

Terraform

Who am I

- My name is Edward Viaene
- I am a **consultant** and **trainer** in Cloud and Big Data technologies
- I apply **DevOps techniques** in all my IT projects
- I held various roles from **banking** to **startups**
- I have a **background** in Unix/Linux, Networking, Security, Risk, and distributed computing
- Nowadays I specialize in everything that has to do with **Cloud** and **DevOps**

Online Training

- **Online training** on Udemy
 - **DevOps, Distributed Computing, Cloud, Big Data (Hadoop)**
 - Using online video lectures
 - 5,000+ enrolled students in 95+ countries

Course Layout

Introduction	Terraform basics	Terraform on AWS	Terraform with Packer	Docker
What is Terraform	Variables	VPC	Building AMIs	Introduction
Installing	Provisioning on Linux / Windows	EC2 instances and EBS	A Jenkins workflow	Docker Repository
First steps	Output	Security Groups		Docker Orchestration
	State	Userdata		Terraform with ECR
	DataSources	Static IPs and DNS		Terraform with ECS
	Template Provider	RDS		A Jenkins workflow with terraform, ECR, ECS
	Modules	IAM Users and Groups		
	Other providers	IAM Roles		
		Autoscaling, Load Balancers		

Course objectives

- To be able to understand, use and apply terraform
- To know when to use different features
- To use terraform with AWS
- To use terraform with Packer to create custom images
- To be able to apply DevOps techniques using terraform

Feedback and support

- To provide feedback or get support, use the discussion groups
- We also have a Facebook group called Learn DevOps: Continuously Deliver Better Software
- You can scan the following barcode or use the link in the next document after this introduction movie

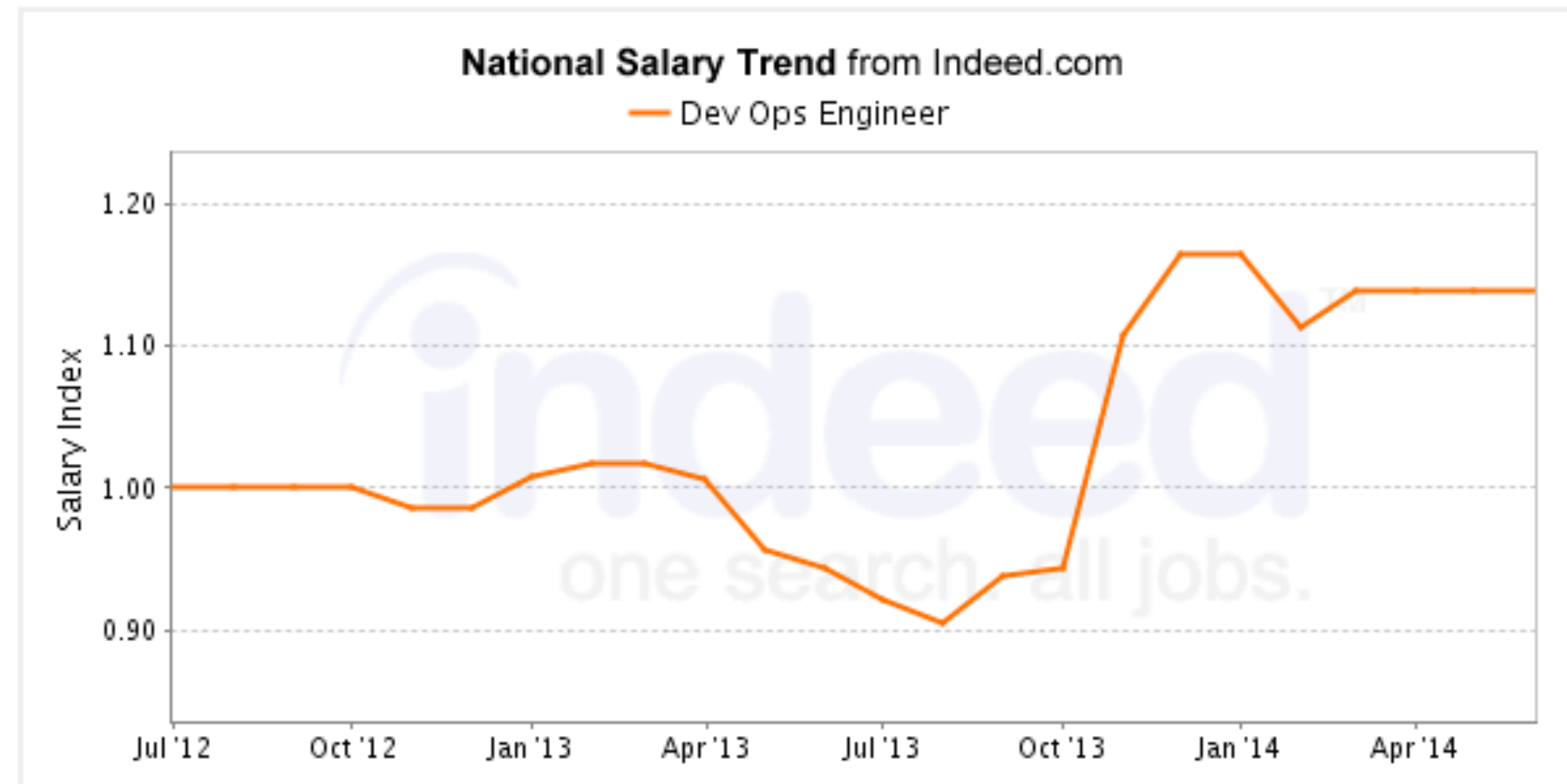


Procedure Document

- Use the next procedure document in the next lecture to download all the resources for the course
- All resources are in a github repository
 - You can clone the git repository
 - You can download a zip file on the github website

Terraform

Why Terraform



Average Salary of Jobs with Related Titles

Senior Dev Ops Engineer in SAN Francisco, CA	\$146,000	
Dev Ops Engineer in SAN Francisco, CA	\$137,000	
Dev Ops Release Engineer in SAN Francisco, CA	\$153,000	
Dev OP in SAN Francisco, CA	\$124,000	
Senior Linux Devops Engineer in SAN Francisco, CA	\$153,000	
Dev Ops Automation Engineer in SAN Francisco, CA	\$138,000	

- The average salary of a DevOps / Automation engineer goes up to \$146,000 in San Fransisco
- Terraform has gained a lot in popularity over the last years
- It is now one of the most sought-after skills in this field
- It's a tool that you will need to master when provisioning any cloud environment

Terraform

- If you are in a Ops or DevOps role, you'll want to **automate** your infrastructure
- Terraform is a great tool to do this:
 - It can automate your infrastructure **in code**
 - It will provision your **cloud infrastructure**, e.g. AWS
 - You can keep a history of your changes made to your infrastructure
 - Collaborate within a team to build infrastructure

Who am I

- My name is Edward Viaene
- I am a **consultant** and **trainer** in Cloud and Big Data technologies
- I apply **DevOps techniques** in all my IT projects
- I held various roles from **banking** to **startups**
- I have a **background** in Unix/Linux, Networking, Security, Risk, and distributed computing
- Nowadays I specialize in everything that has to do with **Cloud** and **DevOps**

Online Training

- **Online training** on Udemy
 - **DevOps, Distributed Computing, Cloud, Big Data (Hadoop)**
 - Using online video lectures
 - 5,000+ enrolled students in 95+ countries

Course Layout

Introduction	Terraform basics	Terraform on AWS	Terraform with Packer
What is Terraform	Variables	VPC	Building AMIs
Installing	Provisioning	EC2 instances and EBS	
First steps	Output	Security Groups	
	State	Userdata	
	DataSources	Static IPs and DNS	
	Template Provider	RDS	
	Modules	IAM Users and Groups	
	Other providers	IAM Roles	

What is Terraform

Terraform

- Infrastructure as **code**
- **Automation** of your infrastructure
- Keep your infrastructure in a certain **state** (compliant)
 - e.g. 2 web instances with 2 volumes, and 1 load balancer
- Make your infrastructure **auditable**
 - You can keep your infrastructure change history in a **version control** system like GIT

Terraform

- Ansible, Chef, Puppet, Saltstack have a focus on automating the **installation and configuration** of software
 - Keeping the **machines** in compliance, in a certain state
- Terraform can automate provisioning of the **infrastructure itself**
 - eg. Using the AWS, DigitalOcean, Azure API
 - Works well with automation software like ansible to install software after the infrastructure is provisioned

Demo

Installing terraform

Demo

Installing terraform using Vagrant

First steps in terraform

Spinning up instance on AWS

Terraform

- Spinning up an instance on AWS
 - Open AWS Account
 - Create IAM admin user
 - Create terraform file to spin up t2.micro instance
 - Run terraform apply

Terraform

- Testing what infrastructure would be built based on our terraform files:

```
$ terraform plan
```

```
...
```

```
+ aws_instance.example
  ami:          "ami-0d729a60"
  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>"
  root_block_device.#: "<computed>"
  security_groups.#: "<computed>"
  source_dest_check: "true"
  subnet_id:        "<computed>"
  tenancy:          "<computed>"
  vpc_security_group_ids.#: "<computed>"
```

Terraform

- Then use terraform apply to execute the changes:

```
$ terraform apply
aws_instance.example: Creating...
  ami:                "" => "ami-0d729a60"
  instance_type:      "" => "t2.micro"
  [...]

aws_instance.example: Creation complete

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

Terraform

- Keeping the changes in an out file

```
$ terraform plan -out changes.terraform  
...
```

- Then only apply those changes to the infrastructure

```
$ terraform apply changes.terraform  
...  
$ rm changes.terraform
```

Terraform

- Afterwards the infrastructure can be destroyed again:

```
$ terraform destroy
aws_instance.example: Destroying...

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

...
```

- Now the instances are terminated and you stop paying AWS for the EC2 instances
- Be very careful with this command on a production environment!

First steps in terraform

Summary

Variables in terraform

Terraform

- Everything in one file is not great
- Use variables to **hide secrets**
 - You don't want the AWS credentials in your git repository
- Use variables for elements that **might change**
 - AMIs are different per region
- Use variables to make it yourself easier to reuse terraform files

Terraform

instance.tf

```
provider "aws" {  
  access_key = "ACCESS_KEY_HERE"  
  secret_key = "SECRET_KEY_HERE"  
  region     = "us-east-1"  
}  
  
resource "aws_instance" "example" {  
  ami          = "ami-0d729a60"  
  instance_type = "t2.micro"  
}
```

Terraform

provider.tf

```
provider "aws" {  
  access_key = "${var.AWS_ACCESS_KEY}"  
  secret_key = "${var.AWS_SECRET_KEY}"  
  region = "${var.AWS_REGION}"  
}
```

vars.tf

```
variable "AWS_ACCESS_KEY" {}  
variable "AWS_SECRET_KEY" {}  
variable "AWS_REGION" {  
  default = "eu-west-1"  
}
```

terraform.tfvars

```
AWS_ACCESS_KEY = ""  
AWS_SECRET_KEY = ""  
AWS_REGION = ""
```

instance.tf

```
resource "aws_instance" "example" {  
  ami = "ami-0d729a60"  
  instance_type = "t2.micro"  
}
```

Terraform

provider.tf

```
provider "aws" {  
  access_key = "${var.AWS_ACCESS_KEY}"  
  secret_key = "${var.AWS_SECRET_KEY}"  
  region = "${var.AWS_REGION}"  
}
```

instance.tf

```
resource "aws_instance" "example" {  
  ami = "${lookup(var.AMIS, var.AWS_REGION)}"  
  instance_type = "t2.micro"  
}
```

vars.tf

```
variable "AWS_ACCESS_KEY" {}  
variable "AWS_SECRET_KEY" {}  
variable "AWS_REGION" {  
  default = "eu-west-1"  
}  
variable "AMIS" {  
  type = "map"  
  default = {  
    us-east-1 = "ami-13be557e"  
    us-west-2 = "ami-06b94666"  
    eu-west-1 = "ami-0d729a60"  
  }  
}
```

<https://cloud-images.ubuntu.com/locator/ec2/>

Demo

Using variables

Provisioning software

Terraform

- There are 2 ways to **provision software** on your instances
- You can build your **own custom AMI** and bundle your software with the image
 - **Packer** is a great tool to do this
- Another way is to boot **standardized AMIs**, and then install the software on it you need
 - Using file uploads
 - Using remote exec
 - Using automation tools like chef, puppet, ansible

Terraform

- **Current state** for terraform with automation (Q4 2016):
 - **Chef is integrated** within terraform, you can add chef statements
 - You can run **puppet agent** using remote-exec
 - For ansible, you can first run terraform, and **output** the IP addresses, then run **ansible-playbook** on those hosts
 - This can be automated in a workflow script
 - There are **3rd party initiatives** integrating Ansible with terraform

File uploads

instance.tf

