Show Submission Credentials

P0. Infrastructure as Code Infrastructure as Code

- ✔ Introduction
- ✔ Getting Started with Terraform
- ✔ Basic EC2 Deployment - AWS
- ✔ Basic VM Deployment - Azure
- ✔ Basic VM Deployment - GCP
- ✔ Conclusion

Introduction

Infrastructure As Code

> Infrastructure as code (IaC) is the process of managing and provisioning data center infrastructure through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools. -- Infrastructure as Code - Wikipedia (https://en.wikipedia.org/wiki/Infrastructure_as_Code)

Information

Terraform will be a mandatory task in several projects.

In the age of Cloud Computing, Infrastructure as Code and automation tools have very much been born out of necessity. Managing large scale deployments to cloud infrastructure (and on-premise datacenters) quickly become unwieldy without processes that are repeatable, auditable, manage via source control, versioned, and continuously tested and released. In industry, many of the infrastructure automation responsibilities may belong to an IT Operations department, although the DevOps role - a Software Engineering practice that embraces both software development and operations - has been gaining popularity in many corporations.

Infrastructure as code tools provide a high level software interface (Python / Ruby or JSON / YAML) that allows developers to specify their infrastructure requirements, software dependencies, and the process for building the infrastructure and deploying it to the cloud. There are many tools to choose from, but in general they attempt to codify good software engineering practices and provide engineers with some of the following benefits.

Benefits

- **Cost reduction** - Manual processes consume developer time, which may be better utilized on other engineering tasks. By automating these manual tasks, developers can dedicate more of their time to engineering tasks.
- **Immutability and Versioning** - While not every Infrastructure as Code tool embraces immutability it is an important concept in cloud infrastructure. Immutable infrastructure (https://www.digitalocean.com/community/tutorials/what-is-immutable-infrastructure) provides the benefit of always being able to identify the state (application version, packages installed, OS versions) of your infrastructure. Additionally, immutability can help to ensure parity between developer, staging, and production environments. Every time that an infrastructure change is introduced, the principle of immutability suggests we should build an orchestration new version of our infrastructure that is uniquely identifiable (by applying labels containing the Git commit id, the build id, or other version number).
- **Risk reduction** - In addition to reducing the time burden on developers, infrastructure automation allows for repeatable processes. This means that solutions to issues experienced in the past are codified in your infrastructure automation process, rather than documentation or tribal knowledge.
- **Reproducibility** - In the case of large scale production failures or infrastructure migrations, it is highly beneficial to be able to re-create your infrastructure from a source configuration.
- **Speed of innovation** - Being able to rely on a repeatable and error proof development process allows engineers to make changes with confidence that their changes will not cause production issues. Any issues identified in production can be quickly un-done and prevented in the future by amending the automated deployment process.

Available Tools

Selecting which set of tools to use depends on the size and complexity of the infrastructure you wish to manage and the types of tasks you wish to automate. There are many tools that allow us to fulfill infrastructure automation capabilities and utilize Infrastructure as Code,

varying from home grown solutions, to open source frameworks, and paid solutions. Some of these tools include:

- Bash / Python / Ruby scripts - Custom solutions using cloud APIs
- Ansible (https://www.ansible.com/) - Infrastructure automation, configuration management, and orchestration
- Chef (https://www.chef.io/chef/) - Configure, deploy, and manage infrastructure
- Docker (https://www.docker.com/), Docker Compose (https://docs.docker.com/compose/), and Kubernetes (https://kubernetes.io/) - Containerization
- Packer (https://www.packer.io/) - Automated machine image creation
- Terraform (https://www.terraform.io/) - Reproducible, cross cloud Infrastructure as Code

In the remainder of this primer, we will be using Terraform (https://www.terraform.io/) to ease the configuration and deployment of cloud infrastructure. You are encouraged to explore other infrastructure automation tools as they will be helpful in reducing the burden of repeatable tasks.

