INT 332 – DevOps Virtualization and Configuration Management

Create a Spotify playlist with Terraform

Submitted To –
Arshiya mam

Submitted By –
E ARUNKUMAR
KO129 - 03
12221881

Abstract

This project demonstrates the use of **Terraform**, a popular Infrastructure as Code (IaC) tool, to automate the creation of a **Spotify playlist**. By leveraging a **community provider for Spotify**, Terraform can query Spotify's music database, search for tracks based on artist or album, and create public playlists using Spotify's Web API.

The authentication is handled via a **Dockerized Spotify Authorization Proxy Server**, and the project uses environment variables to manage sensitive API credentials. Terraform data sources retrieve music information, and Terraform resources are used to build and deploy the playlist.

This project is a creative use-case of Terraform beyond traditional cloud infrastructure and shows how IaC principles can be applied to third-party services like Spotify.

Introduction

In today’s automation-driven digital era, Infrastructure as Code (IaC) has become a foundational practice that enables developers and operations teams to manage and provision resources efficiently and consistently. While IaC is widely used in cloud infrastructure, its flexibility extends beyond traditional use cases. This project demonstrates how Terraform, a powerful IaC tool, can be used to interact with third-party APIs like **Spotify** to automate playlist creation.
By integrating Terraform with the Spotify Web API through a custom provider and authorization flow, this project showcases how developers can manage music playlists programmatically. It reflects the growing trend of applying DevOps methodologies to diverse services, emphasizing automation, repeatability, and configuration as code in modern software ecosystems.

Tools & Technologies Used

Terraform: An open-source Infrastructure as Code (IaC) tool used to define and provision Spotify resources like playlists and tracks declaratively.

Docker: Used to run the Spotify authorization proxy server, enabling secure authentication with the Spotify Web API via containerized services.

Spotify API: Provides access to Spotify's track, album, and playlist data, enabling Terraform to search songs and create playlists programmatically.

GitHub: Hosted the example Terraform configuration files used for automation.

Linux Terminal: Used to execute Terraform commands such as init, plan, and apply, and to manage the Dockerized authorization flow.

Architecture Diagram

```
+-------------------------+
|  Terraform (.tf files)  |
|  - Provider config      |
|  - Data sources         |
|  - Playlist resource    |
+-----------+-------------+
            |
            |  API Key (from proxy)
            ↓
+-------------------------+
| Spotify Auth Proxy (Docker) |
|  - Runs locally on port 27228 |
|  - Serves auth tokens     |
+-----------+-------------+
            |
            | Authenticated requests
            ↓
+--------------------------+
|     Spotify Web API      |
|  - Search tracks         |
|  - Create playlist       |
+--------------------------+
```

CI/CD Workflow

CI (Continuous Integration) Process

1. **Terraform Configuration Prepared**

   The developer writes .tf files specifying provider, authentication, data sources, and playlist resource.

   The configuration includes dynamic search for tracks based on artist or album using Spotify's API.

2. **Authorization Server Setup**

   The Spotify Authorization Proxy is run via Docker.

   Developer logs into their Spotify Developer account and retrieves the API token through the proxy interface.

3. **API Key Injection**

   The API key obtained from the Dockerized auth server is injected into the terraform.tfvars file as a secure variable.

4. **Terraform Initialization**

   Terraform is initialized using terraform init, which downloads and installs the Spotify provider.