```
resource "aws_instance" "example" {  
  ami          = "${lookup(var.AMIS, var.AWS_REGION)}"  
  instance_type = "t2.micro"  
  
  provisioner "file" {  
    source = "app.conf"  
    destination = "/etc/myapp.conf"  
  }  
}
```

- File uploads is an **easy way** to upload a file or a script
- Can be used in conjunction with **remote-exec** to execute a script
- The provisioner may use **SSH** (Linux hosts) or **WinRM** (on windows hosts)

File uploads

- To override the SSH defaults, you can use “connection”:

instance.tf

```
resource "aws_instance" "example" {
  ami      = "${lookup(var.AMIS, var.AWS_REGION)}"
  instance_type = "t2.micro"

  provisioner "file" {
    source = "script.sh"
    destination = "/opt/script.sh"
    connection {
      user = "${var.instance_username}"
      password = "${var.instance_password}"
    }
  }
}
```

- When spinning up instances on AWS, ec2-user is the default user for Amazon Linux and ubuntu for Ubuntu Linux

File uploads

- Typically on AWS, you'll use SSH keypairs:

instance.tf

```
resource "aws_key_pair" "mykey" {
  key_name = "mykey"
  public_key = "ssh-rsa my-public-key"
}

resource "aws_instance" "example" {
  ami          = "${lookup(var.AMIS, var.AWS_REGION)}"
  instance_type = "t2.micro"
  key_name     = "${aws_key_pair.mykey.key_name}"

  provisioner "file" {
    source = "script.sh"
    destination = "/opt/script.sh"
    connection {
      user = "${var.instance_username}"
      private_key = "${file("${var.path_to_private_key}")}"
    }
  }
}
```

Terraform

- After you uploaded a script, you'll want to execute it
- You can execute a script using remote-exec:

```
resource "aws_instance" "example" {
  ami          = "${lookup(var.AMIS, var.AWS_REGION)}"
  instance_type = "t2.micro"

  provisioner "file" {
    source = "script.sh"
    destination = "/opt/script.sh"
  }
  provisioner "remote-exec" {
    inline = [
      "chmod +x /opt/script.sh",
      " /opt/script.sh arguments"
    ]
  }
}
```

Demo

Provisioning with script

Demo

Provisioning Windows Server

Output

Terraform

- Terraform keeps **attributes** of all the **resources** you create
 - eg. the **aws_instance** resource has the **attribute public_ip**
- Those attributes can be **queried** and **outputted**
- This can be useful just to output valuable information or to feed information to external software

Terraform

- Use “output” to display the public IP address of an AWS resource:

```
resource "aws_instance" "example" {  
  ami      = "${lookup(var.AMIS, var.AWS_REGION)}"  
  instance_type = "t2.micro"  
}  
  
output "ip" {  
  value = "${aws_instance.example.public_ip}"  
}
```

- You can refer to any attribute by specifying the following elements in your variable:
 - The resource type: `aws_instance`
 - The resource name: `example`
 - The attribute name: `public_ip`

Terraform

- You can also use the attributes in a **script**:

```
resource "aws_instance" "example" {  
  ami      = "${lookup(var.AMIS, var.AWS_REGION)}"  
  instance_type = "t2.micro"  
  provisioner "local-exec" {  
    command = "echo ${aws_instance.example.private_ip} >> private_ips.txt"  
  }  
}
```

- Useful for instance to start automation scripts after infrastructure provisioning
- You can populate the IP addresses in an **ansible host** file
- Or another possibility: execute a script (with attributes as argument) which will take care of a **mapping** of resource name and the IP address

Demo

outputting

Terraform state

Terraform

- Terraform keeps the **remote state** of the infrastructure
- It stores it in a file called **terraform.tfstate**
- There is also a backup of the previous state in **terraform.tfstate.backup**
- When you execute terraform **apply**, a new terraform.tfstate and backup is **written**
- This is how terraform **keeps track** of the **remote state**
 - If the remote state changes and you hit terraform apply again, terraform will make changes to meet the **correct remote state** again
 - e.g. you **terminate** an instance that is managed by terraform, after terraform apply it will be **started** again

Terraform

- You can keep the terraform.tfstate in **version control**
 - e.g. git
- It gives you a **history** of your terraform.tfstate file (which is just a big JSON file)
- It allows you to **collaborate** with other team members
 - Unfortunately you can get conflicts when 2 people work at the same time
- Local state works well in the beginning, but when your project becomes bigger, you might want to store your state **remote**

Terraform

- The **terraform state** can be saved remote, using the **backend** functionality in terraform
- The default is a **local backend** (the local terraform state file)
- Other backends include:
 - **s3** (with a locking mechanism using dynamoDB)
 - **consul** (with locking)
 - **terraform enterprise** (the commercial solution)

Terraform

- Using the backend functionality has definitely benefits:
- Working in a team: it allows for **collaboration**, the remote state will always be **available** for the whole team
- The state file is not stored locally. Possible **sensitive information** is now only stored in the remote state
- Some backends will enable **remote operations**. The terraform apply will then run completely remote. These are called the enhanced backends (<https://www.terraform.io/docs/backends/types/index.html>)

Terraform

- There are 2 steps to configure a remote state:
 - Add the backend code to a .tf file
 - Run the initialization process

Terraform

- To configure a consul remote store, you can add a file backend.tf with the following contents:

```
terraform {  
  backend "consul" {  
    address = "demo.consul.io" # hostname of consul cluster  
    path    = "terraform/myproject"  
  }  
}
```

Terraform

- You can also store your state in S3:

```
terraform {  
  backend "s3" {  
    bucket = "mybucket"  
    key    = "terraform/myproject"  
    region = "eu-west-1"  
  }  
}
```

- When using an S3 remote state, it's best to configure the AWS credentials:

```
$ aws configure  
AWS Access Key ID []: AWS-key  
AWS Secret Access Key []: AWS_secret_key  
Default region name []: eu-west-1  
Default output format [None]:
```

Terraform

- Next step, terraform init:

```
$ terraform init
Initializing the backend...
region
  The region of the S3 bucket.
```

```
Enter a value: eu-west-1
```

Successfully configured the backend "s3"! Terraform will automatically use this backend unless the backend configuration changes.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see any changes that are required for your infrastructure. All Terraform commands should now work.

If you ever set or change modules or backend configuration for Terraform, rerun this command to reinitialize your environment. If you forget, other commands will detect it and remind you to do so if necessary.

Terraform

- Using a **remote** store for the terraform state will ensure that you always have the **latest version** of the state
- It avoids having to **commit** and **push** the terraform.tfstate to version control
- Terraform remote stores don't always support **locking**
 - The documentation always mentions if locking is available for a remote store
 - S3 and consul support it

Terraform

- You can also specify a (read-only) remote store directly in the .tf file

```
data "terraform_remote_state" "aws-state" {  
  backend = "s3"  
  config {  
    bucket = "terraform-state"  
    key = "terraform.tfstate"  
    access_key = "${var.AWS_ACCESS_KEY}"  
    secret_key = "${var.AWS_SECRET_KEY}"  
    region = "${var.AWS_REGION}"  
  }  
}
```

- This is only useful as a read only feed from your remote file
- It's a datasource (which I'll cover later)
- Useful to generate outputs

Demo

Terraform state with S3

Datasources

Datasources

- For certain providers (like AWS), terraform provides datasources
- Datasources provide you with dynamic information
 - A lot of data is available by AWS in a structured format using their API
 - Terraform also exposes this information using data sources
- Examples:
 - List of AMIs
 - List of availability Zones

Datasources

- Another great example is the datasource that gives you all IP addresses in use by AWS
- This is great if you want to filter traffic based on an AWS region
 - e.g. allow all traffic from amazon instances in Europe
- Filtering traffic in AWS can be done using **security groups**
 - Incoming and outgoing traffic can be filtered by protocol, IP range, and port
 - Similar to iptables (linux) or a firewall appliance

Datasources

```
data "aws_ip_ranges" "european_ec2" {
  regions = [ "eu-west-1", "eu-central-1" ]
  services = [ "ec2" ]
}

resource "aws_security_group" "from_europe" {
  name = "from_europe"

  ingress {
    from_port = "443"
    to_port = "443"
    protocol = "tcp"
    cidr_blocks = [ "${data.aws_ip_ranges.european_ec2.cidr_blocks}" ]
  }
  tags {
    CreateDate = "${data.aws_ip_ranges.european_ec2.create_date}"
    SyncToken = "${data.aws_ip_ranges.european_ec2.sync_token}"
  }
}
```

Example from https://www.terraform.io/docs/providers/aws/d/ip_ranges.html

Demo

Datasources

Template provider

Template provider

- The template provider can help creating **customized configuration files**
- You can build **templates based on variables** from terraform resource attributes (e.g. a public IP address)
- The result is a string that can be used as a variable in terraform
 - The string contains a template
 - e.g. a configuration file
- Can be used to create generic templates or cloud init configs

Template provider

- In AWS, you can pass commands that need to be executed when the instance starts for the first time
- In AWS this is called “user-data”
- If you want to pass user-data that depends on other information in terraform (e.g. IP addresses), you can use the provider template
- There’s a separate section on userdata in this course

Template provider

- First you create a template file:

```
#!/bin/bash  
echo "database-ip = ${myip}" >> /etc/myapp.config
```

- Then you create a `template_file` resource that will read the template file and replace `${myip}` with the IP address of an AWS instance created by terraform

```
data "template_file" "my-template" {  
  template = "${file("templates/init.tpl")}"  
  
  vars {  
    myip = "${aws_instance.database1.private_ip}"  
  }  
}
```

Template provider

- Then you can use the my-template resource when creating a new instance

```
# Create a web server
resource "aws_instance" "web" {
  # ...

  user_data = "${data.template_file.my-template.rendered}"
}
```

- When terraform runs, it will see that it first needs to spin up the database¹ instance, then generate the template, and only then spin up the web instance
- The web instance will have the template injected in the user_data, and when it launches, the user-data will create a file /etc/myapp.config with the IP address of the database

Other providers

Other providers

- Terraform is a tool to create and manage **infrastructure resources**
- Terraform has **many providers** to choose from
- **AWS** is the most **popular** one, and will be the one I'll discuss most in this course
- Potentially **any company** that opens up an API, can be used as a terraform provider

Other providers

- Some other **examples** of Cloud providers supported by terraform are:
 - Google Cloud
 - Azure
 - Heroku
 - DigitalOcean
- And on-premise / private cloud:
 - VMware vCloud / vSphere / OpenStack

Other providers

- It's not only limited to cloud providers:
 - Datadog - monitoring
 - GitHub - version control
 - Mailgun - emailing (SMTP)
 - DNSSimple / DNSMadeEAsy / UltraDNS - DNS hosting
- The full list can be found at <https://www.terraform.io/docs/providers/index.html>

Other providers

- The terraform resources for other providers are **very similar**
- A digitalOcean example:

```
variable "do_token" {}

provider "digitalocean" {
  token = "${var.do_token}"
}

resource "digitalocean_droplet" "mydroplet" {
  image = "ubuntu-14-04-x64"
  name = "web-1"
  region = "nyc2"
  size = "512mb"
}
```


Modules

Modules

- You can use modules to make your terraform more **organized**
- Use **third party** modules
 - Modules from github
- **Reuse** parts of your code
 - e.g. to set up network in AWS - the Virtual Private Network (VPC)

Modules

- Use a module from git

```
module "module-example" {  
  source = "github.com/wardviaene/terraform-module-example"  
}
```

- Use a module from a local folder

```
module "module-example" {  
  source = "./module-example"  
}
```

Modules

- Pass arguments to the module

```
module "module-example" {  
  source = "./module-example"  
  region = "us-west-1"  
  ip-range = "10.0.0.0/8"  
  cluster-size = "3"  
}
```

Modules

- Inside the module folder, you just have again terraform files:

module-example/vars.tf

```
variable "region" {} # the input parameters
variable "ip-range" {}
variable "cluster-size" {}
```

module-example/cluster.tf

```
# vars can be used here
resource "aws_instance" "instance-1" { ... }
resource "aws_instance" "instance-2" { ... }
resource "aws_instance" "instance-3" { ... }
```

module-example/output.tf

```
output "aws-cluster" {
  value = "${aws_instance.instance-1.public_ip},${aws_instance.instance-2.public_ip},${aws_instance.instance-2.public_ip}"
}
```

Modules

- Use the **output** from the module in the main part of your code:

