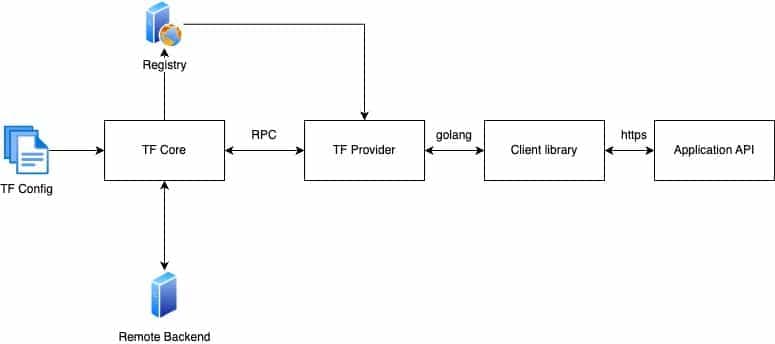
**What is a custom Terraform provider?**

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. Imagine having to work only with AWS resources, but then you get the entire registry downloaded on the host system for no reason.

This is where plugins play an important role. Like any Terraform project, it is mandatory to specify the required providers along with the version number. Based on this information, appropriate plugins are downloaded at the beginning of the first run (the initialization phase). In the subsequent execution calls, Terraform core communicates with plugins via remote procedure calls.

See this process in the diagram below:



Let’s look at the steps taken by Terraform to execute the configuration files:

1. Terraform core identifies the providers used in the configuration files. It attempts to download the corresponding provider plugins from the registry using this information.
2. The plugin downloaded on the host system runs a server process, and Terraform core communicates with the same using RPC. This is the initialization step, usually performed by the terraform init command.
3. When you run a plan/apply command, Terraform creates the state file in either the default local backend or remote backend.
4. To provision the resources using the apply command, the provider plugin consumes the configuration communicated by Terraform core and makes appropriate API calls in Golang. It either leverages the client library or SDK developed by the target platform/application or calls the platform APIs directly.
5. When the configuration is successfully executed (the resources on the target platform/application are successfully provisioned), the provider plugin updates the Terraform state file.
6. Subsequent execution commands to Terraform core – apply for update, or destroy – same steps are performed. The only difference is the update and delete functionality is explicitly implemented within the provider plugin.
7. Any command to just read the state file is executed solely by Terraform core through communicating with state files stored in the backend.

It is possible to develop a Terraform provider plugin for any platform that exposes CRUD operations via a REST API. With this understanding, let’s set up the development environment for our custom provider plugin.

**Why would you need a custom provider?**

There are several scenarios where teams may want to develop custom Terraform plugins – private cloud vendors, platform applications, or any service that exposes the functionality via API. This is evident from the fact that there are so many custom plugins published on the Terraform registry.

**What is required to develop a Terraform custom provider?**

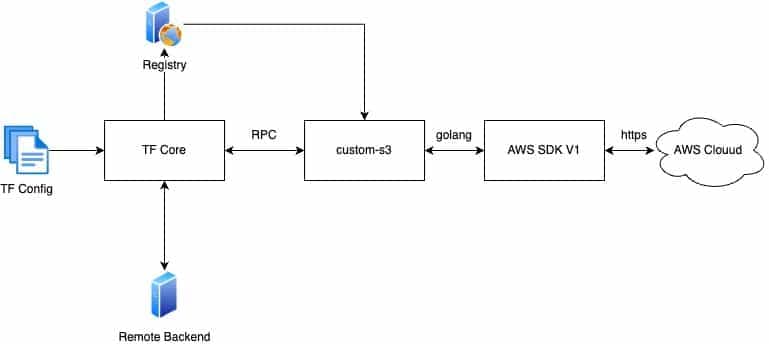
Before moving ahead, here are some prerequisites:

1. **You should know the Go programming language for Terraform provider plugins are developed in Golang.**
2. **This is a relatively advanced Terraform topic, so you should understand how Terraform works.**
3. **Your development system should have Go 1.21 installed and configured. The example in this post uses Go 1.21.**
4. **Terraform v1.15+ is installed.**

For this example, we will assume that we do not want to use the AWS provider plugin to manage S3 buckets, so we will attempt to create a custom provider to do it. We will call this provider “custom-s3”. Feel free to pick any name to follow along. Additionally, we will be using AWS SDK V1 to build this provider.

