Deploying with Terraform

Software Architecture

March 7, 2022

Brae Webb

1 Before Class

Ensure you've had practice using the AWS Academy learner lab. It's preferable if you already have terraform installed[1]. Please also have one of IntelliJ IDEA, PyCharm, or VSCode with the terraform plugin installed. It is also helpful to have read the Infrastructure as Code lecture notes to understand the motivation for using a tool like Terraform.

2 This Week

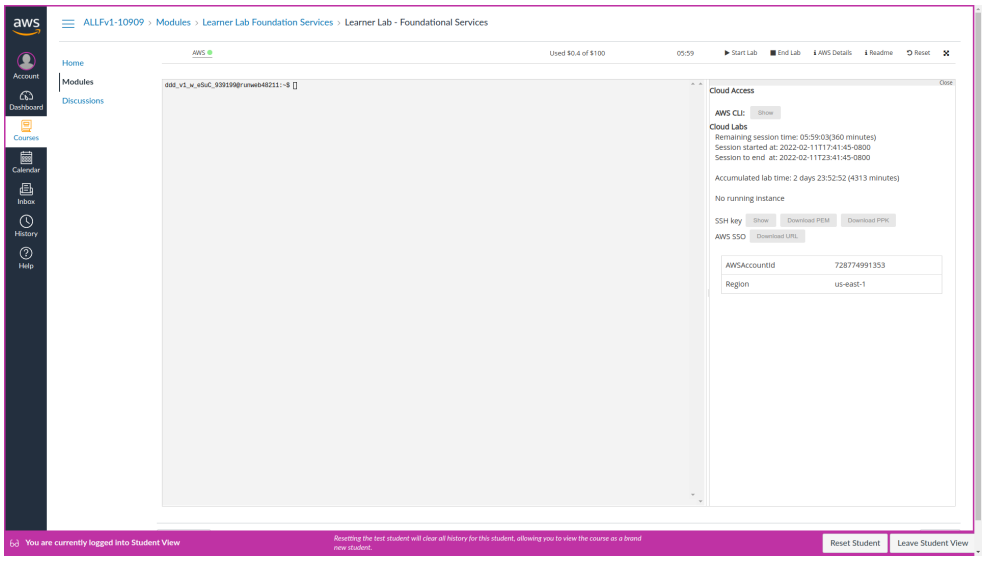
This week our goal is to get experience using an Infrastructure as Code tool, specifically, Terraform, to deploy a service to AWS. Specifically, this week you need to:

- Authenticate terraform to use the AWS learner lab.
- Configure a single server website in terraform and deploy.
- Create a terraform module for deploy arbitrary single server websites.

3 Using Terraform in AWS Leaner Labs

Following the steps from the week one practical, start a learner lab in AWS Academy. For this practical, you don't need to create any resources in the AWS Console. The console can be used to verify that terraform has correctly provisioned resources.

1. Once the learner lab has started, click on 'AWS Details' to display information about the lab.



2. Click on the first 'Show' button next to 'AWS CLI' which will display a text block starting with `[default]`.

[1] https://learn.hashicorp.com/tutorials/terraform/install-cli

3. Create a directory for this weeks practical.
4. Within that directory create a `credentials` file and copy the contents of the text block into the file. **Don't share this file contents — don't commit it**.
5. Create a `main.tf` file in the same directory with the following contents:

```
» cat main.tf
terraform {
    required_providers {
        aws = {
            source = "hashicorp/aws"
            version = "~> 3.0"
        }
    }
}

provider "aws" {
    region = "us-east-1"
    shared_credentials_file = "./credentials"
}
```

   The `terraform` block specifies the required external dependencies, here we need to use the AWS provider. The `provider` clock configures the AWS provider, instructing it which region to use and how to authenticate (using the credentials file we created).

6. We need to initialise terraform which will fetch the required dependencies. This is done with the `terraform init` command.

```
$ terraform init
```

   This command will create a `.terraform` directory which stores providers and a provider lock file, `.terraform.lock.hcl`.

7. To verify that we've setup terraform correctly, use `terraform plan`.

```
$ terraform plan
```

   As we currently have no resources configured, it should find that no changes are required. Note that this doesn't ensure our credentials are correctly configured as terraform has no reason to try authenticating yet.

