Look mom without containers!

An implementation of an immutable infrastructure using packer, terraform and ansible.

nethesis

Matteo Valentini

Who I am

Matteo Valentini

Developer @ Nethesis (mostly Infrastrutture Developer)

Intro

- What is a immutable infrastructure?
- Why the title?
- Index
  - What?
  - Why?
  - How?

What?

Icaro Project

Easy and simple HotSpot for small and medium hotels

- **Fully Open Souce** (AGPL License)
  - Github repo: https://github.com/nethesis/icaro
- **Main site**: https://go.nethesis.it/icaro/
- **Technology used**:
  - Golang for backends
  - VueJs for frontends
  - MariaDB as database
  - coova-chilli for captive portal
  - ansible for provisioning

Icaro Project - Simple deploy

Use Vagrant with DigitalOcean as provider and ansible as provisioner.

1. git clone https://github.com/nethesis/icaro.git
2. cd deploy
3. Add your DigitalOcean's token in to the provided Vagranfile
4. Customize the ansible variables in deploy/ansible/group_vars/all.yml
5. vagrant up --provider digitalocean

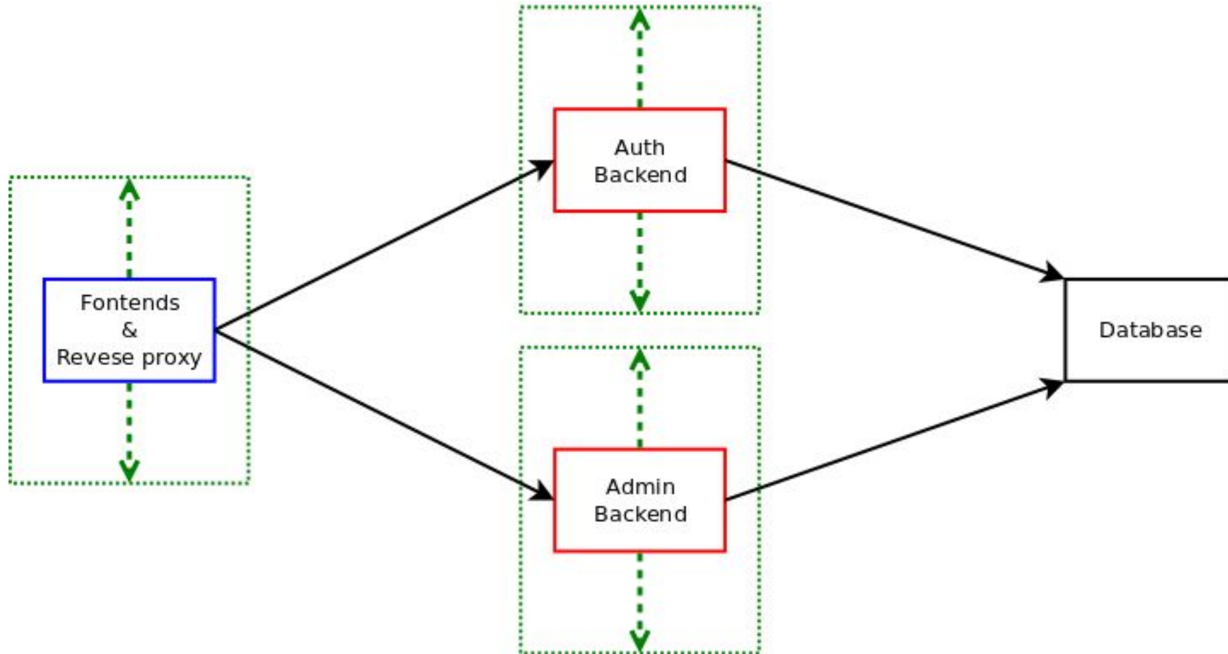
At the end you obtain a system ready to use and all in one instance.

https://go.nethesis.it/icaro/docs/provisioning/

Icaro Project - Nethesis deploy

- On DigitalOcean
- One instance for serving frontends and act as reverse proxy
- Two instance for backends
- One instance for database
- Provisioned with Ansible and Packer
- Orchestrate via Terraform

Icaro Project - Nethesis deploy



Why?