```
output "some-output" {  
  value = "${module.module-example.aws-cluster}"  
}
```

- I'm just using the output resource here, but you can use the variables anywhere in the terraform code

Demo

Modules

Terraform Command Overview

Terraform Command Overview

- Terraform is very much focussed on the **resource definitions**
- It has a **limited toolset** available to modify, import, create these resource definitions
 - Still, every new release there are new features coming out to make it easier to handle your resources
 - For example, today (Q4 - 2016), there's still no good tool to import your non-terraform maintained infrastructure and create the definitions for you
 - There is an external tool called **terraforming** that you can use for now, but it'll take you quite some time to convert your current infrastructure to managed terraform infrastructure (<https://github.com/dtan4/terraforming>)

Terraform Command Overview

Command	Description
terraform apply	Applies state
destroy	Destroys all terraform managed state (use with caution)
fmt	Rewrite terraform configuration files to a canonical format and style
get	Download and update modules
graph	Create a visual representation of a configuration or execution plan
import [options] ADDRESS ID	Import will try and find the infrastructure resource identified with ID and import the state into terraform.tfstate with resource id ADDRESS

Terraform Command Overview

Command	Description
output [options] [NAME]	Output any of your resources. Using NAME will only output a specific resource
plan	terraform plan, show the changes to be made to the infrastructure
push	Push changes to Atlas, Hashicorp's Enterprise tool that can automatically run terraform from a centralized server
refresh	Refresh the remote state. Can identify differences between state file and remote state
remote	Configure remote state storage
show	Show human readable output from a state or a plan

Terraform Command Overview

Command	Description
state	Use this command for advanced state management, e.g. Rename a resource with terraform state mv aws_instance.example aws_instance.production
taint	Manually mark a resource as tainted, meaning it will be destroyed and recreated at the next apply
validate	validate your terraform syntax
untaint	undo a taint

Demo

Command overview

Terraform with AWS

Creating a VPC

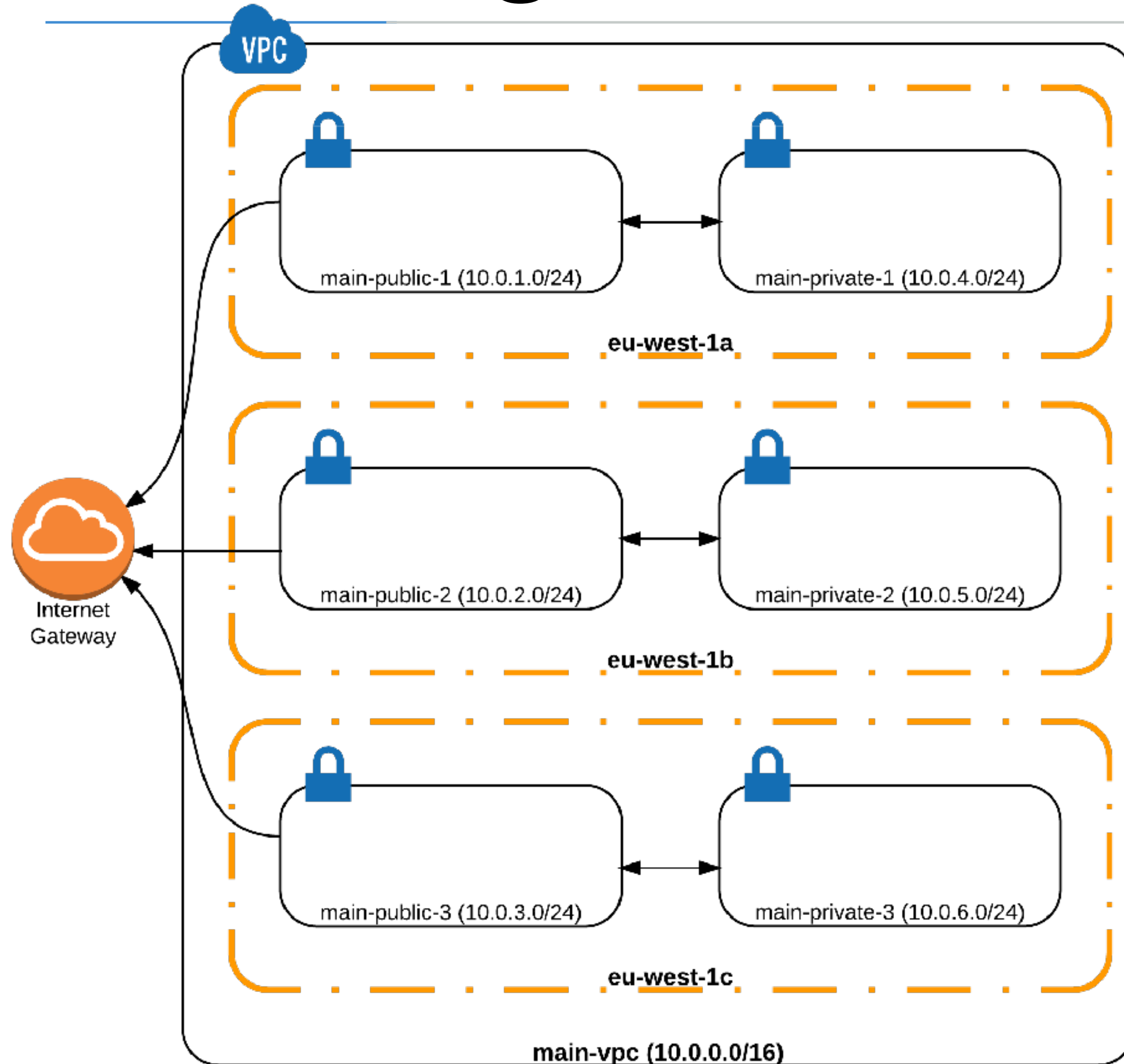
Creating the VPC

- On Amazon AWS, you have a default VPC (**Virtual Private Network**) created for you by AWS to launch instances in
- Up until now we used this **default** VPC
- VPC isolates the instances on a **network** level
 - It's like your **own private network** in the cloud
- Best practice is to always launch your instances in a **VPC**
 - the default VPC
 - or one you create yourself (managed by terraform)

Creating the VPC

- There's also **EC2-Classical**, which is basically one big network where all AWS customers could launch their instances in
- For smaller to medium setups, **one VPC** (per region) will be suitable for your needs
- An instance launched in one VPC can never communicate with an instance in another VPC using their **private IP addresses**
 - They could communicate still, but using their public IP (not recommended)
 - You could also link 2 VPCs, called peering

Creating the VPC



- On Amazon AWS, you start by creating your own **Virtual Private Network** to deploy your instances (servers) / databases in
- This VPC uses the **10.0.0.0/16** addressing space, allowing you to use the IP addresses that start with "10.0.", like this: **10.0.x.x**.
- This VPC covers the **eu-west-1 region**, which is an Amazon AWS Region in Ireland.

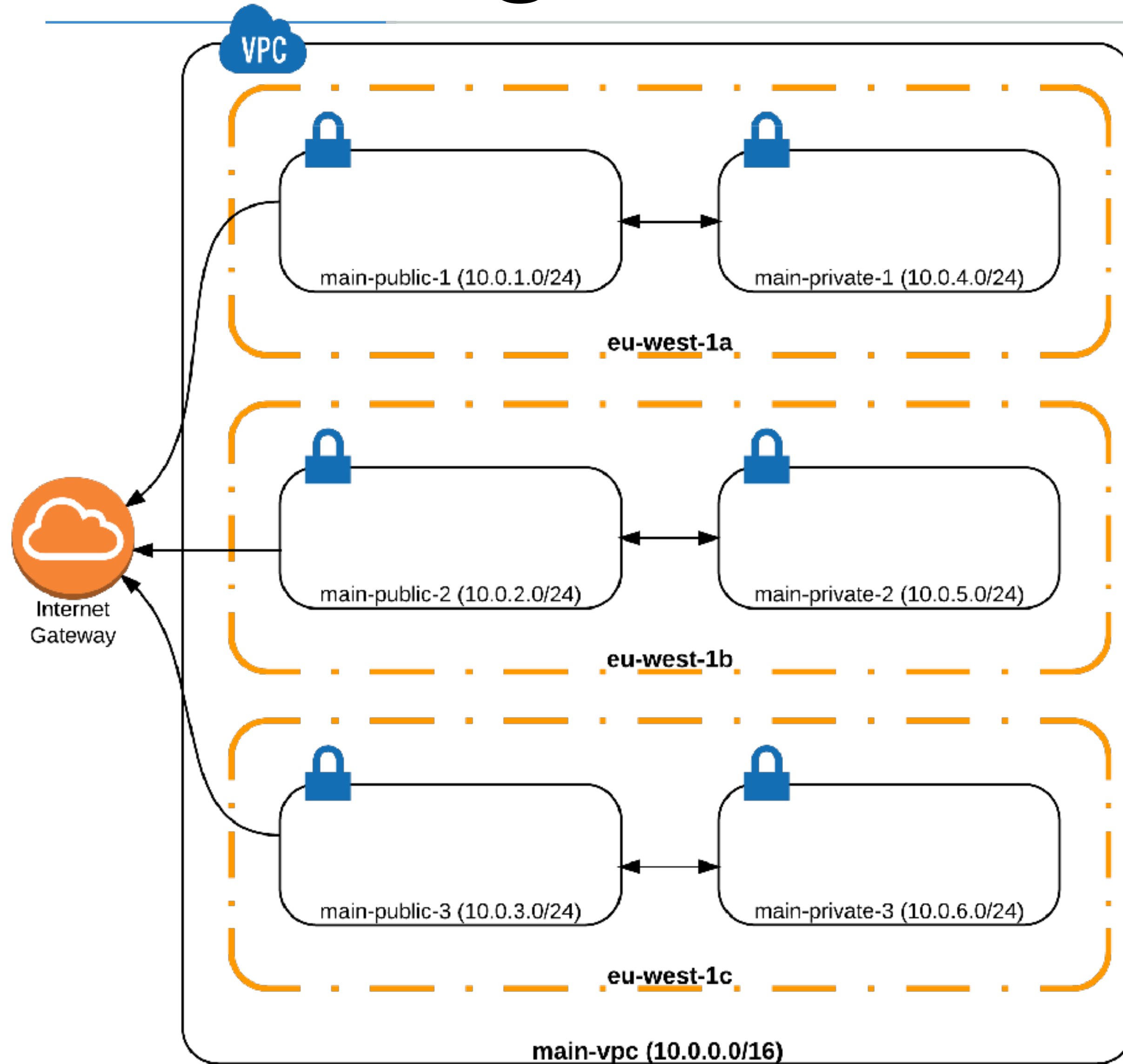
Private Subnets

Range	From	to
10.0.0.0/8	10.0.0.0	10.255.255.255
172.16.0.0/12	172.16.0.0	172.31.255.255.
192.168.0.0/16	192.168.0.0	192.168.255.255

Subnet masks

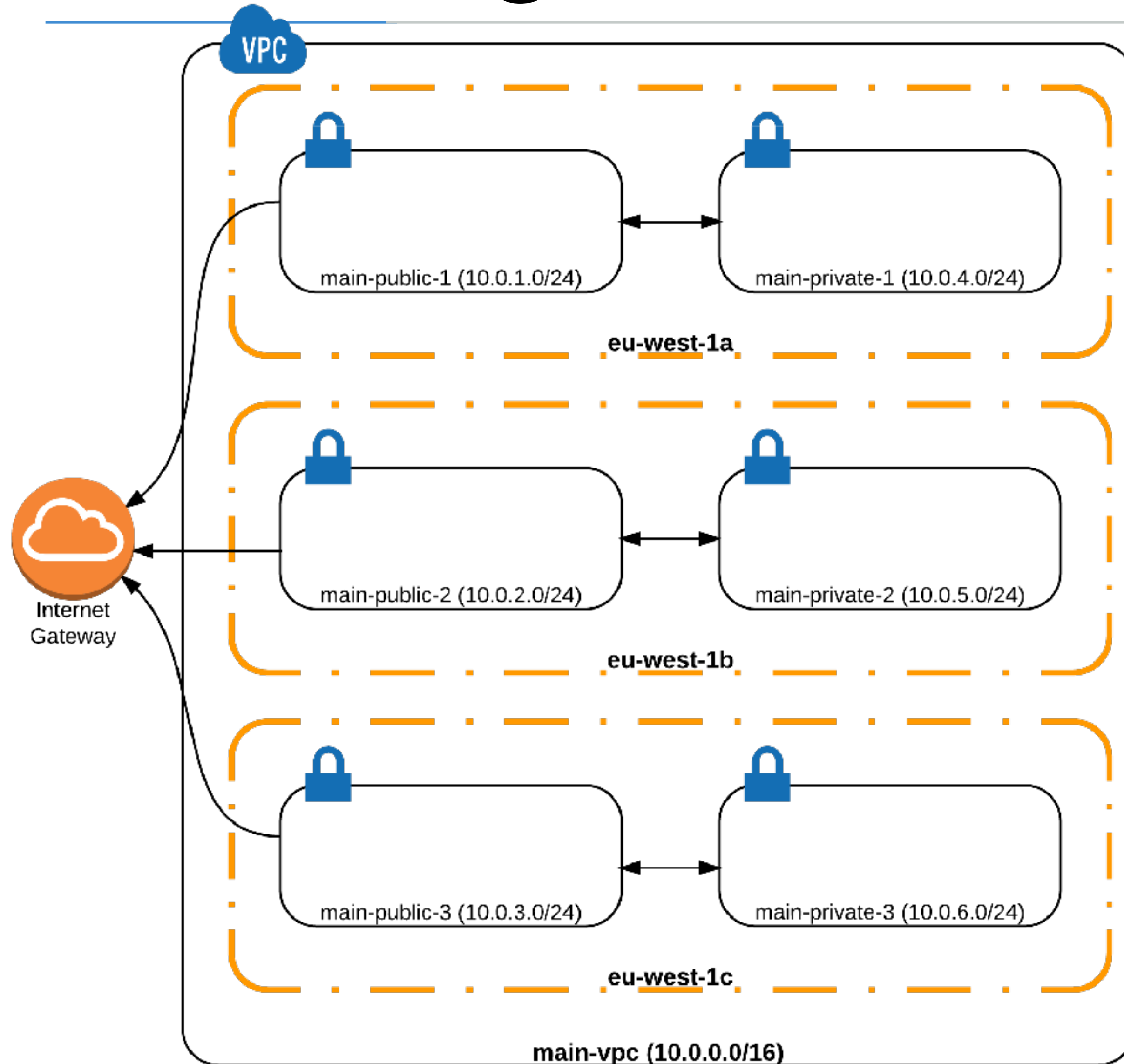
Range	Network mask	Total addresses	Description	Examples
10.0.0.0/8	255.0.0.0	16,777,214	Full 10.x.x.x range	10.0.0.1, 10.100.200.20
10.0.0.0/16	255.255.0.0	65,536	What we used for our VPC	10.0.5.1, 10.0.20.2, 10.0.100.3
10.1.0.0/16	255.255.0.0	65,536	What we can use for another VPC	10.1.5.1, 10.1.20.2, 10.1.100.3
10.0.0.0/24	255.255.255.0	256	All addresses within from 10.0.0.0-10.0.0.255	10.0.0.1, 10.0.0.2, 10.0.0.3
10.0.1.0/24	255.255.255.0	256	All addresses within from 10.0.1.0-10.0.1.255	10.0.1.1, 10.0.1.2, 10.0.1.3
10.0.0.5/32	255.255.255.255	1	Single host	10.0.0.5

Creating the VPC



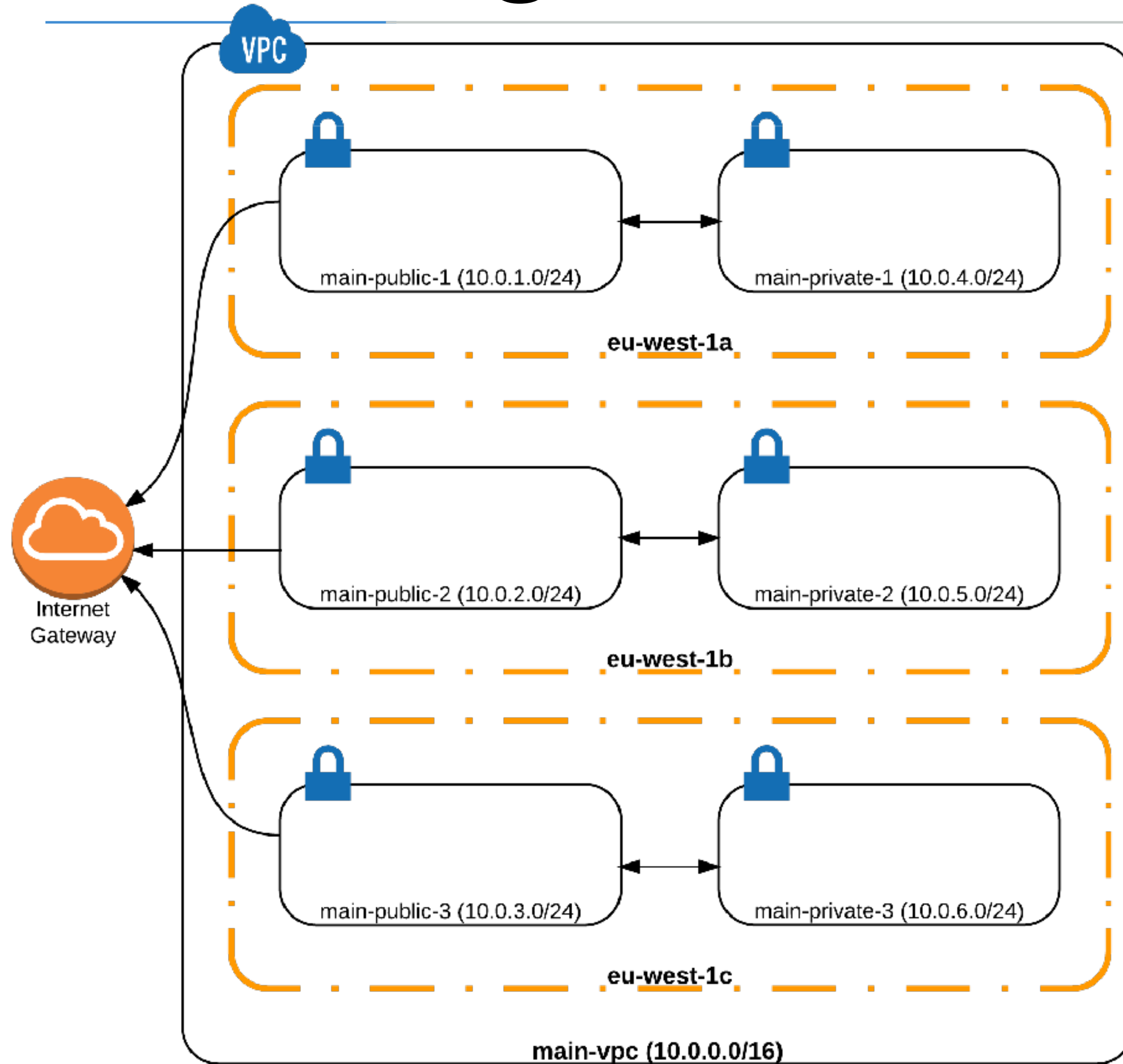
- Every availability zone has its own **public** and **private** subnet
- Instances started in subnet **main-public-3** will have IP address **10.0.3.x**, and will be launched in the eu-west-1c availability zone (Amazon calls 1 datacenter an availability zone)
- An instance launched in **main-private-1** will have an IP address **10.0.4.x** and will reside in Amazon's eu-west-1a Availability Zone (AZ)

Creating the VPC



- All the public subnets are connected to an **Internet Gateway**. These instances will also have a public IP address, allowing them to be reachable from the internet
- Instances launched in the private subnets don't get a public IP address, so they will not be reachable from the internet
- Instances from **main-public** can reach instances from **main-private**, because they're all in the same VPC. This of course if you set the firewall to allow traffic from one to another.

Creating the VPC



- Typically, you use the public subnets for internet-facing services/applications
- Databases, caching services, and backends all go into the private subnets
- If you use a Load Balancer (LB), you will typically put the LB in the public subnets and the instances serving an application in the private subnets.

VPC demo

Spinning up instances

EC2, EBS

EC2 instance

- Spinning up an EC2 instance is very straightforward, let's recap

provider.tf

