Welcome

Working with GoLang Microservices

Serverless Architectures

Serverless / Functions-as-a-Service (FaaS)

- Represents a type of managed service provided by the CSP
- Cost structure is usually consumption-based (i.e., you only pay for what you use)
- Supports many different coding paradigms (C#/.NET, NodeJS, Python, Go ☺, etc.)
- Typically, with Serverless (and PaaS), the consumer is only concerned with the application code and data – elements of the CSP's "backbone" used to support are managed by the CSP
- Includes more sophisticated automated scaling capabilities – built for Internet scale

AWS Lambda Using Go

- Lambdas can be created in the AWS Management Console
- Allows language selection and testing or can use a blueprint as a starting point
- The in-browser IDE does not support Go which prevents editing in the Management Console

AWS Lambda Using Go

- But we’re able to zip it up, push it to an S3 bucket, and then deploy from there
- Likely a better experience that developing than the MC anyway
- In addition to the Lambda, you’ll want an API Gateway in place as well

AWS Lambda Using Go

- This Gateway will provide an HTTP REST interface to the Lambda’s operations
- The Gateway will be setup as a trigger (it’s one of Lambda’s standard triggers)
- The Gateway includes internal components that can proxy to/from the Lambda

AWS Lambda Using Go

- Go packages have been provided by AWS for coding the Lambda’s handler
- Provide several utility functions for managing function execution and results processing

Terraform

- While these components can be created manually in the MC, it’s not ideal
- IaC (Infrastructure-as-Code) is a better alternative
- Infrastructure code (like all other code) is code
- Can be test, versioned, put in source control, etc.
- Multiple options available for tooling to support

What is Terraform?

- “infrastructure as code”
- *declarative* domain-specific language
  - what is declarative?
- used to describe *idempotent* resource configurations, typically in cloud infrastructure
- according to Hashicorp:
  - *Terraform enables you to safely and predictably create, change, and improve infrastructure. It is an open-source tool that codifies APIs into declarative configuration files that can be shared amongst team members, treated as code, edited, reviewed, and versioned*

What is Terraform? (cont'd)

- open source CLI tool for *infrastructure automation*
- utilizes plugin architecture
  - extensible to any environment, tool, or framework and works primarily by making API calls to those environments, tools, or frameworks
- detects implicit dependencies between resources and automatically creates a dependency graph
- builds in dependency order and automatically performs activities in parallel where possible
  - ...sequentially for dependent resources

HashiCorp
Terraform

Why Use Terraform?

- readable
- repeatable
- certainty (i.e., no confusion about what will happen)
- standardized environments
- provision quickly
- disaster recovery

What Does Terraform (HCL) Look Like?

```
resource "aws_instance" "web" {
  ami                  = "ami-
19827362728"
  instance_type = "t2.micro"

  tags = {
    Name = "my-first-instance"
  }
}
```

Hashicorp Configuration Language (HCL)

- The goal of HCL is to build a structured configuration language that is both human and machine friendly for use with command-line tools, but specifically targeted towards DevOps tools, servers, etc.
- Fully JSON compatible
- Made up of **stanzas** or **blocks**, which roughly equate to JSON objects. Each stanza/block maps to an object type as defined by **Terraform providers** (we'll talk more about providers later)
- https://github.com/hashicorp/hcl

Terraform Project Content Types

`*.tf, *.tf.json`

- HCL or JSON
- these files define your declarative infrastructure and resources

`*.tfstate`

- JSON files that store state, reference to resources
- created and maintained by terraform

`terraform.tfvars, terraform.tfvars.json` and/or `*.auto.tfvars, *.auto.tfvars.json`

- HCL or JSON
- variable definitions in bulk
- (more to come on setting variable values at runtime)

Resources

- `*.tf` files contain your **HCL declarative** definitions

```
resource "aws_instance" "web" {
  ami           = "ami-19827362728"
  instance_type = "t2.micro"

  tags {
    Name = "my-first-instance"
  }
}
```

- most **blocks** in your HCL represent a **resource** to be created/maintained by Terraform

