Infrastructure as Code Using Terraform

David D. Riddle <ddriddle@illinois.edu>
Jon R. Roma <roma@illinois.edu>

November 2, 2017

Who are we, and why are we here?

- David Riddle
  - David is a software developer at Technology Services
- Jon Roma
  - Jon is a software developer who has spent 35 years at Technology Services and its predecessors
- Both David and Jon are part of Continuous Improvement team within the Application Services division of Tech Services

Part 1

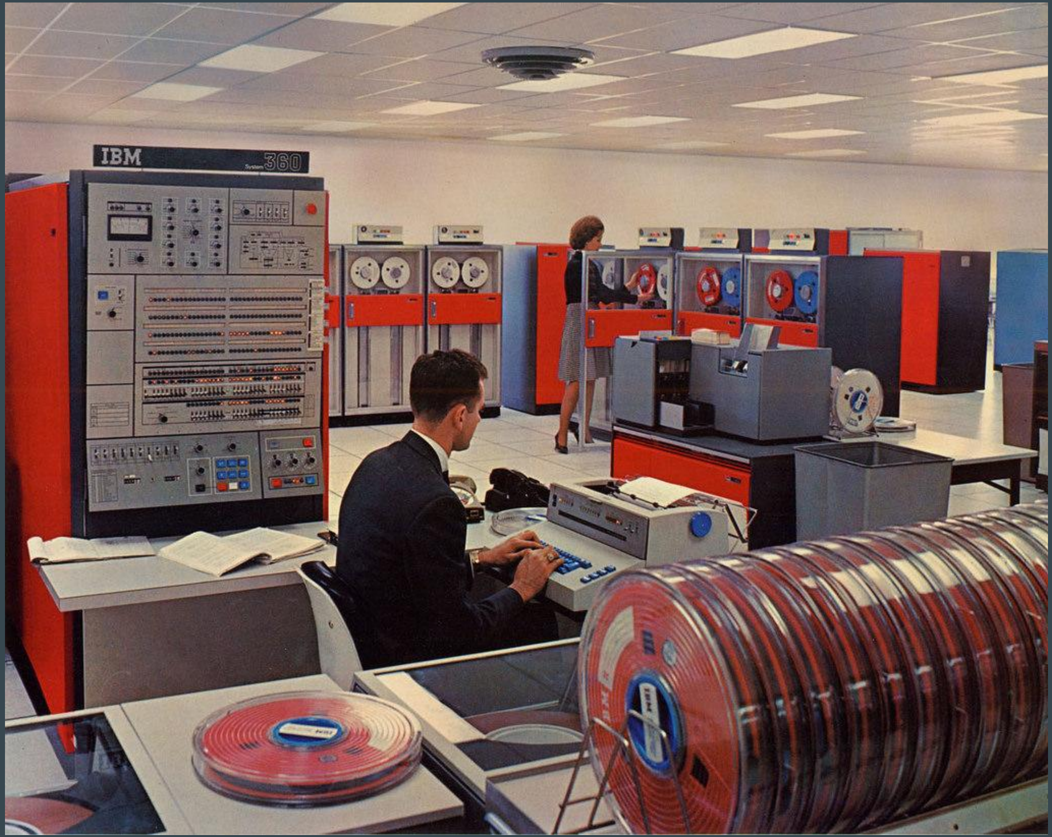
An introduction to Infrastructure as Code and to Terraform

- What is “Infrastructure as Code”?
- What is Terraform and what is the competition?
- Why did we choose Terraform over the others?

Why infrastructure as code?

Once upon a time...

- Infrastructure was physical
- Likewise, configuring and provisioning involved physical activity
    - Plugging in devices, and mounting media
- Nearly everything was a manual, one-off operation



Evolving … but not without pain

- Over time, Technology Services moved from running monolithic mainframes to “one server per service” – and then moved onward to virtualization
- Unfortunately, the resulting sprawl of servers can be difficult and costly to maintain

Evolving … but not without pain

- Configuring and provisioning servers was often a manual, one-off operation – just like in days of old
- Even deploying updates often involved a combination of manual checklists and *ad hoc* scripts
- In short, configuring, provisioning, testing, and deploying services could often be painful
    - As a result, we sometimes avoided making updates to software products that would have been valuable to our customers

Many manual operations

- Resizing of services to meet demand required shuffling hardware
  - Even with virtualization, this was largely manual and hence costly in terms of human effort
  - This workload often led to poor resource utilization
    - Throwing hardware resources at the problem was an unfortunate way to save the human effort

The missing element was treating infrastructure as code

What do we mean by infrastructure?

- What does the term “infrastructure” entail?
  - Web servers, database servers, load balancers, etc.
  - Network components (subnets, VPNs, firewall rules)
  - Persistent storage, logging, and so forth
  - ... and much more!

What is Infrastructure as Code?

- With Infrastructure as Code (IaC), we use the same techniques that were embraced long ago for software – but now we apply them to infrastructure components
- Techniques include:
  - Version control
  - Test Driven Development (TDD)
  - Continuous Integration (CI)

The benefits of automation

- With infrastructure automation using IaC principles, we can repeat operations across a vast swath of servers
- We never have to scratch our heads to recall the steps taken to build a particular environment
  - All the components and all the steps are documented in the infrastructure code
- The tried-and-true techniques, practices, and tools from software development are applied to infrastructure

Improve speed, reliability, security, and quality

- Infrastructure as Code allows us to create repeatable and testable infrastructure definitions before changes are applied to business-critical systems
- We can thereby increase the speed at which changes can be made
  - We also increase the reliability, security, and quality of services
- Does infrastructure created in the AWS console improve on the old manual processes used with physical hardware?
  - **No!**

Why did Terraform become our chosen Infrastructure as Code tool?

There are numerous tools to do Infrastructure as Code

- Ansible
- Chef
- CloudFormation
- Puppet
- Terraform

Choosing an Infrastructure as Code tool

- Note that all the tools we just mentioned are open source, and work with multiple cloud providers – with one exception
  - CloudFormation is closed source and is AWS-only
- All are well-documented and well-supported (with enterprise support available)
- All have a substantial community of contributors and support

Configuration management *vs.* orchestration

- There is some overlap in capabilities between these tools
- Ansible, Chef, and Puppet excel at *configuration management*
  - Their primary function is to install and manage software on existing servers
- In contrast, CloudFormation and Terraform excel at *orchestration*
  - Their function is to define the server infrastructure

Docker and Packer do configuration management

- If you’re using Docker and Packer, most of your configuration management needs are already satisfied
  - Docker creates containers
  - Packer creates virtual machine images
- In either case, all you need to deploy your image is a server
  - Terraform is well-suited for building immutable infrastructure

Mutable *vs.* immutable infrastructure

- If you tell Ansible to install a new version of Java, the changes are made *in place* on your existing servers
  - This can create “configuration drift” where each server can become slightly different than the others
  - These subtle configuration differences are difficult to detect and to debug

Mutable *vs.* immutable infrastructure

- Contrast this with an environment made up of Docker or Packer images:
    - Don't make incremental changes to existing infrastructure
        - Throw it away and redeploy!
- Any software change simply involves building a new image and deploying it on clean servers

Mutable *vs.* immutable infrastructure

- Think of each change as a new deployment, with several benefits:
  - You know *exactly* what software is running on every server
    - The code you deploy is byte-for-byte *identical* to the code you built and tested – you've eliminated configuration drift
  - *Any* version of the code (past, present or future) can be deployed at will
    - This is a prerequisite for continuous integration

Procedural *vs.* Declarative

- Ansible and Chef are *procedural*:
  - The developer writes code that defines the steps to be taken to achieve the desired end state
- Terraform, CloudFormation, and Puppet are *declarative*:
  - The developer writes code that defines the desired end state

Procedural *vs.* Declarative

- Terraform's declarative nature means that the code always represents the desired state of your infrastructure
- It is easy at a glance to determine what's currently deployed and how it's configured
- Terraform figures out how to achieve the desired state from the infrastructure configuration code and state

Why did Technology Services choose Terraform for IaC?