Why no containers?

- **Bad history**: previous attempt to use docker on a service was a total failure
- **Complexity:** no strong knowledge in the team how to manage containers on production
- **Cost**: no managed kubernetes on DigitalOcean at the time

Why no PaaS?

- **Too expensive**
- **No values in the PaaS solution**: we already have the technical skills for manage and maintain a service

Terraform

- Enable Infrastructure as code paradigm
- Can be versioned in a csv
- Simple DSL language
- It’s become a standard de facto for modeling cloud based system

Why DigitalOcean?

- **Simple**
- **Clear on cost**
- **Complete**: have all necessary tool for implement a scalable system
    - Snapshot
    - Floating ip
    - LoadBalancer
    - VM
    - User data
- **Terraform support**

Why Ansible

- **Easy**
- **Simple to manage**: agent less and no need of a *“master”* node
- **Well know:** the technology was already in use and trusted by the team

How?

Intro - What we are going to implement?

- Simple demo app
- One scalable backend
- On connected database

Application - Twelve-factor App

- **Codebase** One codebase tracked in revision control, many deploys
- **Dependencies** Explicitly declare and isolate dependencies
- **Config** Store config in the environment
- **Backing services** Treat backing services as attached resources
- **Build, release, run** Strictly separate build and run stages
- **Processes** Execute the app as one or more stateless processes

Application - Twelve-factor App

- **Port binding** Export services via port binding
- **Concurrency** Scale out via the process model
- **Disposability** Maximize robustness with fast startup and graceful shutdown
- **Dev/prod parity** Keep development, staging, and production as similar as possible
- **Logs** Treat logs as event streams
- **Admin processes** Run admin/management tasks as one-off processes

Application - Example (main.go)

```
func main() {

	// init configuration
	configuration.Init()

	db := database.Init()
	defer db.Close()

	// init routers
	router := gin.Default()

	router.GET("/health/check", methods.HealthCheck)

	// handle missing endpoint
	router.NoRoute(func(c *gin.Context) {
			c.Status(http.StatusNotFound)
	})

	router.Run()
}
```

Application - Example (methods/healtcheck.go)

```go
package methods
import (
        "github.com/Amygos/immutable_deploys/app/database"
        "github.com/gin-gonic/gin"
        "net/http"
)

func HealthCheck(c *gin.Context) {

        db := database.Instance()

        if db := db.Exec("SELECT NOW()"); db.Error != nil {
                c.Status(http.StatusInternalServerError)
        } else {
                c.Status(http.StatusOK)
        }
}
```

Application - Example (database/database.go)

```
var db *gorm.DB
var err error

func Instance() *gorm.DB {
	if db == nil {
		Init()
	}
	return db
}
func Init() *gorm.DB {
	conn_params := "user=" + configuration.Config.Database.User + " " +
		"password=" + configuration.Config.Database.Password + " " +
		"host=" + configuration.Config.Database.Host + " " +
		"port=" + configuration.Config.Database.Port + " " +
		"dbname=" + configuration.Config.Database.Name

	db, err = gorm.Open("postgres", "sslmode=require "+conn_params)
	if err != nil {
		panic(err.Error())
	}
	return db
}
```

Application - Example (configuration/config.go)

```
type Configuration struct {
	Database struct {
			Host   string
			Port   string
			User   string
			Name   string
			Password string
	}
}

var Config = Configuration{}

func Init() {
	Config.Database.User = os.Getenv("DB_USER")
	Config.Database.Password = os.Getenv("DB_PASSWORD")
	Config.Database.Host = os.Getenv("DB_HOST")
	Config.Database.Port = os.Getenv("DB_PORT")
	Config.Database.Name = os.Getenv("DB_NAME")
}
```

Versioning - Git

- Use git tag for make a new release

Provisioning - Ansible (playbook tasks)

```
tasks:
    - name: Create app directory
    file:
    path: /opt/app
    state: directory
    - name: Download and install app executable
    get_url:
    url: "{{ url }}/releases/download/{{ version }}/app"
    force: true
    dest: /opt/app/app
    mode: 0755
    - name: Copy Ade-tasks sytemd unit
    copy:
    src: app.service
    dest: /usr/lib/systemd/system/app.service
    - name: Enable and restart app service
    systemd:
    name: app
    state: restarted
    daemon_reload: yes
    enabled: yes
```

