TEKsystems Global Services

WORKFORCE DEVELOPMENT

Configuration Management and DevOps: Automate Infrastructure Deployments

PARTICIPANT GUIDE

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

Terraform Developer

Lab page

https://jruels.github.io/tf-dev

Defining Variables - Play

You can define different variables for each individual host, or set shared variables for a group of hosts in your inventory. You can also define variables in a play.

```yaml
- hosts: webservers
  vars:
    http_port: 80
```

NOTE: When you define variables in a play, they are only visible to tasks executed in that play.

Defining Variables – External File

You can define variables in reusable variables files and/or in reusable roles. When you define variables in reusable variable files, the sensitive variables are separated from playbooks.
This separation enables storing playbooks in source control and even share them without exposing passwords or other sensitive data.

Defining Variables – External File

This example shows how you can include variables defined in an external file:

```
- hosts: all
  remote_user: root
  vars:
    favcolor: blue
  vars_files:
    - /var/external_vars.yml
```

Defining Variables – External File

The contents of each variables file is a simple YAML dictionary.
For example:

```yaml
---
var1: myfavorite
var2: mysecondfavorite
```

Defining Variables – Command Line

You can define variables when you run your playbook by passing variables at the command line.

--extra-vars (-e)

```
ansible-playbook playbook.yml -extra-vars "red"
```

Defining Variables – Command Line

If you have a lot of special characters, use a JSON or YAML file containing the variable definitions.

```
ansible-playbook release.yml -extra-vars
"@some_var_file.json"
```

Variable Precedence

Ansible does apply variable precedence, and you might have a use for it. Here is the order of precedence from least to greatest (the last listed variables override all other variables):

1. Command line values (for example, -u my_user, these are not variables)
2. role defaults (defined in role/defaults/main.yml)
3. Inventory file or script group vars
4. Inventory group_vars/all
5. Playbook group_vars/all
6. inventory group_vars/*
7. playbook group_vars/*
8. inventory file or script host vars
9. inventory host_vars/*
10. playbook host_vars/*

Variable Precedence

11. host facts / cached set_facts
12. play vars
13. play vars_prompt
14. play vars_files
15. role vars (defined in role/vars/main.yml)
16. block vars (only for tasks in block)
17. task vars (only for the task)
18. include_vars
19. set_facts /registered vars
20. role (and include_role params)
21. include params
22. extra vars (for example –e "user=my_user") – always wins precedence.

Working With Variables - List

A list variable combines a variable name with multiple values. The multiple values can be stored as an itemized list or in square brackets [], separated with commas.

Defining variables as lists:

You can define variables with multiple values using YAML lists. For example:

```
Region:
    -   northeast
    -   southeast
    -   midwest
```

Working With Variables - List

Referencing list variables:

When you use variables defined as a list (also called an array), you can use individual, specific fields from that list. The first item in a list is item 0, the second item is item 1.

For example, to select the first element in the array:

```
region: "{{ region[0] }}"
```

POP QUIZ: DISCUSSION

What does this expression return?

```
region: "{{ region[0] }}"
```

POP QUIZ: DISCUSSION

What does this expression return?

```
region: "{{ region[0] }}"
```

- A: northeast
- B: southeast
- C: midwest

POP QUIZ: DISCUSSION

What does this expression return?

```
region: "{{ region[0] }}"
```

- A: northeast
- B: southeast
- C: midwest

Working With Variables - Dictionary

A dictionary stores the data in key-value pairs. Usually, dictionaries are used to store related data, such as the information contained in an ID or a user profile.

You can define more complex variables using YAML dictionaries. A YAML dictionary maps keys to values. For example:

```
region:
  field1: one
  field2: two
```

Working With Variables - Dictionary

When you use variables defined as a key:value dictionary (also called a hash), you can use individual, specific fields from that dictionary using either bracket notation or dot notation:

```
foo['field']
foo.field
```

Dot notation can cause problems because some keys collide with attributes and methods of python dictionaries.

PROTIP: Use bracket notation in most cases.

