**Lab Guide: Kubernetes Liveness and Readiness Probes with MongoDB**

**Objective:**

In this lab, we will learn how to implement **Liveness** and **Readiness** probes in Kubernetes. We will start with incorrect configurations for both probes, then rectify them to demonstrate their behavior. The **Liveness** probe will restart the pod if the application becomes unhealthy, and the **Readiness** probe will remove the pod from the service endpoint if the application is not ready to serve requests.

**Prerequisites:**

* A running Kubernetes cluster with kubectl configured.
* Basic knowledge of Kubernetes deployments and services.
* Docker installed locally (if running the cluster locally).
* MongoDB Docker image: mongo:4.0.8.

**1. Liveness Probe Example:**

The **Liveness Probe** checks whether the application in the container is running. If the check fails, Kubernetes restarts the container.

We will start with an **incorrect** liveness probe command.

**Step 1.1: Create the Incorrect Liveness Probe Configuration**

yaml

Copy code

# Incorrect Liveness Probe Configuration (liveliness.yml)

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, database 'db1' does not exist

initialDelaySeconds: 1

periodSeconds: 10

timeoutSeconds: 5

successThreshold: 1

failureThreshold: 2

**Step 1.2: Apply the Incorrect Liveness Probe Configuration**

bash

Copy code

kubectl apply -f liveliness.yml

**Step 1.3: Observe Pod Behavior**

Monitor the pods to see if they are being restarted due to the incorrect command.

bash

Copy code

kubectl get pods -w

You should observe that the pods are restarting because the command fails (db1 is not a valid MongoDB database).

**Step 1.4: Fix the Liveness Probe**

Now, we will correct the liveness probe command.

yaml

Copy code

# Corrected Liveness Probe Configuration (liveliness.yml)

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

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

initialDelaySeconds: 1

periodSeconds: 10

timeoutSeconds: 5

successThreshold: 1

failureThreshold: 2

**Step 1.5: Apply the Corrected Liveness Probe Configuration**

bash

Copy code

kubectl apply -f liveliness.yml

**Step 1.6: Verify Pod Health**

Monitor the pods to ensure they are no longer restarting.

bash

Copy code

kubectl get pods -w

**2. Readiness Probe Example:**

The **Readiness Probe** determines whether the application is ready to serve traffic. If the probe fails, Kubernetes will remove the pod from the service's endpoints, but it won't restart the pod.

We will start with an **incorrect** readiness probe command.

**Step 2.1: Create the Incorrect Readiness Probe Configuration**

yaml

Copy code

# Incorrect Readiness Probe Configuration (readiness.yml)

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

**Step 2.2: Apply the Incorrect Readiness Probe Configuration**

bash

Copy code

kubectl apply -f readiness.yml

**Step 2.3: Monitor the Pod and Service**

Monitor the pod's readiness status. The pod will not be part of the service because the readiness probe is failing.

bash

Copy code

kubectl get pods -w

kubectl describe svc mongodb-service

In the output of the service description, notice that no endpoints are available because the readiness probe is failing.

**Step 2.4: Fix the Readiness Probe**

Now, we will correct the readiness probe command.

yaml

Copy code

# Corrected Readiness Probe Configuration (readiness.yml)

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

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

**Step 2.5: Apply the Corrected Readiness Probe Configuration**

bash

Copy code

kubectl apply -f readiness.yml

**Step 2.6: Verify Pod and Service Health**

Monitor the pods again to see that they are now ready and part of the service endpoints.

bash

Copy code

kubectl get pods -w

kubectl describe svc mongodb-service

**3. Conclusion:**

* The **Liveness Probe** ensures that Kubernetes restarts a container when it becomes unhealthy.
* The **Readiness Probe** controls whether the pod can accept traffic by adding or removing it from the service endpoints.

By starting with incorrect commands for both probes and then rectifying them, you observed how Kubernetes handles unhealthy containers and readiness issues.

4o

**Lab Guide: Installing Terraform and Deploying AWS EC2 Instance**

**Objective:**

This guide will walk you through installing **Terraform** on an Ubuntu instance, configuring AWS provider credentials, and deploying an EC2 instance using Terraform. You will also explore Terraform commands for validating, applying, refreshing, and destroying infrastructure.

