Application with 11 Microservices on Local Kubernetes Cluster

Today I was working on **Online Boutique,** a cloud-first application with 11 microservices showcasing Kubernetes, Istio, and gRPC. I created my own YAML configuration files with a few modifications.
Online Boutique consists of an **11-tier Microservices** application. The application is a web-based **E-Commerce** app where users can browse items, add them to the cart, and purchase them.
This application demonstrates the use of technologies like **Kubernetes**, **Istio**, **Stackdriver**, and **gRPC**. This application works on any Kubernetes cluster, as well as Google Kubernetes Engine.
I deploy it on a Local Cluster in **Docker Desktop** but you can deploy it on Google Kubernetes Engine (**GKE**), Amazon *Elastic Kubernetes Service* (**EKS**), Azure Kubernetes Service (**AKS**), Linode Kubernetes Engine (**LKE**) or any other Kubernetes solution.
FEATURES

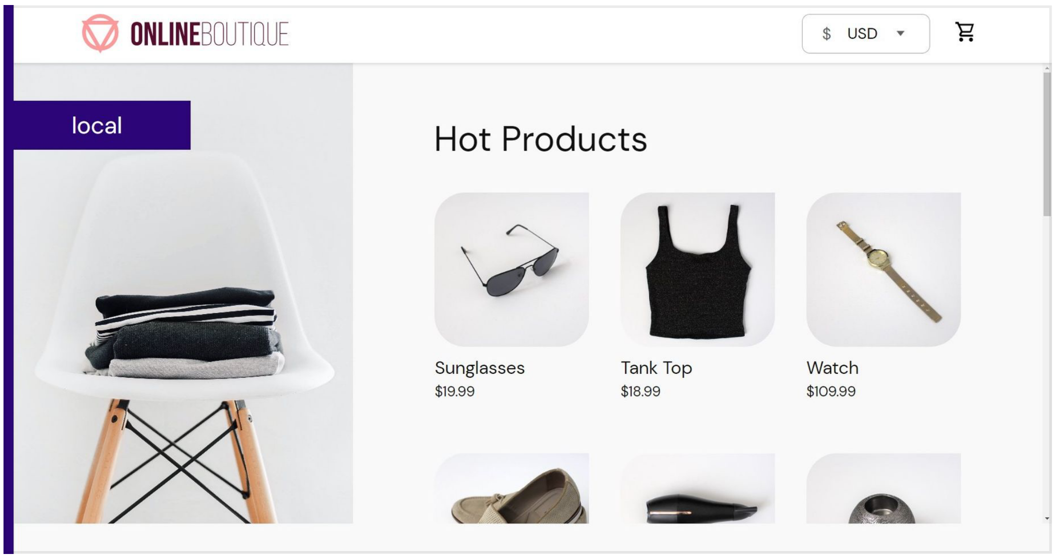
- **Kubernetes/GKE:** The app is designed to run on Kubernetes (both locally on "Docker for Desktop", as well as on the cloud with GKE).
- **gRPC:** Microservices use a high volume of gRPC calls to communicate with each other.
- **Istio:** Application works on Istio service mesh.
- **Cloud Operations (Stackdriver):** Many services are instrumented with **Profiling** and **Tracing**. In addition to these, using Istio enables features like Request/Response **Metrics** and **Context Graph** out of the box. When it is running out of Google Cloud, this code path remains inactive.
- **Skaffold:** Application is deployed to Kubernetes with a single command using Skaffold.
- **Synthetic Load Generation:** The application demo comes with a background job that creates realistic usage patterns on the website using Locust load generator.

