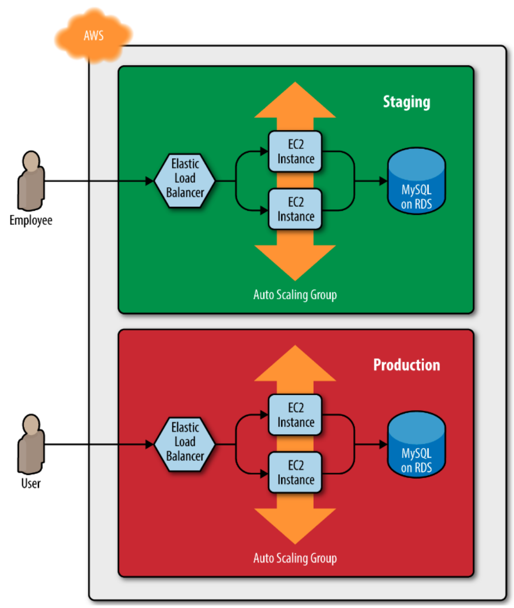
Reusable Infrastructure with Modules

The Plan

- Module basics
- Module inputs
- Module locals
- Module outputs
- Module gotchas
- Module versioning

Multiple Environments

- Cloud advantage: create multiple copies of the same environment
  - Production, Development, Test
  - Environments need to be similar if not identical
- We don't want to be able to re-use Terraform code across environments
  - DRY Principle: "Do not repeat yourself"

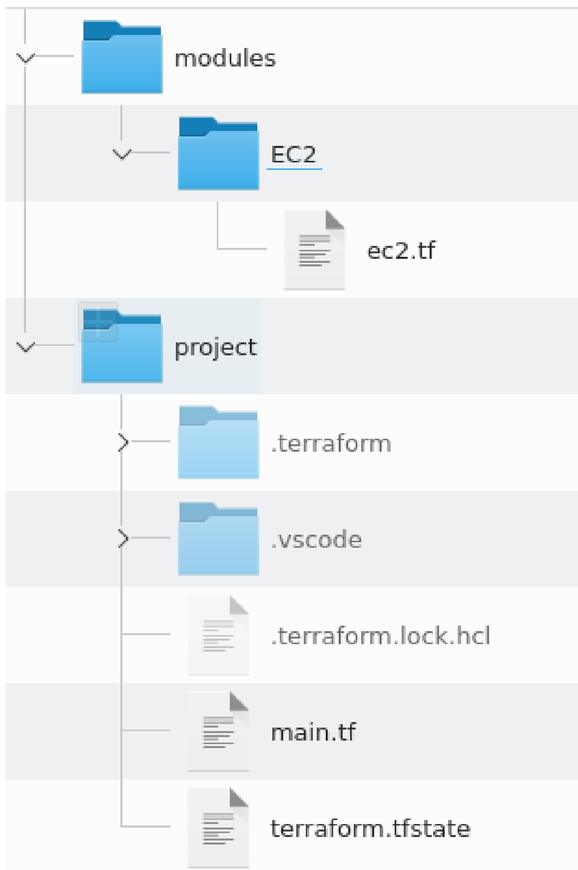


Module Basics

- Any folder containing terraform files is a module
  - There are no special declarations or syntax required
  - Modules are containers for multiple resources that are used together
  - Modules are the primary strategy used to package and reuse terraform resources
- Every terraform configuration has at least one module
  - It is referred to as the "root" module
  - It consists of the terraform files in the main working directory
- Modules (usually the root) may import other or "call" other modules
  - Modules that are being called are called "child" modules

Calling Modules Example

- If we are creating the same resource in multiple configurations, we can put it into a module
  - In this example, the demo project uses a module in the *modules/EC2* folder to create an EC2 instance to function like a webserver
  - The folder structure looks like this:



Calling Modules Example

- The EC2 Module code is familiar

```
resource "aws_instance" "alpha" {
    ami = "ami-047a51fa27710816e"
    instance_type = "t2.micro"
    tags = {
        source = "EC2 Module"
    }
}
```

- Calling it as a module is straightforward

```
module "VM-1" {
    source = "../modules/EC2"
  }
```

- The problem is that this module is not easily reusable because the *ami* and *instance type* are hard-coded into the module code
  - We need to parameterize the module to make it reusable

