgrounds

Please, try to modify the infrastructure. Add some nodes, remove previous ones, play with security settings, scale up worker nodes and whatever. After any change call for the *terraform apply*. Every time terraform will calculate the changes and modify infrastructure accordingly.

Clean up

Now, when it's done, don't forget to delete our instances with *terraform destroy*. Notice that disabled instance are still count for bills from cloud providers, so we have to wipe it out completely.

```
$ terraform destroy

Do you really want to destroy?
  Terraform will delete all your managed infrastructure.
  There is no undo. Only 'yes' will be accepted to confirm.
  Enter a value: yes
digitalocean_ssh_key.default: Refreshing state... (ID: 2365956)
digitalocean_droplet.redis: Refreshing state... (ID: 19537746)
digitalocean_droplet.voting: Refreshing state... (ID: 19537783)
aws_key_pair.deployer: Refreshing state... (ID: Terraform)
aws_security_group.redis: Refreshing state... (ID: sg-4b4ee423)
...
aws_key_pair.deployer: Destruction complete
aws_security_group.postgres: Destruction complete
aws_security_group.outgoing: Destruction complete
aws_security_group.ssh: Destruction complete
Apply complete! Resources: 0 added, 0 changed, 19 destroyed.
```

Assure in control panels that all your instances was terminated and floating IPs dissassociated.

Conclusion

Long time ago the Source Code first fell down into the Version Control World and never got back. The next one was the Configuration. It looks like the time has come for the Infrastructure to join them.

Let's recap the most important points of this impressive demonstration:

- Track a history of changes
- Keep names of authors
- Be able to research/rollback any previous state, yesterday or one year ago
- Avoid a lot of possible mistakes by manual verification of planned changes
- Automate it in more simple way instead of calling API from custom scripts
- Keep all things together readable and clean
- Work with multiply infrastructure providers easily

It was a long way, but we did it. Thanks for joining!

Links

- laboratory source code
- terraform.io
- aws.amazon.com
- digitalocean.com
- Infrastructure as Code (wiki)
- AWS CloudFormation - AWS-only tool to solve the same issue.