- Terraform runs on most common platforms:
  - Mac OS X
  - FreeBSD
  - Linux
  - OpenBSD
  - Solaris
  - Windows

Why did Technology Services choose Terraform for IaC?

- Terraform supports numerous cloud providers
  - This allows using the same tool across multiple providers
- The Terraform community is enthusiastic, and the development team is prolific
  - Bugs are fixed rapidly
  - New features are added rapidly
    - Includes support for new infrastructure components

Why did Technology Services choose Terraform for IaC?

- Infrastructure should be immutable
  - Immutable infrastructure is better suited for dynamic scaling
  - Immutable infrastructure is easier to test
- Terraform was designed from the ground up with these attributes in mind
  - Terraform rapidly gained grassroots support within several different groups in Technology Services

Application Services' preferred technology

- **Amazon Web Services** – is our default platform for services moving to the cloud
  - Example of an exceptional case: Microsoft Azure for Windows applications like Exchange or Active Directory
- **Terraform** – Infrastructure as Code
- **Terragrunt** – a thin wrapper for Terraform supporting remote state and locking
- **Docker** – for containerizing our services

Application Services' preferred technology

- **GitHub** – version control and issue tracking
- **Drone CI** – continuous integration engine
- **Behave, Gherkin, and Selenium** – for automated integration testing

Application Services' implementation model

- Application Services favors pre-built Amazon machine images (AMIs) and Docker images
  - Images are faster to launch
    - Scaling up and scaling down are efficient
  - Docker images are platform-agnostic
    - They can run on any base platform (Amazon, Google Cloud, VMware, etc.)

Application Services’ implementation model

- We use Terraform to provision the infrastructure on which our containers reside
    - Our Docker clusters consist of virtual machines built from images that support Amazon EC2 Container Service (ECS)
- This means we don’t need traditional configuration management tools like Ansible and Chef

Questions?

Part 2

In-depth view of Terraform concepts

- Configuration files
- Command-line options
- State
- Providers
  - Resources
  - Data sources

Terraform configuration files

How Terraform infrastructure definitions are organized

- Terraform manages infrastructure components by directory
  - A Terraform directory typically contains definitions for related components
  - Terraform doesn’t care how the individual infrastructure definitions are organized in files within a directory
  - Furthermore, the order of definitions of resources and variables in these files doesn’t matter

How Terraform infrastructure definitions are organized

- Consider that a web server might consist of several components
- These components might include:
  - The server itself (one or more Amazon EC2 instances)
  - A load balancer
  - An autoscaler
  - Security groups (a.k.a. firewall rules)
  - A back-end database server

How Terraform infrastructure definitions are organized

- These infrastructure components are intimately related
    - The Terraform definitions for these components – *except* the database tier – thus belong in a single directory
- Why might we maintain the database infrastructure separately?
    - By their very definition, databases are stateful
        - Their entire purpose is to persist valuable data
        - We ordinarily want the data to remain intact when reconfiguring the application

How Terraform infrastructure definitions are organized

- Web servers, on the other hand, are stateless
  - We can treat these components as ephemeral infrastructure
    - We want to build, destroy, and reconfigure at will
- As a result, our general recommendation is to divide the Terraform configuration based on their statefulness
  - Give yourself some flexibility by considering your infrastructure's lifecycle

Terraform configuration files – two file formats

- There are two accepted formats of writing Terraform infrastructure definition files:
  - HashiCorp Configuration Language (HCL)
  - JSON

Terraform configuration files in HCL

- HCL is Terraform's native format
  - HCL is designed to strike a balance between being human-readable and machine-friendly
  - HCL is recommended format for Terraform configuration files
  - HCL configuration files are text files with names ending in `.tf`

Terraform configuration files in JSON

- Terraform's support for JSON configuration files allows software generation of Terraform infrastructure definitions
- Terraform processes JSON files that end with `.tf.json`

Terraform configuration files

- Any files in the working directory that do *not* end in either `.tf` or `.tf.json` are ignored
    - The supported formats (HCL and JSON) can be mixed freely
- We recommend sticking with HCL configuration files
    - All of our examples use HCL

An example Terraform configuration file

```
# Define the Amazon machine image (AMI) to use.
variable "ami" {
    description = "Amazon machine image"
}

/* A multi
   line comment. */

resource "aws_instance" "web" {
    ami           = "${var.ami}"
    count         = 2
    instance_type = "${var.instance_type}"
    tags {
        Name = "example"
    }
}
```

- This Terraform configuration file is written in HCL
- It defines a variable named `ami`
- Second, it declares an `aws_instance` (EC2) resource
  - The `count` parameter indicates that two instances are to be created
  - We add a tag for our instances

Terraform command-line options

Terraform commands (Command line interface)

```
Common commands:
    apply              Builds or changes infrastructure
    console            Interactive console for Terraform interpolations
    destroy            Destroy Terraform-managed infrastructure
    env                Environment 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 new or existing Terraform configuration
    output             Read an output from a state file
    plan               Generate and show an execution plan
    push               Upload this Terraform module to Terraform Enterprise to run
    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
```

The three basic Terraform commands

- The principal Terraform verbs you'll encounter are few in number
  - `plan` – Show Terraform's execution plan
    - Tells the developer what infrastructure will be created, modified, or destroyed
  - `apply` – Apply the infrastructure definition
    - Create, modify, and destroy components as required
  - `destroy` – Destroy the defined infrastructure

Additional commonly-used Terraform commands

- The following Terraform verbs are also useful:
  - `init` – Initialize a Terraform configuration
    - Downloads and initializes any plugins (such as providers)
  - `get` – Initialize modules used in a Terraform configuration
    - Downloads module from source and initializes
  - `fmt` – Rewrites configuration files to canonical format

Additional commonly-used Terraform commands (continued)

- Still more commonly-used Terraform verbs:
  - `show` – Inspect Terraform state or plan
  - `output` – Extract one or more outputs from a Terraform state file
    - Useful for visual confirmation of completed plan
    - Also useful to import Terraform outputs into a shell script or program

Terraform state

Terraform state

- Terraform maintains state about each infrastructure configuration
    - Allows tracking real resources and metadata
    - Improves performance for large infrastructure configurations
- Terraform state is maintained *per directory*
    - By default, the state resides in a file named `terraform.tfstate` within each directory
    - Remote Terraform state storage is better suited to team environments

