Terraform Workflow

Terraform is an open source infrastructure as code (IAC) tool. It allows us to build, change and version the infrastructure efficiently. It allows us to declaratively define a set of configuration files. We can evolve our infrastructure as move on to provisioning different cloud resources. For example, you may want to create a VPC and on top of it, you want to provision security group rules and within that environment, we want to spin up some virtual machines. With terraform, we define the high-level VPC resources and the fields that are required to create the element of our infrastructure. It also allows us to refer other components of our infrastructure. Terraform gives us a workflow for creating and managing our infrastructure resources.

There are three main commands that we run, when we want to provision an Infrastructure.

- `terraform init` : This command is used to initialize the working directory containing terraform configuration files. It is safe to run multiple times. The command will never delete your existing infrastructure or a state. During initialization, it searches the terraform configuration files for the providers defined. These providers are published as a plugins. Once it finds the providers, it tries to download and install the providers in the terraform directory.
- `terraform plan` : With this command, terraform figures out what it needs to do. It sees the real provisioned resources and compares it to the desired configuration. If Terraform detects that no changes are needed to resource instances or to root module output values, `terraform plan` will report that no actions need to be taken. You can use the optional `-out=FILE` option to save the generated plan to a file on disk.
- `terraform apply` : This command provisions the resources proposed in `terraform plan` . Before actually applying the plan, it prompts us to confirm the changes. We can also pass the filename of a saved plan file created earlier with `terraform plan -out=...`. In this case Terraform will apply the changes in the plan without any confirmation prompt.

When we provide the terraform configurations, the terraform builds a dependency graph. By looking into dependency graph, it can make sure that, the resources are provisioned accordingly. Whenever we run the `terraform plan` and `terraform apply`, the terraform looks into dependency graph to generate the plans and refresh the states.

Workflow in Action:

1. Create a new directory and write the configuration Files.

```
$ mkdir terraform-in-action && cd terraform-in-action

$ touch main.tf
```

Add the below code in **main.tf**

```
 1 terraform {
 2   required_providers {
 3     aws = {
 4       source  = "hashicorp/aws"
 5       version = "3.50.0"
 6     }
 7   }
 8 }
 9 
10 provider "aws" {
11   # Configuration options
12   region                  = var.region
13   profile                 = var.aws_profile
14   shared_credentials_file = var.shared_credentials_file
15   default_tags {
16     tags = var.tags
17   }
18 }
```

You might see an error on the line `12`,`13`,`14`. It is because, we are referencing the variables, which are not present. So let us create a new file **variable.tf**

```
$ touch variables.tf
```

Add the below code in **variables.tf**

```
variable "region" {
  description = "Deployment Region"
  default     = "ap-south-1"
}

variable "aws_profile" {
  description = "Given name in the credential file"
  type        = string
  default     = "rahul-admin"
}

variable "shared_credentials_file" {
  description = "Profile file with credentials to the AWS account"
  type        = string
  default     = "~/.aws/credentials"
}

variable "tags" {
  description = "A map of tags to add to all resources."
  type        = map(string)
  default = {
    application = "Learning-Tutor"
    env         = "Test"
  }
}
```

Here you can define the region and aws_profile. We have to point to the location of credentials of the AWS account. We can also define the tags.

Let us now create the new file **lambda.tf** to create the lambda function.

```
$ touch lambda.tf
```

Add the below code in **lambda.tf**

```
module "profile_generator_lambda" {
  source  = "terraform-aws-modules/lambda/aws"
  version = "2.7.0"
  # insert the 28 required variables here
  function_name = "profile-generator-lambda"
  description   = "Generates a new profiles"
  handler       = "index.handler"
  runtime       = "nodejs14.x"
  source_path   = "${path.module}/resources/profile-generator-lambda"
  tags = {
    Name = "profile-generator-lambda"
  }
}
```

- ***source*** field is used to indicate the terraform module which we intend to use.
- ***function_name*** field is the unique name for our lambda function.
- ***handler*** field is the Lambda Function entry point for our code.
- ***runtime*** field is Lambda Function runtime
- ***source_path*** field is the absolute path to a local file or directory containing our Lambda source code

Let us now create a file **index.js** in the folder **resources/profile-generator-lambda**

```
$ mkdir resources/profile-generator-lambda
$ touch index.js
```

