COSC349 Assignment 2

Matthew Dockerty (7516648)

Application Deployment

For this second assignment, I decided to use the same simple recipe book application I developed for assignment 1. Since I made use of environment variables for all the application configuration, I did not have to make a single change to the application code.

I decided to use Terraform to deploy my application because I wanted to learn more about it as a tool, and because I wanted to be able to automate the deployment of cloud services (something which I wouldn't have been able to do with Vagrant).

I defined my infrastructure using HCL (HashiCorp Configuration Language) in several .tf files. The deployment takes two configurable inputs, namely:

- AWS CLI credentials: stored in the file *~/.aws/credentials*
- Database password: defined in the environment variable *TF_VAR_db_password* (will be used as the password for the DocumentDB instance)

Once these credentials have been set, deploying the entire infrastructure for the application is as simple as running the command *terraform plan* to generate a list of actions that Terraform will perform, and *terraform apply* to perform those actions. The plan step is not required but is useful to gain an idea of what Terraform will do when applying the configuration and deploying.

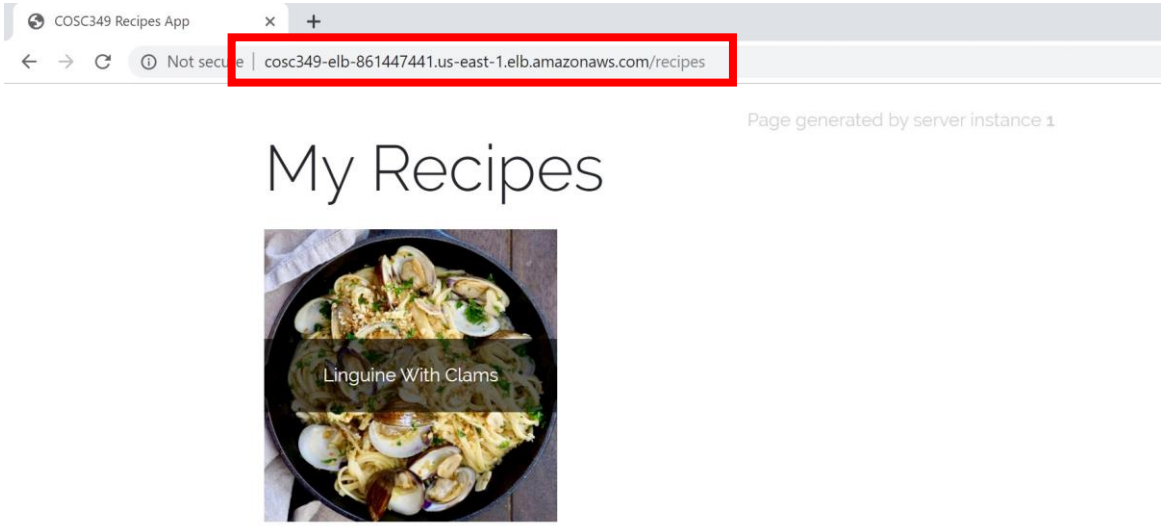
After running the apply command, Terraform handles deploying the infrastructure to AWS based on the specifications declared in the .tf configuration files. It also provisions the application servers, which consists of cloning the application files from GitHub (A public repository containing the build output files from assignment 1 for simplicity https://github.com/matthewdockerty/cosc349-assignment2-deployment), and configuring a system service. This simple and automatic deployment and provisioning process was one of the major appeals of Terraform.

I also configured an output as a part of the configuration which displays the domain name a user needs to connect to in order to access the deployed application (see below). In the case of this particular deployment, it will be accessible at http://cosc349-elb-861447441.us-east-1.elb.amazonaws.com

```
aws_docdb_cluster_instance.db_ci[0]: Creation complete after 6m18s [id=cosc349-d
ocdb-0]

Apply complete! Resources: 9 added, 1 changed, 0 destroyed.

Outputs:

elb-dns-name = cosc349-elb-861447441.us-east-1.elb.amazonaws.com
```