Terraform state

- Terraform uses its state to create its execution plan and to make changes to your infrastructure
  - Before any operation, Terraform synchronizes the state to reflect the real infrastructure

Terraform state

- Terraform uses its state to map the infrastructure definitions in your Terraform files to the real world infrastructure
- You might have a resource definition like this:

```
resource "aws_instance" "web" {
    ...
    }
```

- Terraform's state file allows mapping this resource definition to a running Amazon EC2 instance `i-006534aca5bcf2164`, for example

Terraform state

- Terraform must also maintain metadata
    - The most obvious metadata element is dependency order
        - This is used to allow Terraform to create and destroy resources in proper sequence
    - Terraform stores additional metadata like creation time, update time, last run time, etc.
    - Cached metadata can improve performance under some conditions

Terraform import

- We often experimentally build and configure infrastructure manually
- We eventually want Terraform to manage this as Infrastructure as Code
- Terraform can import existing infrastructure:

```
% terraform import aws_instance.web i-006534aca5bcf2164
```

Terraform import

- Currently, Terraform's import feature has some major limitations
  - Import isn't supported for all resources
  - Terraform only imports resources into the state file
    - Import does not generate Terraform configuration code (`*.tf` files) at this time

Terraform import

- Currently, you must manually write configuration code to preserve what terraform import has stored in the state file
    - This is a little bit tedious, but `terraform plan` can help verify that the configuration code you write accurately describes the manually-created infrastructure
- A future version of Terraform will generate complete Infrastructure as Code in the form of `*.tf` files

Terraform providers

Terraform providers

- A key Terraform concept is that of providers
  - Terraform is agnostic to the underlying infrastructure platform
- Each provider is responsible for API interactions with the underlying platform
  - A provider exposes platform functionality in the form of resources and data sources
- Providers are initialized with the `terraform init` command

Terraform has dozens of providers … a few are listed below

- AWS
- DNS
- Docker
- GitHub
- Google Cloud
- Heroku
- HTTP
- Local
- Microsoft Azure
- MySQL
- PostgreSQL
- Random
- Template
- VMware vSphere

Terraform providers

- This provider block defines parameters used to interact with AWS:

```
provider "aws" {
    region  = "us-east-2"
    profile = "roma-test-readonly"
}
```

- This provider block accomplishes two things:
    - Terraform loads the plugin supporting AWS
    - We specify the AWS region and identify the authentication credentials

Terraform providers

- We normally don't hardcode the AWS profile in the Terraform provider block
- Instead, we set two system environment variables:
    - `AWS_DEFAULT_REGION`
    - `AWS_PROFILE`
- Then, all we need is a trivial Terraform provider block:

```
provider "aws" {}
```

Using multiple providers is supported

```
#   Provider for AWS region in North America.
provider "aws" {
    alias    = "north-america"
    region   = "ca-central-1"
}

#   Provider for AWS region in Europe.
provider "aws" {
    alias    = "europe"
    region   = "eu-central-1"
}
```

NOTE: This example assumes that suitable Amazon credentials are configured

Setting the `AWS_PROFILE` environment variable is one possible method to do this

```
#   Resource for VPC in North America.
resource "aws_vpc" "vpc_na" {
    cidr_block   = "10.0.0.0/16"
    provider     = "aws.north-america"
}

#   Resource for VPC in Europe.
resource "aws_vpc" "vpc_europe" {
    cidr_block   = "10.0.0.0/16"
    provider     = "aws.europe"
}

#   Output the id from each of the VPCs.
output "vpc_na_id" {
    value        = "${aws_vpc.vpc_na.id}"
}
output "vpc_europe_id" {
    value        = "${aws_vpc.vpc_europe.id}"
}
```

Terraform resources

- A resource usually corresponds to an infrastructure component such as:
  - Virtual machines (EC2 instances)
  - DNS records
  - Database (RDS or DynamoDB) instances
  - Security groups
  - Virtual private clouds (VPCs)
  - Lambda functions

Some Terraform resources for AWS

- After specifying the AWS provider to Terraform, the infrastructure developer gains access to numerous AWS resources
- A few of the many available AWS resources are shown below:
  - aws_instance – An Amazon EC2 instance
  - aws_s3_bucket – An S3 bucket
  - aws_s3_bucket_object – An object in an S3 bucket
  - aws_db_instance – An RDS database instance

Wait … there's more!

- The above list is just a brief example
    - There are many AWS resources supported by Terraform
        - Support for new platform features in Terraform is rapid
- A resource can be almost anything – a physical machine, a VM, a network switch, a subnet, a container, etc.

Terraform data sources

- Terraform providers also provide data sources, which allow retrieving data external to your Terraform configuration
  - Information not maintained in Terraform
  - Information defined by independent Terraform configurations
- Data sources present read-only views into pre-existing data, or compute values at runtime
  - Allows separate management of shared infrastructure components in isolation from consumers of those components

Example Terraform data sources for the AWS provider

- aws_ami – given a search string, get a list of Amazon machine images (AMIs) matching the search criteria
- aws_availability_zones – given an AWS region, get a list at runtime of AWS availability zones within that region
- aws_db_instance – gives information about an RDS instance
- aws_iam_role – given the name of an IAM role, get information about the role like its Amazon Resource Name (ARN)

Example: Show available AWS availability zones in a region

```
provider "aws" {
    region = "us-east-2"
}

data "aws_availability_zones" "available" {
    state    = "available"
}

output "az_available" {
    value   = "${data.aws_availability_zones.available.names}"
}
```

The value in the output block is a Terraform variable reference – more about variables later

Output from our example

```
% terraform apply
data.aws_availability_zones.available: Refreshing state...

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

Outputs:

az_available = [
    us-east-2a,
    us-east-2b,
    us-east-2c
]
```

Terraform provisioners

- A provisioner can be used to execute scripts on a local or remote server while creating or destroying resources
  - Examples: bootstrap a resource, clean up before destroy, run configuration management, etc.
  - This provisioning might take the form of a Chef or Ansible script – or even a shell script
- Technology Services doesn't make heavy use of provisioners
  - We do provisioning at the time we build Docker images

Terraform provisioners

- A provisioner block can be inserted inside any resource block:

```
resource "aws_instance" "web" {
  ...
  provisioner "local-exec" {
    command = "echo ${self.private_ip} > ip_addr.txt"
  }
}
```

- Remote provisioners will need a connection block to specify connection method (e. g., ssh or winrm), username, and password

Terraform provisioners

```
resource "aws_instance" "web" {
  ...
  # Copy the conf/myapp.key file to /etc/myapp.key.
  provisioner "file" {
    source      = "conf/myapp.key"
    destination = "/etc/myapp.key"
  }
  # Copy a string's content into /tmp/myapp.log.
  provisioner "file" {
    content     = "ami used: ${self.ami}"
    destination = "/tmp/myapp.log"
  }
  # Copy everything in config/myapp to /etc/myapp/config.
  provisioner "file" {
    source      = "config/myapp/"
    destination = "/etc/myapp/config"
  }
}
```

- Multiple provisioners can be defined
  - They run in sequence
- Each of the three provisioners here copies data into the resource from the host running Terraform
  - The first example copies a single file
  - The second example copies text from a Terraform variable
  - The final example copies a whole directory tree

Questions?

