Terraform Walk-Through

Our Plan

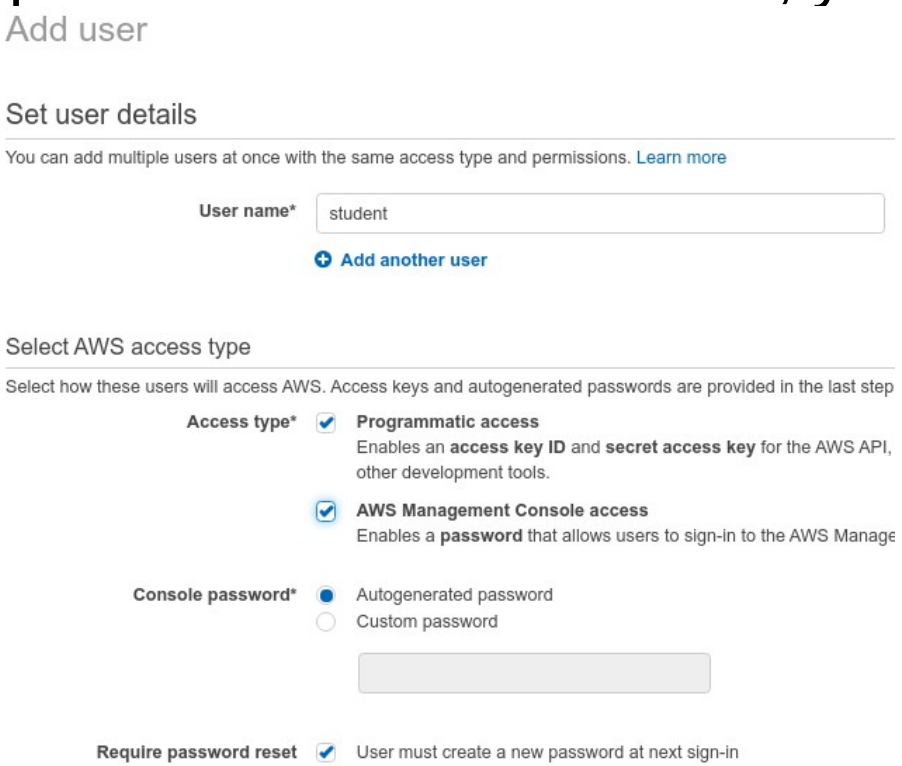
- Getting started with Terraform
  - Setting up your AWS account
  - Installing Terraform
  - Deploying a single server
  - Deploying a single web server
- More advanced configuration
  - Deploying a configurable web server
  - Deploying a cluster of web servers
  - Deploying a load balancer
  - Cleaning up

We Will Use AWS

- The largest market share
  - Although for you it may be different
- Will try to provide examples for other clouds
- All clouds give you a free tier
  - A trick to user after a year:
  - Use kind of address
  - If you already used up your free tier credits, the examples in the labs should still cost you no more than a few dollars.

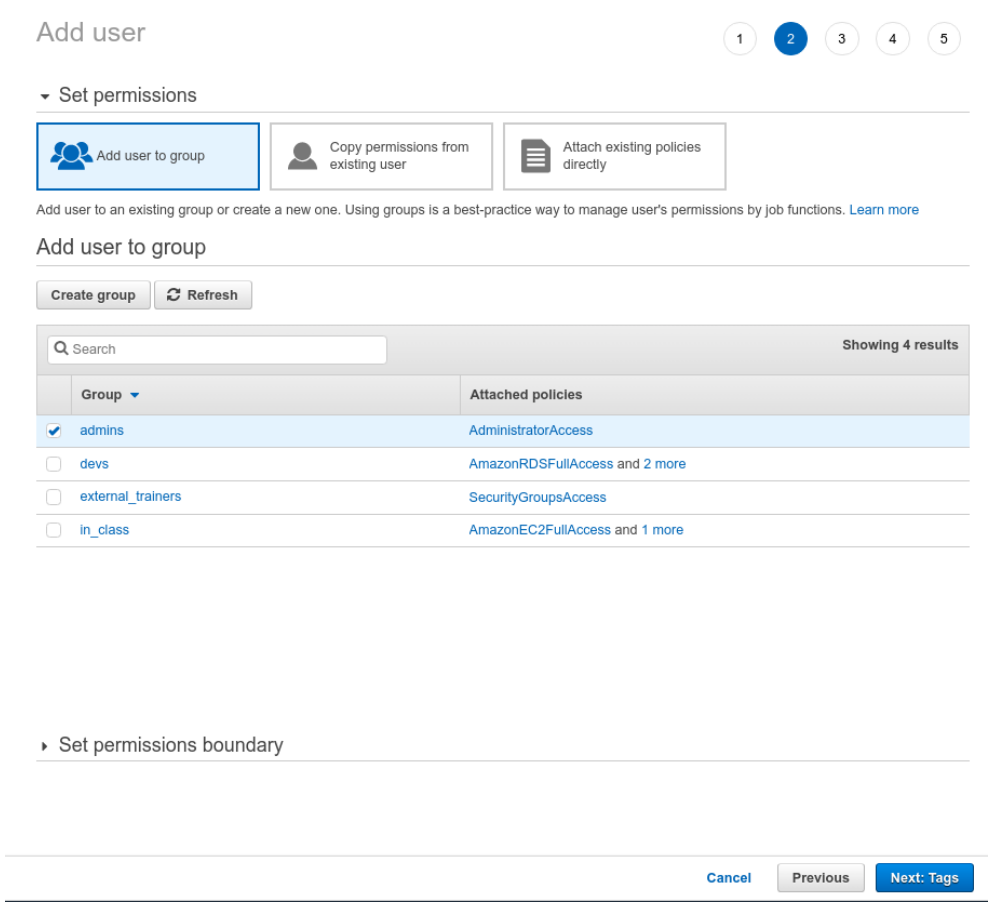
Setting Up Your AWS Account

- If you don't already have an **AWS account** , head over to https://aws.amazon.com and sign up
- The *only* thing you should use the root user for is to create other user accounts with more-limited permissions, and then switch to one of those accounts immediately
- If you are using an existing AWS account, it must have a **Default VPC** in it.
- If the instructor provided a student account, you can use that



Giving AWS Permissions

- Here are the permissions you will need (for some labs)
  - AmazonEC2FullAccess
  - AmazonS3FullAcces
  - AmazonDynamoDBFullAccess
  - AmazonRDSFullAccess
  - CloudWatchFullAccess
  - IAMFullAccess



Install Terraform

- Download the `terraform` executable from the Terraform home page
- Depending on your OS, you may also install a native package
- For Mac
  - `brew install terraform`
- You may use a cloud server if provided by the instructor

Verify

- Ubuntu example
- Terraform v0.12.20
- Your version of Terraform is out of date! The latest version is 0.12.24. You can update by downloading from https://www.terraform.io/downloads.html
- OK... Update here