Registering Variables

You can create variables from the output of an Ansible task with the task keyword `register`. You can use registered variables in any later tasks in your play.

```
- hosts: web_servers
  tasks:
    - name: Register shell command output as variable
      shell: /usr/bin/foo
      register: foo_result
      ignore_errors: true

    - name: Run shell command using output from
previous task
      shell: /usr/bin/bar
      when: foo_result.rc == 5
```

Registering Variables

Registered variables are stored in memory. You cannot cache registered variables for use in future plays. Registered variables are only valid on the host for the rest of the current playbook run.

If a task fails or is skipped, Ansible still registers a variable with a failure or skipped status, unless the task is skipped based on tags.

Querying Nested Data

Many registered variables (and facts) are nested YAML or JSON data structures. You cannot access values from these nested data structures with the simple {{ foo }} syntax. You must use either bracket notation or dot notation. For example, to reference an IP address from your facts using the bracket notation:

```
{{ ansible_facts["eth0"]["ipv4"]["address"] }}
```

Optional Variables

By default, Ansible requires values for all variables in a templated expression. However, you can make specific variables optional.

For example, you might want to use a system default for some items and control the value for others. To make a variable optional, set the default value to the special variable omit:

Optional Variables

In this example, the default mode for the files `/tmp/foo` and `/tmp/bar` is determined by the umask of the system.

Ansible does not send a value for `mode`. Only the third file, `/tmp/baz`, receives the *mode=0444* option.

```
- name: Touch files with optional mode
  ansible.builtin.file:
    dest: "{{ item.path }}"
    state: touch
    mode: "{{ item.mode | default(omit) }}"
  loop:
    - path: /tmp/foo
    - path: /tmp/bar
    - path: /tmp/baz
      mode: "0444"
```

Handlers

Sometimes you want a task to run only when a change is made on a machine. For example, you may want to restart a service if a task updates the configuration of that service, but not if the configuration is unchanged. Ansible uses handlers to address this use case. Handlers are tasks that only run when notified.

POP QUIZ: DISCUSSION

How have you used handlers?

POP QUIZ: DISCUSSION

When should handlers be used?

- When a configuration file is updated, and the service needs to be restarted.
- When a new release is rolled out

Handlers

```
---
- name: Verify Apache installation
  hosts: web
  vars:
    http_port: 80
    max_clients: 200
  tasks::
    - name: Install Apache
    - yum:
    - name: httpd
...
    - name: Apache config
      template:
        src: httpd.j2
        dest: /etc/httpd.conf
      notify:
      - Restart Apache
  handlers:
    - name: Restart Apache
      service:
        name: httpd
        state: restarted
```

Notify Handlers

Tasks can instruct one or more handlers to execute using the `notify` keyword. The `notify` keyword can be applied to a task and accepts a list of handler names that are notified on a task change.

```
tasks:
    name: Template file
    template:
      src: template.j2
      dest: /etc/foo.conf
    notify:
    - Restart apache
    - Restart memcached
```

Notify Handlers

Tasks can instruct one or more handlers to execute using the `notify` keyword. The `notify` keyword can be applied to a task and accepts a list of handler names that are notified on a task change.

```
tasks:
    notify:
    - Restart apache
    - Restart memcached
handlers:
  - name: Restart memcached
    service:
      name: memcached
      state: restarted
  - name: Restart Apache
    service:
      name: apache
      state: restarted
```

Specifying Handlers

Handlers must be named in order for tasks to be able to notify them using the notify keyword.

Alternately, handlers can utilize the listen keyword. Using this handler keyword, handlers can listen on topics that can group multiple handlers as follows:

Specifying Handlers

```yaml
tasks::
    - name: Restart everything
      command: echo "this task will restart the web services"
      notify: "restart web services"

handlers:
  - name: Restart memcached
    service:
      name: memcached
      state: restarted
    listen: "restart web services"

  - name: Restart Apache
    service:
      name: httpd
      state: restarted
    listen: "restart web services"
```

Blocks

Example with Ansible Blocks

```
---
- name: Ansible Blocks
  hosts: server1
  gather_facts: false

  tasks:
    - block:
        - name: List usr directory
          command: "ls -l /usr/"

        - name: List root directory
          command: "ls -l /root"
      become: yes

    - name: List home directory
      command: "ls -l ~/"
```

Recovery

An additional benefit of using ansible blocks is to perform recovery operations.
If any of the tasks within a block fail, the playbook will exit.

