Lab: Validate a Terraform configuration with `terraform validate`

The `terraform validate` command validates the configuration files in a directory, referring only to the Terraform configuration files. Validate runs checks that verify whether a configuration is syntactically valid and internally consistent.

- Task 1: Validate Terraform configuration
- Task 2: Terraform Validate False Positives
- Task 3: JSON Validation Output

Task 1: Validate Terraform configuration

Validation requires an initialized working directory with any referenced plugins and modules installed. It is safe to run `terraform validate` at anytime.

Run a `terraform validate` against your Terraform configuration.

```
terraform validate
```

```
Success! The configuration is valid.
```

Make a change to the configuration that is not validate within Terraform and re-run a validate to see if the syntax error is picked up. We will include an argument in our resource block that isn't supported and see what `validate` reports.

Update the VPC resource block to include a `region` argument `main.tf`

```
resource "aws_vpc" "vpc" {
  region = "us-east-1"
  cidr_block = var.vpc_cidr

  tags = {
    Name        = var.vpc_name
    Environment = "demo_environment"
    Terraform   = "true"
  }
}
```

```
terraform validate
```

```
  Error: Unsupported argument
```


```
    on main.tf line 25, in resource "aws_vpc" "vpc":
    25:   region = "us-east-1"

  An argument named "region" is not expected here.
```

Notice that the `terraform validate` command will return the problem(s) along with file and line number in where they occur. Now replace the offending piece of code and re-issue a `terraform validate`.

```
resource "aws_vpc" "vpc" {
  cidr_block = var.vpc_cidr

  tags = {
    Name        = var.vpc_name
    Environment = "demo_environment"
    Terraform   = "true"
  }
}
```

```
terraform validate
```

```
Success! The configuration is valid.
```

Task 2: Terraform Validate False Positives

The `terraform validate` commands runs checks that verify whether a configuration is syntactically valid and internally consistent, but it may not catch everything. We typically refer to these as false positives where the `terraform validate` reports successful but our plan/apply may still fail.

Let's look a at simple example with the configuration that stores our SSH Key using the Terraform `local` provider.

```
resource "tls_private_key" "generated" {
  algorithm = "RSA"
}

resource "local_file" "private_key_pem" {
  content  = tls_private_key.generated.private_key_pem
  filename = "MyAWSKey.pem"
}
```

If we run a `terraform validate` against we will generate a success.

```
Success! The configuration is valid.
```

Now let's update the `local_file` resource to specify a file path that is incorrect/we don't have permissions to.

```
resource "tls_private_key" "generated" {
  algorithm = "RSA"
}

resource "local_file" "private_key_pem" {
  content  = tls_private_key.generated.private_key_pem
  filename = "/bad_path/MyAWSKey.pem"
}
```

```
terraform validate
Success! The configuration is valid.
```

```
terraform apply

tls_private_key.generated: Refreshing state... [id=502
    de2c3e0a15eaf6e20a5a8cd616378daaafd62]
local_file.private_key_pem: Refreshing state... [id=81
    a9356c0bca8224ea96ec7add0ac28eac442d07]

Terraform used the selected providers to generate the following execution
    plan. Resource actions are
indicated with the following symbols:
-/+ destroy and then create replacement

Terraform will perform the following actions:

  # local_file.private_key_pem must be replaced
-/+ resource "local_file" "private_key_pem" {
      ~ filename             = "MyAWSKey.pem" -> "/bad_path/MyAWSKey.pem"
          # forces replacement
      ~ id                   = "81a9356c0bca8224ea96ec7add0ac28eac442d07"
          -> (known after apply)
        # (3 unchanged attributes hidden)
    }

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

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes
```


```
local_file.private_key_pem: Destroying... [id=81
    a9356c0bca8224ea96ec7add0ac28eac442d07]
local_file.private_key_pem: Destruction complete after 0s
local_file.private_key_pem: Creating...

| Error: mkdir /bad_path: read-only file system
|
|   with local_file.private_key_pem,
|   on main.tf line 5, in resource "local_file" "private_key_pem":
|    5: resource "local_file" "private_key_pem" {
}
```

As you can see the execution of our configuration fails. This is because we don't have permission to create the key in the directory we specified. It is important to know that `terraform validate` will check that the HCL syntax is valid but not if the values provided for arguments are correct.

Task 3: JSON Validation Output

You can also produce the output of a `terraform validate` in JSON format in the event you need to pass this data to another system.

```
terraform validate -json
```

```
{
  "format_version": "0.1",
  "valid": true,
  "error_count": 0,
  "warning_count": 0,
  "diagnostics": []
}
```

You can make the same invalid code change as performed in Task #1 to see how `terraform validate` returns an error in JSON format

```
terraform validate -json
```

```
{
  "format_version": "0.1",
  "valid": false,
  "error_count": 1,
  "warning_count": 0,
  "diagnostics": [
    {
      "severity": "error",
```



```
      "summary": "Unsupported argument",
      "detail": "An argument named \"region\" is not expected here.",
      "range": {
        "filename": "main.tf",
        "start": {
          "line": 25,
          "column": 3,
          "byte": 511
        },
        "end": {
          "line": 25,
          "column": 9,
          "byte": 517
        }
      },
      "snippet": {
        "context": "resource \"aws_vpc\" \"vpc\"",
        "code": "  region = \"us-east-1\"",
        "start_line": 25,
        "highlight_start_offset": 2,
        "highlight_end_offset": 8,
        "values": []
      }
    }
  ]
}
```