Getting Started with Terraform

Terraform

Terraform (https://www.terraform.io/) is a popular open source tool, that takes a declarative approach to orchestration of cloud resources in a reliable and consistent manner. Taking advantage of Terraform will allow you to write a configuration file that describes what infrastructure you want to setup, without having to describe how to set it up. This means resources can be launched in parallel and errors are handled by the system software. Further, with Terraform your solution shares a common abstraction among cloud providers, such as AWS, Azure, and Google Cloud. A full list of supported providers is in the Terraform providers documentation (https://www.terraform.io/docs/providers/index.html).

Notice that taking advantage of Terraform may require you to have a mix of traditional code (such as Boto3 with Python) with Terraform configuration files. Libraries such as python3-terraform (https://github.com/beelit94/python-terraform) allow you to easily integrate Terraform with languages such as Python.

Installing Terraform

Download the appropriate binary package (https://www.terraform.io/downloads.html) for your system. After downloading Terraform, unzip the package. Terraform runs as a single binary named terraform. Any other files in the package can be safely removed and Terraform will still function.

Add the terraform binary to your path so that you run it from any location. Follow these steps for setting the PATH on Linux and Mac (https://stackoverflow.com/questions/14637979/how-to-permanently-set-path-on-linux-unix) or Windows (https://stackoverflow.com/questions/1618280/where-can-i-set-path-to-make-exe-on-windows).

For Linux, you may download the latest Terraform release and move the `terraform` binary to be on the shell PATH:

```
version="<a.b.c>" # find the latest version from https://www.terraform.io/downlo
ads.html
# you should refer to https://releases.hashicorp.com/terraform/ to get the lates
t version
wget https://releases.hashicorp.com/terraform/"$version"/terraform_"$version"_li
nux_amd64.zip
unzip terraform_"$version"_linux_amd64.zip -d /usr/local/bin
```

HashiCorp Configuration Language (HCL) and Terraform Configurations

Terraform configuration file syntax is based on the HashiCorp Configuration Language (https://github.com/hashicorp/hcl). It is meant to strike a balance between human readable and editable as well as being machine-friendly. Terraform can also support JSON configurations, however, it is recommended to use the HCL Terraform syntax. One of the main reasons for developing and adopting HCL is because JSON doesn't support comments. More details about the syntax can be found here (https://www.terraform.io/docs/configuration/syntax.html).

Command Line

Terraform is controlled via a command-line interface (CLI) using a single command-line application: terraform. This application takes various subcommands such as "apply" or "plan". The terraform CLI is a well-behaved command line application. In erroneous cases, a non-zero exit status will be returned. To view a list of the available commands at any time, run `terraform` with no arguments.

Common commands:

- **apply** - Builds or changes infrastructure
- **destroy** - Destroy Terraform managed infrastructure
- **plan** - Generate and show an execution plan

State

Everytime you run Terraform, it stores state about your managed infrastructure and configuration. This state is used by Terraform to map real world resources to your configuration, keep track of metadata, and to improve performance for large infrastructures. This is used to create plans and add incremental changes to your infrastructure.

In this primer, we can use local state files, however you should explore the Terraform documentation (https://www.terraform.io/docs/state/purpose.html) to learn about other options for storing infrastructure state.

Providers

Terraform is used to create, update, manage and deploy infrastructure resources such as VMs, containers, storage and more. Almost any infrastructure type can be represented as a resource in Terraform. A provider is responsible for understanding API interactions and

exposing resources. (e.g. AWS, GCP, Microsoft Azure, OpenStack), PaaS (e.g. Heroku), or SaaS services (e.g. Terraform Enterprise, DNSimple, CloudFlare). The complete list of providers that are supported by Terraform can be found in the providers reference (https://www.terraform.io/docs/providers/) .

Provisioners

Provisioners allow execution of scripts on a local or remote machine as part of resource creation or destruction. They can also be used to bootstrap a resource, cleanup before destroy, run configuration management, etc. For example, they can be used to run a bash script that starts up a web service upon instance creation (might be useful in the team project). More information about provisioners can be found in the provisioners documentation (https://www.terraform.io/docs/provisioners/index.html).