With blocks, we can assign a `rescue` block that can contain a bunch of tasks. If any of the tasks within the block fail, the tasks from the recovery block will automatically be executed to perform clean-up activity.

Rescue Variables

Ansible provide a couple of variables for tasks in the `rescue` portion of a block:

- `ansible_failed_task`

  The task that returned 'failed' and triggered the rescue. For example, to get the name use `ansible_failed_task.name`

- `ansible_failed_result`
  The captured return result of the failed task that triggered the rescue. The same as registering the variable.

Rescue Block

Example with Rescue block

```
---
- name: Ansible Blocks
  hosts: server1
  gather_facts: false

  tasks:
    - block:
        - name: List home directory
          command: "ls -l ~/"

        - name: Failing intentionally
          command: "ls -l /tmp/not-home"
      rescue:
       - name: Rescue block (perform recovery)
         debug:
           msg: "Something broke! Cleaning up.."
```

Always Block

An `always` block will be called independent of the task execution status. It can be used to give a summary or perform additional tasks whether the block tasks fail or not.

Always Block

Example with Always block

```yaml
---
- name: Ansible Blocks
  hosts: server1
  gather_facts: false

  tasks:
    - block:
        - name: List home directory
          command: "ls -l ~/"

        - name: Failing intentionally
          command: "ls -l /tmp/not-home"
      always:
       - name: This always executes
         debug:
           msg: "Can't stop me..."
```

Block Practical Example

Now that we've discussed how a block can be used, let's look at practical examples.

- Install, configure, and start a service
- Apply logic to all tasks in the block
- Enable error handling

Block Practical Example

Practical example (Install, configure, and start Apache

```
---
tasks:
  - name: Install, configure, start Apache
    block:
      - name: Install httpd and memcached
        ansible.builtin.yum:
          name:
          - httpd
          - memcached
          state: present
      - name: Apply config template
        ansible.builtin.template:
          src: templates/src.j2
          dest: /etc/template.conf

      - name: Start/enable service
        ansible.builtin.service:
          name: httpd
          state: started
          enabled: true
    when: ansible_facts['distribution'] == 'CentOS'
    become: true
    become_user: root
```

Block Practical Example

The `when` condition evaluated for all tasks in block.

Practical example (Install, configure, and start Apache

```
---
tasks:
  - name: Install, configure, start Apache
    block:
     - name: Install httpd and memcached
       ansible.builtin.yum:
         name:
         - httpd
         - memcached
         state: present
     - name: Apply config template
       ansible.builtin.template:
         src: templates/src.j2
         dest: /etc/template.conf

     - name: Start/enable service
       ansible.builtin.service:
         name: httpd
         state: started
         enabled: true
    when: ansible_facts['distribution'] == 'CentOS'
    become: true
    become_user: root
```

Block Rescue Task Status

If an error occurs in the block and the rescue task succeeds, Ansible reverts the failed status of the original task for the run and continues to run the play as if the original task had succeeded.

The rescued task is considered successful. However, Ansible still reports a failure in the playbook statistics.

Block Handlers

You can use blocks with `flush_handlers` in a rescue task to ensure that all handlers run even if an error occurs:

```yaml
---
tasks:
  - name: Attempt graceful rollback
    block:
     - name: Print a message
       ansible.builtin.debug:
         msg: 'I execute normally'
       changed_when: yes
       notify: run me even after an error

     - name: Force a failure
       ansible.builtin.command: /bin/false
    rescue:
      - name: Make sure all handlers run
        meta: flush_handlers

handlers:
  - name: Run me even after an error
    ansible.builtin.debug:
      msg: 'This handler runs even on error'
```

Lab: Ansible error handling

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.