**Prerequisites:**

* Ubuntu system (t2.micro, 8-10 GB storage).
* AWS account with IAM user credentials (Access Key ID and Secret Access Key) configured.
* Basic understanding of AWS services (like EC2) and Terraform.

**Part 1: Install Terraform**

**Step 1.1: Update and Install Prerequisites**

On your Ubuntu system, start by updating the package lists and installing required dependencies:

bash

Copy code

sudo apt-get update && sudo apt-get install -y gnupg software-properties-common

**Step 1.2: Add the HashiCorp GPG Key**

Download and install the HashiCorp GPG key:

bash

Copy code

wget -O- https://apt.releases.hashicorp.com/gpg | gpg --dearmor | sudo tee /usr/share/keyrings/hashicorp-archive-keyring.gpg > /dev/null

**Step 1.3: Verify the GPG Key**

Ensure the key was added successfully by checking its fingerprint:

bash

Copy code

gpg --no-default-keyring --keyring /usr/share/keyrings/hashicorp-archive-keyring.gpg --fingerprint

**Step 1.4: Add the Terraform Repository**

Add HashiCorp's official repository to the package manager sources list:

bash

Copy code

echo "deb [signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg] https://apt.releases.hashicorp.com $(lsb\_release -cs) main" | sudo tee /etc/apt/sources.list.d/hashicorp.list

**Step 1.5: Install Terraform**

After adding the repository, update the package list and install Terraform:

bash

Copy code

sudo apt update

sudo apt-get install terraform

**Step 1.6: Verify Terraform Installation**

Check the Terraform version to confirm it's installed:

bash

Copy code

terraform --version

**Part 2: Create Terraform Configuration**

**Step 2.1: Create a New Directory**

Navigate to your working directory and create the main.tf file that defines your infrastructure.

bash

Copy code

cd ~

vi main.tf

**Step 2.2: Define the AWS Provider and EC2 Instance**

In the main.tf file, define the AWS provider, specify the region, and create an EC2 instance:

hcl

Copy code

provider "aws" {

region = "us-west-1"

access\_key = "AKIAZ7FSO3B5SQQKBSHW" # Replace with your AWS Access Key

secret\_key = "eYRe3RqiQIfMLx3ld6ACC12LkA75CVjcMDWJe+Be" # Replace with your AWS Secret Key

}

resource "aws\_instance" "rk" {

ami = "ami-0d53d72369335a9d6" # AMI ID for your region

instance\_type = "t2.micro"

tags = {

Name = "raman-HelloWorld"

}

}

**Note**: Replace the **Access Key**, **Secret Key**, and **AMI ID** with your own credentials. Avoid hardcoding sensitive credentials in production environments. Consider using environment variables or other methods.

**Part 3: Initialize Terraform**

**Step 3.1: Initialize Terraform Configuration**

Before applying your configuration, run the following command to initialize Terraform. This will download the necessary provider plugins (AWS in this case):

bash

Copy code

terraform init

**Step 3.2: Check the Initialization Files**

After initialization, navigate to the .terraform directory to inspect the downloaded AWS provider plugin:

bash

Copy code

cd .terraform/providers/registry.terraform.io/hashicorp/aws/5.67.0/linux\_amd64/

ls

You should see the Terraform provider plugin binaries in this directory.

**Part 4: Terraform Commands**

**Step 4.1: Validate the Configuration**

To check if your Terraform configuration is syntactically valid, run:

bash

Copy code

terraform validate

**Step 4.2: Plan the Infrastructure**

Run the terraform plan command to see the resources that will be created:

bash

Copy code

terraform plan

This command shows an execution plan of what Terraform will create, change, or destroy.

**Step 4.3: Apply the Infrastructure**

To create the EC2 instance on AWS, apply the configuration:

bash

Copy code

terraform apply

Terraform will prompt you to confirm. Type yes to continue. Once applied, you will see the instance details in the output.