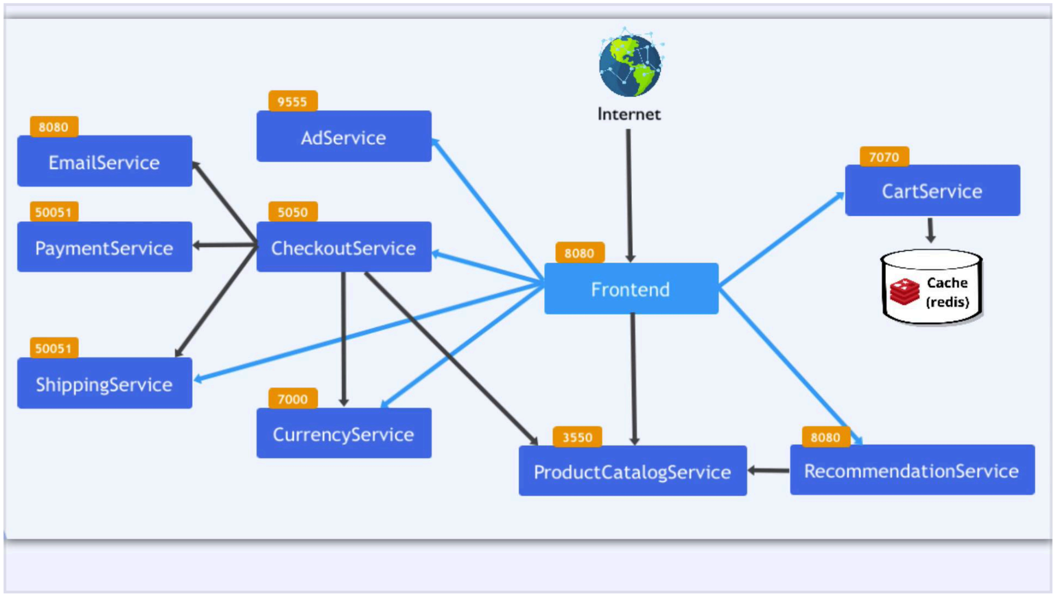
MICROSERVICES

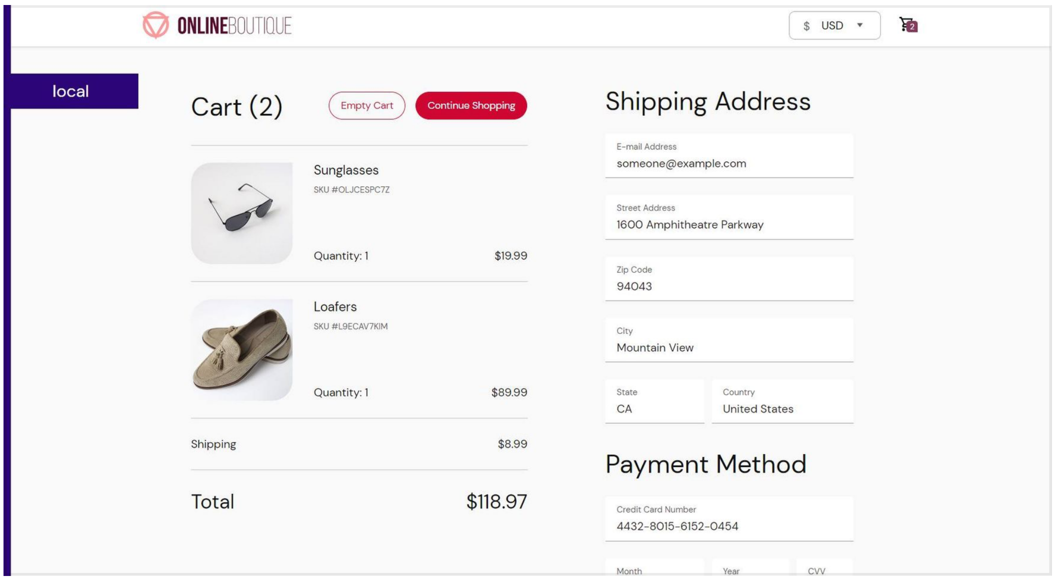
- Frontend: Go Language, Exposes an HTTP server to serve the website. Does not require signup/login and generates session IDs for all users automatically.
- Cart Service: C# Language, Stores the items in the user's shopping cart in Redis and retrieves it.
- Product Catalog Service: Go Language, Provides the list of products from a JSON file and ability to search products and get individual products.
- Currency Service: Node.js, Converts one money amount to another currency. Uses real values fetched from European Central Bank. It's the highest QPS service.
- Payment Service: Node.js, Charges the given credit card info (mock) with the given amount and returns the transaction ID.