See the updated diagram below.

**Note:**The write-up below describes and mentions the main code snippets. Import statements and struct definitions are not included here to manage the length of this blog post.



**How to develop a custom provider in Terraform**

By default, Terraform core attempts to download the provider plugin from the registry. Since we are building a new provider, we want Terraform to fetch it locally.

As mentioned earlier, the provider plugin runs as a server process in one of the processor cores. When we build the Golang code, the executable is placed in the path specified in the GOBIN environment variable. Set the path “/Users/<username>/go/bin”.

Next, in your home directory, create a .terrformrc file and paste the content below.

This file overrides the setting to attempt fetching this particular plugin from the registry and instead uses the path specified here. Since we want to create a custom plugin to manage S3 buckets, we have named its source “hashicorp.com/edu/custom-s3” for development purposes.

provider\_installation {

dev\_overrides {

"hashicorp.com/edu/custom-s3" = "/Users/<username>/go/bin"

}

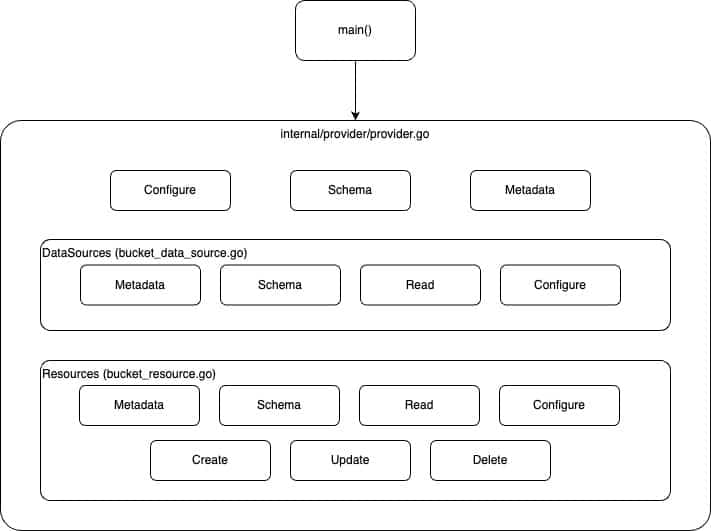
direct {}

}

**1. Plugin source code structure**

Before we start writing some Golang code, it’s important to understand the code structure for ease of development.

The diagram below 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.



**2. main()**

In your project directory, create a main.go file containing the main function. As in any Go program, the main() function serves as the entry point and initializes relevant structs, functions, and objects.

Paste the code below to this file.

package main

import (

"context"

"flag"

"log"

"terraform-provider-custom-s3/internal/provider"

"github.com/hashicorp/terraform-plugin-framework/providerserver"

)

var (

version string = "dev"

)

func main() {

var debug bool

flag.BoolVar(&debug, "debug", false, "set to true to run the provider with support for debuggers like delve")

flag.Parse()

opts := providerserver.ServeOpts{

Address: "hashicorp.com/edu/custom-s3",

Debug: debug,

}

err := providerserver.Serve(context.Background(), provider.New(version), opts)

if err != nil {

log.Fatal(err.Error())

}

}

We use terraform-provider-framework, as it provides the underlying Terraform core’s application protocol interface. We use the “providerserver” package to run our custom provider as a server thread. Note how we have set the Address of our custom provider.

**3. provider.go**

In the project root directory, create a file in the path “internal/provider/provider.go”. This file contains three main functions – Configure, Schema, and Metadata.

**Metadata function**

The Metadata function specifies the identity of the custom provider. Adhering to Terraform’s HCL naming convention, this is part of the identity of each resource defined by this provider.

