Terraform Dice Application & Kubernetes Deployment
README

Braden Gant

November 30, 2025

Project Overview

This project provides hands-on practice with Terraform using two small infrastructure stacks.

- **Part 1 – Docker Terraform Project:** Provision a local "mini cloud" consisting of three Docker containers: an Nginx frontend, a Python backend, a Postgres database, and an enhancement.
- **Part 2 – Kubernetes Terraform Project:** Deploy the backend into a small `kind` Kubernetes cluster using the Terraform `kubernetes` provider consisting of a (Namespace + Deployment + Service) with an added enhancement.

Prerequisites

- Windows host with Docker Desktop installed and running
- PowerShell
- Docker image `hashicorp/terraform:1.9.0`
- Network access to pull base images (e.g., `nginx:alpine` , `postgres:16-alpine` , `python:3.11-slim`)
- Terraform CLI installed locally (for the Kubernetes portion)
- `kind` (Kubernetes in Docker) installed and working
- `kubectl` configured to talk to the `kind-cluster` context

Terraform for Part 1 is not installed directly on the host. Instead, all Part 1 Terraform commands are executed inside the official Terraform Docker image. For Part 2, Terraform is ran directly on the host machine.

Repository Layout

The top level of the repository for the Docker portion is organized as follows:

- `main.tf` – Root Terraform and module configuration
- `variables.tf` – Root variable declarations for (`Database_User`, `Database_Password`, `Database_Name`)
- `terraform.tfvars` – Local values for the database variables
- `modules/`
  - `db/main.tf` – Postgres Docker image, volume, and container
  - `backend/main.tf` – Python backend Docker image and container
  - `frontend/main.tf` – Nginx frontend Docker image and container
- `backend/`
  - `Dockerfile` – Builds the Python/Flask dice API image
  - `app.py` – Flask microservice implementing a dice-rolling REST endpoint
- `frontend/`
  - `Dockerfile` – Builds the Nginx image serving static HTML
  - `index.html` – Simple UI for choosing dice and calling the backend API

In a separate directory the Kubernetes top level portion is as follows:

- `terraform-k8s/`
  - `main.tf` – Terraform Kubernetes provider, Namespace, Deployment, Service, ConfigMap.
  - `variables.tf` – variables such as `namespace_name`, `app_image`, `replicas`, and `service_node_port`.

Part 1: Docker Terraform Project

Architecture

Part 1 provisions the following containers on a custom Docker network:

- **Frontend** (`dice-frontend`): Nginx serves `index.html`, which is exposed on the `localhost:8080` and attached to the shared network `dice_app_net`.
- **Backend** (`dice-backend`): Python/Flask is being used as the microservice which exposes `/roll` on the containers `port 5000`, which is then mapped to the host `port 5001`.
- **Database** (`dice-postgres`): Postgres 16 running on `port 5432` with a Docker-managed volume denoted as `dice_postgres_data` for persistent storage.

The backend also exposes an HTTP API:

- `GET /roll?die={NdS}` where `N` is the number of dice and `S` is the sides (e.g., `d6`, `3d6`, `d20`).

The frontend is utilizing a static HTML/JavaScript page that allows for the user to choose a die, it then sends a request to the backend API, and displays the random results and total.

Running the Terraform Commands

All Terraform commands for Part 1 are executed from the project root utilizing the Terraform Docker image.

```
C:\Users\Braden\Desktop\Terraform\terraform-dice
```

Initialize

```
docker run --rm -it ‘
  -v "${PWD}:/workspace" ‘
  -w /workspace ‘
  -v /var/run/docker.sock:/var/run/docker.sock ‘
  hashicorp/terraform:1.9.0 ‘
  init
```

This downloads the `kreuzwerker/docker` provider and sets up the working directory.

Plan

```
docker run --rm -it ‘
  -v "${PWD}:/workspace" ‘
  -w /workspace ‘
  -v /var/run/docker.sock:/var/run/docker.sock ‘
  hashicorp/terraform:1.9.0 ‘
  plan -var-file="terraform.tfvars"
```

The planing stage shows the creation of:

- One `docker_network` (`dice_app_net`)
- One `docker_volume` (`dice_postgres_data`)
- Three `docker_image` resources
- Three `docker_container` resources

Apply

```
docker run --rm -it ‘
  -v "${PWD}:/workspace" ‘
  -w /workspace ‘
  -v /var/run/docker.sock:/var/run/docker.sock ‘
  hashicorp/terraform:1.9.0 ‘
  apply -var-file="terraform.tfvars"
```

After confirming with `yes`, Terraform provisions the full stack. On success, the outputs are:

- `Backend_URL = "http://localhost:5001/roll?die=d20"`
- `Frontend_URL = "http://localhost:8080"`

Destroy

The following command sequence is used to tear down all of the containers, images, networks, and volumes.

```
docker run --rm -it ‘
  -v "${PWD}:/workspace" ‘
  -w /workspace ‘
  -v /var/run/docker.sock:/var/run/docker.sock ‘
  hashicorp/terraform:1.9.0 ‘
  destroy -var-file="terraform.tfvars"
```

Variables and Secrets

The root `variables.tf` defines the following inputs for the Postgres Database

- `Database_User` – Postgres username
- `Database_Password` – Postgres password (marked `sensitive = true`)
- `Database_Name` – Name of the application database

Concrete values are supplied within the `terraform.tfvars`

```
Database_User = "diceuser"
Database_Password = "supersecretpassword"
Database_Name = "dicelog"
```

These variables are then passed into the `db` module, which configures the Postgres container via the environment variables `POSTGRES_USER`, `POSTGRES_PASSWORD`, and `POSTGRES_DB`.