Lab 5-1

- Please do lab 5-1

Module Inputs

- Following the example of calling functions in a programming language, we want to be able to pass values to a module as parameters
- In the webserver example, we want to parameterize the module code by adding three variables:

```
variable "ami_type" {
    type = string
    default = "ami-00399ec92321828f5"
}

variable "inst_type" {
    type = string
    default = "t2.nano"
}

variable VM_name {
    type = string
    default = "EC2-Module"
}
```

Inside the EC2 Module

- We can parameterize the EC2 definitions

```
# In the EC2 Module

resource "aws_instance" "alpha" {
    ami = var.ami_type
    instance_type = var.inst_type
    tags = {
        source = "EC2 Module"
        Name = var.VM_name
    }
}
```

Lab 5-2

- Please do lab 5-2

Passing Parameters

- We can pass the values to the variables when the EC2 module is called

```
module "VM-1" {
    source = "../modules/EC2"
    ami_type = "ami-00399ec92321828f5"
    inst_type = "t2.micro"
    VM_name = "alpha"
  }

module "VM-2" {
    source = "../modules/EC2"
    ami_type = "ami-00399ec92321828f5"
    inst_type = "t2.small"
    VM_name = "beta"
  }
```

Variable Passthroughs

- We don't want to have to hardcode the parameters so we can use root module variables to pass the values through

```
variable machine_names {
    type = list(string)
    default = ["Singleton"]
}

variable machine_amis {
    type = list(string)
    default = ["ami-077e31c4939f6a2f3"]
}

variable machine_types {
    type = list(string)
    default = ["t2.nano"]
}
```

```
module "VM-1" {
    source = "../modules/EC2"
    ami_type = var.machine_amis[0]
    inst_type = var.machine_types[0]
    VM_name = var.machine_names[0]
  }
```

Lab 5-3

- Please do lab 5-3

Using "count" with Modules

- We can create multiple copies of modules with count

```
module "VM" {
    count = length(var.machine_names)
    source = "../modules/EC2"
    ami_type = var.machine_amis[count.index]
    inst_type = var.machine_types[count.index]
    VM_name = var.machine_names[count.index]
  }
```

```
# terraform.tfvars

machine_names = ["alpha","beta"]
machine_types = ["t2.small","t2.nano"]
machine_amis = ["ami-077e31c4939f6a2f3","ami-00399ec92321828f5"]
```

Lab 5-4

- Please do lab 5-4

Combining Modules

- Generally, we need to use more than one module because we need more than one type of resource
- In addition to the Ec2 instance module, we define a security group module

```
resource "aws_security_group" "app_port" {
    description = " Security group to allow access app instance"
    ingress {
    description = "OpenPort"
    from_port   = var.access_port
    to_port     = var.access_port
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags = {
    Name = var.sg_name
  }
}
```

Lab 5-5

- Please do lab 5-5

Combining Modules

- The security group has its own variables and called just like the EC2 module

```
variable "access_port" {
    description = "Access port to use for the application"
    type = number
    default = 80
}

variable "sg_name" {
    description = "The name of the security group"
    type = string
    default = "My SG"
}
```

```
module "VM" {
    count = length(var.machine_names)
    source = "../modules/EC2"
    ami_type = var.machine_amis[count.index]
    inst_type = var.machine_types[count.index]
    VM_name = var.machine_names[count.index]
  }

module "SG" {
  source = "../modules/SGroup"
}
```

Module Return Value

- To use the security group created by the SG module, we need to be able to reference it
- An output variable is passed back to the calling module with the id of the security group

```
output "secgps" {
    value =aws_security_group.app_port.id
    description = "Returns the id of the security group"
}
```

- And an input variable for the EC2 module so it can be passed the security group reference

```
variable "sg_groups" {
    description = "Associated security groups"
    type = string
}
```

- And add the reference to the resource definition

```
resource "aws_instance" "alpha" {
    ami = var.ami_type
    instance_type = var.inst_type
    tags = {
        source = "EC2 Module"
        Name = var.VM_name
    }
    vpc_security_group_ids = [var.sg_groups]
}
```