- Shipping Service: Go Language, Gives shipping cost estimates based on the shopping cart. Ships items to the given address (mock)
- Email Service: Python Language, Sends users an order confirmation email (mock).
- Checkout Service: Go Language, Retrieves user cart, prepares order and orchestrates the payment, shipping and the email notification.
- Recommendation Service: Python Language, Recommends other products based on what's given in the cart.
- Ad Service: Java Language, Provides text ads based on given context words.
- Load Generator: Python/Locust Language, Continuously sends requests imitating realistic user shopping flows to the frontend.

#DevOps #Kubernetes #Linux #AWS #Docker #Git #EKS #AKS #LKS #Python #GoLang #NodeJS #Java #javaScript #CSharp #Istio #Microservices #gRPC #Teraform #GCP #AWS #Azure #eCommerce #Stackdriver #Container #Kubectl #Minikube







Cart (2)

Sunglasses SKU #OLJCESPC7Z	Quantity: 1	$19.99
Loafers SKU #L9ECAV7KIM	Quantity: 1	$89.99
Shipping		$8.99
Total		$118.97

Shipping Address

E-mail Address: someone@example.com

Street Address: 1600 Amphitheatre Parkway

Zip Code: 94043

City: Mountain View

State: CA

Country: United States

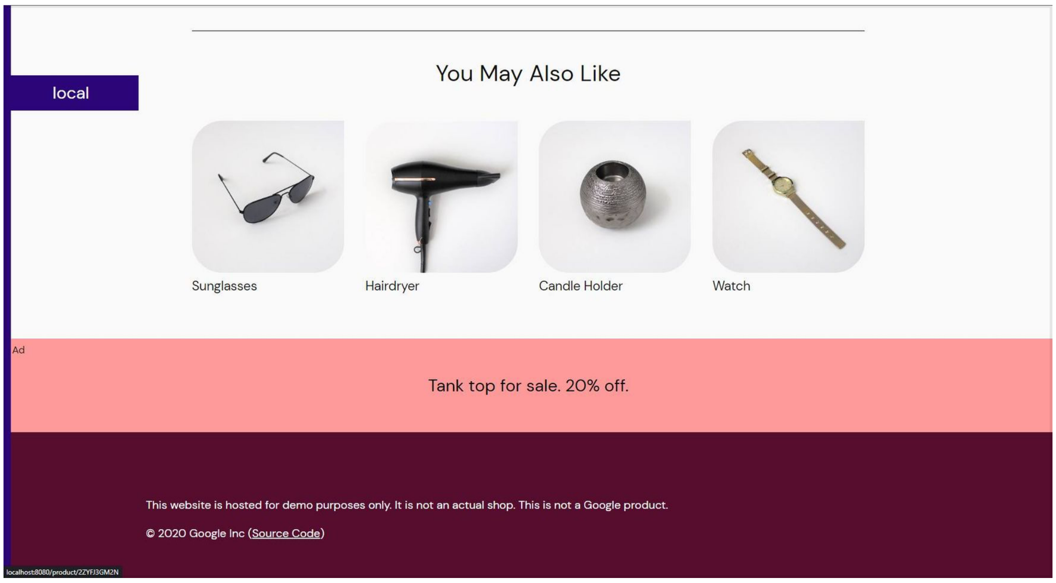
Payment Method

Credit Card Number: 4432-8015-6152-0454

Month

Year

CVV



local

You May Also Like

Sunglasses

Hairdryer

Candle Holder

Watch

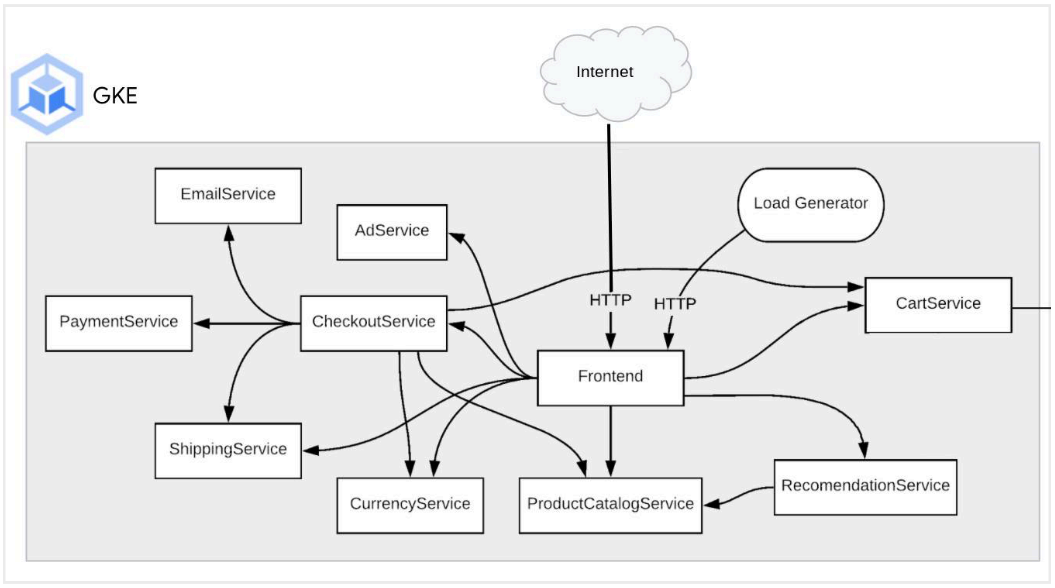
Ad

Tank top for sale. 20% off.

This website is hosted for demo purposes only. It is not an actual shop. This is not a Google product.

© 2020 Google Inc (Source Code)

localhost:8080/product/2ZYFJ3GM2N



+6

How to Manage Secrets in AWS with Secrets Manager and Terraform

AWS Secrets Manager helps us to create, manage, rotate and retrieve database credentials, API keys, and other secrets through their lifecycle

In this story, we will learn how to create and consume secrets using AWS Secrets Manager and Terraform.

Creating a Secret for a Variable

In this first example, we will create a secret using a variable for an API username.

The first step is to define the variable, and we are using sensitive = true to protect the values of the variable from being printed in the logs and console output.