Provisioning - Ansible (systemd unit)

```
[Unit]
Description=App server
After=network.target cloud-init.service

[Service]
Type=simple
User=root
EnvironmentFile=-/opt/app/conf.env
WorkingDirectory=/opt/app
Environment=GIN_MODE=release
ExecStart=/opt/app/app

[Install]
WantedBy=multi-user.target
```

Orchestration - Terraform (Droplet)

```
resource "digitalocean_droplet" "app" {
  image  = "centos-7-0-x64"
  name   = "app"
  region = "ams3"
  size   = "s-1vcpu-1gb"
  user_data = "${data.template_cloudinit_config.app.rendered}"
}
```

Orchestration -Terraform (Domains)

```
resource "digitalocean_domain" "app" {
  name = "deploy.example.com"
}

resource "digitalocean_record" "app" {
  domain = "${digitalocean_domain.app.name}"
  type   = "A"
  name   = "app"
  ttl    = "60"

  value = "${digitalocean_droplet.app.ipv4_address}"
}
```

Orchestration - Terraform (Database)

```
resource "digitalocean_database_cluster" "app" {
  name       = "app"
  engine     = "pg"
  version    = "11"
  size       = "db-s-1vcpu-1gb"
  region     = "ams3"
  node_count = 1
}
```

Orchestration - Terraform (user-data)

```
data "template_cloudinit_config" "app" {
  gzip          = false
  base64_encode = false
  part {
    content_type = "text/cloud-config"
    content      = "${data.template_file.config_app.rendered}"
  }
}

data "template_file" "config_app" {
  template = "${file("${path.module}/config_app.tpl")}"
  vars {
    db_host     = "${digitalocean_database_cluster.app.host}"
    db_port     = "${digitalocean_database_cluster.app.port}"
    db_user     = "${digitalocean_database_cluster.app.user}"
    db_password = "${digitalocean_database_cluster.app.password}"
    db_name     = "${digitalocean_database_cluster.app.database}"
  }
}
```

Orchestration - Terraform (Cloud-init template)

```
#cloud-config
write_files:
  - path: /opt/app/conf.env
      content: |
      DB_HOST="${db_host}"
      DB_PORT="${db_port}"
      DB_USER="${db_user}"
      DB_PASSWORD="${db_password}"
      DB_NAME="${db_name}"
```

What we are obtained now?

What we are obtained now?

- A twelve factor app
- Automated application provisioning
- Infrastructure as code
- A vertically scalable system

What we are not obtained?

- A immutable infrastructure: we still need to live provisioning the application instance
- A horizontally scalable system
- A versionable history of deploys

Provision - Packer (Intro)

What packer do?

1. Create a instance
2. Run a provisioner
3. Shutdown the instance
4. Create and save a snapshot of the istance

Provision - Packer (Config)

```
{
  "builders": [{
    "type": "digitalocean",
    "image": "centos-7-x64",
    "region": "ams3",
    "size": "s-1vcpu-1gb",
    "ssh_username": "root",
    "snapshot_name": "app-{{timestamp}}"
  }],
  "provisioners": [{
    "type": "ansible",
    "playbook_file": "ansible/playbook.yml"
  }]
```

Provision - Packer (output)

```
    digitalocean: PLAY RECAP *********************************************************************
    digitalocean: default                    : ok=5    changed=4    unreachable=0    failed=0
    digitalocean:
==> digitalocean: Gracefully shutting down droplet...
==> digitalocean: Creating snapshot: app-1556824617
==> digitalocean: Waiting for snapshot to complete...
==> digitalocean: Destroying droplet...
==> digitalocean: Deleting temporary ssh key...
Build 'digitalocean' finished.

==> Builds finished. The artifacts of successful builds are:
--> digitalocean: A snapshot was created: 'app-1556824617' (ID: 46661831) in regions 'ams3
```