Module Return Value

- We can reference the value returned by the module with the following syntax
  - Reminder that the module name is the value we create when we *call* the module

```
module.< MODULE_NAME >.< OUTPUT_NAME >
```

```
module "VM" {
    source = "../modules/EC2"
    ami_type = var.machine_ami
    inst_type = var.machine_type
    VM_name = var.machine_name
    sg_groups = module.SG.secgps
  }
```

Lab 5-6

- Please do lab 5-6

Module Gotchas - Paths

- The hard-coded file paths are interpreted as relative to the current working directory
  - The problem is that this will not work if we are working with a module in a different directory
- To solve this issue, you can use an expression known as a path reference, which is of the form path.<TYPE>. Terraform supports the following types of path references:
  - *path.module* : Returns the file system path of the module where the expression is defined
  - *path.root* : Returns the file system path of the root module
  - *path.cwd* : Returns the file system path of the current working directory, usually the same as path.root

Module Path

- In this example, the template file is located with a path relative to the module, but if we hard-code the path, it will be interpreted as relative to the current working directory
  - By using the `path.module` construct, we ensure the file reference remains relative to the module

```
data "template_file" "user_data" {
  template = file("${path.module}/user-data.sh")

  vars = {
    server_port = var.server_port
    db_address  = data.terraform_remote_state.db.outputs.address
    db_port     = data.terraform_remote_state.db.outputs.port
  }
}
```

Module Gotcha - Inline Blocks

- The configuration for some Terraform resources can be defined either as inline blocks or as separate resources
  - When creating a module, you should always prefer using a separate resource
- Inline block example

```
resource "aws_security_group" "alb" {
  name = "${var.cluster_name}-alb"

  ingress {
    from_port   = local.http_port
    to_port     = local.http_port
    protocol    = local.tcp_protocol
    cidr_blocks = local.all_ips
  }

  egress {
    from_port   = local.any_port
    to_port     = local.any_port
    protocol    = local.any_protocol
    cidr_blocks = local.all_ips
  }
}
```

Separate Resource

- You should change this module to define the exact same ingress and egress rules by using separate aws_security_group_rule resources

```
resource "aws_security_group" "alb" {
name = "${var.cluster_name}-alb"
}

resource "aws_security_group_rule" "allow_http_inbound" {
  type                     = "ingress"
  security_group_id = aws_security_group.alb.id

  from_port    = local.http_port
  to_port      = local.http_port
  protocol     = local.tcp_protocol
  cidr_blocks = local.all_ips
}

resource "aws_security_group_rule" "allow_all_outbound" {
  type                     = "egress"
  security_group_id = aws_security_group.alb.id

  from_port    = local.any_port
  to_port      = local.any_port
  protocol     = local.any_protocol
  cidr_blocks = local.all_ips
}
```

Inline Blocks

- Using a mix of inline blocks and separate resources may cause errors where routing rules conflict and overwrite one another
  - Use one or the other
  - When creating a module, you should always try to use a separate resource instead of the inline block
  - This allows for more flexible modules
- For example, changing a security group rule to allow a testing port is easier to do with a separate resource than having to edit inline blocks