Now What?

```
(base) mark@mark-workstation:~$ terraform
Usage: terraform [-version] [-help] <command> [args]

The available commands for execution are listed below.
The most common, useful commands are shown first, followed by
less common or more advanced commands. If you're just getting
started with Terraform, stick with the common commands. For the
other commands, please read the help and docs before usage.

Common commands:
    apply              Builds or changes infrastructure
    console            Interactive console for Terraform interpolations
    destroy            Destroy Terraform-managed infrastructure
    env                Workspace management
    fmt                Rewrites config files to canonical format
    get                Download and install modules for the configuration
    graph              Create a visual graph of Terraform resources
    import             Import existing infrastructure into Terraform
    init               Initialize a Terraform working directory
    login              Obtain and save credentials for a remote host
    logout             Remove locally-stored credentials for a remote host
    output             Read an output from a state file
    plan               Generate and show an execution plan
    providers          Prints a tree of the providers used in the configuration
    refresh            Update local state file against real resources
    show               Inspect Terraform state or plan
    taint              Manually mark a resource for recreation
    untaint            Manually unmark a resource as tainted
    validate           Validates the Terraform files
    version            Prints the Terraform version
    workspace          Workspace management
```

Connect to AWS - Way 1

- Way 1: Set up AWS credentials
- `$export AWS_ACCESS_KEY_ID=(you access key id)`
- `$export AWS_SECRET_ACCESS_KEY=(your secret access key)`
- This will only give you the setup for this shell
- To make it work after reboot put it into `.bashrc`
- `vi ~/.bashrc`

Connect to AWS - Way 2

- Way 2
- Terraform supports the same authentication mechanisms as all AWS CLI and SDK tools
- Therefore, it'll also be able to use credentials in `$HOME/.aws/credentials`
  - These are automatically generated if you run the `configure` command on the AWS CLI, or IAM

Let Us Prepare to Deploy a Server

- Create an empty folder, `lab01`
- Put in a file called `main.tf`

- Now you will be adding resources, like this:

```
resource "<PROVIDER>_<TYPE>" "<NAME>" {
   [CONFIG ...]
}
```

Server Resource

- ami
  - The Amazon Machine Image (AMI) to run on the EC2 Instance.
  - You can find free and paid AMIs in the AWS Marketplace
  - or create your own using tools such as Packer
  - This ami parameter to the ID of an Ubuntu 18.04 AMI in us-east- This AMI is free to use
- instance_type
  - The type of EC2 Instance to run
  - Each type of EC2 Instance provides a different amount of CPU, memory, disk space, and networking capacity.
  - The EC2 Instance Types page lists all the available options
  - t2.micro, which has one virtual CPU, 1 GB of memory, and is part of the AWS free tier

Do Not Try to Remember by Heart

- Terraform supports dozens of providers
- Each of which supports dozens of resources
- Each resource has dozens of arguments
- We recommend using the documentation.
  - Here is an example for ami
- Now run `terraform init`

Result of

```
(base) mark@mark-workstation:~/projects/ES/terraform-class/lab01$ terraform init

Initializing the backend...

Initializing provider plugins...
- Checking for available provider plugins...
- Downloading plugin for provider "aws" (hashicorp/aws) 2.60.0...

The following providers do not have any version constraints in configuration,
so the latest version was installed.

To prevent automatic upgrades to new major versions that may contain breaking
changes, it is recommended to add version = "..." constraints to the
corresponding provider blocks in configuration, with the constraint strings
suggested below.

* provider.aws: version = "~> 2.60"

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 working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Result of "terraform plan"

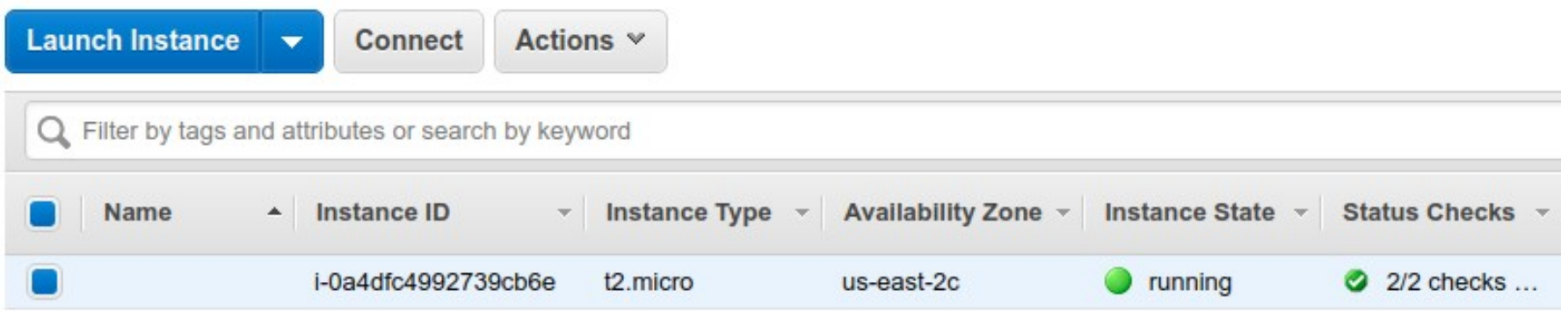
```
$ terraform plan Refreshing
Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.
------------------------------------------------------------------------
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.example will be created
  + resource "aws_instance" "example" {
      + ami                          = "ami-0c55b159cbfafe1f0"
Plan: 1 to add, 0 to change, 0 to destroy.

------------------------------------------------------------------------

Note: You didn't specify an "-out" parameter to save this plan, so Terraform
can't guarantee that exactly these actions will be performed if
"terraform apply" is subsequently run.
```

Result of "terraform apply"

```
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.example will be created
  + resource "aws_instance" "example" {
      + ami                          = "ami-0c55b159cbfafe1f0"
...
}

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

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

aws_instance.example: Creating...
aws_instance.example: Still creating... [10s elapsed]
aws_instance.example: Still creating... [20s elapsed]
aws_instance.example: Creation complete after 24s [id=i-0a4dfc4992739cb6e]