INFRASTRUCTURE AS CODE

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

HashiCorp
Terraform

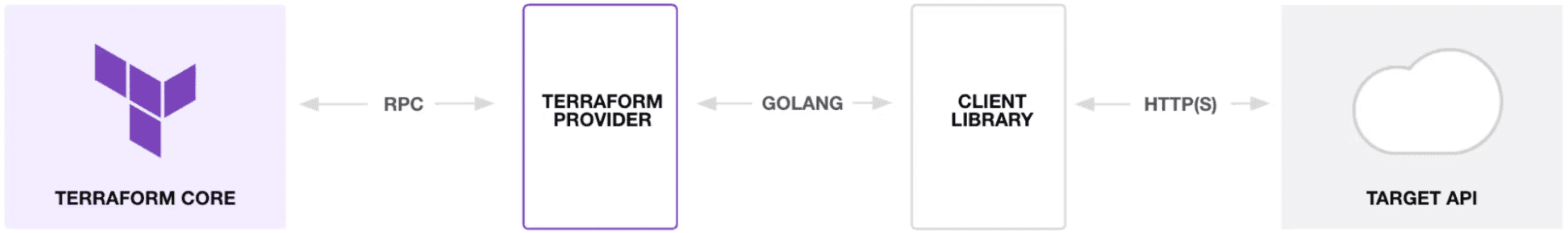
Declarative IaC also enhances consistency and scalability. Configurations can be versioned in tools like Git, enabling tracking, reviews, and reusability across environments. Automated change detection and planning provide visibility into updates, ensuring safer and more predictable infrastructure management.

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.

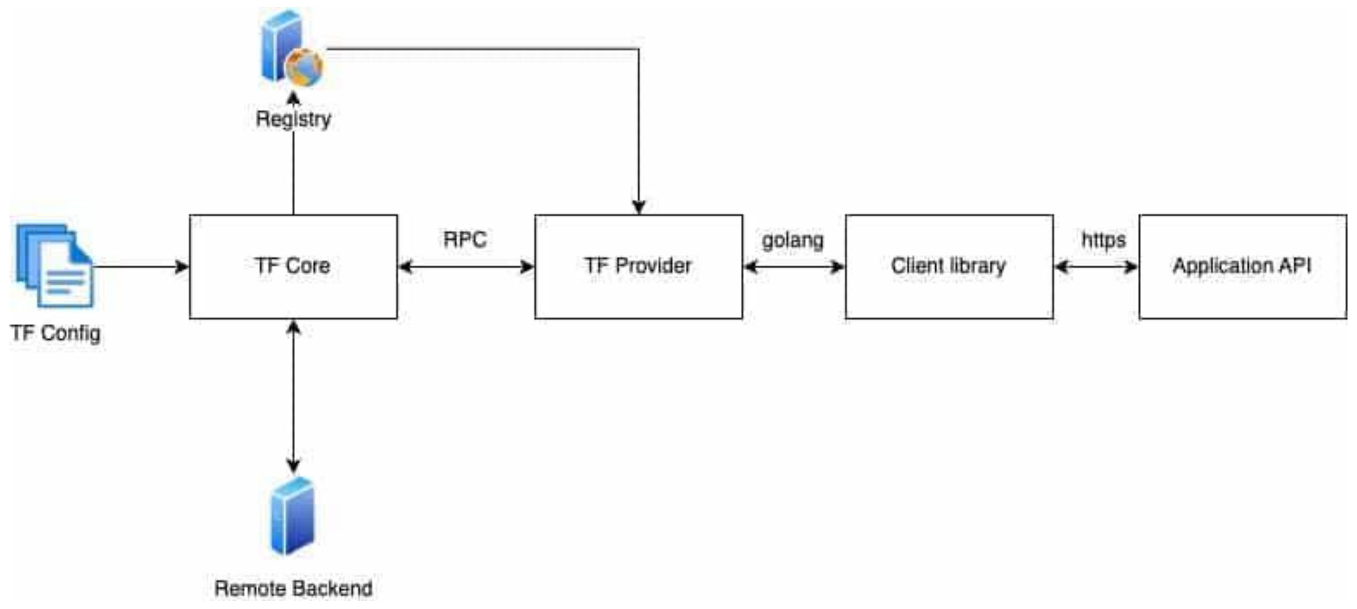


TERRAFORM PROVIDER

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.

RESOURCE MANAGEMENT

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.



POP QUIZ: DISCUSSION

Why build a custom provider?

POP QUIZ: DISCUSSION

Why build a custom provider?