Resources

- *resources* are key elements and captured as top-level objects (stanzas) in Terraform configuration files
- each resource stanza indicates the intent to *idempotently* create that resource
- body of resource contains configuration of attributes of that resource
- each provider (e.g., AWS, Azure, etc.) provides its own set of resources and defines the configuration attributes
- when a resource is created by Terraform, it's tracked in Terraform *state*
- resources can refer to attributes of other resources, creating implicit dependencies
    - dependencies trigger sequential creation

Terraform Commands and the CLI

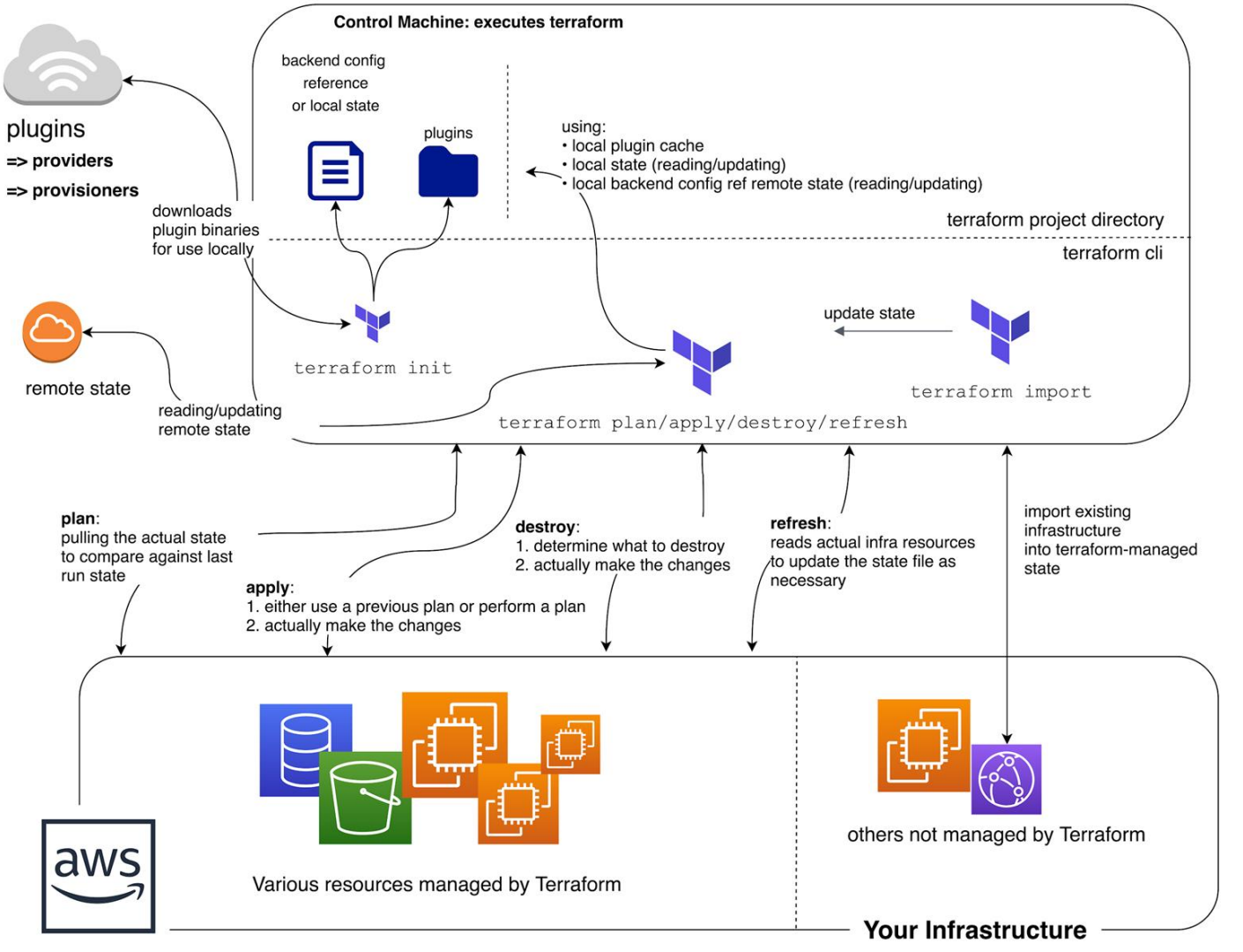
- The CLI is how you'll most often use terraform

```
terraform init …
terraform plan …
terraform apply …
```