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

Verify the Deployment Result

- Go to AWS dashboard
- Verify that the server was indeed created



Next Step

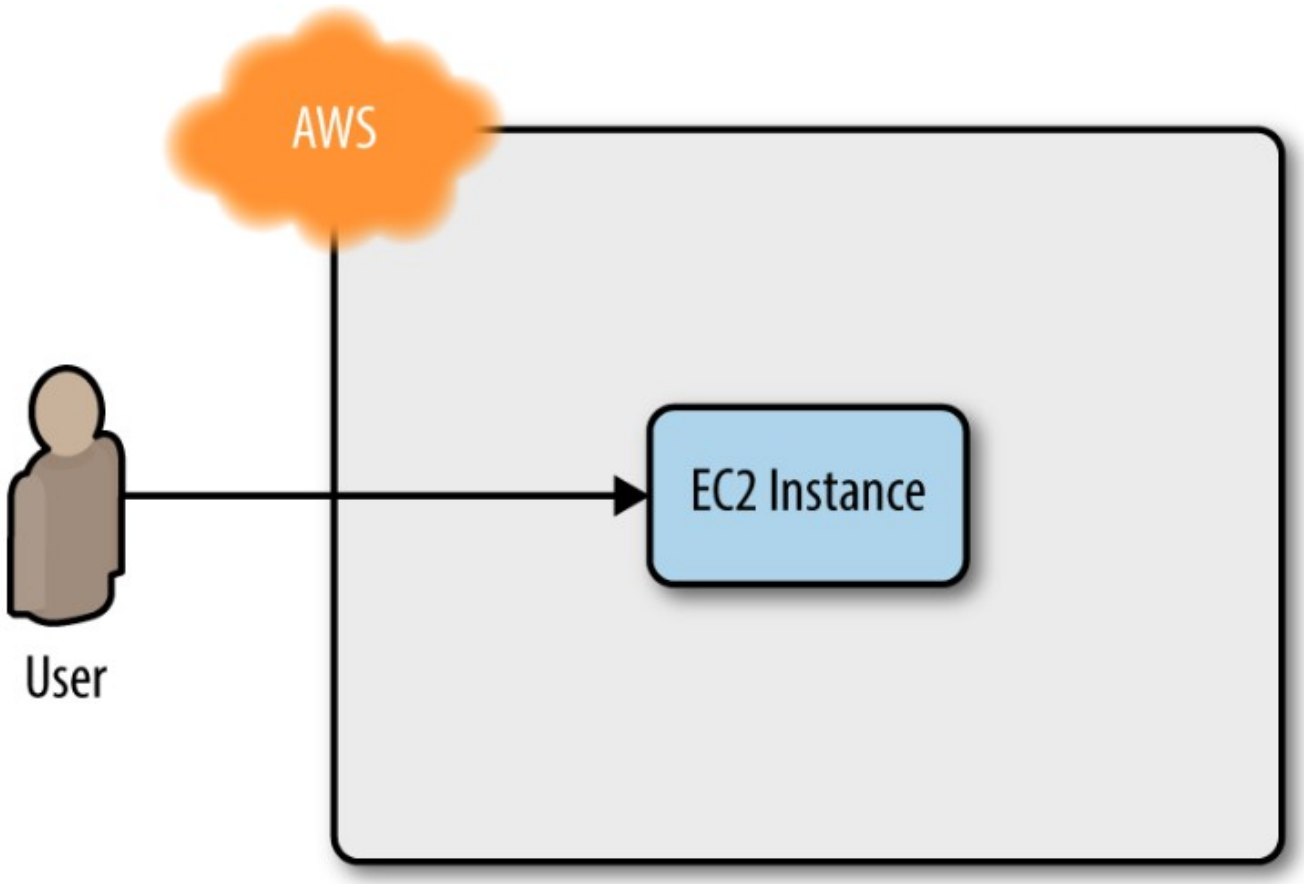
- Let's give our server a name tag
- Add to your `main.tf`
  - (remove the previous server definition)

```
resource "aws_instance" "example" {
  ami           = "ami-0c55b159cbfafe1f0"
  instance_type = "t2.micro"

  tags = {
    Name = "terraform-example"
  }
}
```

- Run `terraform apply`
- Verify that we gave our server a name

```
Plan: 2 to add, 0 to change, 1 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

aws_instance.example: Destroying... [id=i-0a4dfc4992739cb6e]
aws_security_group.instance: Creating...
aws_security_group.instance: Creation complete after 2s [id=sg-02b2ec7c6f419809a]
aws_instance.example: Still destroying... [id=i-0a4dfc4992739cb6e, 10s elapsed]
aws_instance.example: Still destroying... [id=i-0a4dfc4992739cb6e, 20s elapsed]
aws_instance.example: Still destroying... [id=i-0a4dfc4992739cb6e, 30s elapsed]
aws_instance.example: Destruction complete after 31s
aws_instance.example: Creating...
aws_instance.example: Still creating... [10s elapsed]
aws_instance.example: Still creating... [20s elapsed]
aws_instance.example: Creation complete after 23s [id=i-01f311884533ef669]

Apply complete! Resources: 2 added, 0 changed, 1 destroyed.
```

Deploy a Single Web Server



Make a Web Server

- In the real world, you would build a real web server such as Flask
- We will, instead, do a one-command

```
1 #!/bin/bash
2 echo "Hello, World" > index.html
3 nohup busybox httpd -f -p 8080 &
```

- However, how should we put it into our instance?
- We will add it to the aws_instance, as *User Data* configuration

Adding a Script to the Instance

- You pass a shell script to User Data by setting the user_data argument in your Terraform code as follows:
- The `<<-EOF` and `EOF` are Terraform's `heredoc` syntax, which allows you to create multiline strings without having to insert newline characters all over the place

```
1 user_data = <<-EOF
2               #!/bin/bash
3               echo "Hello, World" > index.html
4               nohup busybox httpd -f -p 8080 &
5               EOF
```

Wait! One More Thing!

- By default, AWS does not allow any incoming or outgoing traffic from an EC2 Instance.
- To allow the EC2 Instance to receive traffic on port 8080, you need to create a security group:
- Creates a new resource called `aws_security_group`

```
resource "aws_security_group" "instance" {
  name = "terraform-example-instance"

  ingress {
    from_port   = 8080
    to_port     = 8080
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

CIDR Blocks

- The `ingress` in this group allows incoming TCP requests
  - on port 8080 from the CIDR block 0.0.0.0/0
- CIDR blocks are a concise way to specify IP address ranges
- For example
  - a CIDR block of 10.0.0.0/24
  - represents all IP addresses between 10.0.0.0 and 10.0.0.255
- The CIDR block 0.0.0.0/0 is an IP address range that includes all possible IP addresses, so this security group allows incoming requests on port 8080 from any IP

Passing the Security Group ID

- You also need to tell the EC2 instance to actually use the security group by passing the group's ID
- It goes into the `vpc_security_group_ids` argument of the `aws_instance resource.`
- This is done with Terraform expressions

Terraform "expressions"

- An expression in Terraform is anything that returns a value
- The simplest type of expressions are literals
  - strings: "ami-0c55b159cbfafe1f0"
  - numbers: 7
- Here we need an expression which is a reference

