**Helm Lab Guide**

**Prerequisites**

* Kubernetes cluster running (e.g., minikube or a cloud-based cluster).
* kubectl and helm installed and configured to interact with the cluster.

Refer to the [Helm Installation Guide](https://helm.sh/docs/intro/install/) to set up Helm if you haven’t done so already.

**Lab Outline**

1. **Install Helm**
2. **Adding and Managing Helm Repositories**
3. **Exploring Helm Charts**
4. **Deploying Applications with Helm**
5. **Managing Helm Releases**
6. **Advanced Helm Commands**
7. **Rollback and History Management**
8. **Pulling and Inspecting Helm Charts**

**1. Install Helm**

Follow the instructions on the official [Helm Installation Guide](https://helm.sh/docs/intro/install/) to download and install Helm for your operating system.

**Verify Installation**

bash

Copy code

helm version

**2. Adding and Managing Helm Repositories**

**List Current Repositories**

bash

Copy code

helm repo list

**Add Bitnami Repository**

The Bitnami repository contains various useful charts, including tomcat and jenkins.

bash

Copy code

helm repo add bitnami https://charts.bitnami.com/bitnami

**Verify Repository Addition**

bash

Copy code

helm repo list

**Search for Charts in Bitnami Repository**

1. Search for the tomcat chart:

bash

Copy code

helm search repo bitnami/tomcat

1. Search for the jenkins chart:

bash

Copy code

helm search repo bitnami/jenkins

**Remove Stable Repository (Optional)**

If you previously added the stable repository, you can remove it.

bash

Copy code

helm repo remove stable

**3. Exploring Helm Charts**

**View Chart Details**

To explore detailed information about the chart, you can use the following commands:

1. **Show Chart Values**: View configurable values of the tomcat chart.

bash

Copy code

helm show values bitnami/tomcat

1. **Show Chart Information**: Retrieve metadata about the chart.

bash

Copy code

helm show chart bitnami/tomcat

1. **Show All Information**: Display both the metadata and values.

bash

Copy code

helm show all bitnami/tomcat

**4. Deploying Applications with Helm**

**Basic Deployment**

Deploy the tomcat application with the bitnami/tomcat chart:

bash

Copy code

helm install <release-name> bitnami/tomcat -n <namespace>

Example:

bash

Copy code

helm install raman-test-tomcat bitnami/tomcat

**Verify Deployment**

List all Helm releases to confirm the deployment:

bash

Copy code

helm list

**Check Deployed Resources**

Use kubectl to inspect the resources created by the Helm release:

bash

Copy code

kubectl get all

**5. Customizing Helm Releases**

**Custom Deployment with Configuration**

Deploy the tomcat application with custom values, including setting the service type to NodePort, disabling persistence, and setting the replica count to 3.

bash

Copy code

helm install test-tomcat bitnami/tomcat --set service.type="NodePort" --set persistence.enabled=false --set replicaCount=3

**Update an Existing Helm Release**

To update the test-tomcat release with new configurations:

bash

Copy code

helm upgrade test-tomcat bitnami/tomcat --set replicaCount=2 --set service.type="NodePort" --set persistence.enabled=false

**View Helm Release Details**

Check the status of a specific Helm release:

bash

Copy code

helm status test-tomcat

**6. Advanced Helm Commands**

**Check Helm Release History**

View the history of the test-tomcat release to track changes over time.

bash

Copy code

helm history test-tomcat

**Rollback Helm Release**

Rollback the test-tomcat release to a previous revision:

bash

Copy code

helm rollback test-tomcat <revision-number>

Example:

bash

Copy code

helm rollback test-tomcat 1

**7. Rollback and History Management**

**View Current Values of a Release**

Display the values used in the test-tomcat release:

bash

Copy code

helm get values test-tomcat

**Rollback to a Specific Revision**

Rollback to a specified revision, e.g., revision 3:

bash

Copy code

helm rollback test-tomcat 3

Verify the rollback by inspecting the release’s history:

bash

Copy code

helm history test-tomcat

**8. Pulling and Inspecting Helm Charts**

**Download the Chart Package**

Download the bitnami/tomcat chart package:

bash

Copy code

helm pull bitnami/tomcat

**Download and Extract the Chart**

Extract the chart files for inspection:

bash

Copy code

helm pull bitnami/tomcat --untar

**Explore the Chart Files**

Navigate to the extracted chart directory and explore its contents.