```
# Secret Variables
variable "api_username" {
  description = "API service username"
  type        = string
  sensitive   = true
}
```
then we will create the secret:
```
# Creating a AWS Secret for API Service User
resource "aws_secretsmanager_secret" "service_user" {
  name        = "service_user"
  description = "Service Account Username for the API"
  recovery_window_in_days = 0
  tags = {
    Name        = "service_user"
    Environment = var.app_environment
  }
}
resource "aws_secretsmanager_secret_version" "service_user" {
  secret_id     = aws_secretsmanager_secret.service_user.id
  secret_string = var.api_username
}
```
Define the username in the "terraform.tfvars" file:
```
api_username = "srv_apiprod"
```
Note: *the "recovery_window_in_days" is an* ***optional*** *setting for the number of days that AWS Secrets Manager waits before it can delete the secret. The* ***default value is 30****. This value can be* ***0 to force deletion without recovery*** *(recommended for development) or range from 7 to 30 days.*

Creating a Secret for a Password

In this second example, we will create a secret for an API password.

The first step is using the "randon_password" resource to create a random password. This function is identical to "random_string" except that the result is treated as sensitive and not displayed in console output.
```
resource "random_password" "service_password" {
  length  = 16
  special = true
  numeric = true
  upper   = true
  lower   = true
}
```
Then we create the create secret, using the randon_password function:
```
resource "aws_secretsmanager_secret" "service_password" {
  name        = "service_pass"
  description = "Service Account Password for the API"
  recovery_window_in_days = 0
  tags = {
    Name        = "service_pass"
    Environment = var.app_environment
  }
}
resource "aws_secretsmanager_secret_version" "service_password" {
  secret_id     = aws_secretsmanager_secret.service_password.id
  secret_string = random_password.service_password.result
}
```

and we can consume the secret like this:

```
output "service_user" {
  value = local.service_user
  sensitive = true
}
```