Part 3

Advanced Terraform: Resources, variables, and interpolation

- Resources and data sources
- Terraform configuration file semantics
- Declaring, assigning, and using Terraform variables
- Terraform variable interpolation

More about Terraform resources and data sources

Resource declarations

- In the last session, we introduced the concepts of Terraform resources and data sources
- Resources are declared as a block that identifies the infrastructure component to be managed by Terraform:

```
resource "aws_instance" "web" {
  ami           = "${var.ami}"
  instance_type = "${var.instance_type}"
  count         = 2
}
```

Resource declarations

```
resource "aws_instance" "web" {
  ami           = "${var.ami}"
  instance_type = "${var.instance_type}"
  count         = 2
}
```

- The literal `resource` identifies the block as a resource declaration
- The quoted value `aws_instance` identifies the resource `type`
- The quoted value `web` is used by the developer to assign a `name` to this resource instance

Resource declarations

```
resource "aws_instance" "web" {
  ami           = "${var.ami}"
  instance_type = "${var.instance_type}"
  count         = 2
}
```

- Taken together, the `type` and `name` must be unique across your configuration
  - A Terraform configuration resides in a single directory
- The arguments used to configure a resource declaration are enclosed within the block inside braces

Resource declarations

```
resource "aws_instance" "web" {
  ami           = "${var.ami}"
  instance_type = "${var.instance_type}"
  count         = 2
}
```

- The specific configuration arguments supported vary by resource
    - The `aws_instance` object declared here has arguments that make sense for a virtual server, like `ami`, `instance_type`, and `count`
    - A different resource like `aws_s3_bucket` would support a different set of arguments appropriate for that resource

Resource declarations

- When `terraform apply` runs, Terraform
  - Saves the state of the new or updated infrastructure
  - Produces one or more `attributes`
    - These are akin to return values from a subroutine call
    - An example of an attribute might be a unique identifier for a virtual machine

Resource declarations

```
resource "aws_instance" "web" {
  ami           = "${var.ami}"
  instance_type = "${var.instance_type}"
  count         = 2
}
```

- The attributes produced by Terraform are useful in interpolations
  - One infrastructure component's declaration can thus use another component's [output] attributes as input
  - This allows Terraform to identify dependencies

Terraform configuration file semantics

HCL configuration file semantics

- Single-line comments start with `#`
- Multi-line comments are enclosed within `/*` and `*/`
- Values are assigned as `key = value`
  - Whitespace doesn't matter
  - The RHS (value) must be an expression that evaluates to one of the valid Terraform types

Terraform native types

- Terraform has three native types:
  - `string`
  - `list`
  - `map`

Terraform `string` type

- String values are expressed using double quotes
  - Example: `"silly little string"`
- Multi-line strings can use shell-style “here document” syntax
  - We discourage “here documents” for the sake of clarity
    - Inline documents impede the readability of your declarations and can often be avoided

Using “here documents” *vs.* reading from a file

```
resource "aws_iam_policy" "policy" {
  description = "Not recommended"
  policy = <<EOF
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "ec2:Describe*"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
EOF
}
```

```
resource "aws_iam_policy" "policy" {
  description = "More recommended"
  policy      = "${file("policy.json")}"
}
```

At left, we use a “here document” to assign an inline JSON policy document to our resource

At right, we use Terraform’s `file` function to specify a file containing the JSON document

Even better is a data source to declare the policy document

```
data "aws_iam_policy_document" "policy" {
  statement {
    actions = [
      "ec2:Describe*",
    ]
    resources = ["*"]
  }
}

resource "aws_iam_policy" "policy" {
  description = "Most recommended"
  policy      = "${data.aws_iam_policy_document.policy.json}"
}
```

- The document at left uses no JSON
- Instead, a Terraform data source defines the policy document
- The policy document resource in turn is used to initialize the policy

Terraform `string` type (continued)

- At this time, no native numeric or boolean types are available in Terraform
  - For consistent behavior, you should express numeric or boolean values as quoted strings
    - Example: “`42`” or “`false`”

Terraform `list` type

- A list is expressed using square brackets (`[]`)
  - Example: `["foo", "bar", "baz"]`

Terraform `map` type

- A map is expressed with braces (`{}`) and colons (`:`)
  - Example: `{ "foo": "bar", "bar": "baz" }`

Declaring Terraform variables

Declaring variables with Terraform

- Variables used in a Terraform declaration *must* be declared
- The most trivial example of a variable declaration in Terraform is

```
variable "foobar" {}
```

- The variable `foobar` is declared without specifying a type, description, or default value
  - The type thus defaults to string

Declaring variables with Terraform

- A string variable declaration with *explicit* typing:

```
variable "key" {
  type = "string"
}
```

- A list variable declaration with *implicit* typing:

```
variable "aws_zones" {
  default = ["us-east-1a", "us-east-1b"]
}
```

Declaring variables with Terraform

- A map variable declaration with *explicit* typing:

```
variable "images" {
  type = "map"

  default = {
    us-east-1 = "image-1234"
    us-west-2 = "image-4567"
  }
}
```

Declaring variables with Terraform

- A Terraform variable declaration block can include three optional parameters:

```
variable NAME {
   [default = default]
   [description = description]
   [type = type]
}
```

Declaring variables with Terraform

- The optional `default` parameter assigns a default value to the variable
- The optional `description` parameter provides a human-friendly description of the variable
  - This is useful when a user runs Terraform interactively
    - The description becomes part of the interactive prompt

Declaring variables with Terraform

- The optional `type` may be any of the valid Terraform types, namely `string`, `list`, or `map`
  - If no type is specified, the type is inferred from the default
  - If no default is provided, the type is assumed to be `string`
    - If you intend a variable to be a type other than `string`, you *must* declare the type or set a default value
  - Terraform will fail if the variable has no value at runtime

Terraform variable assignment

Assigning Terraform variables

- Terraform variables can be assigned in a variety of ways:
  - From a variables file
  - From an environment variable
  - From the command line
  - From an interactive prompt
    - Only strings can be assigned interactively
- We'll give some examples in the next several slides

Assigning Terraform variables

- If a file named `terraform.tfvars` is found in the current directory, Terraform loads it

An example `terraform.tfvars` file

```
# Set values for scalars.
is_prod = "false"
pi      = "3.14159"

# Set values for a list.
some_list = [
  "1",
  "2",
]

# Set values for a map.
some_map = {
  foo = "bar"
  bax = "qux"
}
```

Assigning Terraform variables

- If you want to use a different variables file, you can specify it using `-var-file` on the command line
- For example:

```
% terraform plan -var-file foobar
```

- The `-var-file` flag can be used multiple times per command invocation

Assigning Terraform variables

- Terraform variables can be assigned using environment variables
- The environment variables must be named `TF_VAR_`*name*
- For example, the following statement:

  ```
  % TF_VAR_image=foo terraform apply
  ```

  would set a Terraform variable named `image` to the value `foo`
- Maps and lists can be specified using environment variables in HCL format

Assigning Terraform variables from the command line

- You can set variables from the command line with the `-var` flag:

```
% terraform plan -var access_key=foo -var secret_key=bar
```

Assigning Terraform variables interactively

- Finally, if Terraform lacks assignment for any variables, it will prompt you interactively
    - Interactive input is only supported for string variables
    - Terraform will produce an error if any list or map variable is undefined at runtime