bash

Copy code

cd tomcat/

Inspect specific chart template files in the templates directory:

bash

Copy code

cd templates/

ls

**Clean-Up**

**Uninstall Helm Releases**

To remove a release and all associated resources, use helm uninstall:

bash

Copy code

helm uninstall <release-name>

Example:

bash

Copy code

helm uninstall test-tomcat

**Delete Kubernetes Resources**

If additional resources were created manually, delete them as follows:

bash

Copy code

kubectl delete hpa --all

kubectl delete svc --all

This concludes the Helm lab guide, covering fundamental to advanced Helm commands for Kubernetes application management. Use this guide to get comfortable with Helm for deployment, updates, rollbacks, and troubleshooting within Kubernetes.

4o

**Kubernetes Health Checks Lab Guide**

**Prerequisites**

* Access to a Kubernetes cluster (e.g., Minikube, GKE, etc.)
* kubectl installed and configured to manage the cluster.

**Key Concepts**

* **Liveness Probe**: Detects if a container is alive. If it fails, Kubernetes restarts the container.
* **Readiness Probe**: Determines if a container is ready to serve traffic. If it fails, the pod is removed from the service endpoint.

**Lab Outline**

1. **Liveness Probe with MongoDB (exec command)**
   * Set up a deployment with incorrect and correct commands.
   * Observe the restart behavior upon probe failure.
2. **Readiness Probe with MongoDB (exec command)**
   * Test incorrect and correct commands.
   * Observe the pod’s removal from the service endpoint.
3. **HTTP-based Liveness Probes with Apache**
   * Configure HTTP-based probes with incorrect paths to simulate failures.
   * Correct the paths and validate.

**1. Liveness Probe with MongoDB Deployment**

**Step 1: Create an Incorrect Liveness Probe**

Create the file liveliness.yml to deploy a MongoDB pod with an **incorrect liveness probe command**:

yaml

Copy code

apiVersion: apps/v1

kind: Deployment

metadata:

name: mongo

spec:

replicas: 3

selector:

matchLabels:

app: mongo

template:

metadata:

labels:

app: mongo

spec:

containers:

- name: mongo

image: mongo:4.0.8

livenessProbe:

exec:

command:

- mongo

- --eval

- "db1.adminCommand('ping')" # Incorrect command

initialDelaySeconds: 1

periodSeconds: 10

timeoutSeconds: 5

successThreshold: 1

failureThreshold: 2

Apply this configuration:

bash

Copy code

kubectl apply -f liveliness.yml

**Step 2: Monitor the Pod Behavior**

Check the pod status and events:

bash

Copy code

kubectl get pods

kubectl describe pod <pod-name>

* **Expected Outcome**: Since the command is incorrect, the liveness probe fails, causing Kubernetes to restart the pod repeatedly.

**Step 3: Correct the Liveness Probe Command**

Edit the liveliness.yml file to correct the command:

yaml

Copy code

command:

- mongo

- --eval

- "db.adminCommand('ping')" # Correct command

Update the deployment:

bash

Copy code

kubectl apply -f liveliness.yml

**Step 4: Verify Corrected Probe**

Check the pod's status again to ensure it is running stably without restarts.

bash

Copy code

kubectl get pods

kubectl describe pod <pod-name>

* **Expected Outcome**: The liveness probe now passes, and the pod does not restart.

**2. Readiness Probe with MongoDB Deployment**

**Step 1: Configure an Incorrect Readiness Probe**

Create the file readiness.yml with an **incorrect readiness probe command**:

yaml

Copy code

apiVersion: apps/v1

kind: Deployment

metadata:

name: mongo

spec:

replicas: 3

selector:

matchLabels:

app: mongo

template:

metadata:

labels:

app: mongo

spec:

containers:

- name: mongo

image: mongo:4.0.8

readinessProbe:

exec:

command:

- mongo

- --eval

- "db1.adminCommand('ping')" # Incorrect command

initialDelaySeconds: 1

periodSeconds: 10

timeoutSeconds: 5

successThreshold: 1

failureThreshold: 2

---

apiVersion: v1

kind: Service

metadata:

name: mongodb-service

spec:

selector:

app: mongo

ports:

- protocol: TCP

port: 27017

targetPort: 27017

Apply the configuration:

bash

Copy code

kubectl apply -f readiness.yml

**Step 2: Monitor Readiness Probe Failure**

Check the pod status and describe events:

bash

Copy code

kubectl get pods

kubectl describe pod <pod-name>

* **Expected Outcome**: The pod will be excluded from the service endpoints due to readiness probe failures.