Storing Multiples Variables in a Single Secret with Key/Value Pairs in JSON

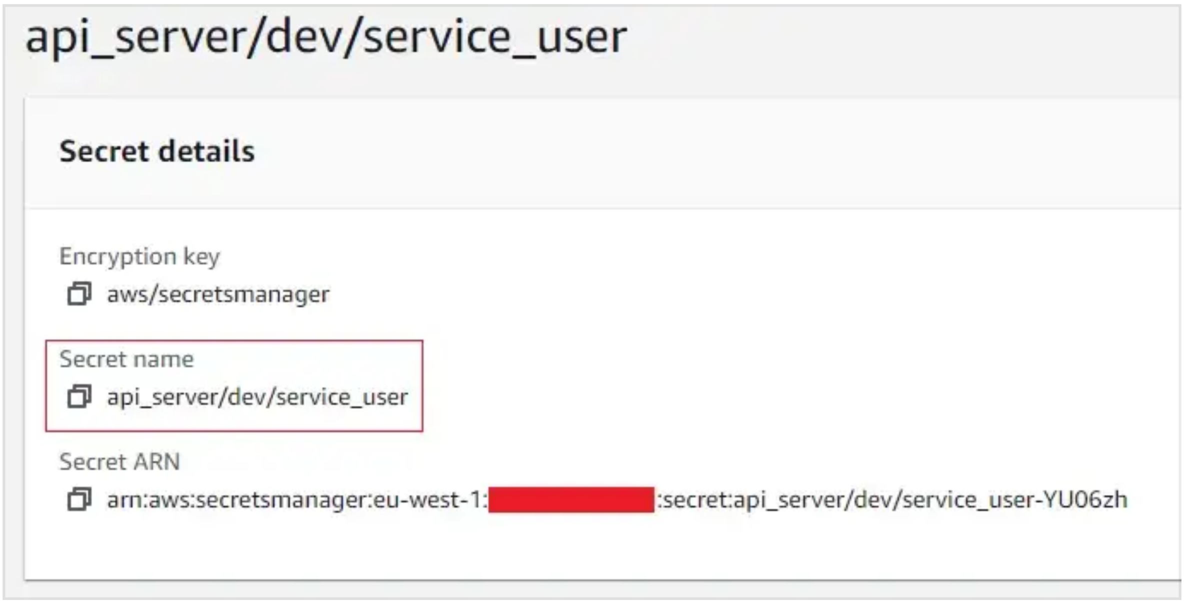
Store secrets cost money! For example, AWS Secrets Manager charges $0.40 per month for each secret we store, plus $0.05 for every 10,000 API calls we make to store or retrieve data.

In this example, we are going to store multiple values for the DNS Configuration in a single secret with key/value pairs in JSON format.

```
# Creating a AWS Secret for DNS Configuration
resource "aws_secretsmanager_secret" "dns_config" {
  name        = "dns_config"
  description = "DNS Configuration"
  recovery_window_in_days = 0
  tags = {
    Name        = "dns_config"
    Environment = var.app_environment
  }
}
resource "aws_secretsmanager_secret_version" "dns_config" {
  secret_id     = aws_secretsmanager_secret.dns_config.id
  secret_string = <<EOF
  {
    "DNSZone": "kopicloud.local",
    "DNSServer1": "10.127.1.6",
    "DNSServer2": "10.127.1.7"
  }
EOF
}
```

Reading a Secret for a Variable

We are going to use the data “aws_secretsmanager_secret” to read values. We will use the Secret name on the AWS Secrets Manager console as a reference for retrieving our secret.



api_server/dev/service_user

Secret details

Encryption key
aws/secretsmanager

Secret name
api_server/dev/service_user

Secret ARN
arn:aws:secretsmanager:eu-west-1:[redacted]:secret:api_server/dev/service_user-YU06zh

This is our code:

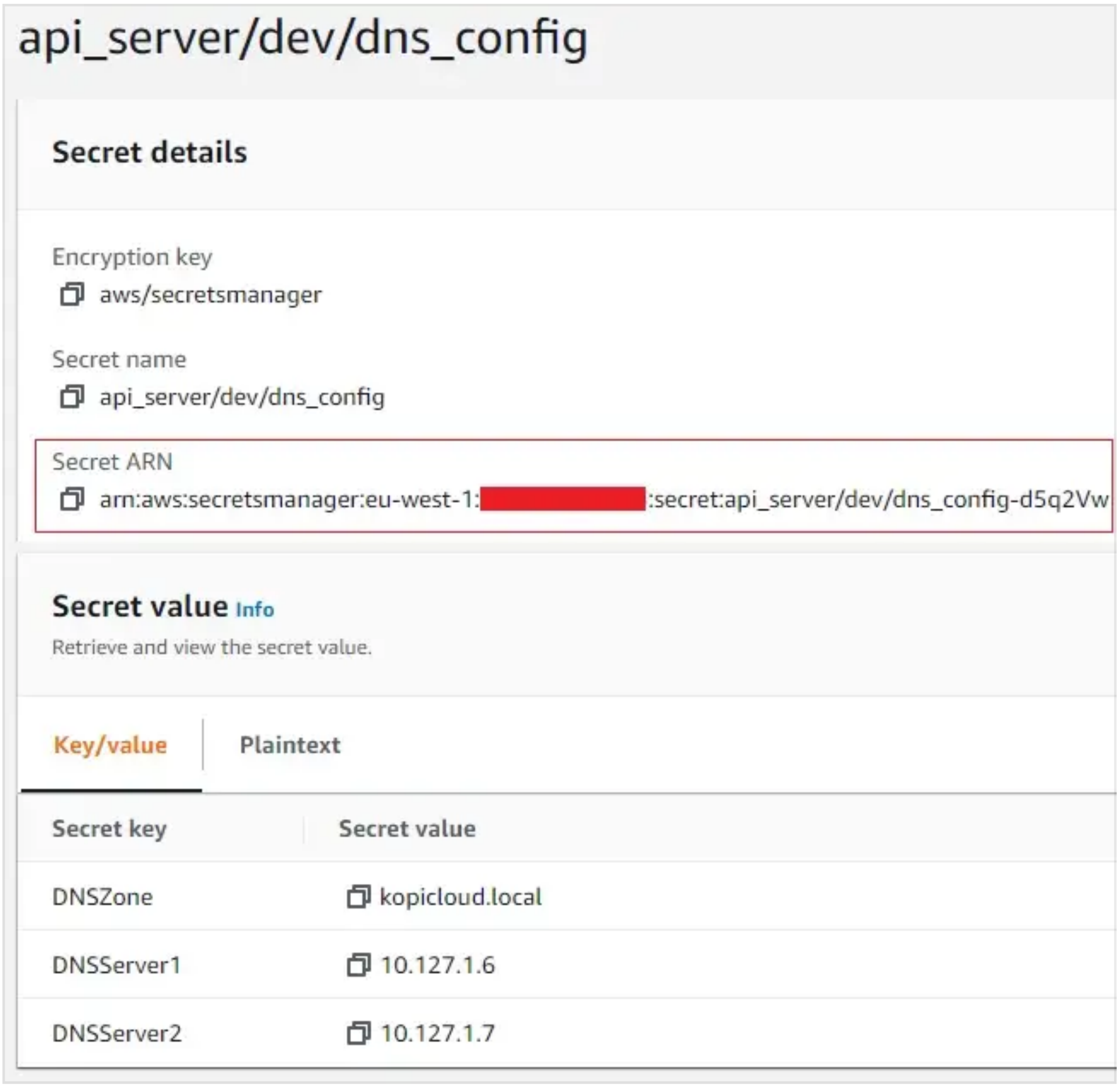
```
# read service user secret
data "aws_secretsmanager_secret" "service_user" {
  name = "api_server/dev/service_user"
}
data "aws_secretsmanager_secret_version" "service_user" {
  secret_id = data.aws_secretsmanager_secret.service_user.id
}
```

Then, we will use locals variables to read the secret:

```
locals {
  service_user =
data.aws_secretsmanager_secret_version.service_user.secret_string
}
```

Reading Key/Value Pairs in JSON Secrets

Now we are going to read the JSON with multiple Key/Value pairs.


api_server/dev/dns_config

Secret details

Encryption key
aws/secretsmanager

Secret name
api_server/dev/dns_config

Secret ARN
arn:aws:secretsmanager:eu-west-1:[redacted]:secret:api_server/dev/dns_config-d5q2Vw

Secret value
Retrieve and view the secret value.