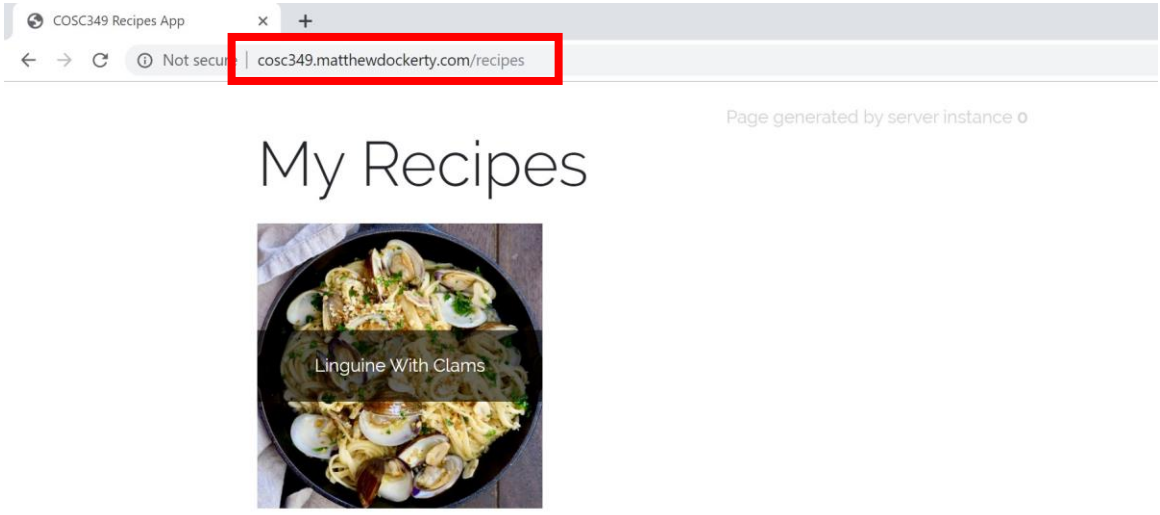
What's also helpful is that if I make any changes to the configuration, I can simply run the *terraform apply* command again, and Terraform will handle updating, destroying, and creating any instances or services that need to be changed. While testing and developing the infrastructure deployment, the domain name never changed, since I never made major changes to the ELB. This means that it is possible to change the configuration (within reasonable bounds) without worrying about the domain name changing and affecting users.

Accessing the application in a browser shows that the deployment did indeed work, and that the application is fully functional.



While it's helpful to have a link to access the application, the one generated by AWS is not particularly user friendly. To further enhance the deployment, I setup an ALIAS DNS record on my personal domain (see below) so that the application could be accessed via a more user-friendly URL.

After waiting a while for the DNS record to propagate, the application was accessible via http://cosc349.matthewdockerty.com (and still should be until the AWS Educate credit runs out!)



Interacting with this application will be exactly the same as in the first assignment (see https://youtu.be/bSMsAYRreik for a video demonstration).

Cloud Services

Elastic Load Balancer (ELB)

My first assignment made use of a virtual machine running an Nginx reverse proxy acting as a load balancer to distribute clients between two application server instances. Since AWS provides load balancing services, I decided to use an ELB as one of my cloud services, replacing the role of the Nginx VM. I used a classic load balancer because it was simpler to set up, and because I did not require the additional features provided by more advanced ELBs.

I found this quite challenging to setup, as several network settings needed to be configured to make everything work. I found an incredibly helpful guide at https://www.bogotobogo.com/DevOps/Terraform/Terraform-VPC-Subnet-ELB-RouteTable-SecurityGroup-Apache-Server-1.php which helped me a lot through this process. The deployed application shows the number of the application server which generated the page, which clearly demonstrates the correct and working behaviour of the ELB.

DocumentDB Database

I also used a MongoDB database to store data for my application in the first assignment. I could have created an AWS EC2 instance and installed MongoDB, but I decided to try and use one of the AWS database services instead. After doing some research I discovered DocumentDB, a proprietary NoSQL document-based database created by Amazon that fully supports the MongoDB API. This meant that I could deploy my application using the DocumentDB database service without having to change my application's code at all.

The deployment process was incredibly straightforward and surprisingly easy – it was literally a matter of creating the DocumentDB instance and changing the database connection details in the application. I made use of a helpful Medium article which explained deploying DocumentDB clusters using Terraform, which can be found at https://medium.com/@geekrodion/amazon-documentdb-and-aws-lambda-with-terraform-34a5d1061c15.

This process was incredibly smooth, although I ran into one major issue: The smallest DocumentDB instance tier available is the *large* tier, which has a price of around $200 per month. Since I was using an Amazon Educate account, I wasn't too concerned about the price, although I realised that leaving it running after submitting the project would quickly exhaust my free credits.

To resolve this issue, I tried to use MongoDB Atlas instead – a MongoDB cloud database service provided by the same group that develops MongoDB. I was able to successfully do so, except MongoDB Atlas does not allow provisioning free tier instances through their API, and I didn't want to pay for the service, and wanted to be able to deploy the whole application without any manual interaction.

As a result of this, I decided to continue using DocumentDB, and to simply stop my DocumentDB cluster until the project is about to be marked, ensuring it doesn't use up all my credit beforehand. I

also contacted Amazon Educate to find out about getting more free credit in case it runs out, which will easily be possible if necessary.

Application Architecture

The application architecture for this assignment is very similar to that of the first assignment. It makes use of two virtual machines, and two cloud services. I chose to omit the build server I had in the first assignment for simplicity. All these services are contained within a single virtual private cloud within a single AWS region.

- **Elastic Load Balancer:** Accepts all incoming requests for the application and distributes the load between the application server instances.
- **2 Application Servers:** Two EC2 instances running the application developed in the first assignment, each in a different availability zone within the region.
- **DocumentDB Cluster:** A database cluster containing a single DocumentDB instance – a MongoDB compatible database – in which all the recipe data for the application are stored.

The following diagram illustrates the architecture of the application and how the different components and services interact.