Terraform:
Configuration Management for Cloud Services

Martin Schütte

26 February 2016

GUUG-Frühjahrsfachgespräch

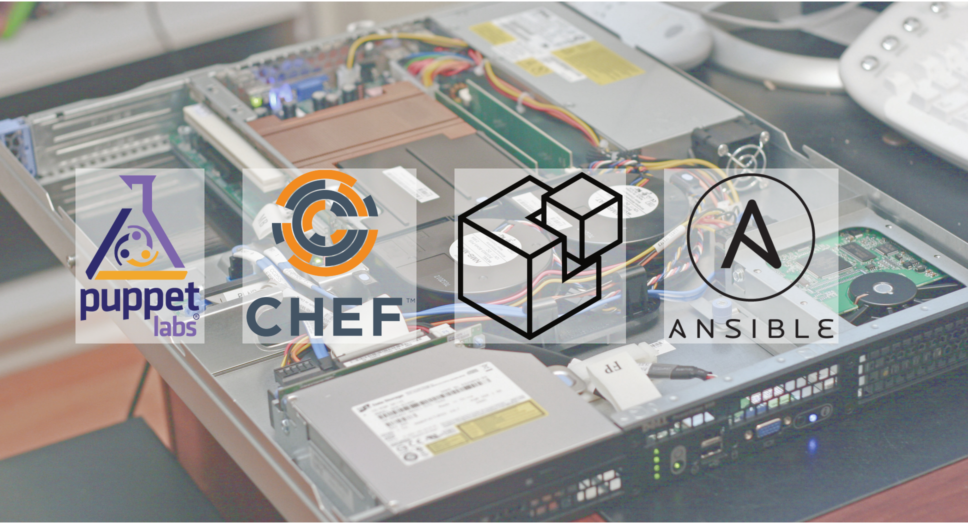
guug
FFG 2016
Köln
23. - 26. Februar

TERRAFORM

Build, Combine, and Launch Infrastructure

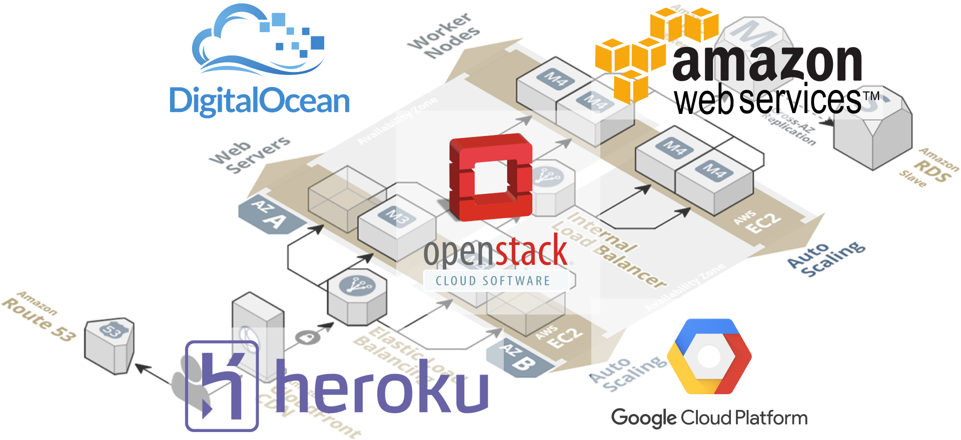
Concepts

From Servers …



by Rodzilla at Wikimedia Commons (CC-BY-SA-3.0)

...to Services


DigitalOcean
amazon web services™
openstack CLOUD SOFTWARE
heroku
Google Cloud Platform
Worker Nodes
Web Servers
AZ A
AZ B
M4
M3
AWS EC2
Internal Load Balancer
Elastic Load Balancing
Auto Scaling
Amazon RDS Slave
Cross-AZ Replication
Amazon Route 53
Amazon CloudFront
Availability Zone

Services also need Configuration Management

- Replace “click paths” with source code in VCS
- Lifecycle awareness, not just a `setup.sh`
- Reproducible environments
- Specification, documentation, policy enforcement

Core Ideas in Terraform

- Simple model of resource entities with attributes
- Stateful lifecycle with CRUD operations
- Declarative configuration
- Dependencies by inference
- Parallel execution

Core Concepts in Terraform

- Provider: a source of resources
  (usually with an API endpoint & authentication)
- Resource: every thing “that has a set of configurable attributes and a lifecycle (create, read, update, delete)” – implies ID and state
- Provisioner: initialize a resource with local or remote scripts

Core Concepts in Terraform

- Order: directed acyclic graph of all resources
- Plan: generate an execution plan for review before applying a configuration
- State: execution result is kept in state file (local or remote)
- Lightweight: little provider knowledge, no error handling

Available services

Providers:

- **AWS**
- **Azure**
- **Google Cloud**
- Heroku
- DNSMadeEasy
- OpenStack
- …

Resources:

- aws_instance
- aws_vpc
- aws_elb
- aws_iam_user
- azure_instance
- heroku_app
- …

Provisioners:

- chef
- file
- local-exec
- remote-exec

DSL Syntax

- Hashicorp Configuration Language (HCL), think “JSON-like but human-friendly”
- Variables
- Interpolation, e. g. `”number ${count.index + 1}”`
- Attribute access with `resource_type.resource_name`
- Few build-in functions, e. g. `base64encode(string)`, `format(format, args…)`

HCL vs. JSON

```
# An AMI
variable ”ami” {
  description = ”custom AMI”
}

/* A multi
   line comment. */
resource ”aws_instance” ”web” {
  ami = ”${var.ami}”
  count = 2
  source_dest_check = false

  connection {
    user = ”root”
  }
}
```

```
{
  ”variable”: {
    ”ami”: {
      ”description”: ”custom AMI”
    }
  },
  ”resource”: {
    ”aws_instance”: {
      ”web”: {
        ”ami”: ”${var.ami}”,
        ”count”: 2,
        ”source_dest_check”: false,

        ”connection”: {
          ”user”: ”root”
        }
      }
    }
  }
}
```

Example: Simple Webservice

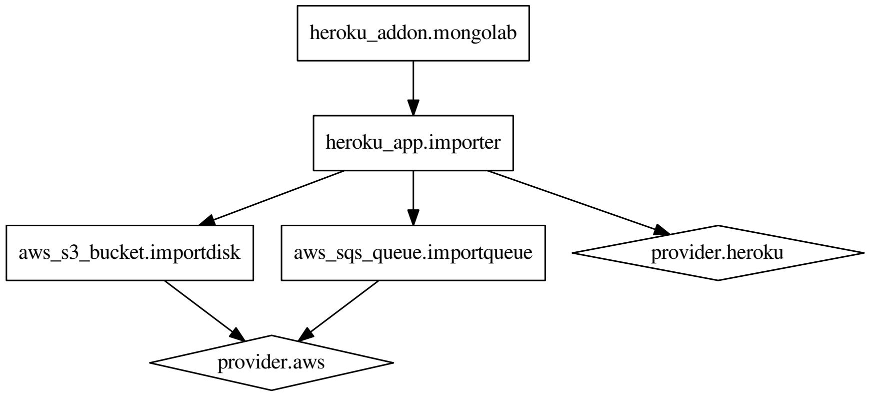
Example: Simple Webservice (part 1)

```
### AWS Setup
provider ”aws” {
  access_key = ”${var.aws_access_key}”
  secret_key = ”${var.aws_secret_key}”
  region     = ”${var.aws_region}”
}

# Queue
resource ”aws_sqs_queue” ”importqueue” {
  name = ”${var.app_name}-${var.aws_region}-importqueue”
}

# Storage
resource ”aws_s3_bucket” ”importdisk” {
  bucket = ”${var.app_name}-${var.aws_region}-importdisk”
  acl    = ”private”
}
```

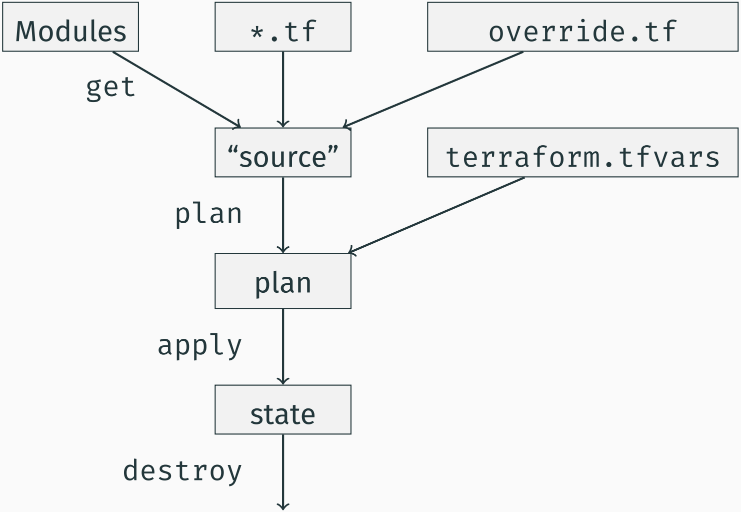
Example: Simple Webservice (part 2)

```
### Heroku Setup
provider ”heroku” { ... }

# Importer
resource ”heroku_app” ”importer” {
  name   = ”${var.app_name}-${var.aws_region}-import”
  region = ”eu”
  config_vars {
    SQS_QUEUE_URL = ”${aws_sqs_queue.importqueue.id}”
    S3_BUCKET     = ”${aws_s3_bucket.importdisk.id}”
  }
}

resource ”heroku_addon” ”mongolab” {
  app  = ”${heroku_app.importer.name}”
  plan = ”mongolab:sandbox”
}
```

terraform graph | dot -Tpdf



Terraform Process



Example: Add Provisioning

```
# Importer
resource ”heroku_app” ”importer” {
  name   = ”${var.app_name}-${var.aws_region}-import”
  region = ”eu”

  config_vars { ... }

  provisioner ”local-exec” {
    command = <<EOT
cd ~/projects/go-testserver &&
git remote add heroku ${heroku_app.importer.git_url} &&
git push heroku master
EOT
  }
}
```