```
provider "aws" {  
  access_key = "${var.AWS_ACCESS_KEY}"  
  secret_key = "${var.AWS_SECRET_KEY}"  
  region = "${var.AWS_REGION}"  
}
```

instance.tf

```
resource "aws_instance" "example" {  
  ami = "${lookup(var.AMIS, var.AWS_REGION)}"  
  instance_type = "t2.micro"  
}
```

vars.tf

```
variable "AWS_ACCESS_KEY" {}  
variable "AWS_SECRET_KEY" {}  
variable "AWS_REGION" {  
  default = "eu-west-1"  
}  
variable "AMIS" {  
  type = "map"  
  default = {  
    us-east-1 = "ami-13be557e"  
    us-west-2 = "ami-06b94666"  
    eu-west-1 = "ami-844e0bf7"  
  }  
}
```

EC2 instance

- Now we want to launch the instance in our newly created VPC
 - With security groups
 - Using a keypair that will be uploaded by terraform

instance.tf

```
resource "aws_instance" "example" {  
  ami          = "${lookup(var.AMIS, var.AWS_REGION)}"  
  instance_type = "t2.micro"  
  
  # the VPC subnet  
  subnet_id = "${aws_subnet.main-public-1.id}"  
  
  # the security group  
  vpc_security_group_ids = ["${aws_security_group.allow-ssh.id}"]  
  
  # the public SSH key  
  key_name = "${aws_key_pair.mykeypair.key_name}"  
}
```

EC2 instance

- We need a new **security group** for this EC2 instance
- A security group is just like a firewall, **managed** by AWS
- You specify **ingress** (incoming) and **egress** (outgoing) traffic rules
- If you only want to access SSH (port 22), then you could create a security group that:
 - Allows ingress port 22 on IP address range 0.0.0.0/0 (all IPs)
 - It's best practice to only allow your work/home/office IP
 - Allow all outgoing traffic from the instance to 0.0.0.0/0 (all IPs, so everywhere)

EC2 instance

securitygroup.tf

```
resource "aws_security_group" "allow-ssh" {  
  vpc_id = "${aws_vpc.main.id}"  
  name = "allow-ssh"  
  description = "security group that allows ssh and all egress traffic"  
  egress {  
    from_port = 0  
    to_port = 0  
    protocol = "-1"  
    cidr_blocks = ["0.0.0.0/0"]  
  }  
  
  ingress {  
    from_port = 22  
    to_port = 22  
    protocol = "-1"  
    cidr_blocks = ["0.0.0.0/0"]  
  }  
  tags {  
    Name = "allow-ssh"  
  }  
}
```

EC2 instance

- To be able to log in, the last step is to make sure AWS installs our public key pair on the instance
- Our EC2 instance already refers to a `aws_key_pair.mykeypair.key_name`, you just need to declare it in terraform:

keypairs.tf

```
resource "aws_key_pair" "mykeypair" {  
  key_name = "mykeypair"  
  public_key = "${file("keys/mykeypair.pub")}"  
}
```

- The `keys/mykeypair.pub` will be uploaded to AWS and will allow an instance to be launched with this **public key** installed on it
- You never upload your **private key**! You use your private key to login to the instance

AWS Instance demo

Spinning up instances

Block storage (EBS)

EC2 instance

- The t2.micro instance with this particular AMI automatically adds 8 GB of EBS storage (= **Elastic Block Storage**)
- Some instance types have **local storage** on the instance itself
 - This is called ephemeral storage
 - This type of storage is always lost when the instance terminates
- The 8 GB EBS **root volume** storage that comes with the instance is also set to be automatically removed when the instance is terminated
 - You could still instruct AWS not to do so, but that would be counter-intuitive (anti-pattern)
- In most cases the 8GB for the OS (root block device) suffices

EC2 instance

- In our next example I'm adding an **extra EBS storage volume**
 - Extra volumes can be used for the **log** files, any **real data** that is put on the instance
 - That data will be persisted until you instruct AWS to remove it
- **EBS storage** can be added using a terraform resource and then attached to our instance

EC2 instance

instance.tf

```
resource "aws_instance" "example" {  
  ...  
}  
  
resource "aws_ebs_volume" "ebs-volume-1" {  
  availability_zone = "eu-west-1a"  
  size = 20  
  type = "gp2" # General Purpose storage, can also be standard or io1 or st1  
  tags {  
    Name = "extra volume data"  
  }  
}  
  
resource "aws_volume_attachment" "ebs-volume-1-attachment" {  
  device_name = "/dev/xvddh"  
  volume_id = "${aws_ebs_volume.ebs-volume-1.id}"  
  instance_id = "${aws_instance.example.id}"  
}
```

EC2 instance

- In the previous example we added an **extra** volume
- The **root volume** of 8 GB still exists
- If you want to increase the storage or type of the root volume, you can use **root_block_device** within the aws_instance resource