- And plenty more: **`terraform --help`** or https://www.terraform.io/docs/commands/index.html
- Third-party SDKs also available for running and interacting with Terraform (e.g., **`scalr`, `terragrunt`, `terratest`**)

Big picture look at

Terraform Command Flow



`terraform init`

- a special command, run before other commands/operations
- what does it do?
  - downloads required provider packages
  - downloads modules referenced in the HCL (more on modules later)
  - initializes state
    - local state: ensuring local state file(s) exist
    - remote state: more complex initialization (more on remote state later)
  - basic syntax check
- idempotent
- remember the **`.terraform`** directory?
  - **`init`** downloads the provider packages and modules to this directory
  - also, where state files live

Input Variables

- enable interchangeable values to be stored centrally and referenced single or multiple times
- similar to variables in other languages
- declared in **`variable`** stanzas
- parsed first
- cannot interpolate or reference other variables
- allow for default values
- optionally specify value type, e.g.,
  - **`List, Map, String`**

Input Variables

- Input variable definitions support the following
  - default – provides default value if not specified; makes optional
  - type – type of value accepted for the variable
  - description – string description/documentation
  - validation – block for defining validation rules for input
  - sensitive – true or false; limits output as part of TF operations (plan or apply)

Example Variable Definition

```
variable "instance_size" {
  default     = "t2.micro"
  type        = string # changed
in 0.12
  description = "Size of EC2
instance"
}
```

Example Variable Definition

```
variable "student_alias" {
  type        = string
  description = "Your student alias"
  validation {
    condition     = trimprefix(var.student_alias, "test") ==
var.student_alias
    error_message = "Please do not use test aliases with this
deployment."
  }
}
```

Data Sources

- logical references to data objects stored externally to the `tfstate` file
- allows you to reference resources not created by Terraform
- examples
  - current default region in AWS CLI
  - AMI ID search
  - AWS ARN lookup
  - AWS VPC CIDR range

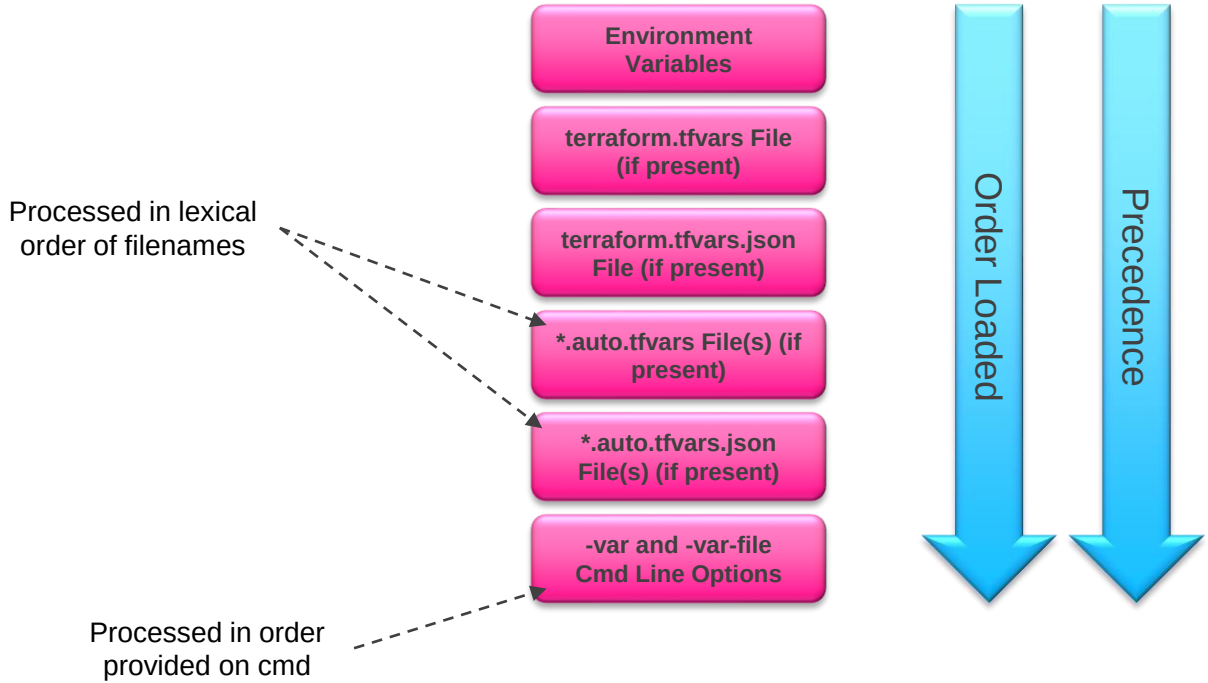
Data Source Example: AWS AMI Lookup

```
data "aws_ami" "latest-ubuntu" {
  most_recent = true
  owners = ["099720109477"]

  filter {
    name   = "name"
    values = ["ubuntu/images/hvm-ssd/ubuntu-xenial-16.04-amd64-
server-*"]
   }

  filter {
    name   = "virtualization-type"
    values = ["hvm"]
  }
}
```