Example: Add Outputs

```
# Storage
resource ”aws_s3_bucket” ”importdisk” { ... }

# Importer
resource ”heroku_app” ”importer” { ... }

# Outputs
output ”importer_bucket_arn” {
  value = ”${aws_s3_bucket.importdisk.arn}”
}

output ”importer_url” {
  value = ”${heroku_app.importer.web_url}”
}

output ”importer_gitrepo” {
  value = ”${heroku_app.importer.git_url}”
}
```

Modules

Modules

“Plain terraform code” lacks structure and reusability

Modules

- are subdirectories with self-contained terraform code
- may be sourced from Git, Mercurial, HTTPS locations
- use variables and outputs to pass data

Module Example

Every Terraform directory may be used as a module.

Here I use the previous webservice example.

Using a Module Example (part 1)

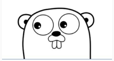
```
module ”importer_west” {
    source     = ”../simple”
    aws_region = ”eu-west-1”

    app_name              = ”${var.app_name}”
    aws_access_key        = ”${var.aws_access_key}”
    aws_secret_key        = ”${var.aws_secret_key}”
    heroku_login_email    = ”${var.heroku_login_email}”
    heroku_login_api_key = ”${var.heroku_login_api_key}”
}

module ”importer_central” {
    source     = ”../simple”
    aws_region = ”eu-central-1”

    # ...
}
```

Using a Module Example (part 2)

```
# Main App, using modules
resource ”heroku_app” ”main” {
  name   = ”${var.app_name}-main”
  region = ”eu”

  config_vars {
    IMPORTER_URL_LIST = <<EOT
[ ”${module.importer_west.importer_url}”,
  ”${module.importer_central.importer_url}” ]
EOT
  }
}

output ”main_url” {
  value = ”${heroku_app.main.web_url}”
}
```

Plugins

How to Write Own Plugins

- Learn you some Golang 
- Use the `schema` helper lib
- Adapt to model of
  Provider (setup steps, authentication) and
  Resources (arguments/attributes and CRUD methods)

Plugin Example

Simple Plugin: MySQL

Implements provider `mysql` with resource `mysql_database`.

Code at `builtin/providers/mysql`

Usage

Issues

Under active development, current version 0.6.12

- Still a few bugs, e. g. losing state info
- Modules are *very* simple
- Lacking syntactic sugar
  (e. g. aggregations, common repetitions)

General problems for this kind of tool

- Testing is inherently difficult
- Provider coverage
- Resource model mismatch, e. g. with Heroku apps
- Ignorant of API rate limits, account ressource limits, etc.

Comparable Tools

Tools:

- AWS CloudFormation (with Generator-Tools)
- OpenStack Heat

Configuration Management:

- SaltStack Salt Cloud
- Ansible v2.0 includes cloud modules

Libraries:

- fog, Ruby cloud abstraction library
- boto, Python AWS library

Workflow

- Use a VCS, i. e. git
- Use PGP to encrypt sensitive data, e. g. with Blackbox
- Use separate user credentials, know how to revoke them
- Take a look at Hashicorp Atlas and its workflow

Hashicorp Workflow

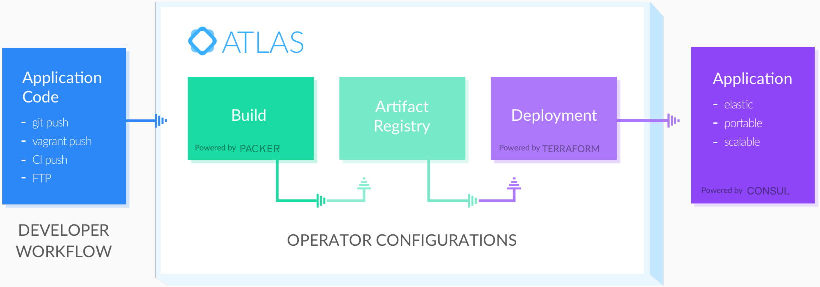
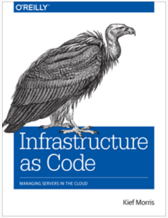


image by Hashicorp Atlas: Artifact Pipeline and Image Deploys with Packer and Terraform

Links and Resources



> *Defining system infrastructure as code and building it with tools doesn't make the quality any better. At worst, it can complicate things.*
> — *Infrastructure as Code* by Kief Morris

- Terraform
- hashicorp/terraform
- StackExchange/blackbox
- Terraform: Beyond the Basics with AWS

The End

Thank You!

Questions?

Martin Schütte
info@mschuette.name

`http://slideshare.net/mschuett/`