Add the below code in **index.js**

```
const faker = require("faker/locale/en_IND");

exports.handler = async (event, context) => {
  let firstName = faker.name.firstName();
  let lastName = faker.name.lastName();
  let phoneNumber = faker.phone.phoneNumber();
  let vehicleType = faker.vehicle.vehicle();

  let response = {
    firstName: firstName,
    lastName: lastName,
    phoneNumber: phoneNumber,

    vehicleType: vehicleType,
  };

  return {
    statusCode: 200,
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify({
      profile: response,
    }),
  };
};
```

Let us initialize the folder **resources/profile-generator-lambda** with **npm** and also install the **faker** dependency which is used in our lambda code to get the random profiles.

```
$ npm init && $ npm install faker
```

2. Terraform Init

At the root of the folder, let us now initialize the directory, which will download all the providers and modules used in the configuration.

```
$ cd ../../
$ terraform init
```

3. Terraform plan

Let us check if a plan matches the expectation and also store the plan in output file ***plan-out***

```
$ terraform plan --out "plan-out"
```

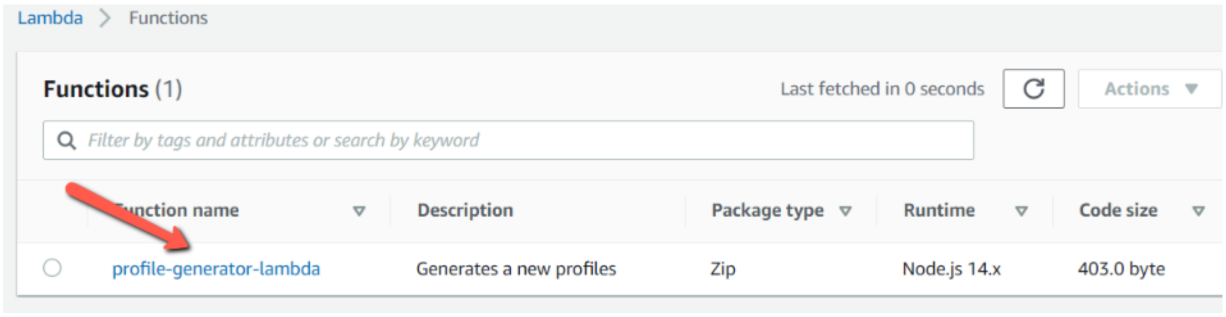
4. Terraform apply

Once the plan is verified, apply the changes to get the desired infrastructure components.

```
$ terraform apply "plan-out"
```

Now, let us verify the infrastructure created on AWS console.

- **Lambda Function**



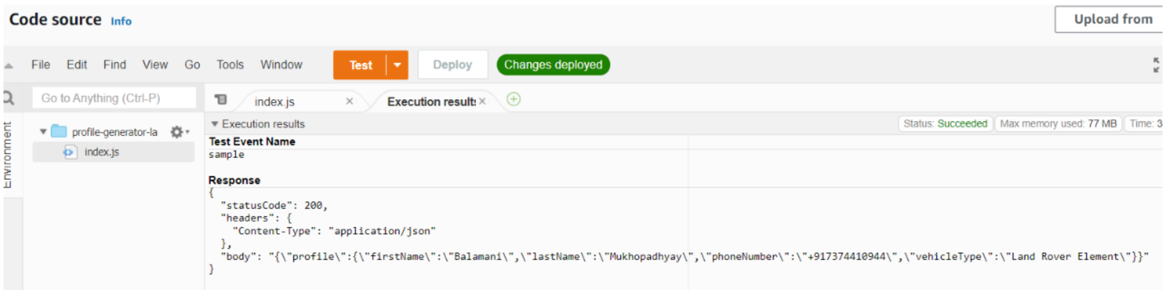
Lambda > Functions

Functions (1)

Last fetched in 0 seconds

	Function name	Description	Package type	Runtime	Code size
○	profile-generator-lambda	Generates a new profiles	Zip	Node.js 14.x	403.0 byte

- **Test the Lambda Function**



Execution results

Status: Succeeded | Max memory used: 77 MB | Time: 3

Test Event Name
sample

Response
```
{
  "statusCode": 200,
  "headers": {
    "Content-Type": "application/json"
  },
  "body": "{\"profile\":{\"firstName\":\"Balamani\",\"lastName\":\"Mukhopadhyay\",\"phoneNumber\":\"+917374410944\",\"vehicleType\":\"Land Rover Element\"}}"
}
```