

# Terraform Developer





## WORKFORCE DEVELOPMENT

# Content Usage Parameters

**Content** refers to material including instructor guides, student guides, lab guides, lab or hands-on activities, computer programs, etc. designed for use in a training program

1

Content is subject to  
copyright protection

2

Content may only be  
leveraged by students  
enrolled in the training  
program

3

Students agree not to  
reproduce, make  
derivative works of,  
distribute, publicly perform  
and publicly display  
content in any form or  
medium outside of the  
training program

4

Content is intended as  
reference material only to  
supplement the instructor-  
led training

Lab page

<https://jrueels.github.io/tf-dev>





# TERRAFORM AND PROVIDERS



# Terraform



A Terraform provider is a translator that helps Terraform talk to different systems, like a cloud service, an API, or a database. When you tell Terraform what you want to build or manage (e.g., a virtual machine), the provider figures out how to make that happen in the real world.



Terraform is Infrastructure as Code, allowing you to write, manage, and version control your infrastructure just like you would with application code.

The declarative approach in Infrastructure as Code (IaC), like Terraform, focuses on defining the desired end state of infrastructure, making it simpler and easier to manage. It ensures idempotency, meaning configurations can be safely reapplied without unintended changes, reducing errors and manual effort. The clear, human-readable format improves understanding and maintenance

# Infrastructure as code - continued



Declarative IaC also enhances consistency and scalability. Configurations can be versioned in tools like Git, enabling tracking, reviews, and reusability across environments. Automated change detection and planning provide visibility into updates, ensuring safer and more predictable infrastructure management.



# Resource management



Resource management under Terraform essentially means performing CRUD operations on the compute, storage, or network resources in the context of cloud platforms. Additionally, Terraform supports multiple IaaS and PaaS platforms. Some of the providers available in the Terraform registry also help manage a variety of components such as container orchestration (K8s), CI/CD pipelines, asset management, etc.

# Terraform provider



The Terraform architecture relies on plugins to satisfy various vendor requirements for their specific resource management. The diagram below shows a very high-level overview of this architecture.

When we install Terraform, the core component is also installed. This component is responsible for carrying out the init-plan-apply-destroy lifecycle of the resources, setting the standard for managing any kind of resource.

# Terraform provider

The core Terraform component does not implement the logic to interact with each vendor's APIs. Implementing CRUD operations or plan-apply-destroy logic for each vendor resource would cause unnecessary bloating of memory resources both on the network and storage of the host system.





# Terraform provider communication

Terraform Core only includes the hashicorp provider. To manage other resources (AWS, Azure, VMware, etc.) you must install the provider (plugin). Terraform Core communicates with plugins via remote procedure calls.



# Terraform provider communication



```
aws_resource.tf

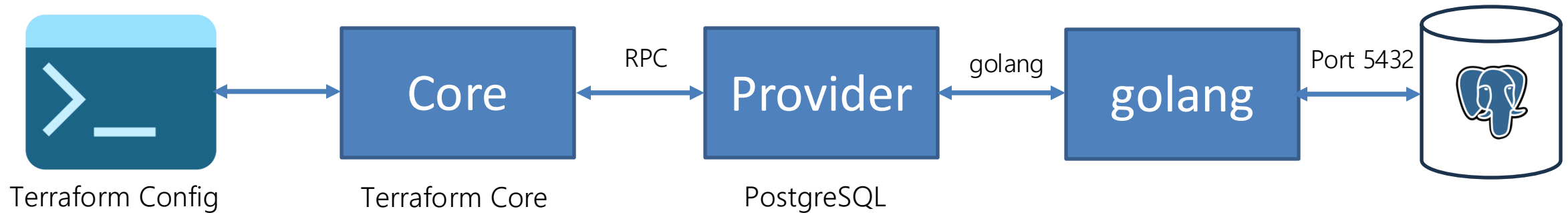
resource "aws_instance" "example" {
  ami           = "ami-123456"
  instance_type = "t2.micro"
}
```

Terraform Core communicates with providers using gRPC (Remote Procedure Call - RPC), enabling Terraform to delegate resource management to providers, which interact with external APIs to create, read, update, and delete infrastructure resources.

- Decoupled Architecture: Terraform Core and providers are separate processes, connected via RPC, allowing independent updates and extensibility.
- Lifecycle Management: Terraform Core invokes provider functions (Create, Read, Update, Delete) via RPC to manage infrastructure state.
- State Synchronization: Providers fetch current resource states, compare with desired configurations, and execute necessary changes.
- Performance and Scalability: Using gRPC instead of older plugin protocols improves efficiency, allowing multi-provider support and parallel execution.

# Terraform is modular

Terraform's modular architecture, powered by gRPC, enables it to run one or multiple providers simultaneously, ensuring efficient communication between Terraform Core and external APIs. This decoupled design improves scalability, supports parallel execution across providers, and allows independent updates for seamless infrastructure management.