Check the service endpoint to verify:

bash

Copy code

kubectl describe service mongodb-service

**Step 3: Correct the Readiness Probe Command**

Update the readiness.yml file with the correct command:

yaml

Copy code

command:

- mongo

- --eval

- "db.adminCommand('ping')" # Correct command

Apply the updated configuration:

bash

Copy code

kubectl apply -f readiness.yml

**Step 4: Verify Readiness Probe Success**

Verify that the pod is now ready and included in the service endpoint:

bash

Copy code

kubectl get pods

kubectl describe pod <pod-name>

kubectl describe service mongodb-service

**3. HTTP-based Liveness Probe with Apache HTTP Server**

This section tests HTTP-based liveness probes with different paths.

**Step 1: Create an HTTP-based Liveness Probe with Incorrect Path**

Create the file pod1.yml with an **incorrect HTTP path** for the liveness probe:

yaml

Copy code

apiVersion: v1

kind: Pod

metadata:

name: app-pod2

spec:

containers:

- name: container1

image: httpd:2.4

livenessProbe:

httpGet:

path: /abc # Incorrect path

port: 80

initialDelaySeconds: 10

timeoutSeconds: 5

Apply the configuration:

bash

Copy code

kubectl apply -f pod1.yml

**Step 2: Monitor Pod Behavior**

Check the pod status and events:

bash

Copy code

kubectl get pods

kubectl describe pod app-pod2

* **Expected Outcome**: Since /abc is incorrect, the probe fails, and Kubernetes restarts the container.

**Step 3: Correct the Liveness Probe HTTP Path**

Update the pod1.yml file to correct the HTTP path:

yaml

Copy code

httpGet:

path: / # Correct path

port: 80

Apply the corrected configuration:

bash

Copy code

kubectl apply -f pod1.yml

**Step 4: Verify Corrected Probe**

Check the pod status to confirm the liveness probe is passing, and the pod remains stable:

bash

Copy code

kubectl get pods

kubectl describe pod app-pod2

**Summary**

This lab demonstrated how to set up and troubleshoot Kubernetes health checks using:

* **Liveness Probes** to restart unhealthy pods
* **Readiness Probes** to control traffic to pods based on their readiness status
* **HTTP-based probes** with Apache to check for responsiveness

4o

**Terraform Lab Guide for AWS EC2 Instance Deployment**

**Overview**

In this guide, we’ll use Terraform to create an AWS EC2 instance with predefined configurations. This guide includes all necessary steps, from installing the Terraform CLI to deploying an instance.

**Prerequisites**

* An AWS account with IAM permissions to create EC2 resources.
* **Access Key** and **Secret Access Key** for AWS.
* A CLI environment with vi or any text editor (Linux/Unix preferred).

**Important Security Note**

Never store sensitive information (e.g., AWS access keys) directly in code. Instead, use environment variables or tools like AWS Vault or HashiCorp Vault to handle credentials.

**Step 1: Install Terraform**

Follow the installation instructions for Terraform from the [official HashiCorp documentation](https://developer.hashicorp.com/terraform/tutorials/aws-get-started/install-cli) to download and install the Terraform CLI for your operating system.

To verify installation, run:

bash

Copy code

terraform -v

You should see the installed Terraform version.

**Step 2: Set Up AWS Credentials**

You will use your **AWS Access Key** and **Secret Access Key** in the Terraform configuration file. Keep these credentials secure.

**Step 3: Create a Terraform Configuration File**

**3.1. Navigate to Your Project Directory**

Open your terminal, navigate to the project directory, and create a new file called ec2.tf:

bash

Copy code

mkdir terra-raman

cd terra-raman

vi ec2.tf

**3.2. Define the AWS Provider**

In ec2.tf, specify the AWS provider and your region, access key, and secret key. Below is the basic structure:

hcl

Copy code

provider "aws" {

region = "us-east-1"

access\_key = "AKIAZ7FSO3B5ZGXBDNF3" # Replace with your actual access key

secret\_key = "N7qXqPhmL6mpiXlE1lc+QmjfMxSeD1X4VOmNo+dA" # Replace with your actual secret key

}

**Note**: Using environment variables is more secure. You can export the credentials to your environment as follows:

bash

Copy code

export AWS\_ACCESS\_KEY\_ID="AKIAZ7FSO3B5ZGXBDNF3"

export AWS\_SECRET\_ACCESS\_KEY="N7qXqPhmL6mpiXlE1lc+QmjfMxSeD1X4VOmNo+dA"

If using environment variables, omit the access\_key and secret\_key lines in the provider block.