Secret key	Secret value
DNSZone	kopicloud.local
DNSServer1	10.127.1.6
DNSServer2	10.127.1.7

We are going to use the data "aws_secretsmanager_secret" to read values and, in this example, the Secret ARN on the AWS Secrets Manager console as a reference for our secret to retrieve.

```
data "aws_secretsmanager_secret" "dns_config" {
  arn = "arn:aws:secretsmanager:eu-west-1:88888888888:secret:api_server/
dev/dns_config-d5q2Vw"
}
data "aws_secretsmanager_secret_version" "dns_config" {
  secret_id = data.aws_secretsmanager_secret.dns_config.id
}
```

Then, we will use locals variables to read the secret:

```
locals {
  dns_config =
jsondecode(data.aws_secretsmanager_secret_version.dns_config.secret_strin
g)
}
```

and we can consume the DNSZone secret like this:

```
output "dns_config_DNS_Zone" {
  value = local.dns_config.DNSZone
  sensitive = true
}
```

Debugging Our Secrets Code

Terraform is very good at protecting our secrets. If we are trying to output the secrets with this Terraform output code:

```
output "dns_config_DNS_Zone" {
  value = local.dns_config.DNSZone
}
```

we are getting this message from Terraform when we execute the code:

```
D:\tf-code> terraform apply -auto-approve
|
| Error: Output refers to sensitive values
|
|   on secret-manager-main.tf line 17:
|   17: output "service_user" {
|
| To reduce the risk of accidentally exporting sensitive data that
| was intended to be only internal, Terraform requires that any root | module
output containing sensitive data be explicitly marked as
| sensitive, to confirm your intent.
|
| If you do intend to export this data, annotate the output value as
| sensitive by adding the following argument: sensitive = true
```

If we add the recommended sensitive = true, the code is executed. However, we cannot see the value, we just consume from Terraform code.

```
D:\tf-code> terraform apply -auto-approve
Outputs:
service_user = <sensitive>
```

So, how can we debug the Terraform code to be sure that we are reading the code properly? We are going to use a "null_resource" with a "local-exec" to redirect the output to a text file:

```
resource "null_resource" "dns_config" {
  provisioner "local-exec" {
    when    = create
    command = "echo ${local.dns_config.DNSZone} >> dns_zone.txt"
  }
}
```

and then look inside the text file...