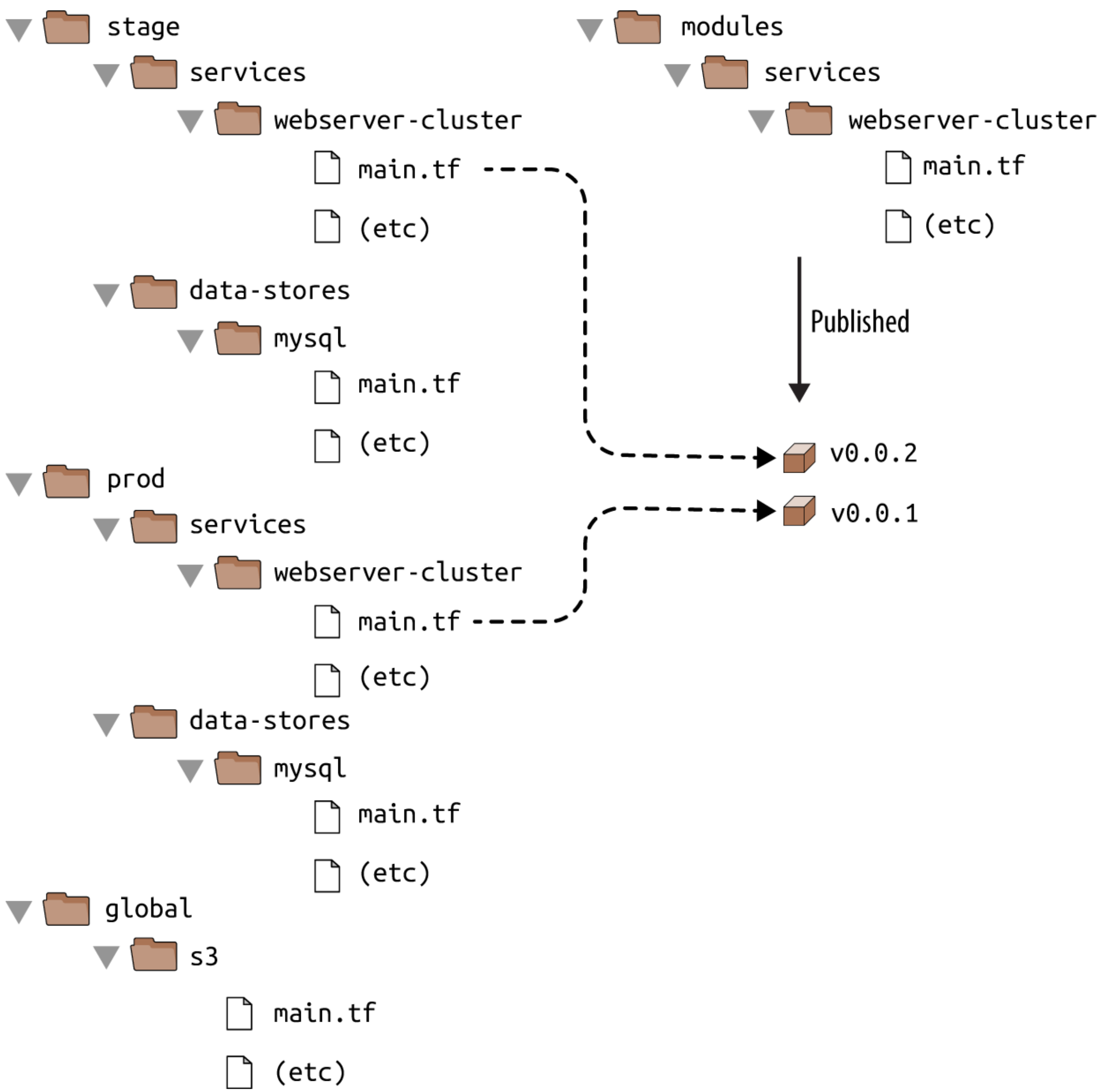
```
resource "aws_security_group_rule" "allow_testing_inbound" {
  type              = "ingress"
  security_group_id = module.webserver_cluster.alb_security_group_id

  from_port   = 12345
  to_port     = 12345
  protocol    = "tcp"
  cidr_blocks = ["0.0.0.0/0"]
}
```

Module Versioning

- If the staging and production environment point to the same module folder, any change in that folder will affect both environments on the very next deployment
  - This creates a coupling between environments and modules that can cause problems
- To solve this problem, we use a standard build management technique of using versions
  - As changes are made to a module, releases or versions of that module are published
  - Part of the configuration of any Terraform configuration plan is identification of which version of a module to include

Module Versioning Layout



Module Versioning

- An effective strategy is to use a repository tool like git and GitHub to publish releases of a module
  - Then the appropriate "release" of a module can be used

```
module "webserver_cluster" {
source = "github.com/foo/modules//webserver-cluster?ref=v0.0.1"

cluster_name           = "webservers-stage"
db_remote_state_bucket = "(YOUR_BUCKET_NAME)"
db_remote_state_key    = "stage/data-stores/mysql/terraform.tfstate"

instance_type = "t2.micro"
min_size      = 2
max_size      = 2
}
```

Semantic Versioning

- A common versioning scheme is "semantic versioning"
  - The format is MAJOR.MINOR.PATCH (e.g., 1.0.4)
  - There are specific rules on when you should increment each part of the version number
- MAJOR version increments when you make incompatible API changes
- MINOR version increments when you add functionality in a backward-compatible manner
- PATCH version increments when you make backward-compatible bug fixes