```
<PROVIDER>_<TYPE>.<NAME>.<ATTRIBUTE>

In our case

aws_security_group.instance.id
```

Altogether

```
resource "aws_instance" "example" {
  ami                    = "ami-0c55b159cbfafe1f0"
  instance_type          = "t2.micro"
  vpc_security_group_ids = [aws_security_group.instance.id]

  user_data = <<-EOF
              #!/bin/bash
                  echo "Hello, World" > index.html
                  nohup busybox httpd -f -p 8080 &
                  EOF

  tags = {
    Name = "terraform-example"
  }
}
```

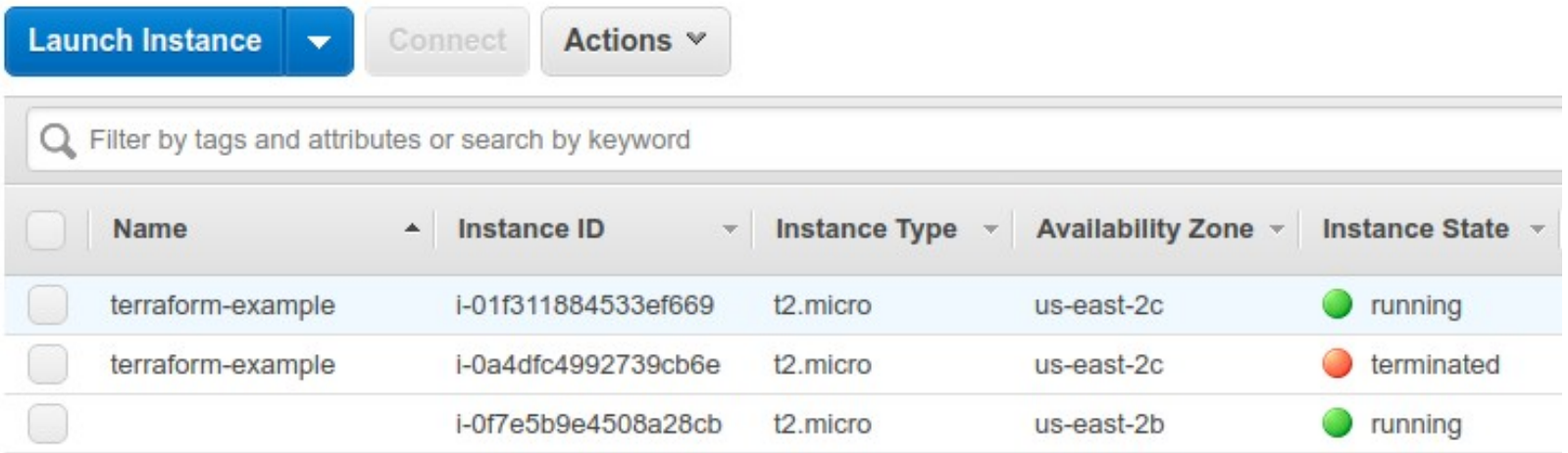
New Result of "terraform plan"

```
Refreshing Terraform state in-memory prior to plan...
The refreshed state will be used to calculate this plan, but will not be
persisted to local or remote state storage.

aws_instance.example: Refreshing state... [id=i-0a4dfc4992739cb6e]
...
  # aws_security_group.instance will be created
  + resource "aws_security_group" "instance" {
...
Plan: 2 to add, 0 to change, 1 to destroy.
```

New Result of "terraform apply"

```
Plan: 2 to add, 0 to change, 1 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

aws_instance.example: Destroying... [id=i-0a4dfc4992739cb6e]
aws_security_group.instance: Creating...
aws_security_group.instance: Creation complete after 2s [id=sg-02b2ec7c6f419809a]
aws_instance.example: Still destroying... [id=i-0a4dfc4992739cb6e, 10s elapsed]
aws_instance.example: Still destroying... [id=i-0a4dfc4992739cb6e, 20s elapsed]
aws_instance.example: Still destroying... [id=i-0a4dfc4992739cb6e, 30s elapsed]
aws_instance.example: Destruction complete after 31s
aws_instance.example: Creating...
aws_instance.example: Still creating... [10s elapsed]
aws_instance.example: Still creating... [20s elapsed]
aws_instance.example: Creation complete after 23s [id=i-01f311884533ef669]

Apply complete! Resources: 2 added, 0 changed, 1 destroyed.
```

Et Voila!



Test the Deployment

```
$ curl http://<EC2_INSTANCE_PUBLIC_IP>:8080
Hello, World
```

```
$ curl http://18.188.2.30:8080
Hello, World
```

Quiz

- Usernames and passwords referenced in the Terraform code, even as variables, will end up in plain text in the state file.
  - A. True
  - B. False

Quiz

- What happens when you apply Terraform configuration? Choose *TWO* correct answers.
  - A. `terraform plan`
  - B. `terraform state`
  - C. `terraform apply`
  - D. `terraform validate`
  - E. `terraform output`

Terraform Dependencies

- When you add a reference from one resource to another, you create an implicit dependency
- Terraform
  - Parses these dependencies
  - builds a dependency graph from them
  - uses that to automatically determine in which order it should create resources
- To see the dependencies, you use the command

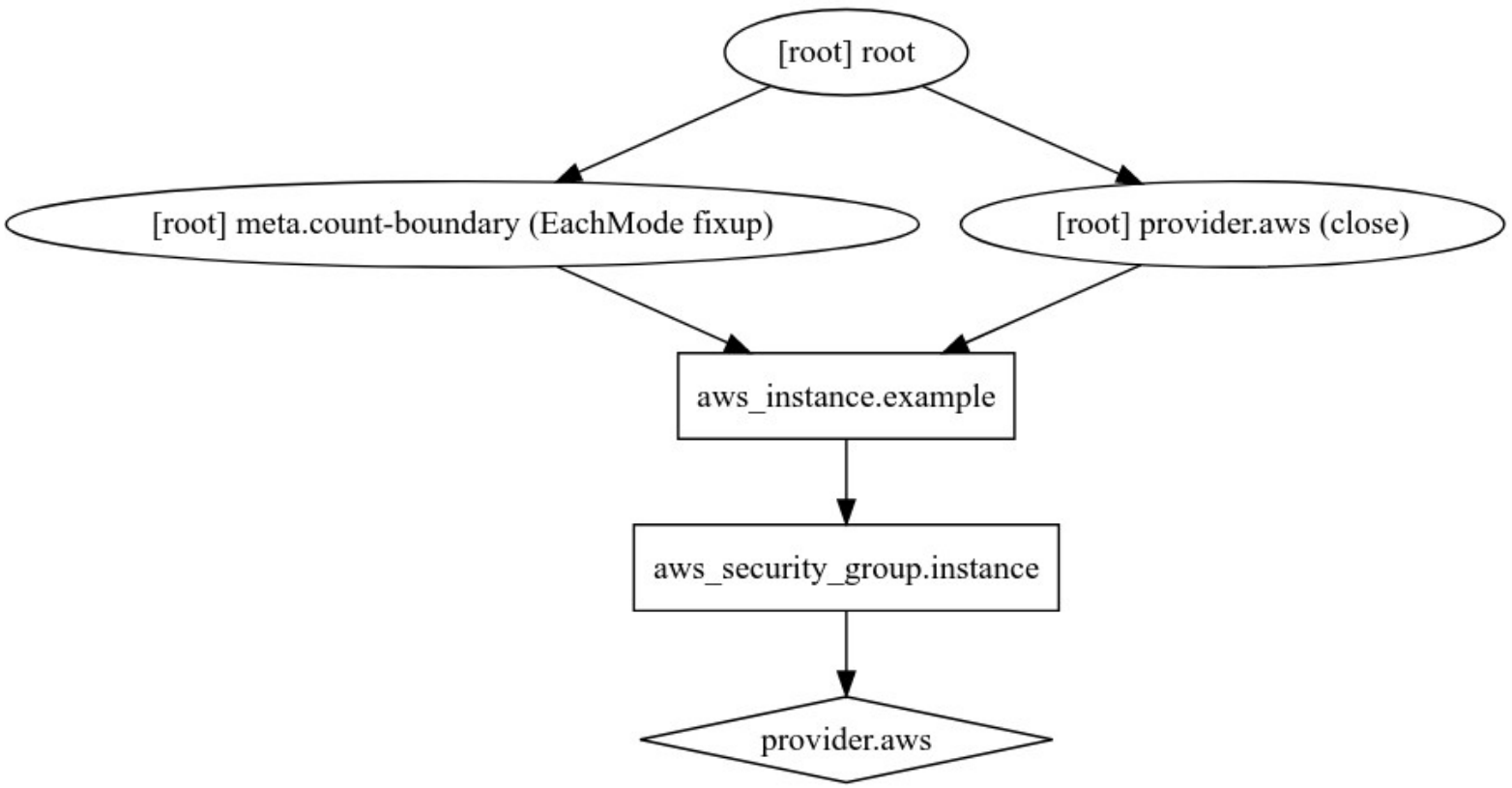
```
1 terraform graph
```

Terraform Graph Output