Basic EC2 Deployment - AWS

Basic EC2 Deployment - AWS

In this section we will introduce how to create cloud infrastructure, particularly an EC2 instance on AWS using Terraform. Before you start writing your Terraform files, note that Terraform will automatically search for saved credentials (for example, in ~/.aws/credentials) or IAM instance profile credentials. If you haven't done so, you can follow the AWS instructions (https://aws.amazon.com/blogs/security/a-new-and-standardized-way-to-manage-credentials-in-the-aws-sdks/) on configuring credentials.

Danger

Don't forget to make sure that all your instances are properly tagged with the current week's project!

Develop a Terraform Configuration

This is a simple Terraform configuration file that creates a t2.micro instance in the us-east-1 region.

- Provider - The AWS Provider (https://www.terraform.io/docs/providers/aws/index.html) is used to indicate that Terraform should use the AWS APIs to launch resource.
- Resource - The aws_instance (https://www.terraform.io/docs/providers/aws/r/instance.html) resource declares the properties of the EC2 instance we wish to launch.
    - Many of the resource's configuration variables have default values. Refer to the resource documentation if you wish to customize the instance further.

```
# terraform init      Initialize a Terraform working directory
# terraform validate  Validates the Terraform files
# terraform fmt       Rewrites config files to canonical format
# terraform plan      Generate and show an execution plan
# terraform apply     Builds or changes infrastructure
# terraform destroy   Destroy Terraform-managed infrastructure

provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "cmucc" {
  ami           = "ami-2757f631"
  instance_type = "t2.micro"

  tags = {
    project = "getting-started-with-cloud-computing"
  }

  # You may optionally specify a SSH key that exists in your AWS account
  # key_name = "my-ssh-key"
}
```

This is a complete configuration that can be applied to AWS by Terraform.

The provider block is used to configure the named provider, in this case "aws". A provider is responsible for creating and managing resources. Multiple provider blocks can exist if a Terraform configuration is composed of multiple providers, which is a common situation.

The resource block defines a resource that exists within the infrastructure, in this case an EC2 instance.

The resource block has two strings before opening the block: the resource type and the resource name. In this example, the resource type is `aws_instance` and the name is `cmucc`. The prefix of the type maps to the provider. In this case, `aws_instance` automatically tells Terraform that it is managed by the "aws" provider.

Give the file a name and save it with a **.tf** extension.

Apply the Terraform Configuration

The first time you are working with a new Terraform configuration you must run ***terraform init*** which downloads a plugin based on the providers you are using.

You should use the commands `terraform validate / terraform fmt` to validate and format the Terraform files before you apply the terraform. Besides, it is recommended to use Jetbrains HashiCorp Terraform / HCL language support plugin (https://plugins.jetbrains.com/plugin/7808-hashicorp-terraform--hcl-language-support) to audit the syntax. Jetbrains IDEs offer free licenses (https://www.jetbrains.com/student/) to students and faculty if you own a university email address.