In the code below, the Metadata function defines the TypeName as “customs3”. Any resource exposed by this provider will be identified as “customs3\_<resource name>” in Terraform configuration files.

func (p \*cs3Provider) Metadata(\_ context.Context, \_ provider.MetadataRequest, resp \*provider.MetadataResponse) {

resp.TypeName = "customs3"

resp.Version = p.version

}

**Schema function**

The Schema function sets the Schema used by the provider.

Since we are dealing with AWS SDK, it defines the parameters required for authenticating against the AWS cloud. Here, we define region, access\_key, and secret\_key attributes.

Similarly, we can also set session tokens not just for AWS but for any service for which we may develop custom Terraform plugins. Note that the “schema” object used here is exposed by the “schema” package of “terraform-plugin-framework”.

func (p \*cs3Provider) Schema(\_ context.Context, \_ provider.SchemaRequest, resp \*provider.SchemaResponse) {

resp.Schema = schema.Schema{

Attributes: map[string]schema.Attribute{

"region": schema.StringAttribute{

Optional: true,

},

"access\_key": schema.StringAttribute{

Optional: true,

},

"secret\_key": schema.StringAttribute{

Optional: true,

Sensitive: true,

},

},

}

}

**Configure function**

The Configure function validates the schema attributes and creates a session object using AWS SDK. The code attempts to read environment variables for the region, access key, and secret key.

After all the validations are passed, it attempts to validate these credentials to build the AWS sessions object. This object is passed as a client to provision resources (S3 buckets in our case), or read the existing S3 buckets as a data source later.

The Configure function is invoked every time any Terraform command is executed.

func (p \*cs3Provider) Configure(ctx context.Context, req provider.ConfigureRequest, resp \*provider.ConfigureResponse) {

// Retrieve provider data from configuration

var config cs3ProviderModel

diags := req.Config.Get(ctx, &config)

resp.Diagnostics.Append(diags...)

if resp.Diagnostics.HasError() {

return

}

if config.Region.IsUnknown() {

resp.Diagnostics.AddAttributeError(

path.Root("region"),

"Unknown Region",

"The provider cannot create the Custom S3 client as there is an unknown configuration value for the AWS Region. ",

)

}

if config.AccessKey.IsUnknown() {

resp.Diagnostics.AddAttributeError(

path.Root("access\_key"),

"Unknown Access Key value",

"The provider cannot create the Custom S3 client as there is an unknown configuration value for the AWS Access Key. ",

)

}

if config.SecretKey.IsUnknown() {

resp.Diagnostics.AddAttributeError(

path.Root("secret\_key"),

"Unknown Secret Key value",

"The provider cannot create the Custom S3 client as there is an unknown configuration value for the AWS Secret Key. ",

)

}

if resp.Diagnostics.HasError() {

return

}

region := os.Getenv("AWS\_REGION")

access\_key := os.Getenv("AWS\_ACCESS\_KEY\_ID")

secret\_key := os.Getenv("AWS\_SECRET\_ACCESS\_KEY")

if !config.Region.IsNull() {

region = config.Region.ValueString()

}

if !config.AccessKey.IsNull() {

access\_key = config.AccessKey.ValueString()

}

if !config.SecretKey.IsNull() {

secret\_key = config.SecretKey.ValueString()

}

if region == "" {

resp.Diagnostics.AddAttributeError(

path.Root("region"),

"Missing Region",

"The provider cannot create the AWS client as there is a missing or empty value for the Region. ",

)

}

if access\_key == "" {

resp.Diagnostics.AddAttributeError(

path.Root("access\_key"),

"Missing Access Key",

"The provider cannot create the AWS client as there is a missing or empty value for the Access Key. ",

)

}

if secret\_key == "" {

resp.Diagnostics.AddAttributeError(

path.Root("secret\_key"),

"Missing Secret Key",

"The provider cannot create the AWS client as there is a missing or empty value for the Secret Key. ",

)

}

if resp.Diagnostics.HasError() {

return

}

// Create AWS client

client, err := session.NewSession(&aws.Config{

Region: aws.String(region), // Specify the AWS region

Credentials: credentials.NewStaticCredentials(access\_key, secret\_key, ""),

})

if err != nil {

fmt.Println(err)

return

}

resp.DataSourceData = client

resp.ResourceData = client

}

**DataSources and Resources function**

The DataSources and Resources functions return the corresponding objects during initialization.