```
digraph {
    compound = "true"
    newrank = "true"
    subgraph "root" {
        "[root] aws_instance.example" [label = "aws_instance.example", shape = "box"]
        "[root] aws_security_group.instance" [label = "aws_security_group.instance", shape = "box"]
        "[root] provider.aws" [label = "provider.aws", shape = "diamond"]
        "[root] aws_instance.example" -> "[root] aws_security_group.instance"
        "[root] aws_security_group.instance" -> "[root] provider.aws"
        "[root] meta.count-boundary (EachMode fixup)" -> "[root] aws_instance.example"
        "[root] provider.aws (close)" -> "[root] aws_instance.example"
        "[root] root" -> "[root] meta.count-boundary (EachMode fixup)"
        "[root] root" -> "[root] provider.aws (close)"
    }
}
```

Terraform Graph Visual

- Use a desktop app such as Graphviz or



NETWORK SECURITY

- All our example deploy not only into your Default VPC (as mentioned earlier), but also the default subnets of that VPC
- Running a server in a public subnet is fine for a quick experiment, but in real-world usage, it’s a security risk
- For production systems, you should deploy all of your servers, and certainly all of your data stores, in private subnets

Using Variables

Using Variables

Adding Scalability

Deploy a Configurable Web Server

- Don’t Repeat Yourself (DRY) principle
- However, we violated it
  - the web server port 8080 is duplicated in both the security group and the `User Data` configuration
- So, DRY:
  - every piece of knowledge must have a single, unambiguous, authoritative representation within a system

Terraform Input Variables

```
variable "NAME" {
  [CONFIG ...]
}
```

- description
  - It's always a good idea to use this parameter to document how a variable is used
- default, or use these ways:
  - passing it in at the command line (using the -var option)
  - via a file (using the -var-file option)
  - via an environment variable
- type
  - enforce type constraints on the variables a user passes in
  - type constraints: string, number, bool, list, map, set, object, tuple, and any

Examples of Terraform Input Variables

- Input variable that checks that the value you pass in is a number:

```
variable "number_example" {
    description = "An example of a number variable
    type        = number
    default     = 42
  }
```

Examples of Terraform Input Variables

- List input variable with all numbers

```
variable "list_numeric_example" {
    description = "An example of a numeric list
    type        = list(number)
    default     = [1, 2, 3]
  }
```

Examples of Terraform Input Variables

- A map of strings

```
variable "map_example" {
  description = "An example of a map
  type        = map(string)

  default = {
    key1 = "value1"
    key2 = "value2"
    key3 = "value3"
  }
}
```

Quiz

- Consider the following Terraform 0.12 configuration snippet. How would you define the `cidr_block` for us-east-1 in the `aws_vpc` resource using a variable?

```
variable "vpc_cidrs" {
  type = map
  default = {
    us-east-1 = "10.0.0.0/16"
    us-east-2 = "10.1.0.0/16"
    us-west-1 = "10.2.0.0/16"
    us-west-2 = "10.3.0.0/16"
  }
}

resource "aws_vpc" "shared" {
  cidr_block = _______________
}
```

- A. var.vpc_cidrs[“us-east-1”]
- B. var.vpc_cidrs.0
- C. vpc_cidrs[“us-east-1”]
- D.var.vpc_cidrs[0]

Quiz

- You have defined the values for your variables in the file terraform.tfvars, and saved it in the same directory as your Terraform configuration. Which of the following commands will use those values when creating an execution plan?
  - A. `terraform plan`
  - B. `terraform plan -var-file=terraform.tfvars`
  - C. All of the above
  - D. None of the above

OO Coding with Terraform!

```
variable "object_example" {
  description = "An example of a structural type"
  type        = object({
    name    = string
    age     = number
    tags    = list(string)
    enabled = bool
  })

  default = {
    name    = "value1"
    age     = 42
    tags    = ["a", "b", "c"]
    enabled = true
  }
}
```

"server_port" Variable

```
variable "server_port" {
  description = "Server port for HTTP requests"
  type        = number
}
```

Using "server_port" Variable

- If you run `terraform apply`, you will get this message:

```
1 var.server_port
2   "Server port for HTTP requests"
3   Enter a value:
```

- Your choices now are:
  - Enter a value :)
  - terraform plan -var "server_port=8080"
  - export TF_VAR_server_port=8080
  - Supply a default

How to Use Your Variable

- Simply, use `var` , like this: `var.<VARIABLE_NAME>`
- For example

```
resource "aws_security_group" "instance" {
  name = "terraform-example-instance"

  ingress {
    from_port   = var.server_port
    to_port     = var.server_port
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

New Expression - Interpolation

- `"${...}"`
- Now, let us use the same `server_port` inside of `User Data`

```
1 user_data = <<-EOF
2   #!/bin/bash
3   echo "Hello, World" > index.html
4    nohup busybox httpd -f -p ${var.server_port} &
5    EOF
```

Setting an "output" variable

- Additional variables
- description
  - It is always a good idea to document
- sensitive
  - true will instruct Terraform not to log this output at the end of terraform apply
  - For sensitive material or secrets such as passwords or private keys

```
output "<NAME>" {
  value = <VALUE>
  [CONFIG ...]
}
```

Output Variable For Our Script

```
output "public_ip" {
  value       = aws_instance.example.public_ip
  description = "The public IP address of the web server"
}
```