Terraform output variables

Terraform output variables

- Output variables specify the Terraform state elements that are exposed to the caller after a successful `terraform apply`
- Example output declaration:

```
output "ip" {
    value = "${aws_eip.ip.public_ip}"
}
```

- This defines an output variable named `ip`
    - This can be derived from interpolation, as seen above

Terraform output variables

- Terraform displays the output when you run `terraform apply`
- Example:

```
% terraform apply
...
Apply complete! Resources: 0 added, 0 changed, 0 destroyed.

Outputs:

  ip = 50.17.232.209
```

Terraform output variables

- You can also query the outputs after a Terraform configuration has been applied by using the `terraform output` command:

```
% terraform output ip
50.17.232.209
```

- This command can be used by scripts to extract outputs

Interpolating Terraform variables

Terraform’s interpolation syntax

- Terraform’s interpolation syntax is powerful
  - You can reference variables, attributes of resource definitions, outputs of module instances, and so forth
- Variable interpolations are wrapped in the construct `${}`

Interpolation operators

- Terraform interpolations support a number of common operators
  - Math operators `+`, `-`, `*`, and `/` for floating point values
  - Math operators `+`, `-`, `*`, `/`, and `%` for integer values
  - Equality operators `==` and `!=`
  - Numerical comparison operators `>`, `<`, `>=`, and `<=`
  - Boolean operators `&&`, `||`, and unary `!`
  - The well-known ternary operator from C and other languages

```
condition ? true_value : false_value
```

Interpolating a user string variable

- Example: `${var.foo}` interpolates the value of the variable `foo`
- Next, we'll look at interpolation of each of Terraform's data types

Interpolating a user map variable

- Example: `${var.amis["us-east-2"]}`
  - The variable `amis` is a map object
  - The variable reference interpolates the value of the `us-east-2` key within the `amis` map

Interpolating a user list variable

- Example 1: `${var.subnets}`
    - This interpolates the subnets variable as a list
- Example 2: `${var.subnets[3]}`
    - This returns the element at the specified list index

Interpolating attributes from a resource

- Example: `${aws_instance.web.id}`
  - The reference consists of three components:
    - The `type` of Terraform resource referenced
    - The `name` of a specific resource instance
    - The resource `attribute` to be interpolated
- Attribute values are computed by `terraform apply`

Example: Interpolating attributes from a resource

```
resource "aws_security_group" "ssh" {
  ingress {
    from_port   = "22"
    to_port     = "22"
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

resource "aws_instance" "default" {
  ami             = "${var.ami}"
  instance_type   = "${var.instance_type}"
  security_groups = ["${aws_security_group.ssh.name}"]
}
```

- We're creating an AWS security group (firewall rule) and an Amazon EC2 instance that uses that rule
- Interpolation is used to specify the Amazon machine image and instance type for the EC2 instance
- Interpolation is used to attach the security group to the EC2 instance

Breaking down a resource interpolation reference

- Let’s break down the interpolation reference we just showed:

```
${aws_security_group.ssh.name}
```

type name attribute

Interpolating count information

- A Terraform resource block can use the `count` meta-parameter to declare multiple instances of the resource
- The “magic” interpolation `${count.index}` can be used to specify different parameters for each resource instance
    - This construct takes the place of a loop, since Terraform is declarative and not iterative

Example: Interpolating count information

```
variable "name" {
  default = [ "Moe", "Larry", "Curly" ]
}

resource "aws_instance" "stooge" {
  ami           = "${var.ami}"
  instance_type = "${var.instance_type}"
  count         = "${length(var.name)}"

  tags {
    Name = "${element(var.name, count.index)}"
    Type = "Stooge"
  }
}
```

In this example, we use `${count.index}` to assign each EC2 instance a unique tag

Interpolating an attribute from a resource

- For resources declared to have multiple instances, two additional interpolations are available
  - Example 1: `${aws_instance.stooge.0.id}`
    - Returns the id attribute of the first resource
  - Example 2: `${aws_instance.stooge.*.id}`
    - Returns a list of id attributes for each resource
    - The asterisk is sometimes referred to as “splat syntax”

Interpolating an attribute from a data source

- Example: `${data.aws_ami.debian.id}`
- The reference consists of four components:
  - The literal string `data`
  - The `type` (here, a Terraform data source)
  - The `name` of the specific data source instance
  - The `attribute` of the data source

Example: Interpolating an attribute from a data source

```
data "aws_ami" "amazon_linux" {
  most_recent = "true"
  owners      = ["amazon"]

  filter {
    name   = "name"
    values = ["amzn-ami-hvm-*-x86_64-gp2"]
  }
}

output "amazon_linux_id" {
  value = "${data.aws_ami.amazon_linux.id}"
}
```

- We're using a data source to select a suitable Amazon machine image
  - Runs Linux on a 64-bit x86 platform
  - Is an official Amazon image
  - Is the most recent build
- We interpolate an attribute from the data source and output it
- The interpolated value could be used in an `aws_instance` resource

Breaking down a resource interpolation reference

- Let's break down the interpolation reference we just showed:

```
${data.aws_ami.amazon_linux.id}
```

Literal string **data**

type

name

attribute

Denotes that this is a reference to a data source

Interpolating conditionals

- One can conditionally configure the count:

```
resource "aws_instance" "web" {
  ami           = "${data.aws_ami.amazon_linux.id}"
  instance_type = "${var.instance_type}"
  count         = "${var.define_web ? 1 : 0}"
}
```

- The `web` resource is only configured if `${var.define_web}` evaluates to true
- If the VPN count is zero, the resource is not created, or is destroyed

Interpolating conditionals

- In this example, the conditional uses the ternary operation:

```
resource "aws_instance" "web" {
  ami           = "${data.aws_ami.amazon_linux.id}"
  instance_type = "${var.instance_type}"
  count         = "${var.env == "prod" ? var.prod_num : var.dev_num}"
}
```

- The values returned by the true and false sides of the conditional must have the same data type

Interpolation functions

- Terraform has a rich list of functions that can be used in interpolations:
  - Examples: `abs`, `basename`, `ceil`, `cidrhost`, `dirname`, `element`, `file`, `floor`, `format`, `index`, `join`, `lookup`, `max`, `min`, `pow`, `sha512`, `sort`, `split`, `substr`
- Here's a bit of sample code:

```
output "hostname" {
  value = "${var.app}${lookup(var.suffix, var.env)}.${var.domain}"
}
```

Questions?