**3.3. Define the EC2 Instance Resource**

Add the aws\_instance resource to create an EC2 instance:

hcl

Copy code

resource "aws\_instance" "myec2" {

availability\_zone = "us-east-1a"

ami = "ami-0866a3c8686eaeeba" # Amazon Linux 2 AMI ID, ensure this is available in your region

instance\_type = "t2.micro"

tags = {

Name = "raman-firstserver"

}

}

**Explanation of Fields**:

* **availability\_zone**: Specifies the availability zone for the instance.
* **ami**: Amazon Machine Image ID, which is specific to the region.
* **instance\_type**: Specifies the type of instance to create.
* **tags**: Adds metadata to the instance.

**Step 4: Initialize the Terraform Configuration**

Run terraform init to initialize the configuration directory and download the necessary provider plugins.

bash

Copy code

terraform init

**Expected Output**: Terraform will download provider plugins and prepare the directory for use.

**Step 5: Validate the Configuration**

Before deploying, validate the configuration file to ensure syntax and resources are correctly defined.

bash

Copy code

terraform validate

If successful, it will display Success! The configuration is valid. If there are errors, fix them before proceeding.

**Step 6: Preview the Deployment Plan**

Run terraform plan to generate and review an execution plan without making changes. This helps you understand what resources Terraform will create.

bash

Copy code

terraform plan

Look for aws\_instance.myec2 in the output, which indicates that Terraform plans to create this instance.

**Step 7: Apply the Configuration to Deploy Resources**

Run terraform apply to execute the configuration and create resources in AWS.

bash

Copy code

terraform apply

Terraform will ask for confirmation. Type yes to proceed.

**Expected Output**: After a few moments, Terraform will complete and display details of the created instance.

**Step 8: Verify the EC2 Instance**

After successful deployment, verify the EC2 instance on the AWS Management Console:

1. Go to **EC2 Dashboard** on AWS.
2. Check the **Instances** section to see the newly created instance.
3. Confirm that the instance has the tag Name: raman-firstserver.

Alternatively, you can verify using the CLI:

bash

Copy code

aws ec2 describe-instances --filters "Name=tag:Name,Values=raman-firstserver" --query "Reservations[\*].Instances[\*].InstanceId"

**Step 9: Check Terraform State**

Terraform keeps track of resources it manages in a state file. To list the resources created, run:

bash

Copy code

terraform state list

**Expected Output**: The EC2 instance resource (aws\_instance.myec2) should be listed.

**Step 10: Modify EC2 Instance Configuration (Optional)**

You can modify the configuration file (ec2.tf) to change parameters like instance\_type or add security groups. Run terraform plan and terraform apply again to update the infrastructure according to the changes.

**Step 11: Destroy Resources**

To clean up and remove the created resources, run:

bash

Copy code

terraform destroy

Terraform will ask for confirmation. Type yes to proceed. This will delete the EC2 instance and other associated resources.

**Summary**

This guide demonstrated:

* Installing Terraform.
* Setting up AWS credentials and provider configuration.
* Defining and deploying an EC2 instance.
* Verifying resources and managing the Terraform state.
* Modifying and destroying resources as part of the infrastructure lifecycle.

**Important Note**: Always remove sensitive data, like AWS access keys, from code files and environment variables after completing a project.

**Lab Guide: Kubernetes Deployment and Service using Terraform**

**Overview**

This guide will walk you through creating a Kubernetes namespace, deploying a containerized application in it, and exposing it with a NodePort service, all using Terraform.

**Prerequisites**

* **Kubernetes Cluster**: An operational Kubernetes cluster with kubectl and sufficient permissions to create namespaces, deployments, and services.
* **Kubernetes Config File**: Access to a Kubernetes config file (config) located in the same directory as your Terraform files or at a specified path.
* **Terraform**: Ensure that Terraform is installed on your system.