In the same directory as the above configuration file, run `terraform plan` and `terraform apply` You should see output similar to this:

```
# If there are no existing Terraform state files or downloaded modules run `init
`
$ terraform init
...

$ terraform validate # no output if the validation passed

$ terraform plan
...
Terraform will perform the following actions:

  # aws_instance.cmucc will be created
  + resource "aws_instance" "cmucc" {
      + ami                          = "ami-2757f631"
      + arn                          = (known after apply)
      + associate_public_ip_address  = (known after apply)
      + availability_zone            = (known after apply)
      + cpu_core_count               = (known after apply)
      + cpu_threads_per_core         = (known after apply)
      + get_password_data            = false
      + host_id                      = (known after apply)
      + id                           = (known after apply)
      + instance_state               = (known after apply)
      + instance_type                = "t2.micro"
      + ipv6_address_count           = (known after apply)
      + ipv6_addresses               = (known after apply)
      + key_name                     = (known after apply)
      + network_interface_id         = (known after apply)
      + password_data                = (known after apply)
      + placement_group              = (known after apply)
      + primary_network_interface_id = (known after apply)
      + private_dns                  = (known after apply)
      + private_ip                   = (known after apply)
      + public_dns                   = (known after apply)
      + public_ip                    = (known after apply)
      + security_groups              = (known after apply)
      + source_dest_check            = true
      + subnet_id                    = (known after apply)
      + tags                         = {
          + "project" = "getting-started-with-cloud-computing"
        }
      + tenancy                      = (known after apply)
      + volume_tags                  = (known after apply)
      + vpc_security_group_ids       = (known after apply)

      + ebs_block_device {
          + delete_on_termination = (known after apply)
          + device_name           = (known after apply)
          + encrypted             = (known after apply)
          + iops                  = (known after apply)
          + kms_key_id            = (known after apply)
          + snapshot_id           = (known after apply)
          + volume_id             = (known after apply)
          + volume_size           = (known after apply)
          + volume_type           = (known after apply)
```

```
        }

      + ephemeral_block_device {
          + device_name  = (known after apply)
          + no_device    = (known after apply)
          + virtual_name = (known after apply)
        }

      + network_interface {
          + delete_on_termination = (known after apply)
          + device_index          = (known after apply)
          + network_interface_id  = (known after apply)
        }

      + root_block_device {
          + delete_on_termination = (known after apply)
          + encrypted             = (known after apply)
          + iops                  = (known after apply)
          + kms_key_id            = (known after apply)
          + volume_id             = (known after apply)
          + volume_size           = (known after apply)
          + volume_type           = (known after apply)
        }
    }

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


$ terraform apply
...
```

This is the execution plan which describes the actions Terraform is going to take for you. If this step is successful, terraform will ask for your permission before actually going ahead and creating these resources for you.

The deployment operation can then be validated by visiting the AWS Dashboard and verifying that the EC2 instance defined in the Terraform file has been created.

Change cloud resources

As described earlier, Terraform maintains the state of your infrastructure. This means that you can use Terraform to change your infrastructure and version control not only your configurations but also your state so that you can see how the infrastructure evolved over time.

Modify the ami used earlier to ***ami-b374d5a5*** and run the following commands to observe the changes Terraform will make:

```
$ terraform validate
# Review changes that are planned
$ terraform plan
....

$ terraform apply
...
```

The AMI change can be validated by verifying the EC2 instance description on the AWS Dashboard.

Destroy cloud resource

Since Terraform maintains state, you can use the `destroy` command to clean up any cloud infrastructure that is managed by Terraform. After running this command, you must review the resources that are to be destroyed and follow the prompts:

```
$ terraform destroy
...
```

Terraform Resource Dependency Graph

In this section, we are going to introduce resource dependencies, where we'll not only see a configuration with multiple resources for the first time but also scenarios where resource parameters use information from other resources.

In the previous section, the example configuration had a single resource, however, real infrastructure has a diverse set of resources and resource types. It is a common scenario that Terraform configurations contain multiple resources, multiple resource types, and these types even span multiple providers.

We will show you a basic example of multiple resources and how to reference the attributes of other resources to configure subsequent resources.

You should know from the previous study that a VM resource is created with its associated resources such as a security group, a public IP address, a virtual network etc. These resources are configured when you select them on the AWS portal or use CLI where it will apply the default value if you don’t manually set the argument in command.