Part 4

Advanced Terraform

- Visualizing your infrastructure
- Terraform modules
- More Terraform interpolations
- Terraform remote state
- Adding Terragrunt to enhance our Terraform experience
- How Application Services uses Terraform and Terragrunt today
- Terraform best practices

Visualizing your infrastructure

Terraform infrastructure can be rendered into a graph

- The `terraform graph` command produces a graphical representation of a Terraform configuration or execution plan
- Usage is simple:

```
% terraform graph | dot -T png > infra.png
```

- The output is in the DOT format, and is supported by the open-source Graphviz package (see http://www.graphviz.org/) to generate diagrams
- A variety of web-based applications can also process DOT files

Sample output from the `terraform graph` command

Terraform modules

Terraform modules – setting the stage

- Much of the Terraform documentation refers to a collection of Terraform configuration files as a *module*
    - Strictly speaking, a Terraform module is nothing more than a directory containing Terraform code
- Here, we'll show how we can take advantage of Terraform modules to define reusable infrastructure configuration

Terraform modules – setting the stage

- Making a Terraform module reusable is almost effortless
  - Any directory with Terraform configuration files is inherently a Terraform module
  - Making this module reusable is simple:
    - We simply reference the module from within another Terraform configuration!
- If you refactor a Terraform module to be reusable, remove the state and variable files from the module directory – they're obsolete

Terraform modules – setting the stage

- All our previous Terraform examples have been run from within the module's native directory
  - Unless we plan to define module variables interactively, we define their values in the `terraform.tfvars` file in the module directory
  - We've already seen that Terraform maintains the infrastructure state in the file `terraform.tfstate` in the module directory
    - This state includes any outputs defined by the developer

Terraform modules – setting the stage

- Before we delve into invoking Terraform modules externally, we'll introduce a trivial code example
  - Our example module creates no infrastructure
  - The module defines two variables used as inputs
  - The module interpolates these two inputs, and performs exponentiation on them
  - The module produces two outputs

A simple example of code to be used as a Terraform module

```
variable "base" {
  description = "Base"
}

variable "exponent" {
  description = "Exponent"
}
```

```
output "debug_output" {
  value = "base: ${var.base} | exponent: ${var.exponent}"
}

output "result" {
  value = "${pow(var.base,var.exponent)}"
}
```

- This example code defines variables named `base` and `exponent`
- The example code provides outputs with the names `debug_output` and `result`

Terraform modules – setting the stage

- We'll first run the module natively
  - This is the ordinary way we've used modules in the past:
    - We navigate to the module directory and run `terraform plan` and `terraform apply` as usual

A simple example of code to be used as a Terraform module

```
% terraform apply
var.base
  Base
  Enter a value: 3

var.exponent
  Exponent
  Enter a value: 2

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

Outputs:

debug_output = base: 3 | exponent: 2
result = 9
```

- We run the configuration *directly* here
  - Later, we'll call this code externally as a module
- We supply variable values interactively
- Terraform records state in the module directory

Terraform modules – invoking externally

- We've shown how the Terraform module is run natively – i. e., from its own directory
    - Now, let's invoke the module's code externally
        - In other words, let's reuse that module again and again in other Terraform configurations
- Terraform handles variable definitions and state a bit differently when the module is invoked externally

Calling a Terraform module

- When invoking a module, Terraform manages the infrastructure defined in that module as if it were defined by the caller itself
- An important distinction is that the caller must provide any non-default variable values required by the module
- The infrastructure state (including outputs) managed by the module is *not* maintained in the module directory as before
    - Instead, the state produced by the module is part of the caller's state

Calling a Terraform module

- Calling Terraform modules resembles a resource definition:

```
module "1k" {
  base     = "10"
  exponent = "3"
  source   = "github.com/cites-illinois/terraform-demo/module/power"
}
```

- The identifier on the module declaration (“`1k`” in this example) denotes a name that refers to the module’s exported attributes
- The `source` identifies to Terraform where the module code resides
    - This is the only module argument required by Terraform itself

Providing inputs to a Terraform module

- We pass two other arguments into this module:

```
module "1k" {
  base     = "10"
  exponent = "3"
  source   = "github.com/cites-illinois/terraform-demo/module/power"
}
```

- These other arguments on the module block are simply input arguments required by the module itself
  - The module's maintainer uses `variable` declarations to determine what inputs the module accepts

Terraform modules can reside in a variety of locations

- Terraform’s modules can reside in a variety of locations:
  - GitHub repository URLs
  - Generic Git and Mercurial repository URLs
  - Amazon S3 buckets
  - Generic HTTP URLs
  - Local file paths

Initializing module references before first use

- Before running any Terraform command using modules, you need to download the modules:

```
% terraform get
```

- This command is fast, and does not check for updates by default
  - You can thus run it repeatedly (i. e., in a CI pipeline)
  - If desired, add the `-update` flag to check for and download updates

Receiving outputs from a module

- A module uses Terraform's `output` declarations to specify the output arguments to be returned to the caller
- We can thus interpolate a module output argument by using the format `${module.1k.result}`
    - The literal `module`
    - The specific module instance name `1k`
    - The attribute `result` refers to an output defined by the module

Using multiple instances of a module

- One can use multiple instances of a module simply by providing a different name for each `module` declaration

```
module "1k" {
  base     = "10"
  exponent = "3"
  source   = "github.com/cites-illinois/terraform-demo/module/power"
}

module "1m" {
  base     = "10"
  exponent = "6"
  source   = "github.com/cites-illinois/terraform-demo/module/power"
}
```

More Terraform interpolations

Interpolating filesystem paths

- Example: `${path.module}`
  - The reference consists of two components:
    - The literal `path`
    - One of the following literal type values
      - `cwd` – This interpolates the current working directory
      - `module` – This interpolates the current module's path
      - `root` – This interpolates the root module's path

An example of filesystem path interpolation

- Here's a trivial Terraform configuration that uses the `${path.module}` interpolation
- The code uses the current module's path to refer to a template file:

```
output "template_path" {
  value = "${path.module}/dumb.template"
}
```

An example of filesystem path interpolation

- Running `terraform apply` on the above configuration might produce the following output:

```
% terraform apply

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

Outputs:

template_path = /var/tmp/roma/terraform-demo/part4/path/dumb.template
```

Local variables

- Terraform’s local variables are analogous to a function's local variables
  - A local variable assigns a name to an expression
    - Useful for complex expressions evaluated repeatedly
- Local variable names must be unique throughout a module
- The value can be any valid Terraform expression

Assigning values to local variables

- Local variables are assigned values using a `locals` block:

```
locals {
  fqdn = "${hostname}${suffix}.${domain}"
}
```

- Each `locals` block can declare as many local variables as needed
- A module may contain any number of `locals` blocks
  - The Terraform documentation recommends grouping related local variable declarations together

Interpolating local variables

- Interpolation of local variables takes place via a reference like `${local.fqdn}`
  - The reference consists of two components:
    - The literal `local`
    - The variable name (`fqdn` in this example)

Interpolating local variables

```
locals {
  gigabyte = "${floor(pow(10,  9))}"
  terabyte = "${floor(pow(10, 12))}"
}

output "gigabyte" {
value = "${local.gigabyte}"
}

output "terabyte" {
value = "${local.terabyte}"
}
```

- Within the locals block, we declare two local variables
  - Each of these variables is assigned the result of an interpolation
- We then define outputs that in turn interpolate the local variables

Interpolating local variables

```
% terraform apply

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

Outputs:

gigabyte = 1000000000
terabyte = 1000000000000
```

Terraform remote state

Remote state

- As we mentioned earlier in this series, Terraform by default stores state in a file named `terraform.tfstate` in the working directory
    - This is not well-suited to multiple people doing development and testing on multiple workstations

Remote state

- Terraform supports remote state storage as a means of maintaining consistent Terraform configurations in a multi-developer environment
    - Terraform supports several remote state backends, including AWS S3, Microsoft Azure, and Google Cloud
    - Many remote state backends support locking, which mitigates problems in a multi-developer environment