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

POP QUIZ: DISCUSSION

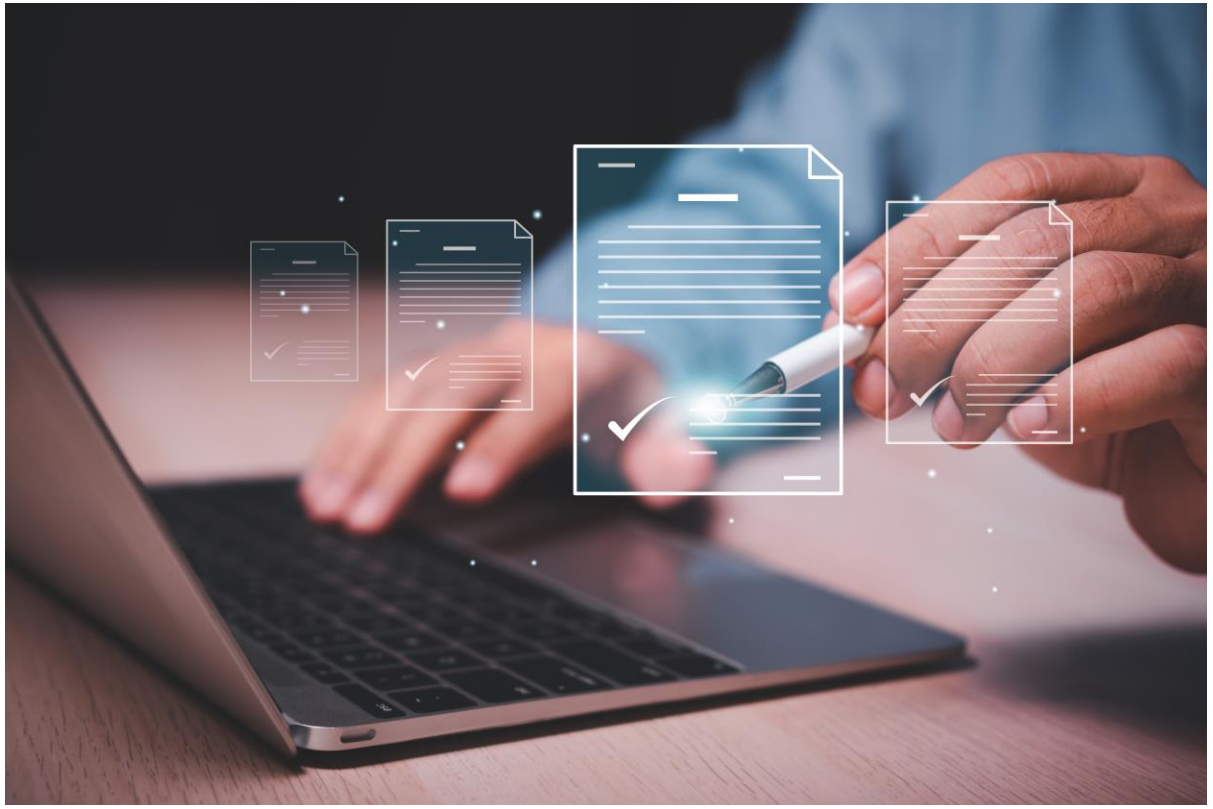
Why build a custom provider?