**Step 1: Setup Project Directory and Files**

1. **Create a directory** to organize your Terraform files.

bash

Copy code

mkdir k8s-terraform-deployment

cd k8s-terraform-deployment

1. **Create main configuration files**:
   * main.tf for the Kubernetes provider, namespace, and deployment.
   * svc.tf for the service resource.

**Step 2: Configure the Kubernetes Provider and Namespace**

**2.1. Define the Provider in main.tf**

In main.tf, configure the Kubernetes provider to use your Kubernetes config file:

hcl

Copy code

provider "kubernetes" {

config\_path = "config" # Path to your kubeconfig file

}

This ensures that Terraform will connect to your Kubernetes cluster for all further commands.

**2.2. Create the Namespace**

Add the kubernetes\_namespace resource to define the namespace where the application will be deployed.

hcl

Copy code

resource "kubernetes\_namespace" "raman" {

metadata {

name = "raman1"

}

}

**Step 3: Create the Kubernetes Deployment**

The deployment configuration creates and manages pods for your application, ensuring the specified number of replicas.

**3.1. Define Deployment in main.tf**

Add the following block to create a deployment named dep1 in the raman1 namespace:

hcl

Copy code

resource "kubernetes\_deployment" "dep1" {

metadata {

name = "dep1"

namespace = kubernetes\_namespace.raman.metadata[0].name

}

spec {

replicas = 2 # Number of replicas for high availability

strategy {

type = "Recreate" # Specifies strategy for updating pods

# Uncomment below for RollingUpdate strategy

# type = "RollingUpdate"

# rolling\_update {

# max\_surge = "25%"

# max\_unavailable = "25%"

# }

}

selector {

match\_labels = {

app = "dep1"

}

}

template {

metadata {

labels = {

app = "dep1"

}

}

spec {

container {

name = "my-flask-app"

image = "ramann123/natwest:my-flask-appV1"

port {

container\_port = 5000

}

resources {

requests = {

cpu = "100m"

memory = "64Mi"

}

limits = {

cpu = "500m"

memory = "128Mi"

}

}

}

}

}

}

}

**Explanation**:

* **metadata.name**: Sets the deployment name to dep1.
* **replicas**: Specifies the number of pod replicas.
* **strategy.type**: Chooses the update strategy (Recreate or RollingUpdate).
* **container**: Defines the container details, including the image and resources.
* **resources**: Specifies CPU and memory requests/limits for container resource management.

**Step 4: Define the NodePort Service**

NodePort services expose the application on a static port on each node’s IP, allowing external access.

**4.1. Configure the Service in svc.tf**

Add the following code to svc.tf to define the NodePort service:

hcl

Copy code

resource "kubernetes\_service" "dep1" {

metadata {

name = "dep1-service"

namespace = kubernetes\_namespace.raman.metadata[0].name

}

spec {

type = "NodePort" # Expose service on a specific port

selector = {

app = "dep1" # Matches the label on the pods

}

port {

protocol = "TCP"

port = 5000 # Service port

target\_port = 5000 # Port where the container listens

node\_port = 30000 # External NodePort (must be between 30000 and 32767)

}

}

}

**Explanation**:

* **metadata.name**: Sets the service name.
* **type**: Specifies NodePort to expose the service on a node's IP.
* **selector**: Matches the app label defined in the deployment.
* **port.target\_port**: Routes traffic to the container’s port 5000.
* **node\_port**: Exposes the application on port 30000 on each node.

**Step 5: Initialize and Apply the Terraform Configuration**

**5.1. Initialize the Project Directory**

Run terraform init to download the Kubernetes provider plugin and prepare the configuration.

bash

Copy code

terraform init

**5.2. Validate the Configuration**

To ensure everything is set up correctly, validate the configuration files:

bash

Copy code

terraform validate

**5.3. Plan the Configuration**

Generate an execution plan with terraform plan to verify the resources that Terraform will create.

bash

Copy code

terraform plan

**5.4. Apply the Configuration**

Apply the configuration to create the namespace, deployment, and service.

bash

Copy code

terraform apply

Terraform will prompt for confirmation; type yes to proceed.

**Expected Output**: Terraform should output the resources created, including the deployment and service.

**Step 6: Verify the Deployment**

**6.1. Check Namespace**

To verify the namespace was created, run:

bash

Copy code

kubectl get namespaces

You should see raman1 listed.