# Terraform is modular

Terraform's modular architecture, powered by gRPC, enables it to run one or multiple providers simultaneously, ensuring efficient communication between Terraform Core and external APIs. This decoupled design improves scalability, supports parallel execution across providers, and allows independent updates for seamless infrastructure management.



# POP QUIZ: DISCUSSION

Why build a custom provider?



© 2024 by Innovation In Software Corporation



# POP QUIZ: DISCUSSION

Why build a custom provider?

- Private cloud vendors, platform applications, or services exposing functionality via APIs.





# POP QUIZ: DISCUSSION

Why build a custom provider?

- Private cloud vendors, platform applications, or services exposing functionality via APIs.
- Resources not covered by existing Terraform providers



© 2024 by Innovation In Software Corporation



# Custom Terraform provider

Developing a Terraform provider plugin for any platform that exposes CRUD operations via a REST API is possible.





# Golang



With Golang, you have a toolbox for creating custom providers. Here are the required files:

- Main.go
- Provider.go
- Main.tf
- Terraform.rc

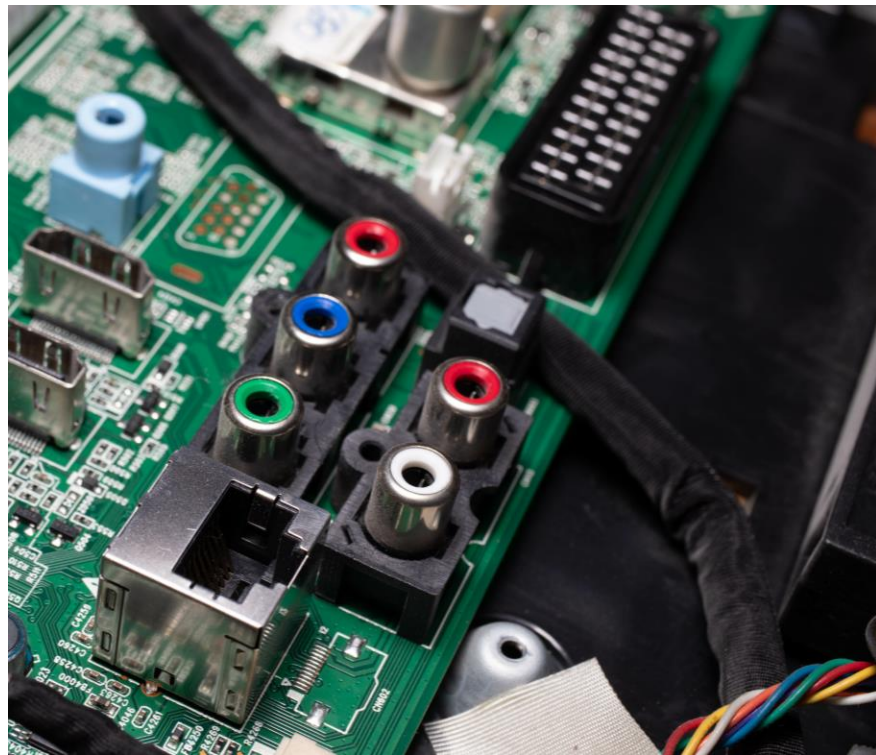


# Main.go



The main.go source file is the on / off switch that begins the conversation between the remote *service*, whatever it is, and the custom provider.

It is responsible for creating an instance of the provider and passing configuration information to it, if necessary.



The provider.go source file implements the standard protocol (like an infrared receiver in the TV) for a remote control. In a television, the infrared receiver processes signals from the remote and tells the TV what to do.

Responsibilities include:

- Decodes signals – Converts infrared pulses into electrical signals.
- Sends commands – Passes instructions to the TV's control system.
- Confirms execution – Ensures the TV responds correctly.

# Custom Terraform provider



The main.tf configuration file figuratively controls the button presses on the remote control (provider). It represents the actions the user wants to perform. Main.tf defines the desired state of infrastructure in Terraform. It specifies providers, resources, and settings that Terraform uses to create, manage, and destroy infrastructure.

# Resource files



`resource_<name>.go` – Defines Terraform resources and their CRUD operations : Create, Read, Update, and Delete. It manages infrastructure objects (e.g., databases, VMs, etc.).

`datasource_<name>.go` – Defines Terraform data sources, which retrieve existing infrastructure details without modifying them. It allows Terraform to read external resources, such as fetching customer data from a database.



# Terraform Plugin Framework



The Terraform plugin framework is an SDK that you can use to develop Terraform providers, and it is HashiCorp's recommended way to develop Terraform Plugins on protocol version 6 or protocol version 5.

The Terraform Plugin Framework simplifies the development of custom Terraform providers by providing structured APIs for managing resources, data sources, and provider configurations. It is the foundation for building providers interacting with external APIs while ensuring compatibility with Terraform Core.