Project Enhancement (Part 1)

For the required enhancement in Part 1, I first added persistent storage for the Postgres database using a dedicated Docker volume called, `dice_postgres_data`. In the Terraform configuration, this is modeled as a `docker_volume` resource that is mounted into the Postgres container at `/var/lib/postgresql/data`. Allowing for database state to survive container restarts, image rebuilds, and `terraform apply`/`destroy` cycles, instead of being lost whenever the container is recreated.

In addition to persistence, I also implemented a simple frontend GUI and a JSON-based API for the backend. The frontend is an Nginx container that serves a static HTML/JavaScript page where the user can select a die (for example `d20` or `3d6`), click a button, and see the result in the browser. This was done by the page calling the Python/Flask backend, which then returns the structured JSON output `{"count":1,"die":"1d20","results":[12],"sides":20,"total":12}`. This makes the service usable from both the GUI and within the terminal and browser using tools like `curl`. Together, with the persistent Postgres volume, and the frontend GUI/JSON API, this enhancement demonstrates hopw Terraform can manage a small but fully functioning web application stack on a local cloud contained network.

Part 1 Summary

In Part 1, the project implements a dice application as a three-tier Docker stack that is fully managed by Terraform. The configuration is organized into separate modules under `modules/` denoted as `db`, `backend`, and `frontend`, where each has its own `main.tf` file that encapsulates the Docker resources for that component. At the root level, a custom Docker network `dice_app_net` is created using the `docker_network` resource, and each module defines the appropriate `docker_image` and `docker_container` resources in order to run its service on the shared network. Postgres database credentials are then modeled as Terraform variables and supplied through a `terraform.tfvars` file, with the password marked as sensitive so it does not appear in plaintext in logs or outputs. The frontend module maps the container port 80 to `localhost:8080`, which is also exposed as a Terraform output, while the backend module exposes the dice-rolling API on the host port 5001. For the implementation of an enhancement beyond the baseline requirements, I added the Postgres database that is configured with a dedicated Docker volume `dice_postgres_data` so that its

data can persist across container restarts and rebuilds. Together, these pieces satisfy all of the Part 1 requirements while keeping the infrastructure modular, repeatable, and easy to manage with `terraform apply` and `terraform destroy`.

Part 2: Kubernetes Terraform Project

Part 2 migrates the dice roller backend into a small Kubernetes cluster managed by Terraform. Instead of running the container directly within the Docker, the application is deployed to a `kind` cluster and exposed via a Kubernetes Service.

All of the Kubernetes-related Terraform code was deployed within:

```
C:\Users\Braden\Desktop\Terraform\terraform-k8s
```

Architecture

The Kubernetes stack consists of:

- A `kind` cluster named `dice-cluster`.
- A dedicated **Namespace** called `dice-app`.
- A **Deployment** of the `dice-backend` running the `dice-backend:latest` image.
- A **NodePort Service** `dice-backend-service` that exposes the backend on node port `30001`.
- A **ConfigMap** in `dice-settings` which is used for the enhancement to inject a configuration into the pod (e.g. `DEFAULT_DIE`).

The dice backend still listens on container port 5000 and exposes the same HTTP API:

```
GET /roll?die=d20
```

Cluster Setup

Before running Terraform for Part 2, the `kind` cluster and tools are set up in the following way:

1. Creating the cluster:

```
kind create cluster --name dice-cluster
```

2. Pointing `kubectl` at the new cluster:

```
kubectl config use-context kind-dice-cluster
kubectl get nodes
```

3. Building the backend image locally:

```
docker build -t dice-backend:latest
```

4. Loading the image into the `kind` cluster so it can be utilized by Pods:

```
kind load docker-image dice-backend:latest --name dice-cluster
```

Terraform Configuration

For Part 2 Terraform is ran directly on the host in `terraform-k8s/main.tf`, where the `kubernetes` provider is configured to read the user's kubeconfig file:

```
terraform {
  required_providers {
    kubernetes = {
      source = "hashicorp/kubernetes"
      version = "~> 2.25"
    }
  }
}

provider "kubernetes" {
  config_path = "C:/Users/Braden/.kube/config"
}
```

The namespace is then created as:

```
resource "kubernetes_namespace" "dice" {
  metadata {
    name = var.namespace_name # defaults to "dice-app"
  }
}
```

ConfigMap Enhancement

For the enhancement in Part 2, a ConfigMap is used to provide configuration data to the backend Pod:

```
resource "kubernetes_config_map" "dice_settings" {
  metadata {
    name = "dice-settings"
    namespace = kubernetes_namespace.dice.metadata[0].name
  }

  data = {
    DEFAULT_DIE = "d20"
    APP_NAME = "Terraform Dice Demo"
  }
}
```

The Deployment then reads `DEFAULT_DIE` from the ConfigMap and exposes it as an environmental variable inside the container.