**6.2. Check Deployment and Pods**

Verify that the deployment and pods are running correctly:

bash

Copy code

kubectl get deployments -n raman1

kubectl get pods -n raman1

The output should show a deployment named dep1 with two pods (replicas) running in the raman1 namespace.

**6.3. Check Service**

To confirm the NodePort service was created and is accessible, run:

bash

Copy code

kubectl get services -n raman1

The output will show dep1-service with an external NodePort at 30000.

**Step 7: Access the Application**

1. Find the IP address of one of your Kubernetes nodes:

bash

Copy code

kubectl get nodes -o wide

1. Access the application in a web browser or via curl using the <NodeIP>:30000 format:

bash

Copy code

curl http://<NodeIP>:30000

You should see a response from the my-flask-app application.

**Step 8: Clean Up Resources**

To remove the resources created, run:

bash

Copy code

terraform destroy

Confirm with yes to delete the namespace, deployment, and service.

**Summary**

This guide covered:

* Setting up a Kubernetes namespace, deployment, and NodePort service using Terraform.
* Validating and applying Terraform configurations.
* Verifying deployment and accessing the service.

By following this guide, you've learned how to deploy applications in Kubernetes using Infrastructure as Code with Terraform, a crucial skill for managing Kubernetes resources at scale.

4o

**Lab Guide: Kubernetes HPA, Secrets, and Pod Configurations using Terraform**

**Overview**

In this lab, we will:

1. Create a Horizontal Pod Autoscaler (HPA) to dynamically adjust the number of pods based on CPU usage.
2. Define Kubernetes Secrets to store sensitive information.
3. Deploy two different pods that consume these secrets using different methods.

**Prerequisites**

* **Kubernetes Cluster**: A working Kubernetes cluster.
* **Terraform Installed**: Make sure Terraform is installed.
* **Access to Kubernetes Config**: Ensure you have the Kubernetes config file for authentication with the cluster.

**Step 1: Setup Project Directory and Files**

1. **Create a directory** to store Terraform configuration files.

bash

Copy code

mkdir k8s-hpa-secrets-lab

cd k8s-hpa-secrets-lab

1. **Create the following files**:
   * main.tf for namespace, deployment, and Kubernetes provider configuration.
   * svc.tf for service.
   * hpa.tf for HPA configuration.
   * secret.tf for secrets management.
   * podsForSecrets.tf for pod definitions.

**Step 2: Kubernetes Provider, Namespace, and Deployment Configuration**

Refer to the previous lab guide to set up the Kubernetes provider, namespace, deployment, and service. Save these configurations in main.tf and svc.tf.

**Step 3: Configure Horizontal Pod Autoscaler (HPA)**

**3.1. Define HPA in hpa.tf**

Create a hpa.tf file and add the following configuration:

hcl

Copy code

resource "kubernetes\_horizontal\_pod\_autoscaler" "hpa" {

metadata {

name = "myhpa"

namespace = kubernetes\_namespace.raman.metadata[0].name

}

spec {

scale\_target\_ref {

api\_version = "apps/v1"

kind = "Deployment"

name = kubernetes\_deployment.dep1.metadata[0].name

}

min\_replicas = 1

max\_replicas = 5

target\_cpu\_utilization\_percentage = 70

}

}

**Explanation**:

* **scale\_target\_ref**: Specifies the target deployment (dep1) to autoscale.
* **min\_replicas**: Sets the minimum number of replicas to 1.
* **max\_replicas**: Sets the maximum number of replicas to 5.
* **target\_cpu\_utilization\_percentage**: Sets the CPU utilization threshold at 70%.

**Step 4: Create Kubernetes Secrets**

**4.1. Define Secrets in secret.tf**

In secret.tf, configure the my-secrets Kubernetes secret to store sensitive data (username and password) in base64 encoding:

hcl

Copy code

resource "kubernetes\_secret" "my\_secrets" {

metadata {

name = "my-secrets"

namespace = "raman1"

}

data = {

username = base64encode("ramankhanna")

password = base64encode("ramankhanna123")

}

type = "Opaque"

}

**Explanation**:

* **data**: Stores sensitive data encoded in base64 format for security.
* **type**: Sets the secret type to Opaque, the default type used for generic secrets.