As seen in the diagram above, the DataSources (bucket\_data\_source.go) and Resources (bucket\_resource.go) have further detailed implementations.

// DataSources defines the data sources implemented in the provider.

func (p \*cs3Provider) DataSources(\_ context.Context) []func() datasource.DataSource {

return []func() datasource.DataSource{

NewBucketDataSource,

}

}

// Resources defines the resources implemented in the provider.

func (p \*cs3Provider) Resources(\_ context.Context) []func() resource.Resource {

return []func() resource.Resource{

NewOrderResource,

}

}

**4. DataSources**

Every provider mainly has two components – data sources and resources. To implement a data source for our custom provider, create a new file named “bucket\_data\_source.go”. Add the code related to the functions – Metadata, Schema, Read, and Configure as explained below.

**Metadata function**

This Metadata function sets the value for the identifier of this specific data source in your custom provider. Any value being set here will be used by the potential Terraform developers to use this in their configuration files.

Here, we have set the metadata to be “\_buckets”. Terraform developers would use “customs3\_buckets”, if they have to access bucket information using this custom provider.

func (d \*bucketDataSource) Metadata(\_ context.Context, req datasource.MetadataRequest, resp \*datasource.MetadataResponse) {

resp.TypeName = req.ProviderTypeName + "\_buckets"

}

**Schema function**

We want to create this data source to read the names, creation dates, and tags of the S3 bucket currently existing in the AWS account. The Schema function defines the schema accordingly by leveraging the schema package provided by the Terraform plugin framework package.

Note that this Schema function differs from the one defined at the provider level.

func (d \*bucketDataSource) Schema(\_ context.Context, \_ datasource.SchemaRequest, resp \*datasource.SchemaResponse) {

resp.Schema = schema.Schema{

Attributes: map[string]schema.Attribute{

"buckets": schema.ListNestedAttribute{

Computed: true,

NestedObject: schema.NestedAttributeObject{

Attributes: map[string]schema.Attribute{

"date": schema.StringAttribute{

Computed: true,

},

"name": schema.StringAttribute{

Computed: true,

},

"tags": schema.StringAttribute{

Required: true,

},

},

},

},

},

}

}

**Read function**

It’s time to read the bucket information using AWS SDK.

Add the Read function below to the file, which uses the client object created at the provider level. This client object is an authenticated session object provided by AWS SDK.

Using the client library – AWS SDK – we make a ListBuckets() function call to read information about all the buckets currently existing in our AWS account. We can append the same to the Diagnostics object for further processing.

Here, further processing simply means storing this information in memory during run-time.

func (d \*bucketDataSource) Read(ctx context.Context, req datasource.ReadRequest, resp \*datasource.ReadResponse) {

var state bucketDataSourceModel

svc := s3.New(d.client)

buckets, err := svc.ListBuckets(nil)

if err != nil {

resp.Diagnostics.AddError(

"Unable to Read Bucket data",

err.Error(),

)

return

}

for \_, bucket := range buckets.Buckets {

bucketState := bucketModel{

Date: types.StringValue(bucket.CreationDate.Format("2006-01-02 15:04:05")),

Name: types.StringValue(\*bucket.Name),

}

state.Buckets = append(state.Buckets, bucketState)

}

diags := resp.State.Set(ctx, &state)

resp.Diagnostics.Append(diags...)

if resp.Diagnostics.HasError() {

return

}

}

**Configure function**

The Diagnostics object collects information about errors and passes it to the Terraform core. Based on the feedback obtained from this Diagnostics object, the Terraform core either exits the process, highlighting the error message, or continues to function if no error is found.