# Terraform Plugin Framework



## Key Features and Benefits:

- **Simplifies Provider Development:** Offers pre-built methods for implementing CRUD operations (Create, Read, Update, Delete) on resources.
- **Better Performance & Stability:** Uses gRPC for efficient communication between Terraform Core and the provider.
- **Strong Type Safety:** Uses Go's strong typing system to reduce bugs and improve reliability.
- **State Management:** Provides tools for handling Terraform state, reducing drift and ensuring accurate updates.
- **Backward Compatibility & Maintenance:** Designed to support long-term compatibility with Terraform, making it easier to upgrade providers.

# Custom provider requirements

All providers must be named terraform-provider-\${API}. So, we must name our project appropriately.

Examples:

- terraform-provider-greeting
- terraform-provider-postgres
- terraform-provider-s3

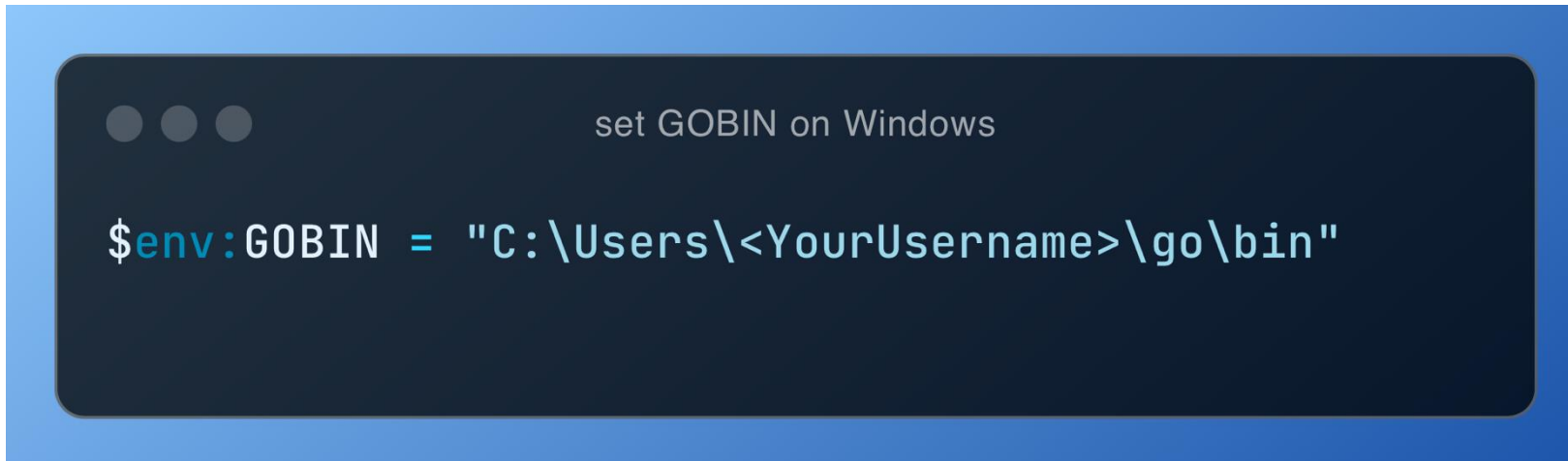


# Creating a custom provider

Terraform core, by default, downloads provider plugins from the registry. When building a new provider, we need to configure Terraform to fetch it locally. The provider plugin runs as a server process on one of the processor cores, and the executable is generated when the Golang code is built. This executable is stored in the path specified by the GOBIN environment variable.

Steps to Set the GOBIN Path:

- Use a directory like C:\Users\TEKsystems\go\bin.
- Ensure the GOBIN environment variable points to this directory.

A terminal window with a dark blue background and a light blue border. The title bar at the top says "set GOBIN on Windows". The command being entered is "\$env:GOBIN = \"C:\\Users\\<YourUsername>\\go\\bin\"".

```
set GOBIN on Windows

$env:GOBIN = "C:\\Users\\<YourUsername>\\go\\bin"
```

# Creating a custom provider

You must create a .terraformrc file in your home directory to instruct Terraform to fetch the provider plugin locally. This file overrides the default behavior of fetching plugins from the registry.

Configuration Details:

- The dev\_overrides block specifies the local path for the plugin.
- In this example, we're developing a custom plugin named registry.terraform.io/example/banking to manage data in a PostgreSQL database.

A code editor window with a dark blue background and a light blue border. The title bar at the top right says 'terraform.rc'. The code inside is as follows:

```
provider_installation {  
  
  dev_overrides {  
    "registry.terraform.io/example/banking" = "C:\Users\TEKsystems\go\bin"  
  }  
}
```

# Walkthrough: Custom providers



# Lab: Custom providers