```
resource "aws_instance" "example" {  
  ...  
  root_block_device {  
    volume_size = 16  
    volume_type = "gp2"  
    delete_on_termination = true # whether to delete the root block device when the instance gets terminated or not  
  }  
}
```

EBS demo

Userdata

Userdata

- Userdata in AWS can be used to do any customization at **launch**:
 - You can install extra software
 - Prepare the instance to join a cluster
 - e.g. consul cluster, ECS cluster (docker orchestration)
 - Execute commands / scripts
 - Mount volumes
- Userdata is only executed at the creation of the instance, not when the instance reboots

Userdata

- Terraform allows you to add userdata to the `aws_instance` resource
 - Just as a string (for simple commands)
 - Using templates (for more complex instructions)

Userdata

- This is an example of userdata just using a string
- It'll install an OpenVPN application server at boot time

instance.tf

```
resource "aws_instance" "example" {
  ami          = "${lookup(var.AMIS, var.AWS_REGION)}"
  instance_type = "t2.micro"

  # the VPC subnet
  subnet_id = "${aws_subnet.main-public-1.id}"
  # the security group
  vpc_security_group_ids = ["${aws_security_group.allow-ssh.id}"]
  # the public SSH key
  key_name = "${aws_key_pair.mykeypair.key_name}"

  # userdata
  user_data = "#!/bin/bash\nwget http://swupdate.openvpn.org/as/openvpn-as-2.1.2-Ubuntu14.amd_64.deb\nndpkg -i openvpn-as-2.1.2-Ubuntu14.amd_64.deb"
}
```

Userdata

- Another better example is to use the template system of terraform:

instance.tf

```
resource "aws_instance" "example" {
  ami          = "${lookup(var.AMIS, var.AWS_REGION)}"
  instance_type = "t2.micro"

  # the VPC subnet
  subnet_id = "${aws_subnet.main-public-1.id}"
  # the security group
  vpc_security_group_ids = ["${aws_security_group.allow-ssh.id}"]
  # the public SSH key
  key_name = "${aws_key_pair.mykeypair.key_name}"

  # userdata
  user_data = "${data.template_cloudinit_config.cloudinit-example.rendered}"
}
```

Userdata

- Another better example is to use the template system of terraform:

cloudinit.tf

```
provider "cloudinit" {}

data "template_file" "init-script" {
  template = "${file("scripts/init.cfg")}"
  vars {
    region = "${var.AWS_REGION}"
  }
}

data "template_cloudinit_config" "cloudinit-example" {

  gzip = false
  base64_encode = false

  part {
    filename      = "init.cfg"
    content_type  = "text/cloud-config"
    content       = "${data.template_file.init-script.rendered}"
  }
}
```

scripts/init.cfg

```
#cloud-config

repo_update: true
repo_upgrade: all

packages:
- docker

output:
  all: '| tee -a /var/log/cloud-init-output.log'
```

Userdata

cloudinit.tf

```
data "template_cloudinit_config" "cloudinit-example" {

  gzip = false
  base64_encode = false

  part {
    filename    = "init.cfg"
    content_type = "text/cloud-config"
    content     = "${template_file.init-script.rendered}"
  }

  # just a shell script instead of
  part {
    content_type = "text/x-shellscript"
    content     = "#!/bin/bash\nnecho 'hello'"
  }

  # an upstart script (basically an init script to start/stop/restart/reload services)
  part {
    content_type = "text/upstart-job"
    content     = "${file("scripts/start_docker_container.cfg")}"
  }
}
```

Demo

Userdata

Static IPs and DNS

private IP or public EIP, Route 53

Static private IPs

- Private IP addresses will be **auto-assigned** to EC2 instances
- Every **subnet** within the VPC has its own **range** (e.g. 10.0.1.0 - 10.0.1.255)
- By specifying the **private IP**, you can make sure the EC2 instance always uses the same IP address:

```
resource "aws_instance" "example" {  
  ami          = "${lookup(var.AMIS, var.AWS_REGION)}"  
  instance_type = "t2.micro"  
  subnet_id    = "${aws_subnet.main-public-1.id}"  
  private_ip   = "10.0.1.4" # within the range of subnet main-public-1  
}
```

EIP

- To use a public IP address, you can use EIPs (Elastic IP addresses)
- This is a public, static IP address that you can attach to your instance

```
resource "aws_instance" "example" {  
  ami          = "${lookup(var.AMIS, var.AWS_REGION)}"  
  instance_type = "t2.micro"  
  subnet_id    = "${aws_subnet.main-public-1.id}"  
  private_ip   = "10.0.1.4" # within the range of subnet main-public-1  
}  
  
resource "aws_eip" "example-eip" {  
  instance = "${aws_instance.example.id}"  
  vpc      = true  
}
```

- Tip: You can use **aws_eip.example-eip.public_ip** attribute with the output resource to show the IP address after terraform apply

Route53

- Typically, you'll not use IP addresses, but **hostnames**
- This is where **route53** comes in
- You can host a domain name on AWS using **Route53**
- You first need to register a domain name using AWS or any accredited registrar
- You can then create a zone in route53 (e.g. example.com) and add DNS records (e.g. server1.example.com)

Route53

- Adding a zone and records can be done in terraform:

```
resource "aws_route53_zone" "example-com" {  
  name = "example.com"  
}  
resource "aws_route53_record" "server1-record" {  
  zone_id = "${aws_route53_zone.example-com.zone_id}"  
  name = "server1.example.com"  
  type = "A"  
  ttl = "300"  
  records = ["${aws_eip.example-eip.public_ip}"]  
}
```

- Tip: When you register your domain name, you need to add the AWS nameservers to that domain
- Route53 has a lot of nameservers. To know your nameservers for your particular domain, you can use the output resource to output the property **aws_route53_zone.example-com.name_servers**

Demo

Route53

RDS

Relational databases

RDS

- RDS stands for **Relational Database Services**
- It's a managed **database** solution:
 - You can easily setup **replication** (high availability)
 - Automated **snapshots** (for backups)
 - Automated **security updates**
 - Easy **instance replacement** (for vertical scaling)

RDS

- Supported databases are:
 - MySQL
 - MariaDB
 - PostgreSQL
 - Microsoft SQL
 - Oracle

RDS

- Steps to create an RDS instance:
 - Create a **subnet group**
 - Allows you to specify in what subnets the database will be in (e.g. eu-west-1a and eu-west-1b)
 - Create a **Parameter group**
 - Allows you to specify parameters to change settings in the database
 - Create a **security group** that allows incoming traffic to the RDS instance
 - Create the **RDS instance(s)** itself

RDS

- First the **parameter** group:

```
resource "aws_db_parameter_group" "mariadb-parameters" {  
  name = "mariadb-parameters"  
  family = "mariadb10.1"  
  description = "MariaDB parameter group"  
  
  parameter {  
    name = "max_allowed_packet"  
    value = "16777216"  
  }  
}
```

- We also specify a parameter that will be applied to MariaDB

RDS

- Second, we specify the **subnet**:

```
resource "aws_db_subnet_group" "mariadb-subnet" {  
  name = "mariadb-subnet"  
  description = "RDS subnet group"  
  subnet_ids = ["${aws_subnet.main-private-1.id}", "${aws_subnet.main-private-2.id}"]  
}
```

- This subnet group specifies that the RDS will be put in the **private subnets**
 - The RDS will only be accessible from other instances within the same subnet, not from the internet
- The RDS instance will also be placed either in private-1 or private-2, not in the private-3 subnet
 - when you enable High Availability you will have an instance in both subnets

RDS

- Third, the **security group**:

```
resource "aws_security_group" "allow-mariadb" {  
  vpc_id = "${aws_vpc.main.id}"  
  name = "allow-mariadb"  
  description = "allow-mariadb"  
  ingress {  
    from_port = 3306  
    to_port = 3306  
    protocol = "tcp"  
    security_groups = ["${aws_security_group.example.id}"] # allowing access from our example instance  
  }  
  egress {  
    from_port = 0  
    to_port = 0  
    protocol = "-1"  
    cidr_blocks = ["0.0.0.0/0"]  
    self = true  
  }  
  tags {  
    Name = "allow-mariadb"  
  }  
}
```

RDS

- Finally, we specify the **RDS resource**:

```
resource "aws_db_instance" "mariadb" {
  allocated_storage = 100 # 100 GB of storage, gives us more IOPS than a lower number
  engine            = "mariadb"
  engine_version    = "10.1.14"
  instance_class    = "db.t2.small" # use micro if you want to use the free tier
  identifier        = "mariadb"
  name              = "mariadb"
  username          = "root" # username
  password          = "a1c9f8v0zs" # password: create your own random password
  db_subnet_group_name = "${aws_db_subnet_group.mariadb-subnet.name}"
  parameter_group_name = "mariadb-parameters"
  multi_az          = "false" # set to true to have high availability: 2 instances synchronized with each other
  vpc_security_group_ids = ["${aws_security_group.allow-mariadb.id}"]
  storage_type       = "gp2"
  backup_retention_period = 30 # how long you're going to keep your backups
  availability_zone = "${aws_subnet.main-private-1.availability_zone}" # preferred AZ
  tags {
    Name = "mariadb-instance"
  }
}
```

Demo

Using RDS resources

IAM

IAM

- IAM is AWS' **Identity & Access Management**
- It's a service that helps you **control access** to your AWS resources
- In AWS you can create:
 - Groups
 - Users
 - Roles

IAM

- Users can have **groups**
 - for instance an “Administrators” group can give **admin privileges** to users
- Users can authenticate
 - Using a login / password
 - Optionally using a **token**: multifactor Authentication (MFA) using Google Authenticator compatible software
 - an **access key** and **secret key** (the API keys)

IAM roles

- Roles can give users / services (temporary) access that they normally wouldn't have
- The roles can be for instance **attached** to EC2 instances
 - From that instance, a user or service can **obtain** access credentials
 - Using those access credentials the user or service can **assume** the role, which gives them permission to do **something**

IAM roles

- An **example**:
 - You create a **role mybucket-access** and assign the role to an EC2 instance at boot time
 - You give the role the permissions to **read and write** items in “mybucket”
 - When you log in, you can now **assume** this **mybucket-access role**, without using your own credentials - you will be given temporary access credentials which just look like normal user credentials
 - You can now read and write items in “mybucket”

IAM roles

- Instead of a user using aws-cli, a **service** also assume a role
- The service needs to **implement** the **AWS SDK**
- When trying to access the S3 bucket, an API call to AWS will occur
- If roles are configured for this EC2 instance, the AWS API will give temporary access keys which can be used to assume this role
- After that, the SDK can be used just like when you would have normal credentials
- This really happens in the background and you don't see much of it

IAM roles

- IAM Roles only work on **EC2 instances**, and not for instance outside AWS
- The temporary access credentials also need to be **renewed**, they're only valid for a predefined amount of time
 - This is also something the AWS SDK will take care of

IAM

- To create an IAM administrators group in AWS, you can create the group and attach the AWS managed Administrator policy to it

```
resource "aws_iam_group" "administrators" {  
  name = "administrators"  
}  
resource "aws_iam_policy_attachment" "administrators-attach" {  
  name = "administrators-attach"  
  groups = ["${aws_iam_group.administators.name}"]  
  policy_arn = "arn:aws:iam::aws:policy/AdministratorAccess"  
}
```

IAM

- You can also create your own custom policy. This one does the same:

```
resource "aws_iam_group" "administrators" {
  name = "administrators"
}
resource "aws_iam_group_policy" "my_developer_policy" {
  name = "my_administrators_policy"
  group = "${aws_iam_group.administators.id}"
  policy = <<EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "*",
      "Resource": "*"
    }
  ]
}
EOF
}
```

IAM

- Next, create a user and attach it to a group:

```
resource "aws_iam_user" "admin1" {  
  name = "admin1"  
}  
resource "aws_iam_user" "admin2" {  
  name = "admin2"  
}  
resource "aws_iam_group_membership" "administrators-users" {  
  name = "administrators-users"  
  users = [  
    "${aws_iam_user.admin1.name}",  
    "${aws_iam_user.admin2.name}",  
  ]  
  group = "${aws_iam_group.administrators.name}"  
}
```

Demo

IAM user/groups

IAM

IAM Roles

IAM

- Let's create a role now that we want to attach to an EC2 instance:

```
resource "aws_iam_role" "s3-mybucket-role" {
  name = "s3-mybucket-role"
  assume_role_policy = <<EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": "sts:AssumeRole",
      "Principal": {
        "Service": "ec2.amazonaws.com"
      },
      "Effect": "Allow",
      "Sid": ""
    }
  ]
}
EOF
}

resource "aws_iam_instance_profile" "s3-mybucket-role-instanceprofile" {
  name = "s3-mybucket-role"
  roles = ["${aws_iam_role.s3-mybucket-role.name}"]
}
```

IAM

- Attaching this role to an EC2 instance now is pretty easy:

```
resource "aws_instance" "example" {
  ami          = "${lookup(var.AMIS, var.AWS_REGION)}"
  instance_type = "t2.micro"
  subnet_id    = "${aws_subnet.main-public-1.id}"
  vpc_security_group_ids = ["${aws_security_group.allow-ssh.id}"]
  key_name     = "${aws_key_pair.mykeypair.key_name}"

  # role:
  iam_instance_profile = "${aws_iam_instance_profile.s3-mybucket-role-instanceprofile.name}"
}
```

IAM

- Creating the bucket is just another resource:

```
resource "aws_s3_bucket" "b" {  
  bucket = "mybucket-c29df1"  
  acl = "private"  
  
  tags {  
    Name = "mybucket-c29df1"  
  }  
}
```

IAM

- Now we need to add some permissions using a policy document:

```
resource "aws_iam_role_policy" "s3-mybucket-role-policy" {
  name = "s3-mybucket-role-policy"
  role = "${aws_iam_role.s3-mybucket-role.id}"
  policy = <<EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:*"
      ],
      "Resource": [
        "arn:aws:s3:::mybucket-c29df1",
        "arn:aws:s3:::mybucket-c29df1/*"
      ]
    }
  ]
}
EOF
}
```

Demo

IAM Roles

Autoscaling

Autoscaling instances in AWS

Autoscaling

- In AWS autoscaling groups can be created to **automatically add/remove** instances when certain thresholds are reached
 - e.g. your application layer can be **scaled out** when you have more visitors
- To set up autoscaling in AWS you need to setup at least **2 resources**:
 - An AWS **launch configuration**
 - Specifies the properties of the instance to be launched (AMI ID, security group, etc)
 - An **autoscaling group**
 - Specifies the scaling properties (min instances, max instances, health checks)

Autoscaling

- Once the autoscaling group is setup, you can create autoscaling policies
 - A policy is triggered based on a threshold (CloudWatch Alarm)
 - An adjustment will be executed
 - e.g. if the average CPU utilization is more than 20%, then scale up by +1 instances
 - e.g. if the average CPU utilization is less than 5%, then scale down by -1 instances

Autoscaling

- First the launch configuration and the autoscaling group needs to be created:

```
resource "aws_launch_configuration" "example-launchconfig" {
  name_prefix      = "example-launchconfig"
  image_id         = "${lookup(var.AMIS, var.AWS_REGION)}"
  instance_type    = "t2.micro"
  key_name         = "${aws_key_pair.mykeypair.key_name}"
  security_groups  = ["${aws_security_group.allow-ssh.id}"]
}

resource "aws_autoscaling_group" "example-autoscaling" {
  name                  = "example-autoscaling"
  vpc_zone_identifier  = ["${aws_subnet.main-public-1.id}", "${aws_subnet.main-public-2.id}"]
  launch_configuration = "${aws_launch_configuration.example-launchconfig.name}"
  min_size             = 1
  max_size             = 2
  health_check_grace_period = 300
  health_check_type    = "EC2"
  force_delete         = true

  tag {
    key = "Name"
    value = "ec2 instance"
    propagate_at_launch = true
  }
}
```

Autoscaling

- To create a policy, you need a `aws_autoscaling_policy`:

```
resource "aws_autoscaling_policy" "example-cpu-policy" {  
  name           = "example-cpu-policy"  
  autoscaling_group_name = "${aws_autoscaling_group.example-autoscaling.name}"  
  adjustment_type  = "ChangeInCapacity"  
  scaling_adjustment = "1"  
  cooldown         = "300"  
  policy_type      = "SimpleScaling"  
}
```

Autoscaling

- Then, you can create a CloudWatch alarm which will trigger the autoscaling policy

```
resource "aws_cloudwatch_metric_alarm" "example-cpu-alarm" {
  alarm_name      = "example-cpu-alarm"
  alarm_description = "example-cpu-alarm"
  comparison_operator = "GreaterThanOrEqualToThreshold"
  evaluation_periods = "2"
  metric_name      = "CPUUtilization"
  namespace        = "AWS/EC2"
  period           = "120"
  statistic        = "Average"
  threshold        = "30"

  dimensions = {
    "AutoScalingGroupName" = "${aws_autoscaling_group.example-autoscaling.name}"
  }

  actions_enabled = true
  alarm_actions   = ["${aws_autoscaling_policy.example-cpu-policy.arn}"]
}
```

Autoscaling

- If you want to receive an alert (e.g. email) when autoscaling is invoked, need create a SNS topic (Simple Notification Service):

```
resource "aws_sns_topic" "example-cpu-sns" {  
  name      = "sg-cpu-sns"  
  display_name = "example ASG SNS topic"  
} # email subscription is currently unsupported in terraform and can be done using the AWS Web Console
```

- That SNS topic needs to be attached to the autoscaling group:

```
resource "aws_autoscaling_notification" "example-notify" {  
  group_names = ["${aws_autoscaling_group.example-autoscaling.name}"]  
  topic_arn   = "${aws_sns_topic.example-cpu-sns.arn}"  
  notifications = [  
    "autoscaling:EC2_INSTANCE_LAUNCH",  
    "autoscaling:EC2_INSTANCE_TERMINATE",  
    "autoscaling:EC2_INSTANCE_LAUNCH_ERROR"  
  ]  
}
```

Demo

Autoscaling

Load Balancer

Load Balancing autoscaled instances

Load Balancer

- Now that you've autoscaled instances, you might want to put a loadbalancer in front of it
- The AWS Elastic Load Balancer (ELB) automatically distributes incoming traffic across multiple EC2 instances
 - The ELB itself scales when you receive more traffic
 - The ELB will healthcheck your instances
 - If an instance fails its healthcheck, no traffic will be sent to it
 - If a new instances is added by the autoscaling group, the ELB will automatically add the new instances and will start healthchecking it

Load Balancer

- The ELB can also be used as SSL terminator
 - It can offload the encryption away from the EC2 instances
 - AWS can even manage the SSL certificates for you
- ELBs can be spread over multiple Availability Zones for higher fault tolerance
- You will in general achieve higher levels of fault tolerance with an ELB routing the traffic for your application
- ELB is comparable to a nginx / haproxy, but then provided as a service

Load Balancer

- AWS provides 2 different types of load balancers:
 - The Classic Load Balancer (ELB)
 - Routes traffic based on network information
 - e.g. forwards all traffic from port 80 (HTTP) to port 8080 (application)
 - The Application Load Balancer (ALB)
 - Routes traffic based on application level information
 - e.g. can route /api and /website to different EC2 instances

Classic Load Balancer

```
resource "aws_elb" "my-elb" {
  name = "my-elb"
  subnets = ["${aws_subnet.main-public-1.id}", "${aws_subnet.main-public-2.id}"]
  security_groups = ["${aws_security_group.elb-securitygroup.id}"]
  listener {
    instance_port = 80
    instance_protocol = "http"
    lb_port = 80
    lb_protocol = "http"
  }
  health_check {
    healthy_threshold = 2
    unhealthy_threshold = 2
    timeout = 3
    target = "HTTP:80/"
    interval = 30
  }

  instances = ["${aws_instance.example-instance.id}"] # optional, you can also attach an ELB to an autoscaling group
  cross_zone_load_balancing = true
  connection_draining = true
  connection_draining_timeout = 400
  tags {
    Name = "my-elb"
  }
}
```

ELB + autoscaling

- You can attach the ELB to an autoscaling group:

```
resource "aws_launch_configuration" "example-launchconfig" {
  name_prefix      = "example-launchconfig"
  image_id         = "${lookup(var.AMIS, var.AWS_REGION)}"
  instance_type    = "t2.micro"
  key_name         = "${aws_key_pair.mykeypair.key_name}"
  security_groups  = ["${aws_security_group.allow-ssh.id}"]
}

resource "aws_autoscaling_group" "example-autoscaling" {
  name                = "example-autoscaling"
  vpc_zone_identifier = ["${aws_subnet.main-public-1.id}", "${aws_subnet.main-public-2.id}"]
  launch_configuration = "${aws_launch_configuration.example-launchconfig.name}"
  min_size            = 1
  max_size            = 2
  health_check_grace_period = 300
  health_check_type = "ELB"
  force_delete        = true
  load_balancers = ["${aws_elb.my-elb.name}"]
  tag {
    key = "Name"
    value = "ec2 instance"
    propagate_at_launch = true
  }
}
```

Demo

Autoscaling group with load balancer

Application Load Balancer

Rule based load balancing

Application Load Balancer

- For an application load balancer, you first define the general settings:

```
resource "aws_alb" "my-alb" {  
  name = "my-alb"  
  subnets = ["${aws_subnet.main-public-1.id}", "${aws_subnet.main-public-2.id}"]  
  security_groups = ["${aws_security_group.elb-securitygroup.id}"]  
  
  Name = "my-alb"  
}  
}
```

Application Load Balancer

- Then, you specify a target group:

```
resource "aws_alb_target_group" "frontend-target-group" {  
  name      = "alb-target-group"  
  port      = 80  
  protocol  = "HTTP"  
  vpc_id    = "${aws_vpc.main.id}"  
}
```

- You can attach instances to targets:

```
resource "aws_alb_target_group_attachment" "frontend-attachment-1" {  
  target_group_arn = "${aws_alb_target_group.frontend-target-group.arn}"  
  target_id        = "${aws_instance.example-instance.id}"  
  port             = 80  
}  
  
resource "aws_alb_target_group_attachment" "frontend-attachment-2" {  
  [...]   
}
```

Application Load Balancer

- You also need to specify the listeners separately:

```
resource "aws_alb_listener" "frontend-listeners" {  
  load_balancer_arn = "${aws_alb.my-alb.arn}"  
  port = "80"  
  
  default_action {  
    target_group_arn = "${aws_alb_target_group.frontend-target-group.arn}"  
    type = "forward"  
  }  
}
```

- The default action matches always if you haven't specified any other rules

Application Load Balancer

- With ALBs, you can specify multiple rules to send traffic to another target:

```
resource "aws_alb_listener_rule" "alb-rule" {
  listener_arn = "${aws_alb_listener.front_end.arn}"
  priority = 100

  action {
    type = "forward"
    target_group_arn = "${aws_alb_target_group.new-target-group.arn}"
  }

  condition {
    field = "path-pattern"
    values = ["/static/*"]
  }
}

resource "aws_alb_target_group" "new-target-group" {
  [...]
}

resource "aws_alb_target_group_attachment" "new-target-group-attachment" {
  [...]
  target_id = "${aws_instance.other-instances-than-the-first-one.id}"
  [...]
}
```

Elastic Beanstalk

Elastic Beanstalk

- Elastic Beanstalk is AWS's Platform as a Service (PaaS) solution
- It's a platform where you launch your app on without having to maintain the underlying infrastructure
 - You are still responsible for the EC2 instances, but AWS will provide you with updates you can apply
 - Updates can be applied manually or automatically
 - The EC2 instances run Amazon Linux

Elastic Beanstalk

- Elastic Beanstalk can handle application scaling for you
 - Underlying it uses a Load Balancer and an Autoscaling group to achieve this
 - You can schedule scaling events or enable autoscaling based on a metric
- It's similar to Heroku (another PaaS solution)
- You can have an application running just in a few clicks using the AWS Console
 - Or using the elasticbeanstalk resources in Terraform

Elastic Beanstalk

- The supported Platforms are:
 - PHP
 - Java SE, Java with Tomcat
 - .NET on Windows with IIS
 - Node.js
 - Python
 - Ruby
 - Go
 - Docker (single container + multi-container, using ECS)

Elastic Beanstalk

- When you deploy an Elastic Beanstalk environment you'll get a CNAME (hostname) that you can use as endpoint
- You can use Route53 to point your domain to that CNAME
- Once Elastic Beanstalk is running, you can deploy your application on it using the EB Command Line utility
 - Can be downloaded from:
<http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/eb-cli3.html>
- In the next demo I'll go over the terraform code that is necessary to launch an ElasticBeanstalk environment

Demo

Elastic Beanstalk

Packer

creating AMIs with Packer

Packer

- Packer is a commandline tool that can build AWS AMIs based on templates
- Instead of installing the software after booting up an instance, you can create an AMI with all the necessary software on
- This can speed up boot times of instances
- It's a common approach when you run a horizontally scaled app layer or a cluster of something
- It might not be very useful to do this on single instances that cannot be terminated (e.g. a database)

Packer

- The packer template:

```
{
  "variables": {
    "aws_access_key": "",
    "aws_secret_key": ""
  },
  "builders": [{
    "type": "amazon-ebs",
    "access_key": "{{user `aws_access_key`}}",
    "secret_key": "{{user `aws_secret_key`}}",
    "region": "us-east-1",
    "source_ami": "ami-fce3c696",
    "instance_type": "t2.micro",
    "ssh_username": "ubuntu",
    "ami_name": "packer-example {{timestamp}}"
  ]},
  "provisioners": [{
    "type": "shell",
    "scripts": [ "scripts/install_software.sh" ],
    "execute_command": "{{ .Vars }} sudo -E sh '{{ .Path }}'",
    "pause_before": "10s"
  ]}
}
```

Demo

Terraform with packer

Terraform with Packer

Packer

- Packer is a commandline tool that can build AWS AMIs based on templates
- Instead of installing the software after booting up an instance, you can create an AMI with all the necessary software on
- This can speed up boot times of instances
- It's a common approach when you run a horizontally scaled app layer or a cluster of something
- It might not be very useful to do this on single instances that cannot be terminated (e.g. a database)

Packer

- The packer template:

```
{
  "variables": {
    "aws_access_key": "",
    "aws_secret_key": ""
  },
  "builders": [{
    "type": "amazon-ebs",
    "access_key": "{{user `aws_access_key`}}",
    "secret_key": "{{user `aws_secret_key`}}",
    "region": "us-east-1",
    "source_ami": "ami-fce3c696",
    "instance_type": "t2.micro",
    "ssh_username": "ubuntu",
    "ami_name": "packer-example {{timestamp}}"
  ]},
  "provisioners": [{
    "type": "shell",
    "scripts": [ "scripts/install_software.sh" ],
    "execute_command": "{{ .Vars }} sudo -E sh '{{ .Path }}'",
    "pause_before": "10s"
  ]}
}
```

Demo

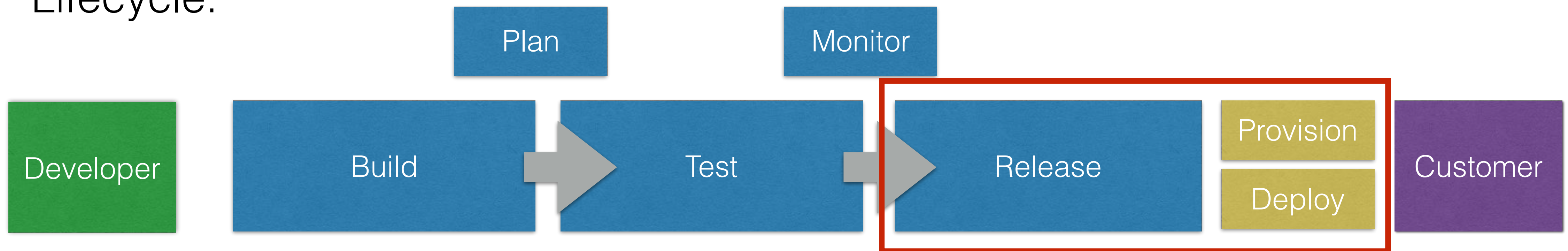
Terraform with packer

DevOps using terraform

Integrating terraform and packer within your workflow

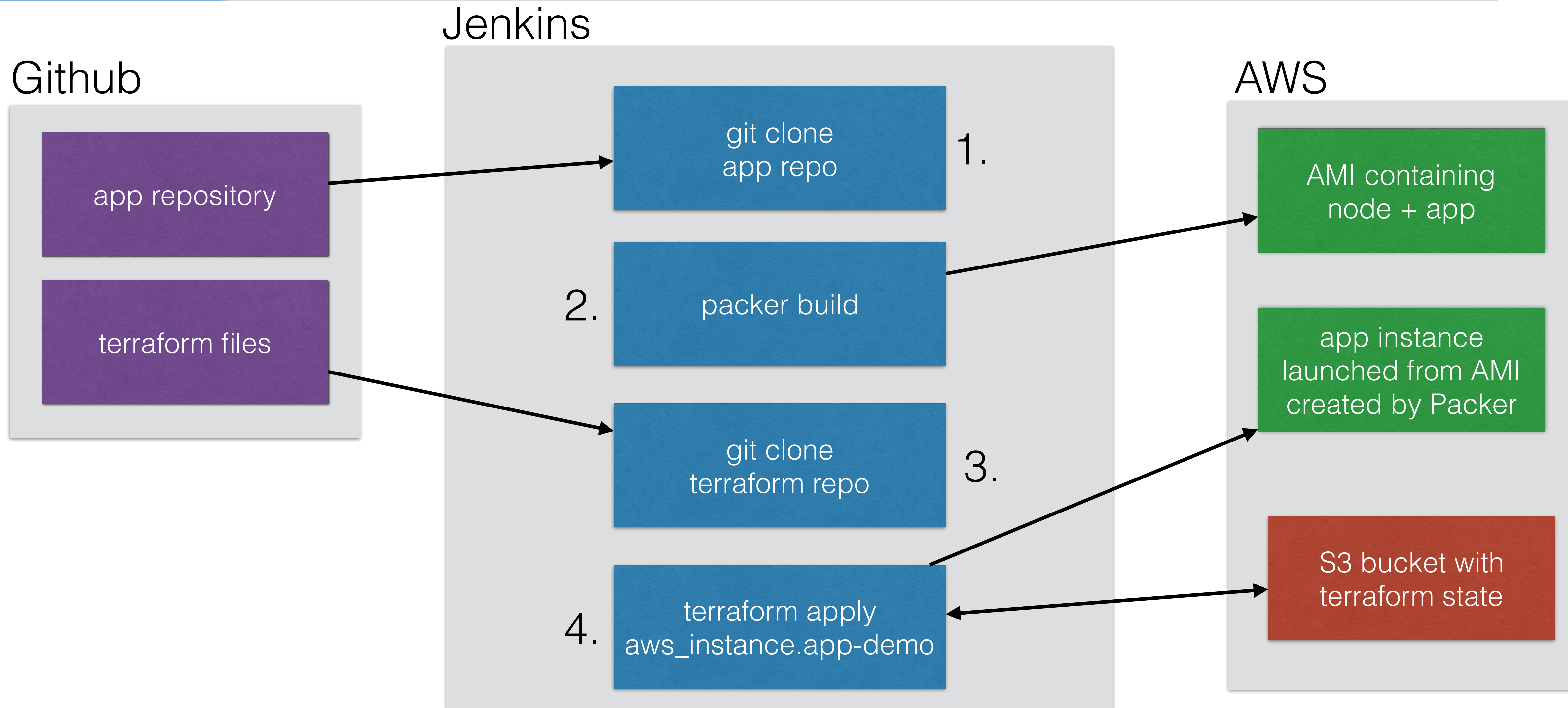
DevOps using terraform

- Terraform is a great fit in a DevOps minded organization
- Tools like Terraform and Packer can be used in the Software Development Lifecycle:



- Release, Provision and Deploy can be done using:
 - Git + Jenkins + Terraform + Packer (Amazon images)
 - Git + Jenkins + Terraform + Docker Orchestration (Docker images)

DevOps using terraform



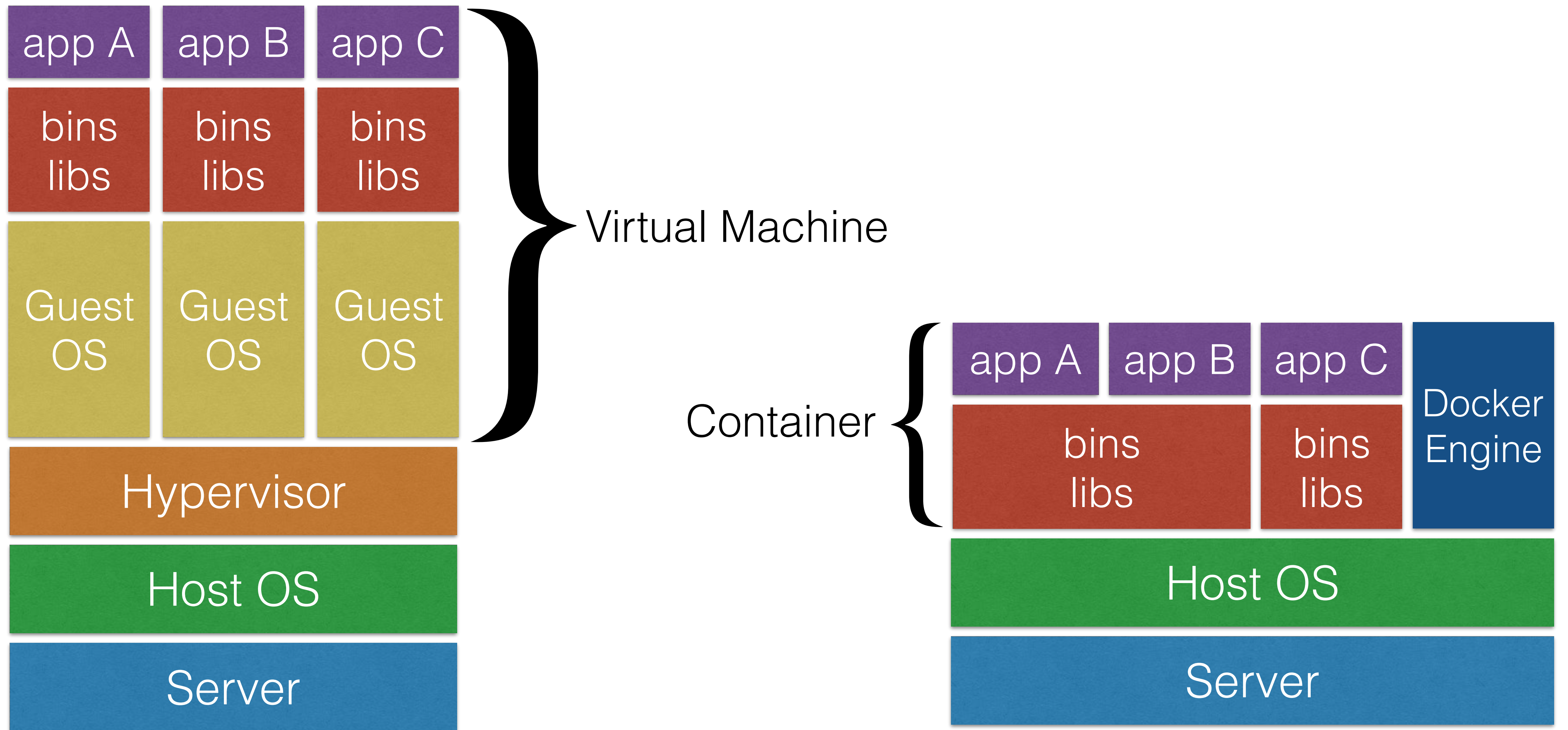
Demo

Terraform with Jenkins and packer

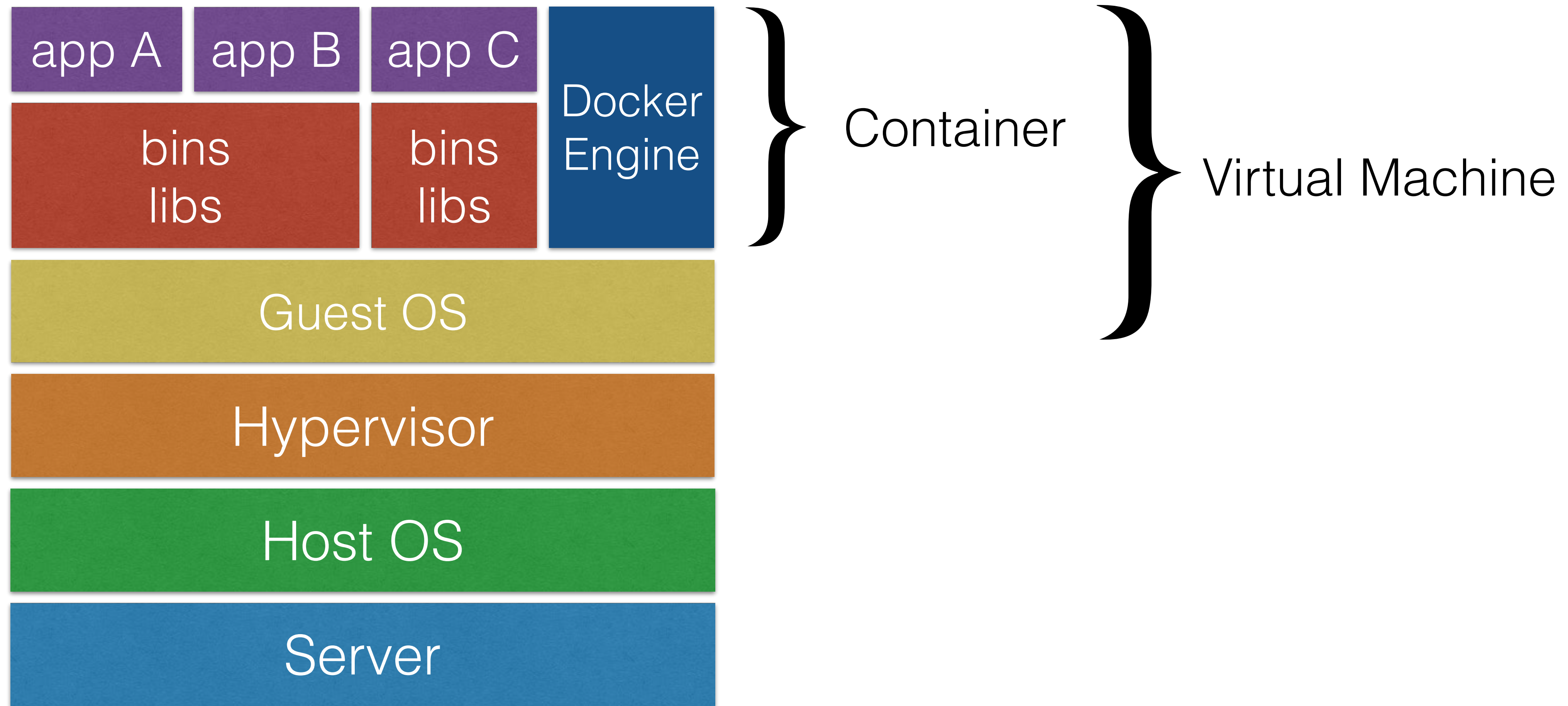
Docker

Introduction

Virtual Machines vs Containers



Containers on AWS

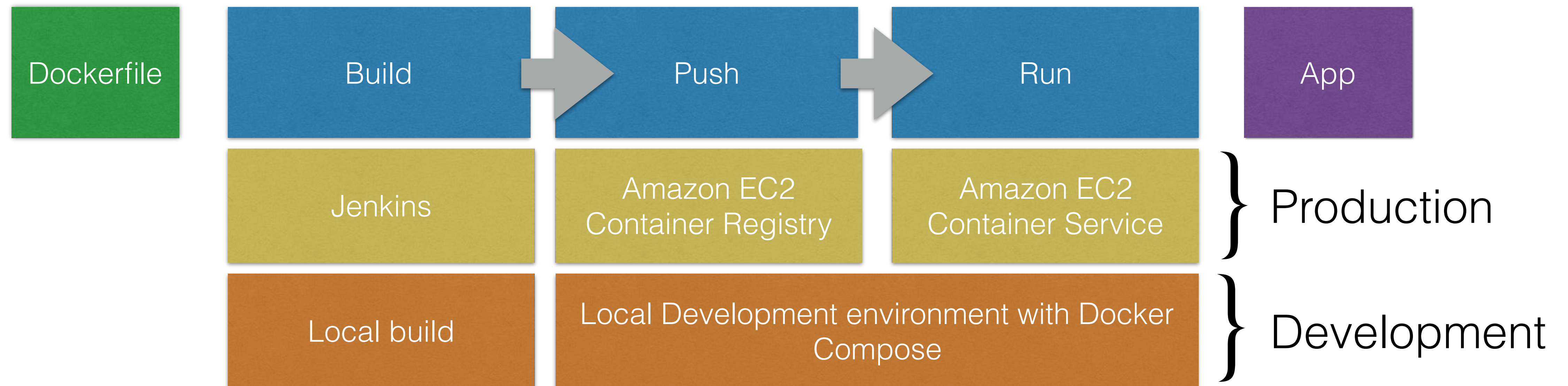


Docker

Docker on AWS

Building Docker images

- Just like packer builds AMIs, you can use docker to build docker images
- Those images can then be run on any Linux host with Docker Engine installed



Running Docker

- Using my vagrant “DevOps Box” (The ubuntu box)
- By downloading Docker for:
 - Windows: <https://docs.docker.com/engine/installation/windows/>
 - MacOS: <https://docs.docker.com/engine/installation/mac/>
 - Linux: <https://docs.docker.com/engine/installation/linux/>
- The demos will be done using Docker Engine installed in the Vagrant DevOps box (<https://github.com/wardviaene/devops-box>)

Docker

Building images

Dockerfile

- Dockerizing a simple nodeJS app only needs a few files:

Dockerfile