Orchestration - Terraform (Image)

```
data "digitalocean_image" "app" {
  name = "app-1556824617"
}
```

Orchestration - Terraform (Droplet)

```
resource "digitalocean_droplet" "app" {
  image   = "${data.digitalocean_image.app.image}"
  name    = "app"
  region = "ams3"
  size    = "s-1vcpu-1gb"
  user_data = "${data.template_cloudinit_config.app.rendered}"

  lifecycle {
    create_before_destroy = true
  }
}
```

Orchestration - Terraform (Floating IP)

```
resource "digitalocean_floating_ip" "app" {
  droplet_id = "${digitalocean_droplet.app.id}"
  region     = "ams3"
}
```

Orchestration -Terraform (Domains)

```
resource "digitalocean_domain" "app" {
  name = "deploy.example.com"
}

resource "digitalocean_record" "app" {
  domain = "${digitalocean_domain.app.name}"
  type   = "A"
  name   = "app"
  ttl    = "60"

  value = "${digitalocean_floating_ip.app.ip}"
}
```

Resulting workflow

1. Do some modifications
2. Create a new version of the application (eg. via git tag and CI/CD pipeline)
3. Run packer for create a new snapshot
4. Put the new image name in terraform config
5. Run terraform for apply you modification

What about horizontal scalability?

Orchestration - Terraform (Variables)

```
variable "instance_count" {
  default = "1"
}
```

Orchestration - Terraform (Droplet)

```
resource "digitalocean_droplet" "app" {
  image  = "${data.digitalocean_image.app.image}"
  count  = "${var.instance_count}"
  name   = "app-${count.index}"
  region = "ams3"
  size   = "s-1vcpu-1gb"

  user_data = "${data.template_cloudinit_config.app.rendered}"
  tags      = ["${digitalocean_tag.app.id}"]

  lifecycle {
    create_before_destroy = true
  }
}

resource "digitalocean_tag" "app" {
  name = "app"
}
```

Orchestration - Terraform (Loadbalancer)

```
resource "digitalocean_loadbalancer" "app" {
  name   = "app"
  region = "ams3"
  count  = "${var.instance_count > 1 ? 1 : 0}"

  forwarding_rule {
    entry_port     = 8080
    entry_protocol = "tcp"

    target_port     = 8080
    target_protocol = "tcp"
  }

  healthcheck {
    port     = 8080
    protocol = "tcp"
  }

  droplet_tag = "${digitalocean_tag.app.id}"
}
```

Orchestration - Terraform (Floating IP)

```
resource "digitalocean_floating_ip" "app" {
  count      = "${var.instance_count > 1 ? 0 : 1}"
  droplet_id = "${digitalocean_droplet.app.id}"
  region     = "ams3"
}
```

Orchestration - Terraform (Domain)

```
resource "digitalocean_domain" "app" {
  name = "deploy.example.com"
}

resource "digitalocean_record" "app" {
  domain = "${digitalocean_domain.app.name}"
  type   = "A"
  name   = "app"
  ttl    = "60"

  value = "${var.instance_count > 1 ?
    join(" ", digitalocean_loadbalancer.app.*.ip) :
    join(" ", digitalocean_floating_ip.app.*.ip_address)}"
}
```

Conclusions

What we are obtain?

- A immutable infrastructure
- A versioned history of deployments
- Automated provisioning
- Vertically and horizontally scalable architecture
- Cost effective architecture: We don’t pay for a load balancer if we don’t use it.

Benefits

- Security: in case of security flaw, we can simply fix the flaw and rebuild all from scratch.
- Provision symplicity: you don't have to care about to previous state or to be idempotency, every time is from scratch.
- Reproducibility: you can redeploy the same snapshot

Limitations/Drawbacks

For this simply example:

- Downtime of ~10 sec for deploy new version (Droplet start up)
- Downtime of 1-2 mins for scale out to form 1 to N > 1 nodes (Loadbalancer creation)

Extra materials

All the code show in this talk is published in this public repository:

https://github.com/Amygos/immutable_deploys

with some bonus points :)

- terraform workspaces
- Travis-CI configuration of automatic application build and release

Thanks for listening!

Questions?