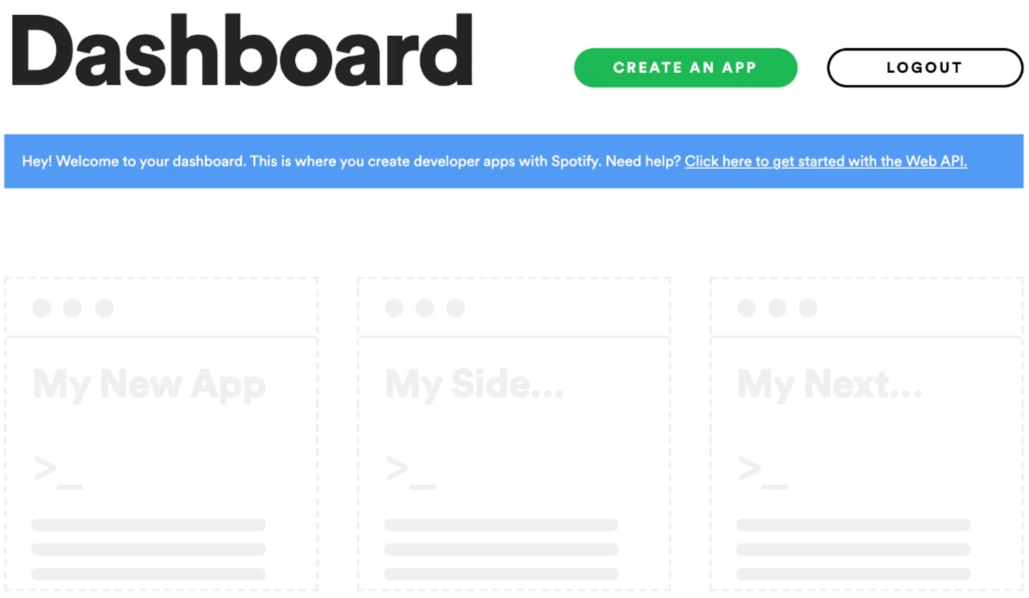
5. **Validation and Plan**

   Developer runs terraform plan to review changes and ensure that the playlist resource will be created correctly.