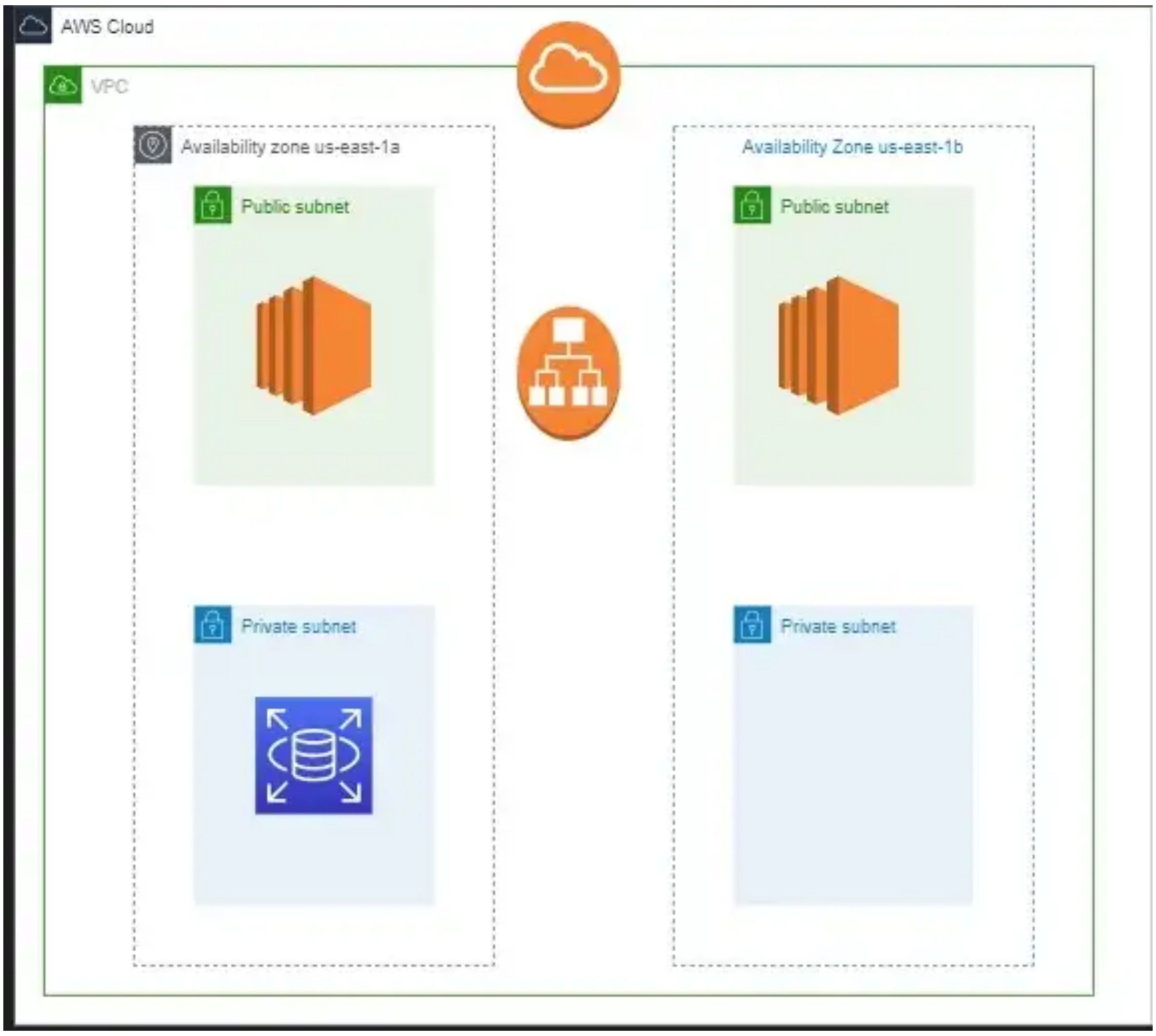
```
D:\tf-code>type dns_zone.txt
kopicloud.local
```

==========

Introduction to Terraform

In this how to lesson you will be deploying a two-tier architecture. The task consists of the following:

- Deploy a VPC with CIDR 10.0.0.0/16 with 2 public subnets with CIDR 10.0.1.0/24 and 10.0.2.0/24. Each public subnet should be in a different AZ for high availability.
- Create 2 private subnet with CIDR '10.0.3.0/24' and '10.0.4.0/24' with an RDS MySQL instance (micro) in one of the subnets. Each private subnet should be in a different AZ. N
- A load balancer that will direct traffic to the public subnets.
- Deploy 1 EC2 t2. micro instance in each public subnet.



Perquisites: Cloud 9 environment with Terraform installed, an AWS account, and Terraform Registry.
This task has several steps- Terraform registry is your best friend!
Step1: Within your cloud9 environment create a new directory. From within the Cloud9 terminal change into the directory by running the change directory command cd. Followed by creating a .tf file <name.tf>. The .tf file tells cloud9 that we are working with terraform.
Step2: Build the infrastructure by entering the below script.

?

?

?

?

?

?

Step3: From the terminal run the below commands in this order
terraform init**.** The terraform init command initializes a working directory containing terraform configuration.
terraform validate to validate the configuration files.
terraform plan which will create an execution plan and point out any errors (I had several).
terraform apply to roll out any changes made and put everything together as it should be.
Below are a few images of Terraform executing after running the mentioned commands. I did have a few errors that I had to correct

Dreadful Errors!!! Fix and run the terraform apply command again. Lesson Learned : It is best to enter your script in parts and validate as you go to avoid a HUGE list of errors. (This quick snip is just that, I had a PAGE of things that I had to fix)

Everything is complete and good to go!!! (An hour and several terraform apply's later)
Step4: Confirm deployment of your terraform baby in AWS. (Yes, each project is my baby as they all take patience and need your undivided attention)

Instances are running

Databases are up
Step5: Once confirmed you can close down the resources created by

running terraform destroy in the terminal.

26

1

26