Deployment and Service

The Deployment then runs the `dice-backend` container using the locally loaded image:

```
resource "kubernetes_deployment" "dice_app" {
  metadata {
    name = "dice-backend"
    namespace = kubernetes_namespace.dice.metadata[0].name
    labels = { app = "dice-backend" }
  }

  spec {
    replicas = var.replicas

    selector {
      match_labels = { app = "dice-backend" }
    }

    template {
      metadata {
        labels = { app = "dice-backend" }
      }

      spec {
        container {
          name = "dice-backend"
          image = var.app_image # "dice-backend:latest"
          image_pull_policy = "IfNotPresent"

          port {
            container_port = 5000
          }

          env {
            name = "DEFAULT_DIE"
            value_from {
              config_map_key_ref {
                name = kubernetes_config_map.dice_settings.metadata[0].name
                key = "DEFAULT_DIE"
              }
            }
          }
        }
      }
    }
  }
}
```

The Service then exposes the Deployment using a NodePort:

```
resource "kubernetes_service" "dice_service" {
  metadata {
    name = "dice-backend-service"
    namespace = kubernetes_namespace.dice.metadata[0].name
  }

  spec {
    selector = {
      app = "dice-backend"
    }

    port {
      port = 5000
      target_port = 5000
      node_port = var.service_node_port # defaults to 30001
    }

    type = "NodePort"
  }
}
```

Running Terraform Commands (Part 2)

All commands for the Kubernetes project are executed from:

`C:\Users\Braden\Desktop\Terraform\terraform-k8s`

Initialize

```
terraform init
```

Plan

```
terraform plan
```

The plan shows creation of:

- one Namespace (`dice-app`),
- one ConfigMap (`dice-settings`),
- one Deployment (`dice-backend`),
- one Service (`dice-backend-service`).

Apply

```
terraform apply
```

After confirming with `yes`, the resources are then created, where they can then be inspected with:

```
kubectl get all -n dice-app
```

Accessing the Dice API

Because the Service is a NodePort on port 30001, and the cluster is running in Docker via kind, the simplest way to access it from the host is to use `kubectl port-forward`:

```
kubectl port-forward -n dice-app svc/dice-backend-service 30001:5000
```

While port-forwarding is running, the dice API is available at:

```
curl "http://localhost:30001/roll?die=d20"
```

Which then returns a JSON response:

```
{"count":1,"die":"1d20","results":[12],"sides":20,"total":12}
```

Destroy

To remove all Kubernetes resources created by Terraform:

```
terraform destroy
```

Afterwards, the `dice-app` namespace is empty:

```
kubectl get all -n dice-app
```

The cluster itself can then be removed with:

```
kind delete cluster --name dice-cluster
```

Project Enhancement (Part 2)

For the enhancement in Part 2, I focused on separating configuration from the container image by creating a Kubernetes ConfigMap and wiring it into the Deployment. The ConfigMap `dice-settings` stores key–value pairs such as `DEFAULT_DIE = "d20"` and an `APP_NAME` label for the service. In the Terraform configuration, this ConfigMap is created in the `dice-app` namespace and then referenced from the `dice-backend` Deployment via an `env` block that uses `config_map_key_ref` to populate environment variables inside the container. This allows for the default die and other settings to be changed by updating the ConfigMap and reapplying Terraform, without rebuilding the entire Docker image or modifying the application code. Combined with the NodePort Service and port-forwarding this is used to verify the `/roll?die=d20` endpoint from the host. Overall, this enhancement demonstrates the Kubernetes workflow where configuration is externalized and managed declaratively alongside the rest of the cluster resources.

Part 2 Summary

In Part 2, the project moves the dice backend into a lightweight Kubernetes cluster managed by Terraform. I first installed and used `kind` to create a local cluster named `dice-cluster`, and configured the `kubectl` to talk to it via the `kind-dice-cluster` context. The setup was then verified with `kubectl get nodes`. Instead of the Docker provider, the `terraform-k8s/main.tf` configuration uses the official `kubernetes` provider, allowing for Terraform to create and manage the cluster resources directly. The infrastructure for the application is then defined using a dedicated namespace called `dice-app`, and a Deployment called `dice-backend` that runs the `dice-backend:latest` image. The NodePort Service `dice-backend-service` would then expose the backend to tghe node

port 30001 so that it can be accessed from the host. As an enhancement, I introduced a ConfigMap named `dice-settings` that stores configuration values such as `DEFAULT_DIE` and injects them into the backend pod via environmental variables. Together, these pieces demonstrate how Terraform can declaratively manage a small Kubernetes-based deployment rather than just standalone Docker containers.