**Step 5: Create Pods that Use the Secrets**

**5.1. Pod 1 Configuration: Mount Secrets as Files**

In podsForSecrets.tf, create the first pod configuration to mount secrets as files:

hcl

Copy code

resource "kubernetes\_pod" "myapp\_pod1" {

metadata {

name = "myapp-pod1"

namespace = "raman1"

labels = {

app = "myapp"

}

}

spec {

container {

name = "httpd-container"

image = "httpd"

volume\_mount {

name = "credentials"

mount\_path = "/tmp/creds"

read\_only = true

}

}

volume {

name = "credentials"

secret {

secret\_name = kubernetes\_secret.my\_secrets.metadata[0].name

}

}

}

}

**Explanation**:

* **volume\_mount**: Mounts the secret as a file in /tmp/creds inside the container, making the secret data accessible in the filesystem.
* **volume.secret.secret\_name**: Refers to the name of the secret (my-secrets) to be mounted.

**5.2. Pod 2 Configuration: Use Secrets as Environment Variables**

In the same podsForSecrets.tf, configure the second pod to use secrets as environment variables:

hcl

Copy code

resource "kubernetes\_pod" "myapp\_pod2" {

metadata {

name = "myapp-pod2"

namespace = "raman1"

labels = {

app = "myapp"

type = "front-end"

}

}

spec {

container {

name = "httpd-container"

image = "httpd"

env {

name = "SECRET\_USERNAME"

value\_from {

secret\_key\_ref {

name = kubernetes\_secret.my\_secrets.metadata[0].name

key = "username"

}

}

}

env {

name = "SECRET\_PASSWD"

value\_from {

secret\_key\_ref {

name = kubernetes\_secret.my\_secrets.metadata[0].name

key = "password"

}

}

}

}

}

}

**Explanation**:

* **env.value\_from.secret\_key\_ref**: Injects the secret values into environment variables (SECRET\_USERNAME and SECRET\_PASSWD) within the container.

**Step 6: Initialize and Apply Terraform Configurations**

**6.1. Initialize the Directory**

Run terraform init to initialize the provider and configurations.

bash

Copy code

terraform init

**6.2. Validate the Configuration**

Use terraform validate to check for syntax errors and validate the configuration.

bash

Copy code

terraform validate

**6.3. Apply the Configuration**

To deploy all resources (namespace, deployment, service, HPA, secrets, and pods), run:

bash

Copy code

terraform apply

Confirm with yes to apply the configuration.

**Expected Output**: Terraform should output details of the resources it created, including namespace, deployment, service, HPA, secrets, and pods.

**Step 7: Verification and Testing**

**7.1. Verify the Horizontal Pod Autoscaler**

Check the HPA configuration by running:

bash

Copy code

kubectl get hpa -n raman1

This should display the HPA configuration, target CPU threshold, and the current/desired replicas.

**7.2. Verify Secrets**

Verify that the secret my-secrets is present in the namespace raman1:

bash

Copy code

kubectl get secret my-secrets -n raman1

kubectl describe secret my-secrets -n raman1

**7.3. Check Pods and Validate Secret Access**

List the pods in the namespace to confirm that both myapp-pod1 and myapp-pod2 are running:

bash

Copy code

kubectl get pods -n raman1

* **Pod 1**: Use the following command to access myapp-pod1 and verify that the secret data is mounted in /tmp/creds.

bash

Copy code

kubectl exec -it myapp-pod1 -n raman1 -- ls /tmp/creds

kubectl exec -it myapp-pod1 -n raman1 -- cat /tmp/creds/username

kubectl exec -it myapp-pod1 -n raman1 -- cat /tmp/creds/password

* **Pod 2**: Verify that the secret data is accessible as environment variables.

bash

Copy code

kubectl exec -it myapp-pod2 -n raman1 -- printenv SECRET\_USERNAME

kubectl exec -it myapp-pod2 -n raman1 -- printenv SECRET\_PASSWD

The output should display the values of username and password for both access methods.

**Step 8: Testing Autoscaling**

**8.1. Simulate CPU Load**

To test the HPA, you can simulate a CPU load on dep1 pods by running a command that consumes CPU in an infinite loop.

**8.2. Monitor HPA Behavior**

Run the following command to observe HPA scaling events based on CPU usage:

bash

Copy code

kubectl get hpa -n raman1 -w

**Expected Output**: As CPU usage increases above the threshold, the HPA will scale up the number of pod replicas. When CPU usage falls, the HPA will scale down.