The Configure function runs at the very beginning of any operation at all levels. Here, the Configure function’s role is to ensure that a properly configured client (session object) is obtained from the provider.

func (d \*bucketDataSource) Configure(\_ context.Context, req datasource.ConfigureRequest, resp \*datasource.ConfigureResponse) {

if req.ProviderData == nil {

return

}

client, ok := req.ProviderData.(\*session.Session)

if !ok {

resp.Diagnostics.AddError(

"Unexpected Data Source Configure Type",

fmt.Sprintf("Expected \*session.Session, got: %T. Please report this issue to the provider developers.", req.ProviderData),

)

return

}

d.client = client

}

**5. Testing the data source**

Before we can test this data source, we need to compile this Go source code. Open the terminal, and run go install . in the root directory. This command builds the source code and places the executable at “/Users/<username>/go/bin: path.

As mentioned previously, any Terraform command execution will refer to this location if this custom provider is being used in that Terraform project.

So far, we implemented a basic data source for our custom provider plugin – customs3\_buckets.

Before we proceed to create the resource, let’s test if the data source works correctly. Create a Terraform project in a separate directory or subdirectory, add a main.tf file and configure this data source as shown below.

terraform {

required\_providers {

customs3 = {

source = "hashicorp.com/edu/custom-s3"

}

}

}

provider "customs3" {

region = "eu-central-1"

access\_key = "AKIAXX2MLEKEWKT24FFC"

secret\_key = "qA8E3wg15KhIVeq0AENoGPjIJA8JzApLC9CX6tK2"

}

data "customs3\_buckets" "example" {}

output "all\_buckets" {

value = data.customs3\_buckets.example

}

We have mentioned our custom provider in the required\_providers block. Since we set this provider’s name as “customs3”, we have to use the same here. We also added the provider block with all the relevant details.

We have used the “customs3\_buckets” data source, which we developed in the previous section.

We use the output variable to print all the data that is read to the terminal.

Run terraform init and observe the output.

terraform init

Initializing the backend...

Initializing provider plugins...

- Finding latest version of hashicorp.com/edu/custom-s3...

╷

│ Warning: Provider development overrides are in effect

│

│ The following provider development overrides are set in the CLI configuration:

│ - hashicorp.com/edu/custom-s3 in /Users/ldt/go/bin

│

│ Skip terraform init when using provider development overrides. It is not necessary and may error unexpectedly.

╵

╷

│ Error: Invalid provider registry host

│

│ The host "hashicorp.com" given in provider source address "hashicorp.com/edu/custom-s3" does not offer a

│ Terraform provider registry.

First, the output throws an important warning message.

It highlights, that we have set the development overrides. Further, it also indicates that “hashicorp.com” is not a provider registry – which is indeed true. While developing the plugin locally, we need to run the init command.

Run terraform plan or apply commands, and observe the output. This should print the list of buckets, with their names, creation date, and tags (if any), as shown below.

...

+ {

+ date = "2022-10-17 13:05:06"

+ name = "tfworkspaces-state"

+ tags = null

},

+ {

+ date = "2022-04-24 09:24:58"

+ name = "www.sumeet.life"

+ tags = null

},

+ {

+ date = "2022-06-03 17:30:17"

+ name = "you.sumeet.life"

+ tags = null

},

]

}

You can apply this plan to save these new output values to the Terraform state, without changing any real

infrastructure.

────────────────────────────────────────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these

actions if you run "terraform apply" now.

If you get a similar output, then you have been able to create a custom provider along with one data source.

**6. Resources**

Programming the data source for our custom provider was easy, as it simply read the existing bucket information from the AWS account.

However, while configuring resources, we have to take the responsibility to perform CRUD operations. Since these operations are interdependent, we have to make sure that the schema is maintained across the implementation.

**Metadata function**

Like all other Metadata functions, this defines how you would want the Terraform developers to represent the bucket configuration block in their projects.

The code below defines the Metadata for S3 bucket resources as “\_s3\_bucket”. This results in leveraging “customs3\_s3\_bucket” as the resource name.

func (r \*orderResource) Metadata(\_ context.Context, req resource.MetadataRequest, resp \*resource.MetadataResponse) {

resp.TypeName = req.ProviderTypeName + "\_s3\_bucket"

}

**Schema function**

Similar to previous experiences, the Schema function here defines the acceptable attributes for this resource.

This time, we added more attributes, such as ID and last updated. These attributes are purely for demonstration purposes and do not serve any real value.