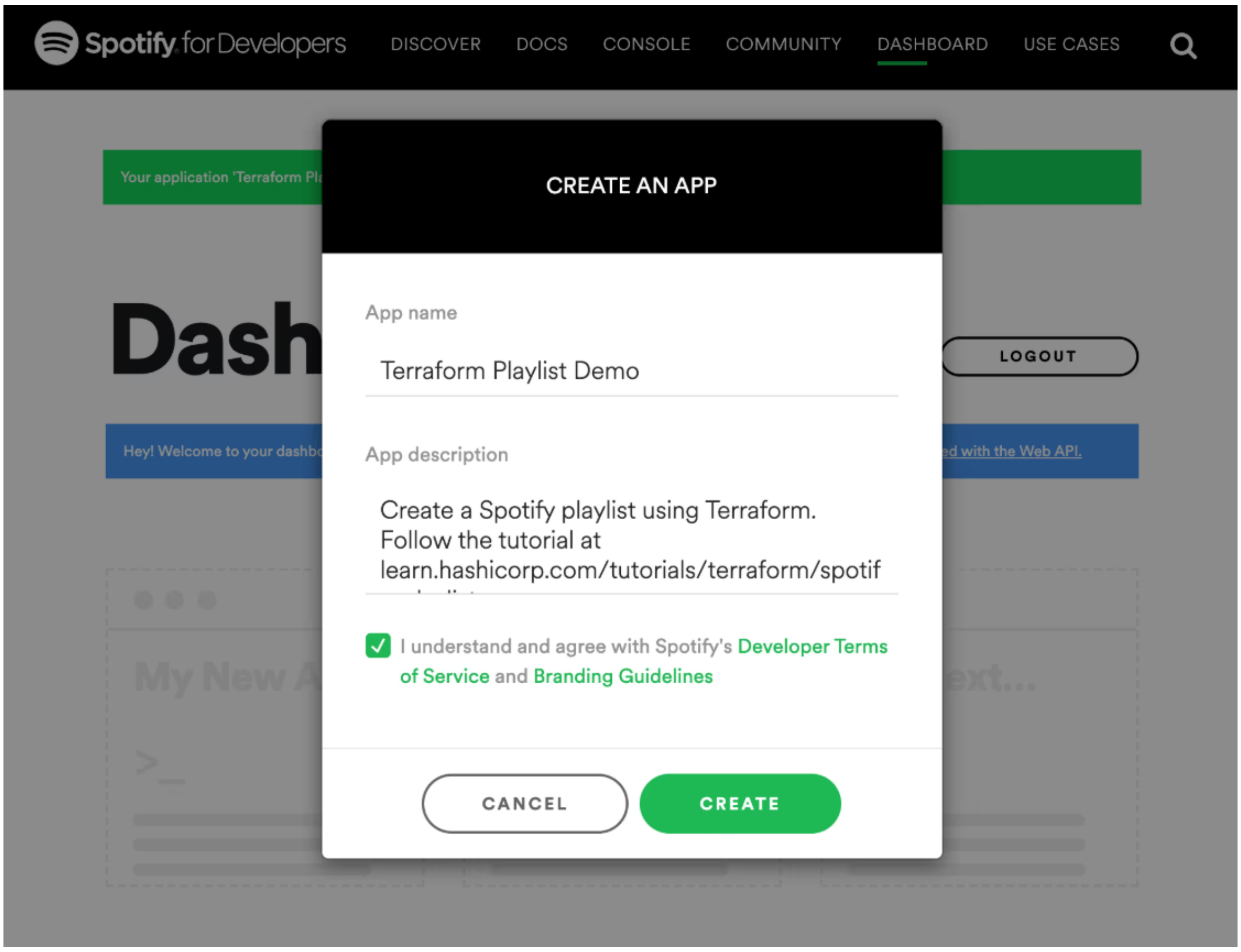
CD (Continuous Delivery) Process

1. **Playlist Provisioning**
   - On running terraform apply, Terraform communicates with the Spotify Web API using the authenticated proxy server.
   - The playlist is created with the defined name, description, and selected tracks.
2. **Output and Verification**
   - Once created, the output displays the playlist URL (e.g., https://open.spotify.com/playlist/...).
   - The playlist is immediately live and can be verified and shared.
3. **Configuration Changes**
   - Updates to the .tf files (like changing the artist, adding specific track IDs, or modifying the playlist name) can be made.
   - Running terraform apply again applies the new configuration, updating the playlist accordingly.

Docker


Dashboard

Hey! Welcome to your dashboard. This is where you create developer apps with Spotify. Need help? Click here to get started with the Web API.



CREATE AN APP

App name

Terraform Playlist Demo

App description

Create a Spotify playlist using Terraform. Follow the tutorial at learn.hashicorp.com/tutorials/terraform/spotif

✓ I understand and agree with Spotify's Developer Terms of Service and Branding Guidelines

Once Spotify creates the application, find and click the green **Edit Settings** button on the top right side.

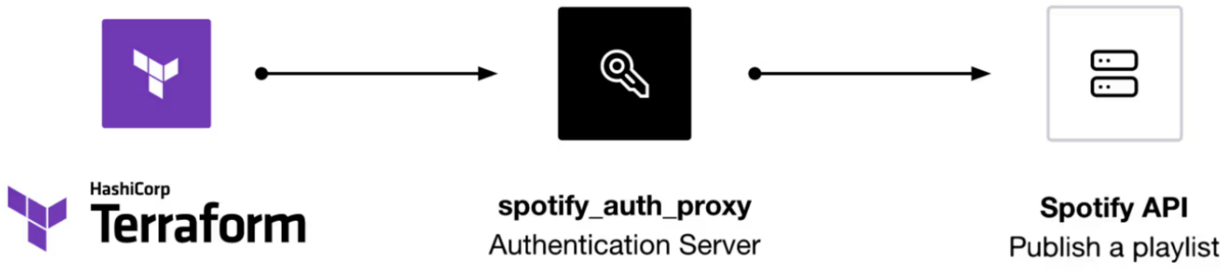
Copy the URI below into the **Redirect URI** field and click **Add** so that Spotify can find its authorization application locally on port 27228 at the correct path. Scroll to the bottom of the form and click **Save**.

```
http://localhost:27228/spotify_callback
 export
SPOTIFY_CLIENT_REDIRECT_URI=http://localhost:27228/spotify_callback
```