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

CUSTOM TERRAFORM PROVIDER

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

CUSTOM PROVIDER REQUIREMENTS



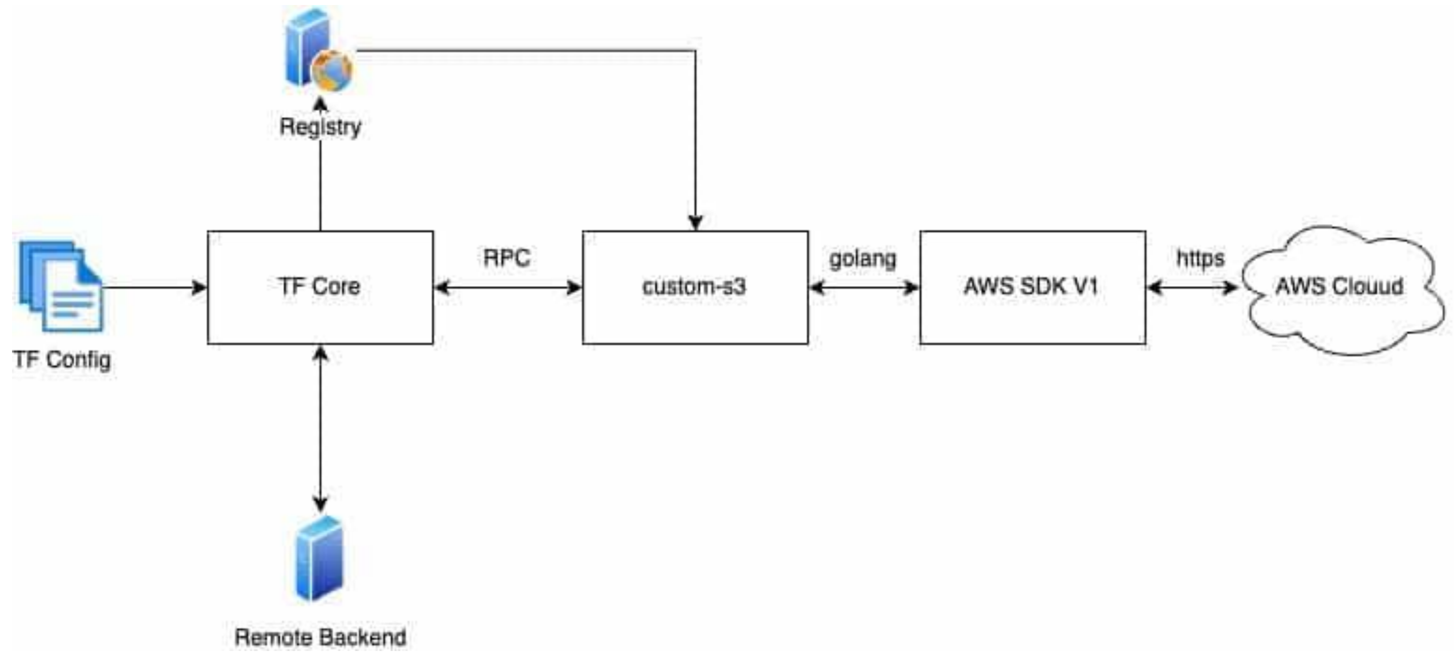
- Providers are written in Go
- An understanding of Terraform concepts
- An API service you want to manage with Terraform

CUSTOM PROVIDER

This example shows a custom provider to manage AWS S3 buckets.

RESOURCE MANAGEMENT

Terraform executes configuration files by first identifying the providers specified in the configuration and downloading their plugins from the registry during initialization (terraform init). The downloaded provider plugin runs a server process, enabling Terraform core to communicate via RPC. When commands like plan or apply are executed, Terraform core processes the configuration and state files, while the provider plugin uses Golang to make API calls, either through the client library or directly to the application API. Once the configuration is applied and resources are provisioned, the provider plugin updates the state file. Subsequent commands like apply or destroy follow the same process, with updates or deletions handled explicitly by the provider plugin.