Backends are configured within a `terraform` block

```
terraform {
  backend "s3" {
    bucket         = "my_bucket"
    key            = "path/to/my/key"
    region         = "us-east-2"
    encrypt        = true
    dynamodb_table = "my_lock_table"
  }
}
```

- This block specifies the Amazon S3 backend for remote state
- Here, we configure the location of the remote state in Amazon S3
- We want our remote state to be encrypted
- Specifying `dynamodb_table` enables state locking with Amazon's DynamoDB

Terraform backend configuration

- Run `terraform init` when configuring a backend for the first time
- Terraform can easily migrate local state to remote state and vice-versa

Adding Terragrunt to enhance our Terraform experience

Advantages of using Terragrunt to wrap Terraform

- Application Services uses Terragrunt on top of Terraform
  - https://github.com/gruntwork-io/terragrunt
- Terragrunt is a thin wrapper used to keep Terraform configurations DRY†
  - Terragrunt suits Application Services well because we manage dozens of Terraform modules
  - Our multi-developer workflow demands remote state

† DRY = Don't repeat yourself!

Advantages of using Terragrunt to wrap Terraform

- Auto init (runs `terraform init` when necessary)
- Can run Terraform commands on <u>*multiple*</u> infrastructure modules:
    - Terragrunt adds new verbs `plan-all`, `apply-all`, `output-all`, and `destroy-all`
- Terragrunt allows enforcing dependencies between modules
- Terragrunt simplifies configuration of Terraform's built-in remote state and locking

How Application Services uses Terraform and Terragrunt today

Infrastructure hierarchy

- The Terraform infrastructure definitions for each product reside in their own GitHub repository
- We maintain a separate directory hierarchy in GitHub for deploying the various products' infrastructure components
  - Top level nodes are environments (`prod`, `qa`, `test`, etc.)
    - Within each environment are nodes with configuration for individual Terraform modules
- We therefore separate the modules from their environment-specific configuration

Modules defining infrastructure reside in their own repositories

```
foo_app/
    └── main.tf
```

```
deploy
├── prod/
│   └── foo_service/
│       ├── provider.tf -> ../../provider.tf
│       └── terraform.tfvars
├── provider.tf
├── terraform.tfvars
└── test/
    └── foo_service/
        ├── provider.tf -> ../../provider.tf
        └── terraform.tfvars
```

- The node `foo_app` represents a Git repository
  - This repository contains infrastructure definitions for `foo_app`, which is an application

Deployment configuration resides in a hierarchical repository

```
foo_app/
    └── main.tf

deploy
 ├── prod/
 │   └── foo_service/
 │       ├── provider.tf -> ../../provider.tf
 │       └── terraform.tfvars
 ├── provider.tf
 ├── terraform.tfvars
 └── test/
     └── foo_service/
         ├── provider.tf -> ../../provider.tf
         └── terraform.tfvars
```

- The `deploy` repository is a base for deploying services, and is used to maintain versioned infrastructure configuration data
- It contains subdirectories for the `prod` and `test` environments
- Below the `prod` and `test` environment subdirectories are found additional directories for configuration of individual services

Deployment configuration resides in a hierarchical repository

```
foo_app/
    └── main.tf

deploy
├── prod/
│   └── foo_service/
│       ├── provider.tf -> ../../provider.tf
│       └── terraform.tfvars
├── provider.tf
├── terraform.tfvars
└── test/
    └── foo_service/
        ├── provider.tf -> ../../provider.tf
        └── terraform.tfvars
```

- The top-level `deploy` directory contains two configuration files, `provider.tf` and `terraform.tfvars`
- These files apply globally to *all* infrastructure managed within the `deploy` hierarchy
    - We'll come back to these files in a few minutes

Deployment configuration resides in a hierarchical repository

```
foo_app/
    └── main.tf

deploy
├── prod/
│   └── foo_service/
│       ├── provider.tf -> ../../provider.tf
│       └── terraform.tfvars
├── provider.tf
├── terraform.tfvars
└── test/
    └── foo_service/
        ├── provider.tf -> ../../provider.tf
        └── terraform.tfvars
```

- Beneath the `prod` and `test` subdirectories is one directory per service – for example, `foo_service`
- The minimal configuration code in these directories is used by Terragrunt to find the infrastructure module, and to configure it with appropriate per-environment settings

A look at the individual elements within the deploy hierarchy

- We've had a brief look at the deployment hierarchy
  - Now, let's look at the key elements

Top-level `terraform.tfvars` file

```
foo_app/
    └── main.tf

deploy
 ├── prod/
 │    └── foo_service/
 │         ├── provider.tf -> ../../provider.tf
 │         └── terraform.tfvars
 ├── provider.tf
 ├── terraform.tfvars
 └── test/
      └── foo_service/
           ├── provider.tf -> ../../provider.tf
           └── terraform.tfvars
```

- Let's look first at the `terraform.tfvars` file at the top level directory of the `deploy` hierarchy

Top-level `terraform.tfvars` file

```
terragrunt = {
  remote_state {
    backend = "s3"
    config {
      bucket         = "terraform-demo-terragrunt"
      key            = "${path_relative_to_include()}/terraform.tfstate"
      region         = "us-east-2"
      encrypt        = true
      dynamodb_table = "terraform-demo-terragrunt"
    }
  }
}
```

- The `terraform.tfvars` file at the top of the `deploy` tree contains the Terragrunt configuration shared by the whole deploy tree
  - It defines the Terraform backend configuration, including the Amazon S3 location, DynamoDB table name, and encryption options

Top-level `provider.tf` file

```
foo_app/
    └── main.tf

deploy
├── prod/
│   └── foo_service/
│       ├── provider.tf -> ../../provider.tf
│       └── terraform.tfvars
├── provider.tf
├── terraform.tfvars
└── test/
    └── foo_service/
        ├── provider.tf -> ../../provider.tf
        └── terraform.tfvars
```

- We'll now examine the `provider.tf` file at the top level directory of the `deploy` tree
  - This file is referenced through symbolic links in the individual service directories at the bottom of the file hierarchy

Top-level `provider.tf` file

```
provider "aws" {
  region = "us-east-2"
}

terraform {
  backend "s3" {}
}
```

- This is the `provider.tf` file in the top level of the `deploy` hierarchy
- Terragrunt fills in the backend information here from the top-level `terraform.tfvars` file
  - These two files are side-by-side in the `deploy` hierarchy

The `terraform.tfvars` file in the bottom-level directory

```
foo_app/
    └── main.tf

deploy
├── prod/
│   └── foo_service/
│       ├── provider.tf -> ../../provider.tf
│       └── terraform.tfvars
├── provider.tf
├── terraform.tfvars
└── test/
    └── foo_service/
        ├── provider.tf -> ../../provider.tf
        └── terraform.tfvars
```

- Last but not least, we examine the `terraform.tfvars` file in the `deploy/test/foo_service` directory

The `terraform.tfvars` file in the bottom-level directory

```
terragrunt = {
  include {
    path = "${find_in_parent_folders()}"
  }

  terraform {
    source = ".../app/foo_app?ref=develop"
  }
}

#   Specify module configuration.
ami            = "ami-c5062ba0"
instance_count = "1"
instance_type  = "t2.nano"
```

- This `terraform.tfvars` file resides in `deploy/test/foo_service`
- It provides the configuration values that are specific to the instance of `foo_service` to be deployed into the `test` environment
- A counterpart file can be found below the `prod` environment directory as well
  - The other `terraform.tfvars` file likely has different configuration values

The Application Services configuration is a work in progress

- Early on, Application Services implemented a configuration hierarchy for our core infrastructure that we manage with Terraform and Terragrunt
  - Terraform did not initially have support for remote state and locking
- Terraform and Terragrunt have evolved, and new capabilities have been added since our first design
  - We recognize shortcomings in our initial design
  - We see opportunities for improvement based on what we've learned
- Further research is required – stay tuned!