Run authorization server

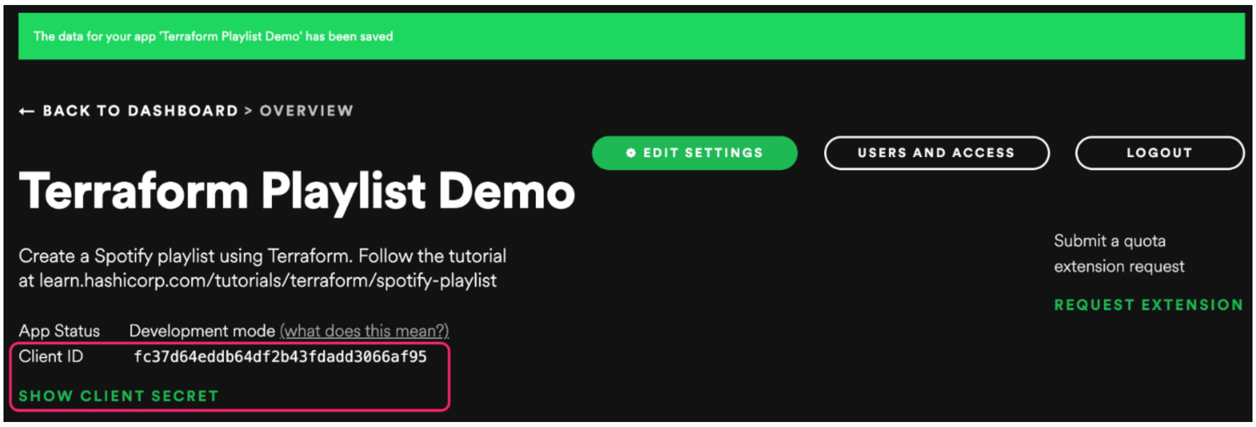
Now that you created the Spotify app, you are ready to configure and start the authorization proxy server, which allows Terraform to interact with Spotify.

Return to your terminal and set the redirect URI as an environment variable, instructing the authorization proxy server to serve your Spotify access tokens on port 27228.



Next, create a file called .env with the following contents to store your Spotify application's client ID and secret.

Copy the **Client ID** from the Spotify app page underneath your app's title and description, and paste it into .env as your SPOTIFY_CLIENT_ID.


The data for your app 'Terraform Playlist Demo' has been saved

← BACK TO DASHBOARD > OVERVIEW

Terraform Playlist Demo

Create a Spotify playlist using Terraform. Follow the tutorial at learn.hashicorp.com/tutorials/terraform/spotify-playlist

Submit a quota extension request

REQUEST EXTENSION

App Status Development mode (what does this mean?)

Client ID fc37d64eddb64df2b43fdadd3066af95

SHOW CLIENT SECRET

.

Docker Compose / Stack File (Optional)

Docker

```
    version: "3.8"

    services:
      spotify-auth-
      proxy:
        image:
      ghcr.io/conrad
      ludgate/spotif
      y-auth-
      proxy:latest

      container_name
      : spotify-
      auth-proxy
        ports:
          -
      "27228:27228"
        env_file:
          - .env
```

docker run --rm -it -p 27228:27228 --env-file ./.env ghcr.io/conradludgate/spotify-auth-proxy

Visit the authorization server's URL by visiting the link that your terminal output lists after Auth:.

The server will redirect you to Spotify to authenticate. After authenticating, the server will display Authorization successful, indicating that the Terraform provider can use the server to retrieve access tokens.

Clone example repository

git clone https://github.com/hashicorp-education/learn-terraform-spotify

Change into the directory.

cd learn-terraform-spotify

Explore the configuration

Open main.tf. This file contains the Terraform configuration that searches Spotify and creates the playlist. The first two configuration blocks in the file:

- **configure Terraform itself and specify the community provider that Terraform uses to communicate with Spotify.**
- **configure the Spotify provider with the key you set as a variable.**

```
terraform {
  required_providers {
    spotify = {
      version = "~> 0.1.5"
      source  = "conradludgate/spotify"
    }
  }
}

provider "spotify" {
  api_key = var.spotify_api_key
}
```

The next block defines a Terraform data source to search the Spotify provider for Dolly Parton songs.

```
data "spotify_search_track" "by_artist" {
  artists = ["Dolly Parton"]
  #  album = "Jolene"
  #  name  = "Early Morning Breeze"
}
```