**Part 5: Managing Infrastructure**

**Step 5.1: View Terraform State**

To see the current resources managed by Terraform, use:

bash

Copy code

terraform state list

**Step 5.2: Check the State File**

To manually inspect the state file, you can view the terraform.tfstate file:

bash

Copy code

cat terraform.tfstate

To search for specific information (e.g., tag name), use:

bash

Copy code

cat terraform.tfstate | grep -i Name

**Step 5.3: Refresh the State**

If changes have been made outside of Terraform, refresh the state to sync it with the real infrastructure:

bash

Copy code

terraform refresh

**Step 5.4: Destroy the Infrastructure**

To remove the created EC2 instance and associated resources, run the destroy command:

bash

Copy code

terraform destroy

Terraform will again prompt you to confirm. Type yes to destroy the infrastructure.

**Part 6: Cleanup**

Once the lab is complete, ensure that all AWS resources created during this lab are cleaned up to avoid incurring charges.

**Conclusion:**

You have successfully installed Terraform, deployed an EC2 instance on AWS using Terraform, and explored Terraform commands to manage your infrastructure. This lab demonstrates the full lifecycle of managing infrastructure as code with Terraform—from initialization and validation to deployment and destruction.

**Lab Guide: Understanding Desired State, Current State, and Last Known Configuration in Terraform**

**Objective:**

This lab will guide you through understanding the concepts of **desired state**, **current state**, and **last known configuration** in Terraform. You will deploy an EC2 instance, manually modify it via the AWS Management Console, and observe how Terraform handles discrepancies between the desired state (from the configuration file) and the current state (from the environment).

**Prerequisites:**

* A running Ubuntu instance with Terraform installed.
* AWS account with an IAM user and programmatic access (Access Key ID and Secret Access Key).
* Basic knowledge of AWS EC2 and Terraform commands.

**Part 1: Desired vs. Current State (EC2 Instance)**

**Step 1.1: Create an EC2 Instance Using Terraform**

1. **Create a new main.tf file** for your EC2 instance:

bash

Copy code

vi main.tf

1. **Define the AWS provider and EC2 instance** in the main.tf file:

hcl

Copy code

provider "aws" {

region = "us-west-1"

access\_key = "AKIAZ7FSO3B5SQQKBSHW" # Replace with your AWS Access Key

secret\_key = "eYRe3RqiQIfMLx3ld6ACC12LkA75CVjcMDWJe+Be" # Replace with your AWS Secret Key

}

resource "aws\_instance" "example" {

ami = "ami-0d53d72369335a9d6" # AMI ID for your region

instance\_type = "t2.micro"

tags = {

Name = "example-instance"

}

}

1. **Initialize Terraform** by running:

bash

Copy code

terraform init

1. **Validate and apply the configuration** to create the instance:

bash

Copy code

terraform apply

Confirm with yes when prompted.

**Step 1.2: Verify the Instance in the AWS Management Console**

1. **Go to the AWS Management Console** and navigate to **EC2 > Instances**.
2. **Check the instance type** (it should be t2.micro as per the Terraform configuration).

**Step 1.3: Modify the Instance in the Console**

1. **In the AWS Management Console**, change the instance type from t2.micro to **t2.nano**.
   * Stop the instance first, then modify the instance type to t2.nano, and start it again.

**Step 1.4: Check the Terraform State**

1. **View the current terraform.tfstate file** to see the last known configuration:

bash

Copy code

cat terraform.tfstate

* Notice that the instance type in the state file is still t2.micro because Terraform hasn't been updated with the changes made via the console.

**Step 1.5: Refresh the State**

1. **Run terraform refresh** to update the Terraform state file to reflect the current state:

bash

Copy code

terraform refresh

1. **Check the terraform.tfstate file again**:

bash

Copy code

cat terraform.tfstate

* You should now see that the instance type has been updated to t2.nano in the state file.

**Step 1.6: Apply the Original Configuration**

1. **Apply the original Terraform configuration** with t2.micro still in the main.tf file:

bash

Copy code

terraform apply

* Confirm with yes when prompted.
* Terraform will recognize the drift between the desired state (in the main.tf file) and the current state (in AWS) and change the instance type back to t2.micro.