Create and apply the following terraform connfiguration as done in the previous step:

```
# terraform init      Initialize a Terraform working directory
# terraform validate  Validates the Terraform files
# terraform fmt       Rewrites config files to canonical format
# terraform plan      Generate and show an execution plan
# terraform apply     Builds or changes infrastructure
# terraform destroy   Destroy Terraform-managed infrastructure

provider "aws" {
  region = "us-east-1"
}

resource "aws_security_group" "student_ami_sg" {
  # inbound internet access
  # allowed: only port 22, 80 are open
  # you are NOT allowed to open all the ports to the public
  ingress {
    from_port = 22
    to_port   = 22
    protocol  = "tcp"

    cidr_blocks = [
      "0.0.0.0/0"
    ]
  }

  ingress {
    from_port = 80
    to_port   = 80
    protocol  = "tcp"

    cidr_blocks = [
      "0.0.0.0/0"
    ]
  }

  # outbound internet access
  # allowed: any egress traffic to anywhere
  egress {
    from_port = 0
    to_port   = 0
    protocol  = "-1"

    cidr_blocks = [
      "0.0.0.0/0"
    ]
  }
}

resource "aws_instance" "cmucc" {
  ami           = "ami-2757f631"
  instance_type = "t2.micro"

  vpc_security_group_ids = [aws_security_group.student_ami_sg.id]

  tags = {
```

```
    project = "getting-started-with-cloud-computing"
  }
  # You may optionally specify a SSH key that exists in your AWS account
  # key_name = "my-ssh-key"
}
```

After the `terraform apply` command is complete, we can use the `terraform graph` command to generate a dependency graph. The output of this command can be supplied to this website (http://www.webgraphviz.com/) for visualization.

Terraform Variables

In this section, you will learn about input and output variables in Terraform. You will create a resource group, configure a VM with a prepared example, identify VM metadata and manipulate the output variable to get the public IP of your VM.

Create a terraform configuration file named `variables.tf` to define input variables and copy the following snippet to it:

```
variable "region" {
  default = "us-east-1"
}

# instance type
variable "instance_type" {
  default = "t2.micro"
}

# Update "project_tag" to match the tagging requirement of the ongoing project
variable "project_tag" {
}

# Update "ami_id"
variable "ami_id" {
}

# Update "key_name" with the key pair name for SSH connection
# Note: it is NOT the path of the pem file
# you can find it in https://console.aws.amazon.com/ec2/v2/home?region=us-east-1
#KeyPairs:sort=keyName
# variable "key_name" {
# }
```

Create the `main.tf` file as follows:

```
# TODO: Create a file named `terraform.tfvars` and set the values of the variabl
es defined in `variables.tf`

# terraform init      Initialize a Terraform working directory
# terraform validate  Validates the Terraform files
# terraform fmt       Rewrites config files to canonical format
# terraform plan      Generate and show an execution plan
# terraform apply     Builds or changes infrastructure
# terraform destroy   Destroy Terraform-managed infrastructure

provider "aws" {
  region = var.region
}

resource "aws_security_group" "student_ami_sg" {
  # inbound internet access
  # allowed: only port 22, 80 are open
  # you are NOT allowed to open all the ports to the public
  ingress {
    from_port = 22
    to_port   = 22
    protocol  = "tcp"

    cidr_blocks = [
      "0.0.0.0/0",
    ]
  }

  ingress {
    from_port = 80
    to_port   = 80
    protocol  = "tcp"

    cidr_blocks = [
      "0.0.0.0/0",
    ]
  }

  # outbound internet access
  # allowed: any egress traffic to anywhere
  egress {
    from_port = 0
    to_port   = 0
    protocol  = "-1"

    cidr_blocks = [
      "0.0.0.0/0",
    ]
  }
}

resource "aws_instance" "cmucc" {
  ami           = var.ami_id
  instance_type = var.instance_type
```

```
  vpc_security_group_ids = [aws_security_group.student_ami_sg.id]

  tags = {
    Project = var.project_tag
  }

  # You may optionally specify a SSH key that exists in your AWS account
  # key_name = var.key_name
}
```

Note that the values for region, ami_id, instance_type and project_tag introduces a new syntax: var.\<variable-name\> OR \<resource-name\>.\<property\>. This is a powerful feature that allows you to reference variables (E.g.: `var.region`), attributes of resources (E.g.: `aws_security_group.student_ami_sg.id`), call functions, and create conditionals that branch based on value.