```
FROM node:4.6
WORKDIR /app
ADD . /app
RUN npm install
EXPOSE 3000
CMD npm start
```

index.js

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('Hello World!');
});

var server = app.listen(3000, function () {
  var host = server.address().address;
  var port = server.address().port;

  console.log('Example app listening at http://
%s:%s', host, port);
});
```

package.json

```
{
  "name": "myapp",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node index.js"
  },
  "engines": {
    "node": "^4.6.1"
  },
  "dependencies": {
    "express": "^4.14.0",
  }
}
```

Dockerfile

- To build this project, docker build can be used
- Docker build can be executed manually or by jenkins
- To build the docker image from the previous slide:

```
$ cd docker-demo  
$ ls  
Dockerfile  index.js  package.json  
$ docker build .  
[...]  
$
```

- After the docker build process you have built an image that can run the nodejs app

Dockerfile

- You could already run the docker app by executing “docker run” locally
 - Docker can be run locally for development purposes
- Instead, we’re immediately going to push this image to Amazon and run this app on AWS
 - The first step will be to **push** this locally built images to **Amazon ECR** (The EC2 Registry - where docker images can be stored in)
 - Secondly, we’ll set up a **docker cluster (ECS)** to run our Docker applications

Dockerfile

- The creation of the ECR repository can be done using terraform:

ecr.tf

```
resource "aws_ecr_repository" "myapp" {  
  name = "myapp"  
}
```

output.tf

```
output "myapp-repository-URL" {  
  value = "${aws_ecr_repository.myapp.repository_url}"  
}
```

Docker build & push command

```
$ docker build -t myapp-repository-url/myapp .  
$ `aws ecr get-login`  
$ docker push myapp-repository-url/myapp
```

Demo

Build docker image and push image to ECR

ECS

Running our dockerized NodeJS app on a Docker cluster

ECS

- Now that your app is dockerized and uploaded to ECR, you can start the ECS cluster
- ECS - EC2 Container Services will manage your docker containers
- You just need to start an autoscaling group with a custom AMI
 - The custom AMI contains the ECS agent
- Once the ECS cluster is online, tasks and services can be started on the cluster

ECS

- First, the ECS cluster needs to be defined:

```
resource "aws_ecs_cluster" "example-cluster" {  
  name = "example-cluster"  
}
```

ECS

- Then, an autoscaling group launches EC2 instances that will join this cluster:

```
resource "aws_launch_configuration" "ecs-example-launchconfig" {
  name_prefix      = "ecs-launchconfig"
  image_id         = "${lookup(var.ECS_AMIS, var.AWS_REGION)}"
  instance_type    = "${var.ECS_INSTANCE_TYPE}"
  key_name         = "${aws_key_pair.mykeypair.key_name}"
  iam_instance_profile = "${aws_iam_instance_profile.ecs-ec2-role.id}"
  security_groups  = ["${aws_security_group.ecs-securitygroup.id}"]
  user_data       = "#!/bin/bash\nnecho 'ECS_CLUSTER=example-cluster' > /etc/ecs/ecs.config\nstart ecs"
  lifecycle {
    create_before_destroy = true
  }
}
resource "aws_autoscaling_group" "ecs-example-autoscaling" {
  name                = "ecs-example-autoscaling"
  vpc_zone_identifier = ["${aws_subnet.main-public-1.id}", "${aws_subnet.main-public-2.id}"]
  launch_configuration = "${aws_launch_configuration.ecs-example-launchconfig.name}"
  min_size            = 1
  max_size            = 1
  tag {
    key = "Name"
    value = "ecs-ec2-container"
    propagate_at_launch = true
  }
}
```

- A list of ECS AMIs can be found at http://docs.aws.amazon.com/AmazonECS/latest/developerguide/ecs-optimized_AMI.html

ECS - Roles

- The iam role policy (aws_iam_role_policy.ecs-ec2-role-policy)

```
"Statement": [  
  {  
    "Effect": "Allow",  
    "Action": [  
      "ecs:CreateCluster",  
      "ecs:DeregisterContainerInstance",  
      "ecs:DiscoverPollEndpoint",  
      "ecs:Poll",  
      "ecs:RegisterContainerInstance",  
      "ecs:StartTelemetrySession",  
      "ecs:Submit*",  
      "ecs:StartTask",  
      "ecr:GetAuthorizationToken",  
      "ecr:BatchCheckLayerAvailability",  
      "ecr:GetDownloadUrlForLayer",  
      "ecr:BatchGetImage",  
      "logs:CreateLogStream",  
      "logs:PutLogEvents"  
    ],  
    "Resource": "*"   
  }  
]
```

ECS - Task definition

- Before the docker app can be launched, a task definition needs to be provided
- The task definition describes what docker container to be run on the cluster:
 - Specifies Docker image (the docker image in ECR)
 - Max CPU usage, max memory usage
 - Whether containers should be linked (e.g. link app container with DB container)
 - Environment variables (e.g. credentials)
 - And any other container specific definitions

ECS - Services definition

- A Service definition is going to run a specific amount of containers based on the task definition
- A Service is always running, if the container stops, it will be restarted
- A Service can be scaled, you can run 1 instance of a container or multiple
- You can put an Elastic Load Balancer in front of a service
- You typically run multiple instances of a container, spread over Availability zones
 - If one container fails, your loadbalancer stops sending traffic to it
 - Running multiple instances with an ELB / ALB allows you to have HA

ECS - Services definition

- The task definition:

myapp.tf

```
data "template_file" "myapp-task-definition-template" {
  template      = "${file("templates/app.json.tpl")}"
  vars {
    REPOSITORY_URL = "${replace("${aws_ecr_repository.myapp.repository_url}", "https://", "")}"
  }
}

resource "aws_ecs_task_definition" "myapp-task-definition" {
  family      = "myapp"
  container_definitions = "${data.template_file.myapp-task-definition-template.rendered}"
}
```

templates/app.json.tpl

```
[
  {
    "essential": true,
    "memory": 256,
    "name": "myapp",
    "cpu": 256,
    "image": "${REPOSITORY_URL}:1",
    "workingDirectory": "/app",
    "command": ["npm", "start"],
    "portMappings": [
      {
        "containerPort": 3000,
        "hostPort": 3000
      }
    ]
  }
]
```


ECS - Services definition

- The Service definition:

myapp.tf

```
resource "aws_ecs_service" "myapp-service" {
  name = "myapp"
  cluster = "${aws_ecs_cluster.example-cluster.id}"
  task_definition = "${aws_ecs_task_definition.myapp-task-definition.arn}"
  desired_count = 1
  iam_role = "${aws_iam_role.ecs-service-role.arn}"
  depends_on = ["aws_iam_policy_attachment.ecs-service-attach1"]

  load_balancer {
    elb_name = "${aws_elb.myapp-elb.name}"
    container_name = "myapp"
    container_port = 3000
  }
  lifecycle { ignore_changes = ["task_definition"] }
}
```


Demo

running Docker images on *AWS* ECS

Demo

Using Jenkins to build docker images, push the image on
AWS ECR and run on AWS ECS

Docker

running Docker images

New Features

v0.8.0

Terraform console

- terraform console: an interactive console for experimenting and using interpolations
 - Discover what variables exist
 - How to manipulate variables
 - How to use interpolations, conditions
 - Quickly output something

Version requirement

- Use a version requirement in tf files
- As new features are developed at a rapid rate, your terraform files might become incompatible with newer/older versions
- You can set a terraform `required_version` to make sure the person executing your tf files will get an error if the requirements are not met:

```
terraform {  
  required_version = "> 0.7.0, < 0.8.0"  
}
```

Conditional values

- A major feature introduced in v0.8.0 is conditional values
- You can now include basic if/else statements
- For example, only build a resource based on a condition:

```
resource "aws_instance" "instance" {  
  count = "${var.env == "prod" ? 1 : 0}"  
  [...]  
}
```

- Only if variable “env” is set to production, the aws_instance “instance” will be launched (count will be set to 1)

Improvements to import

- More resources can be imported
- The import command will now read the configuration from terraform configuration files
 - For example the AWS credentials and region information
- If you have multiple providers (e.g. 2 AWS accounts), import can now be passed a provider alias to specify what provider you want to use

depends_on

- depends_on can be used within resources to specify this resource depends on another resource
 - This already existed
- Now you can specify a module within depends_on
 - e.g. a resource can now depend on a module

Congratulations

Congratulations