**Part 2: Challenge with Desired and Current State (Security Groups)**

**Step 2.1: Create the EC2 Instance Again**

1. **Apply the main.tf file again** to ensure an EC2 instance is created:

bash

Copy code

terraform apply

* This will re-create the instance if it has been destroyed or does not exist.

**Step 2.2: Modify the Security Group in the Console**

1. **Go to the AWS Management Console** and navigate to **EC2 > Instances**.
2. **Create a custom security group** with the necessary rules (e.g., allow SSH and HTTP).
3. **Replace the default security group** attached to the instance with the custom one.
   * Go to **Networking > Security groups**, and associate the custom security group with the instance.

**Step 2.3: Refresh the Terraform State**

1. **Run terraform refresh** to update the Terraform state file:

bash

Copy code

terraform refresh

1. **Check the terraform.tfstate file** to see if the security group has been updated:

bash

Copy code

cat terraform.tfstate | grep -i "security\_group"

* You'll notice that the state file is updated with the current configuration (showing the custom security group).

**Step 2.4: Apply the Terraform Configuration Again**

1. **Run terraform apply** again with the original main.tf file (which doesn't specify a security group):

bash

Copy code

terraform apply

* Confirm with yes when prompted.

**Step 2.5: Observe the Outcome**

* **Check the security groups** associated with the instance in the AWS Management Console.
* You'll notice that Terraform does **not** revert the security group to the default because the **desired state** (from the main.tf file) doesn't mention a security group at all.

**Key Learning Points:**

1. **Desired State**: The state defined in your main.tf file is the desired state that Terraform will attempt to enforce.
2. **Current State**: The actual state of resources in the real environment (AWS in this case).
3. **Last Known Configuration**: The last known configuration is stored in the terraform.tfstate file. It reflects the state of the resources after the last successful Terraform run.
4. **Handling Changes Outside of Terraform**: If you make changes to resources outside of Terraform (e.g., via the AWS Console), running terraform refresh updates the state file to reflect those changes.
5. **Missing Resource Blocks**: If a resource (like a security group) is not defined in the desired state (main.tf), Terraform does not manage it, even if it detects that changes were made to that resource.

**Conclusion:**

This lab illustrates how Terraform tracks the **desired state**, **current state**, and **last known configuration** of resources. When resources are modified manually outside of Terraform, running terraform refresh updates the state file, but unless the desired state explicitly defines those resources, Terraform won’t manage or modify them.

4o

**Lab Guide: Creating Resources on Azure and GitHub using Terraform**

**Objective:**

This lab will guide you through deploying resources using Terraform on **Azure** and **GitHub**. You'll configure providers for both platforms and create resources—a simple repository on GitHub and, optionally, some basic configuration on Azure.

**Part 1: Terraform Setup for Azure and GitHub**

**Step 1.1: Install Terraform (if not installed)**

Follow the steps below if Terraform is not already installed on your Ubuntu machine.

1. **Update the package list and install dependencies**:

bash

Copy code

sudo apt-get update && sudo apt-get install -y gnupg software-properties-common

1. **Add the HashiCorp GPG key**:

bash

Copy code

wget -O- https://apt.releases.hashicorp.com/gpg | gpg --dearmor | sudo tee /usr/share/keyrings/hashicorp-archive-keyring.gpg > /dev/null

1. **Add the Terraform repository**:

bash

Copy code

echo "deb [signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg] \

https://apt.releases.hashicorp.com $(lsb\_release -cs) main" | sudo tee /etc/apt/sources.list.d/hashicorp.list

1. **Install Terraform**:

bash

Copy code

sudo apt-get update && sudo apt-get install terraform

1. **Verify the installation**:

bash

Copy code

terraform --version

**Part 2: Terraform Configuration for Azure**

**Step 2.1: Create the Azure Provider File (azure.tf)**

1. **Create the azure.tf file**:

bash

Copy code

vi azure.tf

1. **Configure the Azure provider**:

hcl

Copy code

provider "azurerm" {}

**Note**: This basic provider block doesn't contain authentication information. In a real-world scenario, you should configure authentication using Azure credentials or environment variables. You can refer to Azure Provider Authentication for more information.