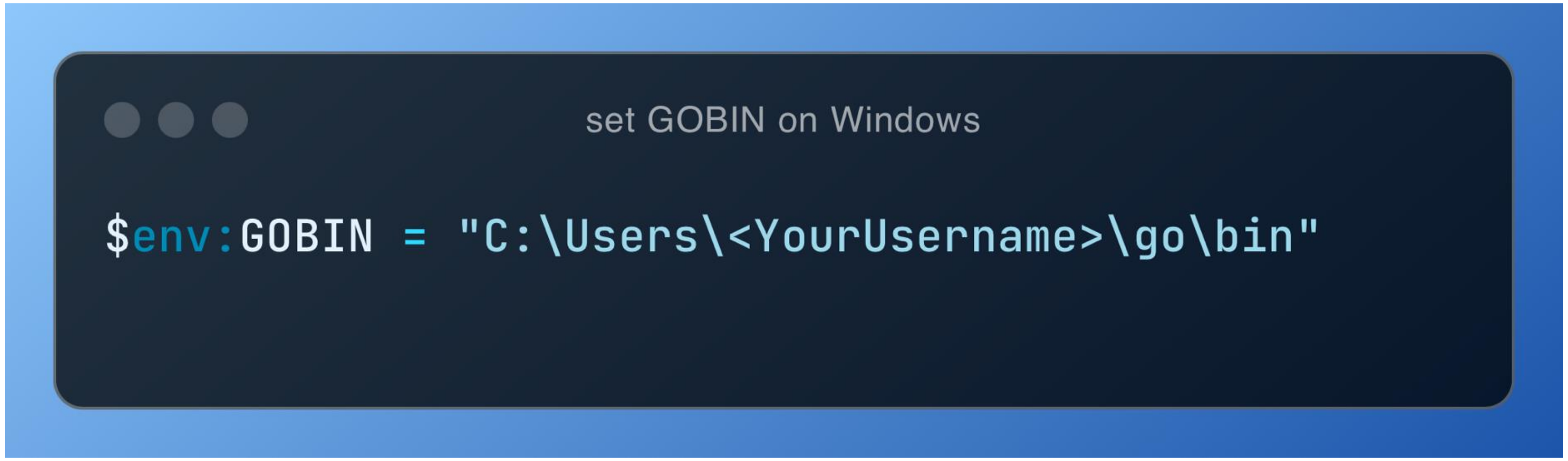


CREATING A CUSTOM PROVIDER

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 /Users/<username>/go/bin.
- Ensure the GOBIN environment variable points to this directory.



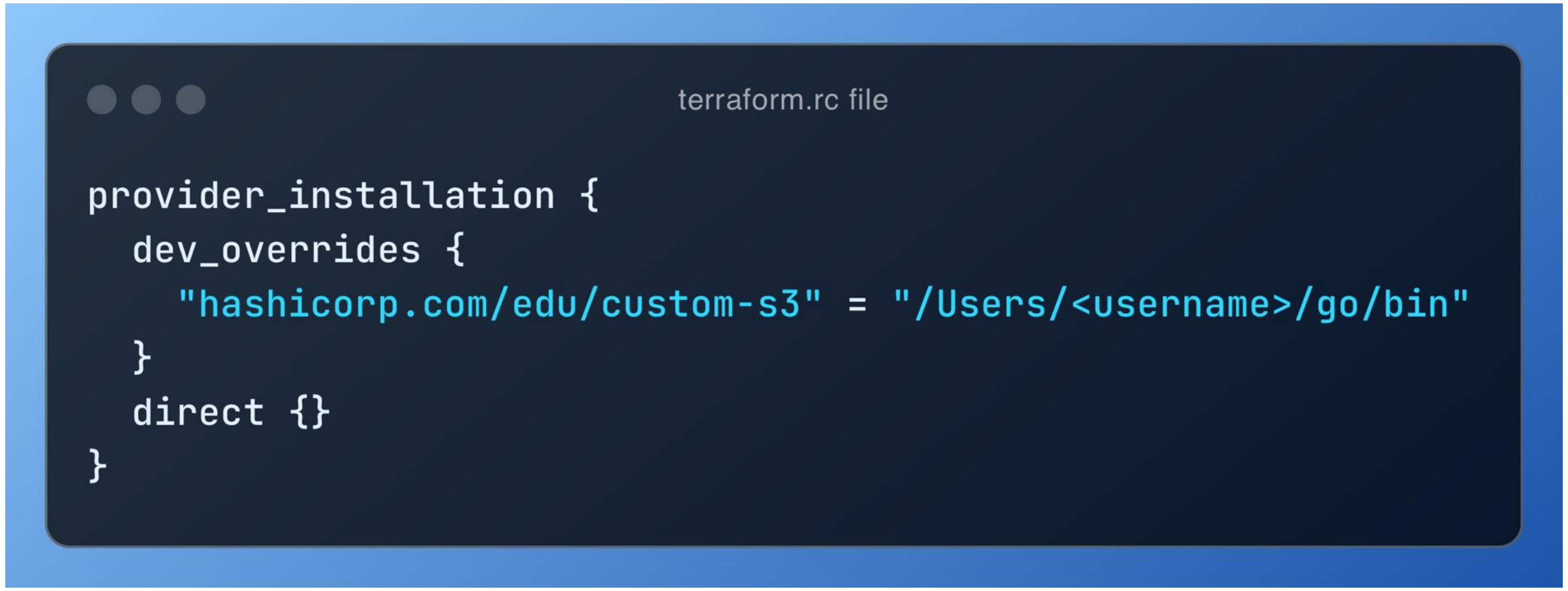
```
$env:GOBIN = "C:\Users\<YourUsername>\go\bin"
```

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.