Adding Scalability

Using Variables

Adding Scalability

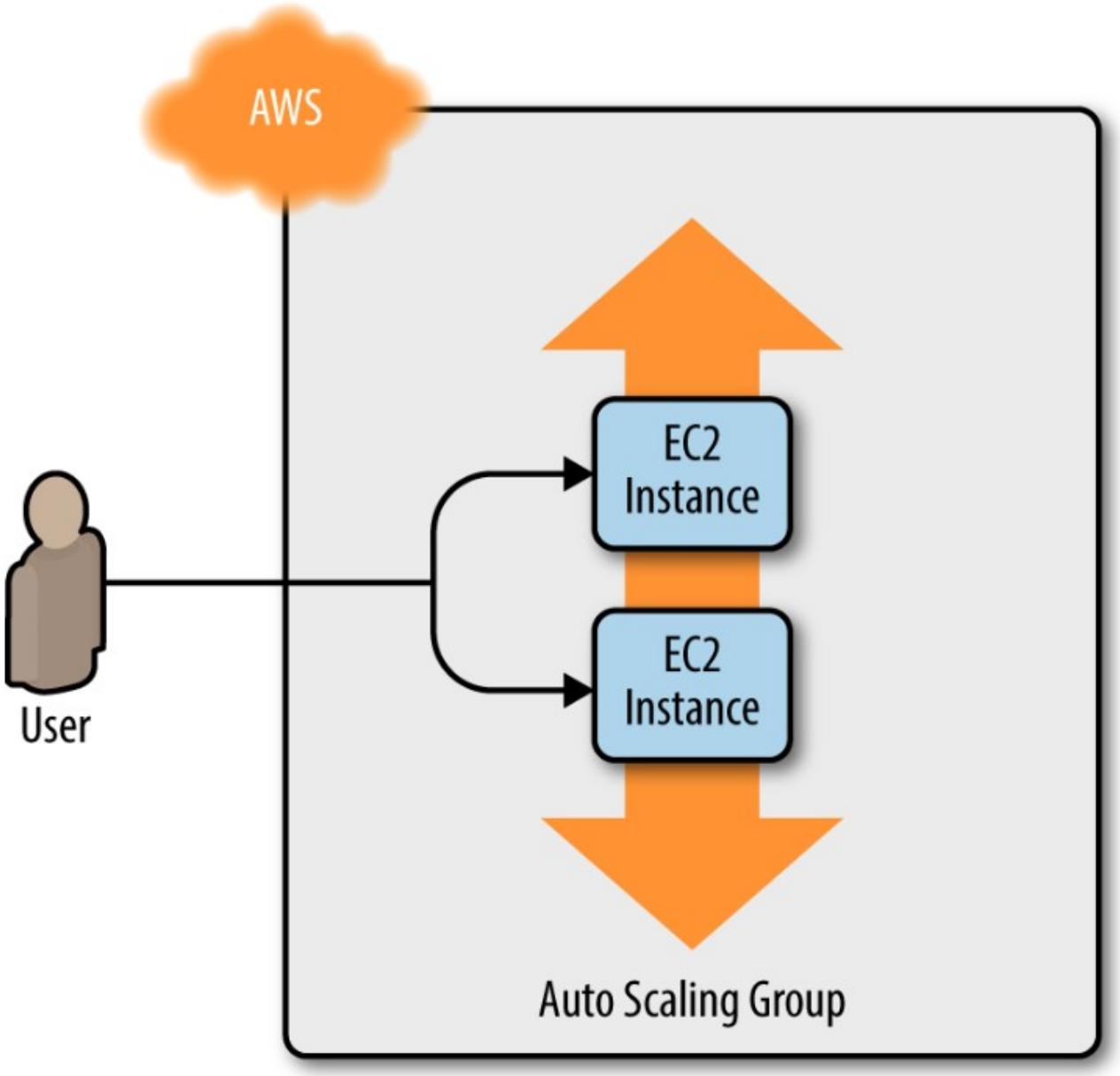
Motivation for a Cluster

- Running a single server is a good start, but in the real world, a single server is a single point of failure
- If that server crashes, or if it becomes overloaded from too much traffic, users will be unable to access your site
- The solution is to run a cluster of servers, routing around servers that go down, and adjusting the size of the cluster up or down based on traffic
- We will need the following
  - Auto-scaling group (ASG)
  - VPC
  - Load balancer

Discussion

- When clusters makes sense and when it is not worth it?
- Imagine a parallel research app for a cluster that would be rearchitected for Cloud
  - would have instances
  - a task dispatcher, etc.
  - would have to manage those instances, etc.
- Alternatives
  - Very large instance with 100+ CPUs
  - Memory pool from Kove

Auto-Scaling Group (ASG)



ASG Described in Terraform

- To create an ASG, first describe the instance that goes into it
  - Create a launch configuration
  - The `aws_launch_configuration` resource
    - uses almost exactly the same parameters as the `aws_instance` `resource`
    - ami is now image_id
    - vpc_security_group_ids is now security_groups
    - put this instead of

```
resource "aws_launch_configuration" "example" {
  image_id        = "ami-0c55b159cbfafe1f0"
  instance_type   = "t2.micro"
  security_groups = [aws_security_group.instance.id]

  user_data = <<-EOF
  #!/bin/bash
      echo "Hello, World" > index.html
      nohup busybox httpd -f -p ${var.server_port} &
      EOF
}
```

The ASG Itself with "aws_autoscaling_group"

- ASG will run between 2 and 10 instances
- each tagged with the name `terraform-asg-example`
- ASG uses a reference to fill in the launch configuration `name`

```
resource "aws_autoscaling_group" "example" {
  launch_configuration =
    aws_launch_configuration.example.name

  min_size = 2
  max_size = 10

  tag {
    key                 = "Name"
    value               = "terraform-asg-example"
    propagate_at_launch = true
  }
}
```

"Lifecycle" Setting

- Use `create_before_destroy`
- If you set `create_before_destroy` to true
  - Terraform will invert the order in which it replaces resources
  - create the replacement resource first
  - then deleting the old resource

```
resource "aws_launch_configuration" "example" {
  image_id        = "ami-0c55b159cbfafe1f0"
  instance_type   = "t2.micro"
  security_groups = [aws_security_group.instance.id]

  user_data = <<-EOF
      #!/bin/bash
      echo "Hello, World" > index.html
      nohup busybox httpd -f -p ${var.server_port} &
      EOF
  # Required when launching configuration with an ASG
  lifecycle {
    create_before_destroy = true
  }
}
```

One More Parameter: "subnet_ids"

- specifies to the ASG into which VPC subnets the EC2 Instances should be deployed
- Each subnet lives in an isolated AWS AZ
- By deploying your instances across multiple subnets
  - you make it fault-tolerant
- Instead of hard-coding the list of subnet, we will get them `data sources`

Data Sources

- `data source` a piece of read-only information that is fetched from the provider (in this case, AWS) every time you run Terraform
- `data source` in your configurations is a way to query the provider's APIs for data
- AWS data sources include
  - VPC data
  - subnet data
  - AMI IDs
  - IP address ranges
  - more

Data Source Syntax

```
data "<PROVIDER>_<TYPE>" "<NAME>" {
  [CONFIG ...]
}
```

- Example: Do I have the default VPC?

```
data "aws_vpc" "default" {
  default = true
}
```

Getting data for "data source"

- Syntax
- `data.<PROVIDER>_<TYPE>.<NAME>.<ATTRIBUTE>`
- Example
- `data.aws_vpc.default.id`
- With this, you can find out the default subnet id