The values of the variables defined in variables.tf must be set in a .tfvars file. Create a file named `terraform.tfvars`. A sample of `terraform.tfvars` is as follows:

```
instance_type="t2.micro"
project_tag="getting-started-with-cloud-computing"
ami_id="ami-2757f631"
```

Now we add an output variable in an `outputs.tf` file to output the public IP of the new VM, in this way, you don't need to go to some specific page in the AWS portal to get the public IP.

```
output "instance_ip_addr" {
  value       = aws_instance.cmucc.private_ip
  description = "The private IP address of the instance."
}
```

The steps are the same as before. Run `terraform init` first, and then `terraform plan` to see the actions that will be taken by Terraform, and run `terraform apply` to build the real infrastructure.

After the apply command has completed, you can go to the AWS dashboard to verify the output variable (VM IP) received from Terraform.

Advanced Terraform Usage

The Terraform developers have provided more intricate examples of Terraform configurations using the AWS provider. These examples may be found in the terraform-providers/terraform-provider-aws (https://github.com/terraform-providers/terraform-provider-aws/tree/master/examples) GitHub repository.

We recommend exploring the Terraform documentation and running the provided AWS examples to gain more understanding of developing complex configurations.

Danger

Don't forget to delete (`terraform destroy`) all resources deployed using different terraform templates when you are done.

Basic VM Deployment - Azure

Terraform with Azure

Following is a Terraform template for basic VM deployments on Azure:

```
terraform {
        required_providers {
            azurerm = {
            source  = "hashicorp/azurerm"
            version = "=2.91.0"
            }
    }
}

provider "azurerm" {
    features {}
}

# Create a resource group if it doesn’t exist
resource "azurerm_resource_group" "terraform_rg" {
    name     = "terraform_primer_rg"
    location = "eastus"

tags = {
    environment = "terraform_primer"
    project     = "getting-started-with-cloud-computing"
    }
}

# Create virtual network
resource "azurerm_virtual_network" "terraform_network" {
    name                = "terraform_primer_vnet"
    address_space       = ["10.0.0.0/16"]
    location            = "eastus"
    resource_group_name = azurerm_resource_group.terraform_rg.name

    tags = {
        environment = "terraform_primer"
        project     = "getting-started-with-cloud-computing"
    }
}

# Create subnet
resource "azurerm_subnet" "terraform_subnet" {
    name                 = "terraform_primer_subnet"
    resource_group_name  = azurerm_resource_group.terraform_rg.name
    virtual_network_name = azurerm_virtual_network.terraform_network.name
    address_prefixes     = ["10.0.1.0/24"]
}

# Create public IPs
resource "azurerm_public_ip" "terraform_ip" {
    name                = "terraform_primer_ip"
    location            = "eastus"
    resource_group_name = azurerm_resource_group.terraform_rg.name
    allocation_method   = "Static"

    tags = {
        environment = "terraform_primer"
        project     = "getting-started-with-cloud-computing"
```

```
    }
}

# Create Network Security Group and rule
resource "azurerm_network_security_group" "terraform_sg" {
    name                = "terraform_primer_sg"
    location            = "eastus"
    resource_group_name = azurerm_resource_group.terraform_rg.name

    security_rule {
        name                       = "SSH"
        priority                   = 1001
        direction                  = "Inbound"
        access                     = "Allow"
        protocol                   = "Tcp"
        source_port_range          = "*"
        destination_port_range     = "22"
        source_address_prefix      = "*"
        destination_address_prefix = "*"
}

    tags = {
        environment = "terraform_primer"
        project     = "getting-started-with-cloud-computing"
    }
}

# Create network interface
resource "azurerm_network_interface" "terraform_nic" {
    name                      = "terraform_primer_nic"
    location                  = "eastus"
    resource_group_name       = azurerm_resource_group.terraform_rg.name

    ip_configuration {
        name                          = "terraform_nic_conf"
        subnet_id                     = azurerm_subnet.terraform_subnet.id
        private_ip_address_allocation = "Dynamic"
        public_ip_address_id          = azurerm_public_ip.terraform_ip.id
    }

    tags = {
        environment = "terraform_primer"
        project     = "getting-started-with-cloud-computing"
    }
}

resource "azurerm_subnet_network_security_group_association" "sec-group-associat
ion" {
    subnet_id                 = azurerm_subnet.terraform_subnet.id
    network_security_group_id = azurerm_network_security_group.terraform_sg.id
}

# Create virtual machine
resource "azurerm_virtual_machine" "terraform-vm" {
    name                  = "terraform_primer_vm"
    location              = "eastus"
```

```
    resource_group_name   = azurerm_resource_group.terraform_rg.name
    network_interface_ids = [azurerm_network_interface.terraform_nic.id]
    vm_size               = "Standard_DS1_v2"

    storage_os_disk {
        name              = "terraform_os_disk"
        caching           = "ReadWrite"
        create_option     = "FromImage"
        managed_disk_type = "Premium_LRS"
    }

    storage_image_reference {
        publisher = "Canonical"
        offer     = "UbuntuServer"
        sku       = "16.04.0-LTS"
        version   = "latest"
    }

    os_profile {
        computer_name  = "terraform-vm"
        admin_username = "azureuser"
        # TODO: Create a terraform.tfvars file and a variables.tf file to set th
e password
        admin_password = var.azure_password
    }

    os_profile_linux_config {
        disable_password_authentication = false
    }

    tags = {
        environment = "terraform_primer"
        project     = "getting-started-with-cloud-computing"
    }
}
```