Terraform best practices

Terraform best practices (with an emphasis on AWS)

- <u>Do</u> use remote state with locking
  - Application Services uses
    - Amazon S3 for state
    - Amazon DynamoDB for locking
  - <u>Do</u> enable versioning on the Amazon S3 bucket

Terraform best practices (with an emphasis on AWS)

- <u>Do</u> put your Terraform configuration files in version control
- <u>Do</u> use the same version control best practices for your infrastructure configurations as with your applications
    - Because it's important, we offer some suggestions about version control best practices
        - Because this subject is off-topic to this discussion, we make those suggestions in an appendix

Terraform best practices (with an emphasis on AWS)

- Don’t repeat yourself – use shared modules
  - Do reuse code in “cookie cutter” style in Terraform modules for consistency and reliability
  - Do create a `README.md` (or equivalent) for each module
    - Do describe what the module does
    - Do enumerate the meaning of input and output variables

Terraform best practices (with an emphasis on AWS)

- **Don't** needlessly hardcode values into your configuration files
  - Interpolate data sources and resources where possible
    - Using data from the remote state configured by other Terraform modules is another possibility
      - This uses Terraform data sources
- **Do** use `terraform fmt` to make your configuration files neat

Terraform best practices (with an emphasis on AWS)

- **Do** use `required_version` to specify constraints on the Terraform version
  - Terraform is under intensive development with frequent updates
- **Do** likewise constrain a Terraform provider via the `version` argument in the provider block

```
terraform {
  required_version = ">= 0.10.0"
}
```

```
provider "aws" {
  version = "~> 0.1"
  region  = "${var.region}"
}
```

The `required_version` feature is well-hidden, but is explained at https://www.terraform.io/docs/configuration/terraform.html.

Terraform best practices (with an emphasis on AWS)

- If you use multiple AWS accounts, **do** use the keyword `allowed_account_ids` in the provider block

```
provider "aws" {
  region              = "us-east-2"
  allowed_account_ids = ["0123456789"]
}
```

- Terraform protects you from building, modifying, or removing resources from the wrong Amazon account

```
% terraform plan
------------------------------------------------------------------------
Error running plan: 1 error(s) occurred:

* provider.aws: Account ID not allowed (9876543210)
```

Questions?

Demo code and other resources

We have set up a public repository on GitHub that includes:

- Demonstration code, including most of the examples given in this presentation
- Links to Terraform and Terragrunt distributions and documentation
- Links to useful resources about using these tools

The End

Thank you for your time!

Appendix: Version control best practices

Version control best practices

- <u>Do</u> use Git
  - In other well-known version control systems, branching and merging were feared and therefore seldom-used
- <u>Don't</u> forget that, with Git, a commit is local-only
  - <u>Do</u> remember to push your code to a remote repository

- **Do** branch and merge frequently and fearlessly
    - The ease of branching and merging is a major strength of Git
- **Do** use small, focused commits
    - Don’t mix unrelated changes in the same commit
    - *Each* bug fix, feature addition, etc. can (and often should) be its own branch
    - Most branches should be short-lived!

Version control best practices

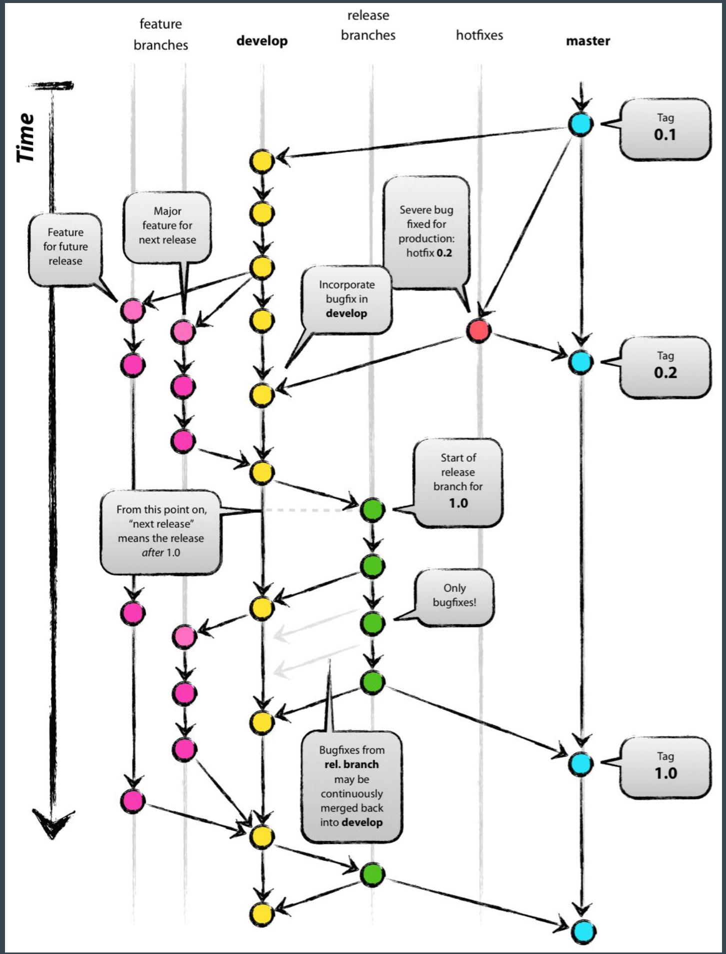
- <u>Do</u> commit and push often
  - Doing so helps collaboration and integration
  - Doing so helps resolve conflicts while they are still bite-sized
- <u>Do</u> avoid committing unfinished or non-working code
  - This can be troublesome in a continuous integration workflow
  - <u>Do</u> use a private branch or use `git stash` if you need to save your work in progress

Version control best practices

- <u>Do</u> write meaningful commit messages
  - Sometimes intensive development results in poor commit message
  - <u>Do</u> get to know the `git rebase` command
- <u>Do</u> agree on a workflow (such as gitflow)
  - See http://nvie.com/posts/a-successful-git-branching-model/
  - There’s a git extension that makes it easy to use gitflow:
    - https://github.com/nvie/gitflow

Gitflow diagrammed

- See the description of gitflow at https://github.com/nvie/gitflow



Version control best practices

- <u>Do</u> test your code regularly
  - Automated test suites can bring big wins
  - Automated deployment can make your life even better
- <u>Do</u> take care to back up your work
  - Git isn’t a substitute for backups
- <u>Do</u> take time to isolate sensitive data from your application code and infrastructure definitions
  - This will also ease transitioning your infrastructure to the cloud