4 Deploying Hextris

If you followed the default instructions in the week one practical, you would have manually deployed the hextris game. Now we'll try to deploy hextris using terraform.

First, we'll need to create an EC2 instance resource. The AWS provider calls this resource an `aws_instance`[2]. Get familiar with the documentation page. Most terraform providers have reasonable documentation, reading the argument reference section helps to understand what a resource is capable of.

We'll start off with the basic information for the resource. We configured it to use a specific AMI and chose the `t2.micro` size. Refer back to the week one practical sheet for a refresher. Add the following basic resource to `main.tf`:

```
   » cat main.tf
resource "aws_instance" "hextris-server" {
    ami = "ami-0a8b4cd432b1c3063"
    instance_type = "t2.micro"

    tags = {
        Name = "hextris"
    }
}
```

Let's create this server. `terraform apply` will first do `terraform plan` and prompt us to confirm if we want to apply to changes.

```
$ terraform apply
```

You should be prompted with something similar to the output below.

```
Terraform used the selected providers to generate the following execution plan.
   Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.hextris-server will be created
  + resource "aws_instance" "hextris-server" {
      + ami = "ami-0a8b4cd432b1c3063"
      (omitted)
      + instance_type = "t2.micro"
      (omitted)
      + tags = {
          + "Name" = "hextris"
        }
    }
```

[2] https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/instance

```
Plan: 1 to add, 0 to change, 0 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:
```

If the plan looks sensible enter `yes` to enact the changes.

```
  Enter a value: yes

aws_instance.hextris-server: Creating...
aws_instance.hextris-server: Still creating... [10s elapsed]
aws_instance.hextris-server: Still creating... [20s elapsed]
aws_instance.hextris-server: Still creating... [30s elapsed]
aws_instance.hextris-server: Still creating... [40s elapsed]
aws_instance.hextris-server: Creation complete after 47s [id=i-08c92a097ae7c5b18]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.
```

You can now check in the AWS Console that an EC2 instance with the name `hextris` has been created. Now we've got a server, we should try to configure it to contain hextris. We'll use the `user_data` field which configures commands to run when launching the instance. First we need a script to provision the server, if we combine all our commands from the last practical, we'll produce this script:

```
   » cat deploy.sh
yum install httpd
systemctl enable httpd
systemctl start httpd

yum install git
cd /var/www/html
git clone https://github.com/Hextris/hextris .
```

Now we can add the following field to our terraform resource. It uses the terraform `file` function to load the contents of a file named `deploy.sh` relative to the terraform directory. The contents of that file is passed to the `user_data` field.

```
user_data = file("${path.module}/deploy.sh")
```

If you run the `terraform plan` command now, you'll notice that terraform has identified that this change will require creating a new EC2 instance. Where possible, terraform will try to update a resource in-place but since this changes how an instance is started, it needs to be replaced. Go ahead and apply the changes.

Now, in theory[3], we should have deployed hextris to an EC2 instance. But how do we access that instance? We *could* go to the AWS Console and find the public IP address. However, it turns out that terraform already knows the public IP address. In fact, if you open the terraform state file (`terraform.tfstate`), you should be able to find it hidden away in there. But we don't want to go hunting through this file all the time. Instead we'll use the `output` keyword.

We can specify certain attributes as 'output' attributes. Output attributes are printed to the terminal when the module is invoked directly but as we'll see later, they can also be used by other terraform configuration files.

```
   » cat main.tf
output "hextris-url" {
  value = aws_instance.hextris-server.public_ip
}
```

This creates a new output attribute, `hextris-url`, which references the `public_ip` attribute of our `hextris-server` resource. Note that resources in terraform are addressed by the resource type (`aws_instance`) followed by the name of the resource (`hextris-server`).

If you plan or apply the changes it should tell you the public IP address of the instance resource.

```
  $ terraform plan
```

```
aws_instance.hextris-server: Refreshing state... [id=i-043a61ff86aa272e0]

Changes to Outputs:
  + hextris-url = "3.82.225.65"

You can apply this plan to save these new output values to the Terraform state,
   without changing any real infrastructure.
```

So let's try and access that url, hmm. That's strange. Something's gone wrong.

5 Security Groups

[3] hint