We recommend experimenting with this template on Azure.

Note: Terraform will automatically search for Azure credentials when used with Azure as a provider. If you haven't set up these credentials in your environment through the CLI ('az login'), refer to the corresponding Azure Primer.

Danger

Don't forget to delete (`terraform destroy`) all resources deployed using different terraform templates when you are done.

Basic VM Deployment - GCP

Terraform with GCP

Following is a Terraform template for basic VM deployments on GCP

```
# Terraform template for deploying a VM on GCP
# TODO: Update the project Id under which the instance must be created
# TODO:  Enable the Compute Engine APIs for the project thorugh the browser dash
board if they have not been enabled.

provider "google" {
  region = "us-east1"
}

resource "google_compute_instance" "student_instance" {
  name         = "gcp-student-instance"
  machine_type = "n1-standard-1"
  zone         = "us-east1-b"

  # Replace this value with the Project ID of an already existing project on GC
P.
  # If there are no projects on GCP, create a new one using CLI/Web Dashboard an
d update the Project ID here.
  # Please note that project ID is different from project name on GCP.
  project = "terraform-primer-123456"

  boot_disk {
    initialize_params {
      image = "debian-cloud/debian-9"
      size  = 10
    }
  }
  network_interface {
    network = "default"
    access_config {
      # Ephemeral IP, leaving this block empty will generate a new external IP a
nd assign it to the machine
    }
  }
  labels = {
    # Only hyphens (-), underscores (_), lowercase characters, and numbers are a
llowed.
    project = "getting-started-with-cloud-computing"
  }
}
```

We recommend experimenting with this template on GCP.

Note: Terraform will automatically search for GCP credentials when used with Google as the provider. If you haven't set up these credentials in your environment through the CLI ('gcloud auth application-default login'), refer to the corresponding GCP Primers.

Danger

Don't forget to delete (`terraform destroy`) all resources deployed using different terraform templates when you are done.

Conclusion

Conclusion

Now that you have gained an understanding of the basic concepts of IaC, you are encouraged to play around with Terraform to automate provisioning of infrastructure enabling best practices in DevOps (https://en.wikipedia.org/wiki/DevOps).