**Part 3: Terraform Configuration for GitHub**

**Step 3.1: Create the GitHub Provider File (git.tf)**

1. **Create the git.tf file**:

bash

Copy code

vi git.tf

1. **Add the provider block for GitHub**:

hcl

Copy code

terraform {

required\_providers {

github = {

source = "integrations/github"

version = "~> 6.0"

}

}

}

provider "github" {

token = "<YOUR\_GITHUB\_TOKEN>"

}

resource "github\_repository" "example" {

name = "raman-test"

description = "My awesome codebase"

visibility = "public"

}

**Note**: Replace <YOUR\_GITHUB\_TOKEN> with a personal GitHub access token. To generate a token:

* Go to GitHub Settings > Developer settings > Personal Access Tokens > Generate new token.
* Ensure you select the necessary permissions (e.g., repo, admin:repo\_hook).

**Step 3.2: Initialize the Terraform Providers**

1. **Run the terraform init command** to download the necessary providers (Azure and GitHub):

bash

Copy code

terraform init

1. **Verify the initialization**:
   * Check the output to confirm that both the **azurerm** and **github** providers have been successfully initialized.

**Part 4: Managing GitHub Resources**

**Step 4.1: Validate the Terraform Configuration**

Run the following command to check if the configuration is valid:

bash

Copy code

terraform validate

* You should see a message that the configuration is valid. If there are any issues, fix them before proceeding.

**Step 4.2: Plan the Terraform Changes**

To see what changes Terraform will apply, run the terraform plan command:

bash

Copy code

terraform plan

* This will show you that Terraform plans to create a new GitHub repository named **raman-test**.

**Step 4.3: Apply the Configuration**

Apply the changes to create the GitHub repository:

bash

Copy code

terraform apply

* You will be prompted to confirm the changes by typing yes. Once confirmed, Terraform will create the repository on GitHub.

**Step 4.4: Verify the Repository on GitHub**

1. **Go to your GitHub account** and navigate to **Repositories**.
2. **Check for the repository** named **raman-test**.

**Part 5: Managing Azure Resources (Optional)**

For this lab, we only set up the Azure provider without defining any resources. To extend this guide with actual Azure resources:

1. **Define an Azure resource** (e.g., a storage account) in the azure.tf file:

hcl

Copy code

resource "azurerm\_resource\_group" "example" {

name = "example-resources"

location = "East US"

}

resource "azurerm\_storage\_account" "example" {

name = "examplestoracc"

resource\_group\_name = azurerm\_resource\_group.example.name

location = azurerm\_resource\_group.example.location

account\_tier = "Standard"

account\_replication\_type = "LRS"

}

1. **Run terraform plan and terraform apply** to create these resources.
2. **Verify the resources** in the Azure portal.

**Part 6: Terraform State Management and Updates**

**Step 6.1: Check the Terraform State**

To see what Terraform has created and is managing, run:

bash

Copy code

terraform state list

This command will display a list of all resources tracked in the state file. You should see the GitHub repository listed.

**Step 6.2: Modify the Configuration**

1. **Change the visibility of the GitHub repository** from public to private in the git.tf file:

hcl

Copy code

resource "github\_repository" "example" {

name = "raman-test"

description = "My awesome codebase"

visibility = "private"

}

1. **Run terraform apply** again to apply the changes:

bash

Copy code

terraform apply

1. **Verify the repository's visibility** on GitHub.

**Part 7: Destroying Resources**

**Step 7.1: Destroy the GitHub Repository**

To clean up the resources created by Terraform, run:

bash

Copy code

terraform destroy

* Confirm with yes to delete the GitHub repository.

**Step 7.2: Destroy Azure Resources (Optional)**

If you have created any Azure resources, run terraform destroy to remove them as well:

bash

Copy code

terraform destroy

**Conclusion:**

In this lab, you successfully:

1. Configured Terraform to work with **Azure** and **GitHub** providers.
2. Created a **GitHub repository** using Infrastructure as Code (IaC).
3. Managed state with Terraform to track and apply changes.
4. (Optional) Set up Azure provider for further resource management.

This lab demonstrates how to use Terraform to manage multiple cloud providers and platforms in a unified way.

4o