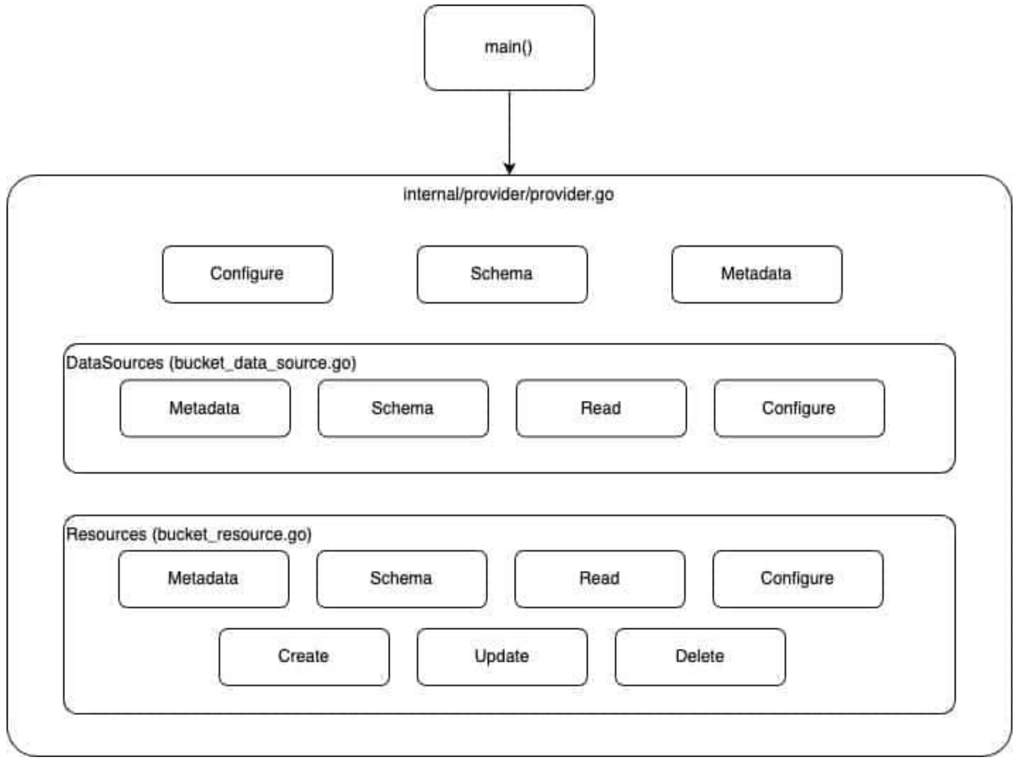
Configuration Details:

- The dev_overrides block specifies the local path for the plugin.
- In this example, we're developing a custom plugin to manage S3 buckets with the source named hashicorp.com/edu/custom-s3.


terraform.rc file

```
provider_installation {
  dev_overrides {
    "hashicorp.com/edu/custom-s3" = "/Users/<username>/go/bin"
  }
  direct {}
}
```

PLUGIN SOURCE CODE STRUCTURE



This diagram shows a high-level overview of how the code for a Terraform provider plugin is organized. Each block below represents a function/module we will implement in this post.

WORKFLOW

- Main.go starts the provider and tells Terraform, "I can help you manage stuff!"
- Provider.go makes sure Terraform has everything it needs to connect (like logging into a system).
- Resource_*name*.go is used to manage things (e.g., creating or deleting), delegating to client.go for specifics.
- Data_source_*name*.go is used to fetch information, delegating to client.go for specifics.
- Client.go communicates with the real world.

Walkthrough: Custom providers

Lab: Custom S3 provider