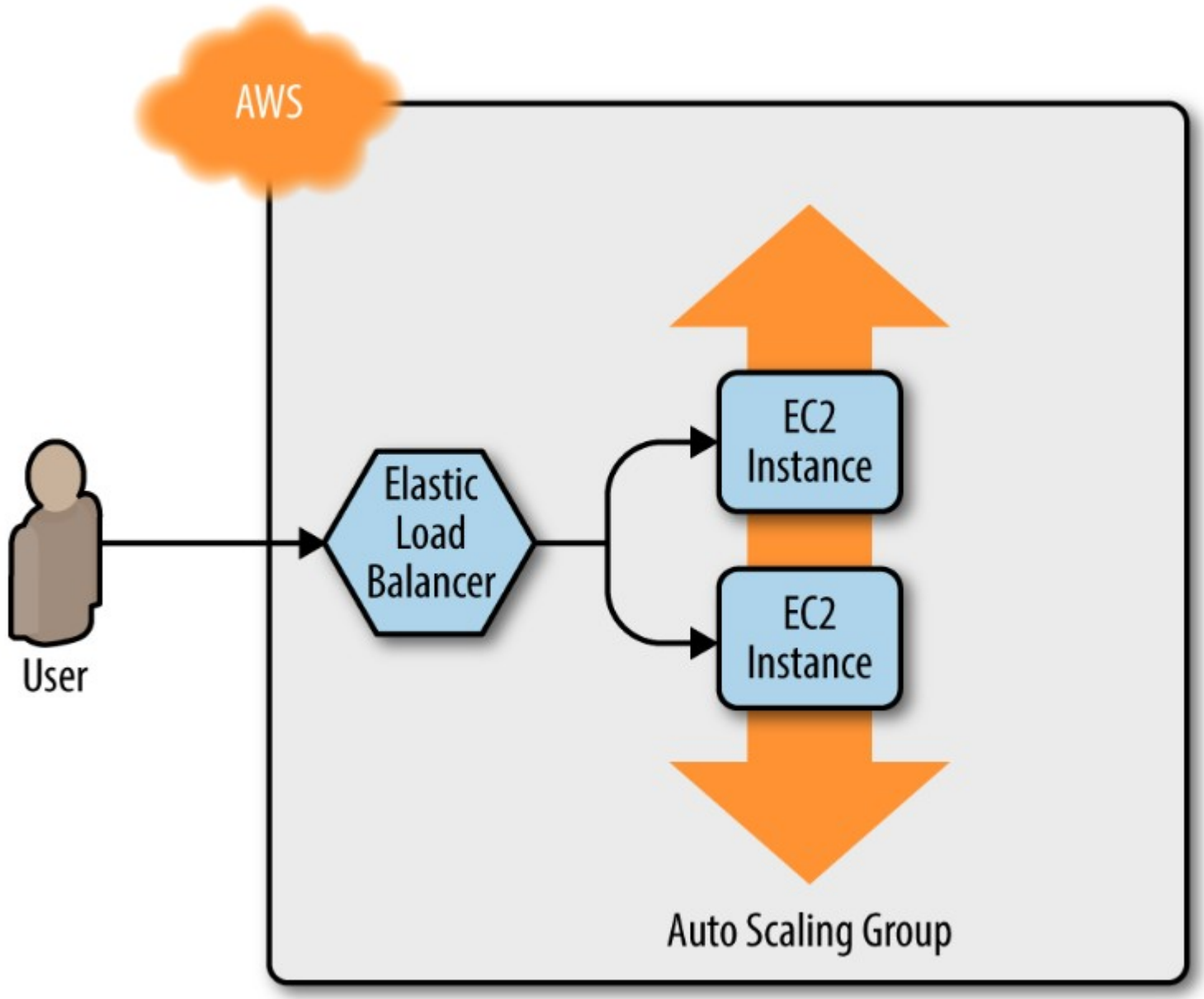
```
1 data "aws_subnet_ids" "default" {
2   vpc_id = data.aws_vpc.default.id
3 }
```

Use the Default Subnet id

- Pull the subnet IDs out of the aws_subnet_ids data source
- Tell your ASG to use those subnets via the `vpc_zone_identifier` argument

```
resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier  = data.aws_subnet_ids.default.ids

  min_size = 2
  max_size = 10

  tag {
    key                 = "Name"
    value               = "terraform-asg-example"
    propagate_at_launch = true
  }
}
```

Load Balancer



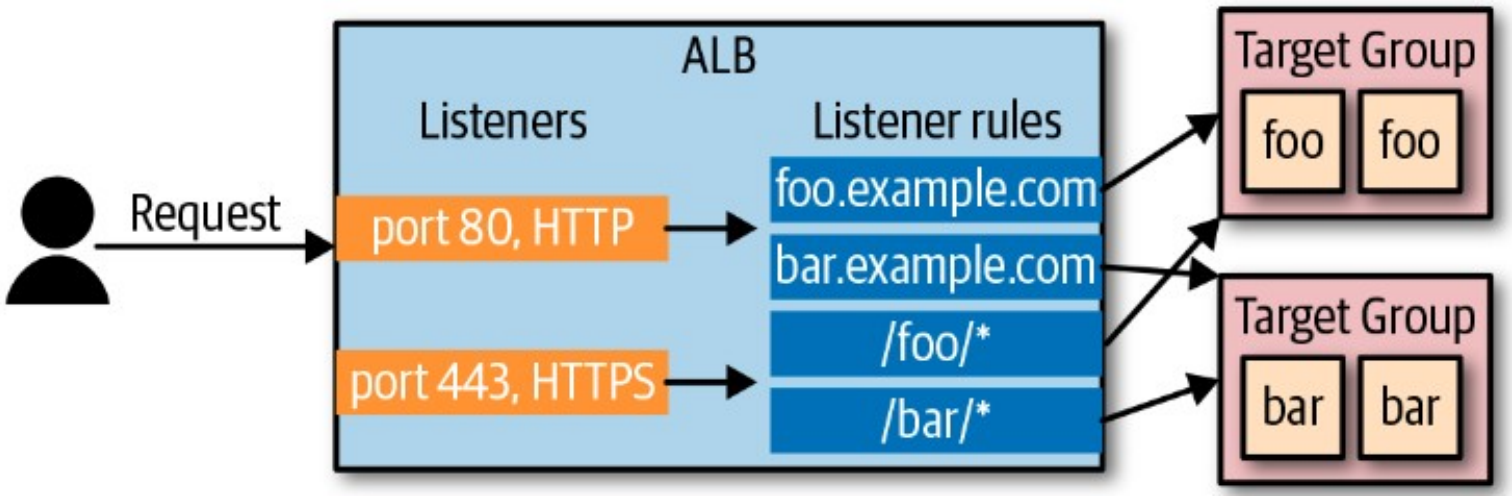
Using Load Balancer

- Problem
  - you now have multiple servers, each with its own IP address
  - you want to give of your end users only a single IP to use
- Solution
  - deploy a load balancer to distribute traffic across your servers
- Advantage
  - highly available and scalable
- ELB to the rescue

Load Balancer Types

- Application Load Balancer (ALB)
  - Best suited for load balancing of HTTP and HTTPS traffic
- Network Load Balancer (NLB)
  - Best suited for load balancing of TCP, UDP, and TLS traffic. Can scale up and down in response to load faster than the ALB (the NLB is designed to scale to tens of millions of requests per second). Operates at the transport layer (Layer 4) of the OSI model.
- Classic Load Balancer (CLB)
  - This is the “legacy” load balancer that predates both the ALB and NLB. It can handle HTTP, HTTPS, TCP, and TLS traffic, but with far fewer features than either the ALB or NLB. Operates at both the application layer (L7) and transport layer (L4) of the OSI model.