Providing Values for Input Variables

- Multiple options
  - using environment variables (prefixed with “TF_VAR_”)
  - defining inputs in a terraform.tfvars file
  - defining inputs in a terraform.tfvars.json file
  - defining inputs in one or more *.auto.tfvars files
  - defining inputs in one or more *.auto.tfvars.json files
  - -var and -var-file options on the command-line

Providing Values for Input Variables



Providing Values for Input Variables

- primarily used when executing Terraform via CLI
- not really used with Terraform Enterprise
- can “push” those variables + values to Enterprise (in files)
- but manage from “Variables” section of the environment

State

- stores information about resources that are created by Terraform
  - also includes values computed by the provider APIs

- local file
  - `.tfstate`

- or backends are also available…

Backends

- determines how state is loaded and how operations like **`apply`** are executed

- enables non-local file state storage, remote execution, etc.

- why use a backend?
  - can store their state remotely and protect it to prevent corruption
    - some backends, e.g., *Terraform Cloud* automatically store all revisions
  - keep sensitive information off local disk
  - remote operations
    - apply can take a *LONG* time for large infrastructures

Backends (cont’d)

- examples
  - S3
  - swift
  - http
  - Terraform Enterprise
  - etc.

Providers

- responsible for understanding API interactions and exposing resources
- Hashicorp helps companies create providers to be added to ecosystem
- declared in HCL config files as a **`provider`** stanza
- each Terraform project can have multiple providers, even of the same type
- describes resources, their inputs, outputs, and the logic to create and change them
- many options
  - AWS, GCP, Azure, and many many others
  - providers available for non-infra services as well such as gmail, MySQL, and Pagerduty

The AWS Provider

- provider documentation
  - https://www.terraform.io/docs/providers/aws/index.html

- HUGE amount of resources

- something like 8 resources per service on average

Configuring the Provider

```
provider "aws" {
  region     = "us-west-1"
  access_key = "[your access key]"
  secret_key = "[your secret access
key]"
}
```

Output Variables

- *inputs* to a Terraform config are declared with variables stanzas
- *outputs* are declared with a special output stanza
- can be referenced through the modules interface or the CLI

Output Variables

- Output variable definitions support the following
  - value – value to be returned as output
  - description – string description/documentation
  - sensitive – true or false; limits output as part of TF operations (plan or apply)

Output Definition

```
output "instance_public_ip" {
  value = aws_instance.web.public_ip
}
```

Demo

https://github.com/ludesdeveloper/terraform-lambda-golang

Thank you!

If you have additional questions, please reach out to me at: (asanders@gamuttechnologysvcs.com)

DevelopIntelligence
A PLURALSIGHT COMPANY