The next block uses a Terraform resource to create a playlist from the first three songs that match the search in the data source block.

```
resource "spotify_playlist" "playlist" {
  name        = "Terraform Summer Playlist"
  description = "This playlist was created by Terraform"
  public      = true

  tracks = [
    data.spotify_search_track.by_artist.tracks[0].id,
    data.spotify_search_track.by_artist.tracks[1].id,
    data.spotify_search_track.by_artist.tracks[2].id,
  ]
}
```

```
resource "spotify_playlist" "playlist" {
  name        = "Terraform Summer Playlist"
  description = "This playlist was created by Terraform"
  public      = true

  tracks = [
    data.spotify_search_track.by_artist.tracks[0].id,
    data.spotify_search_track.by_artist.tracks[1].id,
    data.spotify_search_track.by_artist.tracks[2].id,
  ]
}
```

Open outputs.tf, which defines an output value for the URL of the playlist.

```
output "playlist_url" {
  value       = "https://open.spotify.com/playlist/${spotify_playlist.playlist.id}"
  description = "Visit this URL in your browser to listen to the playlist"
}
```

Set the API key

```
mv terraform.tfvars.example terraform.tfvars
```

```
spotify_api_key = "..."
```

```
variable "spotify_api_key" {
  type        = string
  description = "Set this as the APIKey that the authorization proxy server outputs"
}
```

Install the Spotify provider

```
terraform init
```

Create the playlist

```
terraform apply
```

Confirm the apply with a yes, and Terraform will create your playlist

```
  Enter a value: yes

spotify_playlist.playlist: Creating...
spotify_playlist.playlist: Creation complete after 1s [id=40bGNifvqzwjO8gHDvhbB3]

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

Challenges Faced

Challenges Faced

1. **Authentication & OAuth Flow**
   - **Issue**: Setting up Spotify's OAuth 2.0 flow manually was complex.
   - **Solution**: Used a Dockerized authorization proxy server to handle token exchange securely and automatically.
2. **Environment Variable Management**
   - **Issue**: Spotify credentials (client ID and secret) needed secure handling to avoid exposure.
   - **Solution**: Created a .env file and excluded it from version control; passed it via docker run or docker-compose.
3. **Token Expiry Handling**
   - **Issue**: Spotify access tokens expire after a short time.
   - **Solution**: Reauthenticated periodically by reusing the proxy's /authorize endpoint.
4. **Provider Compatibility**
   - **Issue**: Terraform Spotify provider (conradludgate/spotify) is a third-party plugin with limited support.
   - **Solution**: Thoroughly tested queries using terraform plan to avoid runtime errors.
5. **Track Index Errors**
   - **Issue**: Not all artists return enough tracks for playlist creation (tracks[0], tracks[1], etc.).
   - **Solution**: Added checks in Terraform and tested with multiple known artists like "Dolly Parton".

Results / Output

1. Playlist Creation

- **A Spotify playlist was successfully created using Terraform with the following parameters:**
  - **Name:** ***Terraform Summer Playlist***
  - **Description:** ***This playlist was created by Terraform***
  - **Public: Yes**
  - **Number of tracks: 3 (customizable)**

2. Terraform Terminal Output

```
Apply complete! Resources: 1 added.
```

3. Authentication Proxy Logs

```
APIKey: eyJhbGciOiJIUzI1NiIs...
Token:  xxxxxxxxxxxxxxxxxxx
Auth:   http://localhost:27228/authorize?token=xxxxxxxxxxxxxxxxxx
```

Conclusion

The described setup for handling increased traffic, ensuring consistency, and accelerating the development cycle through Docker and Jenkins-based CI/CD pipelines offers significant benefits for application deployment. By containerizing the application, Docker ensures that the app functions consistently across various environments, eliminating discrepancies between development, staging, and production setups. The pipeline automation reduces the time spent on deployment, allowing developers to focus more on writing code, which leads to faster development cycles.

Security can be enhanced by regularly scanning Docker images for vulnerabilities and implementing robust access controls. Automated testing, such as unit and integration tests within the CI pipeline, ensures early error detection, preventing issues from propagating further down the development process.

Expanding the deployment strategy to multiple cloud platforms like AWS, GCP, or Azure would further improve the availability and redundancy of the application, ensuring resilience in the face of failures. These improvements collectively optimize the efficiency, reliability, and overall performance of the application deployment process, leading to a more robust and scalable infrastructure.

By leveraging resources such as the Maven, Jenkins, and Spring Boot documentation, developers can deepen their understanding of the tools involved and continue to improve and optimize their CI/CD pipeline and cloud deployments.