Application Load Balancer (ALB)



ALB Configuration

```
resource "aws_lb" "example" {
  name               = "terraform-asg-example"
  load_balancer_type = "application"
  subnets            = data.aws_subnet_ids.default.ids
}
```

ALB Listener

```
resource "aws_lb_listener" "http" {
  load_balancer_arn = aws_lb.example.arn
  port              = 80
  protocol          = "HTTP"

  # By default, return a simple 404 page
  default_action {
    type = "fixed-response"

    fixed_response {
      content_type = "text/plain"
      message_body = "404: page not found"
      status_code  = 404
    }
  }
}
```

Security Group for ALB

```
resource "aws_security_group" "alb" {
  name = "terraform-example-alb"

  # Allow inbound HTTP requests
  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  # Allow all outbound requests
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

"aws_lb resource" to Use Our Security Group

```
1 resource "aws_lb" "example" {
2   name               = "terraform-asg-example"
3   load_balancer_type = "application"
4   subnets            = data.aws_subnet_ids.default.ids
5   security_groups    = [aws_security_group.alb.id]
6 }
```

Limits for Your ASG

```
resource "aws_lb_target_group" "asg" {
  name     = "terraform-asg-example"
  port     = var.server_port
  protocol = "HTTP"
  vpc_id   = data.aws_vpc.default.id

  health_check {
    path                = "/"
    protocol            = "HTTP"
    matcher             = "200"
    interval            = 15
    timeout             = 3
    healthy_threshold   = 2
    unhealthy_threshold = 2
  }
}
```

What the Target Group Do?

- health check your Instances by periodically sending an HTTP request to each Instance
- will consider the Instance “healthy” only if the Instance returns a response that matches the configured matcher
- we told the matcher to look for a 200 OK response the target group will automatically stop sending traffic to unhealthy instance

Target Group Knows Its EC2 Instances

```
resource "aws_autoscaling_group" "example" {
  launch_configuration = aws_launch_configuration.example.name
  vpc_zone_identifier  = data.aws_subnet_ids.default.ids

  target_group_arns = [aws_lb_target_group.asg.arn]  # HERE
  health_check_type = "ELB"    # HERE

  min_size = 2
  max_size = 10

  tag {
    key                 = "Name"
    value               = "terraform-asg-example"
    propagate_at_launch = true
  }
}
```

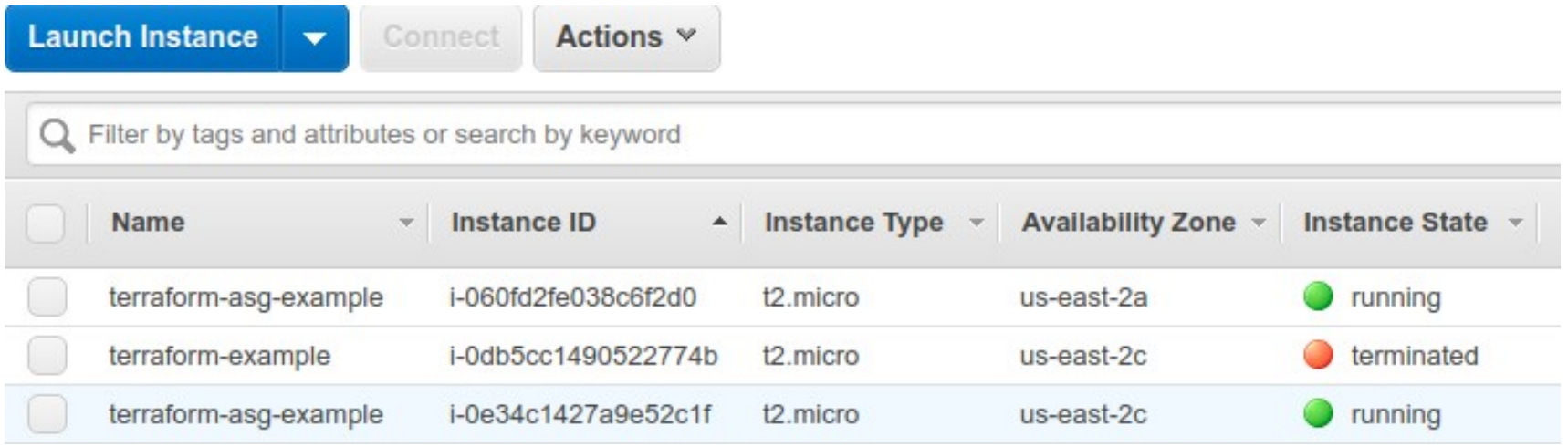
ALB Listener Rule

```
resource "aws_lb_listener_rule" "asg" {
  listener_arn = aws_lb_listener.http.arn
  priority     = 100

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

  action {
    type             = "forward"
    target_group_arn = aws_lb_target_group.asg.arn
  }
}
```

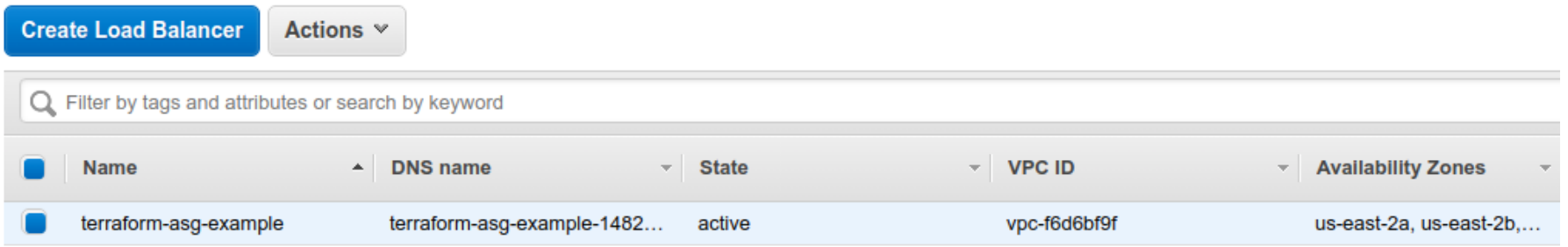
New Output - The DNS Name of the ALB

```
output "alb_dns_name" {
  value       = aws_lb.example.dns_name
  description = "The domain name of the load balancer"
}
```

Results of Upcoming Lab - Instances



Results of Upcoming Lab - Load Balancer



Results of Upcoming Lab - Target Group