**Step 9: Clean Up**

Once testing is complete, destroy all resources by running:

bash

Copy code

terraform destroy

Confirm with yes to remove all created resources from the Kubernetes cluster.

**Summary**

In this lab, you:

* Created an HPA to automatically adjust pod replicas based on CPU usage.
* Defined Kubernetes secrets to securely store sensitive information.
* Deployed two pods, one with secrets mounted as files and the other with secrets injected as environment variables.

This hands-on lab covers essential Kubernetes operations using Terraform for automation and Infrastructure as Code.

**Lab Guide: Provisioning a GCP Instance and Firewall Using Terraform**

This guide provides detailed steps for creating and managing a GCP virtual machine (VM) and firewall rule using Terraform.

**Prerequisites**

1. **GCP Project**: Ensure you have access to the Google Cloud project ID techlanders-nov2024.
2. **Service Account Key**: Store the JSON credentials file (tech.json) for your GCP service account locally.
3. **Terraform Installed**: Ensure Terraform is installed. Verify with terraform -v.

**Lab Setup and Execution**

**Step 1: Project Directory Setup**

1. Create a project directory for your Terraform configuration:

bash

Copy code

mkdir terra-raman

cd terra-raman

1. Initialize Terraform with a GCP provider setup in main.tf.

**Step 2: Configure GCP Provider in Terraform**

1. Open main.tf and define the GCP provider block:

hcl

Copy code

provider "google" {

credentials = file("tech.json")

project = "techlanders-nov2024"

region = "us-central1"

}

* + **credentials**: Points to the local JSON key file (tech.json).
  + **project**: Uses the GCP project ID techlanders-nov2024.
  + **region**: Specifies the primary region as us-central1.

**Step 3: Configure a GCP Compute Instance Resource**

1. In the same main.tf, add a GCP Compute instance resource:

hcl

Copy code

resource "google\_compute\_instance" "mygcpserver" {

name = "techlanders-gcp-instance"

machine\_type = "f1-micro"

zone = "us-central1-a"

boot\_disk {

initialize\_params {

image = "debian-cloud/debian-11"

}

}

network\_interface {

network = "default"

access\_config {

// Assigns an ephemeral external IP

}

}

tags = ["my-gcp-instance"]

}

* + **name**: Assigns the instance name techlanders-gcp-instance.
  + **machine\_type**: Sets the VM type to f1-micro.
  + **zone**: Specifies the zone us-central1-a.
  + **image**: Loads the Debian 11 OS from debian-cloud.
  + **tags**: Adds a custom tag my-gcp-instance.

**Step 4: Configure a Firewall Rule Resource**

1. Add a firewall resource in main.tf:

hcl

Copy code

resource "google\_compute\_firewall" "var\_demo" {

name = "raman-variables"

network = "default"

allow {

protocol = "tcp"

ports = ["443", "80", "53"]

}

source\_ranges = ["0.0.0.0/0"]

}

* + **name**: Names the firewall rule raman-variables.
  + **allow**: Opens TCP ports for HTTP (80), HTTPS (443), and DNS (53).
  + **source\_ranges**: Allows access from any IP.

**Step 5: Initialize and Apply the Configuration**

1. Initialize Terraform in your project directory:

bash

Copy code

terraform init

1. Run a plan to preview changes:

bash

Copy code

terraform plan

1. Apply the configuration:

bash

Copy code

terraform apply -auto-approve

**Note**: The -auto-approve flag skips manual approval.

**Step 6: Verify Resources in GCP**

1. **Compute Instance**:
   * Go to the GCP Console > Compute Engine > VM instances.
   * Confirm that techlanders-gcp-instance is running in the us-central1-a zone.
2. **Firewall Rule**:
   * Go to the GCP Console > VPC Network > Firewall.
   * Check for the raman-variables rule allowing inbound traffic on ports 80, 443, and 53.

**Step 7: Accessing the Instance**

1. In the GCP Console, note the external IP assigned to the instance.
2. Use SSH to connect:

bash

Copy code

ssh <external-ip>

**Step 8: Cleanup**

1. To delete the resources, run:

bash

Copy code

terraform destroy -auto-approve

1. Confirm resources are deleted in the GCP Console.

This completes the lab setup for creating a GCP Compute Instance and Firewall with Terraform.