The rest of the schema remains the same as the data source. This schema will also be reflected in the state files when they are created.

func (r \*orderResource) Schema(\_ context.Context, \_ resource.SchemaRequest, resp \*resource.SchemaResponse) {

resp.Schema = schema.Schema{

Attributes: map[string]schema.Attribute{

"id": schema.StringAttribute{

Computed: true,

},

"last\_updated": schema.StringAttribute{

Computed: true,

},

"buckets": schema.ListNestedAttribute{

Required: true,

NestedObject: schema.NestedAttributeObject{

Attributes: map[string]schema.Attribute{

"date": schema.StringAttribute{

Computed: true,

},

"name": schema.StringAttribute{

Required: true,

},

"tags": schema.StringAttribute{

Required: true,

},

},

},

},

},

}

}

**Create function**

The Create function here begins with initializing the Diagnostics object.

As discussed before, this object is responsible for making sure the configurations are correct throughout and align with the plan output. The plan object is important here since any CRUD operation loops through the JSON stored by the plan. Each iteration in the loop contains the object with attributes as defined in the schema previously.

These iterations use the client object obtained from the Configure function at the provider level to perform AWS operations like CreateBucketInput(), CreateBucket(), and PutBucketTagging().

Each iteration also updates the plan object with appropriate values, following the success of SDK operations.

Lastly, it appends the plan object to the Diagnostics object for later verification.

func (r \*orderResource) Create(ctx context.Context, req resource.CreateRequest, resp \*resource.CreateResponse) {

var plan orderResourceModel

diags := req.Plan.Get(ctx, &plan)

resp.Diagnostics.Append(diags...)

if resp.Diagnostics.HasError() {

return

}

plan.ID = types.StringValue(strconv.Itoa(1))

for index, item := range plan.Buckets {

// Create an S3 service client

svc := s3.New(r.client)

awsStringBucket := strings.Replace(item.Name.String(), "\"", "", -1)

// Create input parameters for the CreateBucket operation

input := &s3.CreateBucketInput{

Bucket: aws.String(awsStringBucket),

}

// Execute the CreateBucket operation

\_, err := svc.CreateBucket(input)

if err != nil {

resp.Diagnostics.AddError(

"Error creating order",

"Could not create order, unexpected error: "+err.Error(),

)

return

}

// Add tags

var tags []\*s3.Tag

tagValue := strings.Replace(item.Tags.String(), "\"", "", -1)

tags = append(tags, &s3.Tag{

Key: aws.String("tfkey"),

Value: aws.String(tagValue),

})

\_, err = svc.PutBucketTagging(&s3.PutBucketTaggingInput{

Bucket: aws.String(awsStringBucket),

Tagging: &s3.Tagging{

TagSet: tags,

},

})

if err != nil {

fmt.Println("Error adding tags to the bucket:", err)

return

}

fmt.Printf("Bucket %s created successfully\n", item.Name)

plan.Buckets[index] = orderItemModel{

Name: types.StringValue(awsStringBucket),

Date: types.StringValue(time.Now().Format(time.RFC850)),

Tags: types.StringValue(tagValue),

}

}

plan.LastUpdated = types.StringValue(time.Now().Format(time.RFC850))

diags = resp.State.Set(ctx, plan)

resp.Diagnostics.Append(diags...)

if resp.Diagnostics.HasError() {

return

}

}

**Read function**

The Read function implemented here serves a different purpose than the one implemented in the DataSources. Here, the Read function is used by the Create, Delete, and Update functions to compare real-world provisioned components.

It is this Read function’s output based on which Terraform does or does not execute apply or destroy commands.

If the real-world provisioned resources match the supplied configuration, then no execution happens. As mentioned earlier, this Read function also depends on the values stored in the state file. It iterates through it to perform its duties.

func (r \*orderResource) Read(ctx context.Context, req resource.ReadRequest, resp \*resource.ReadResponse) {

// Get current state

var state orderResourceModel

diags := req.State.Get(ctx, &state)

resp.Diagnostics.Append(diags...)

if resp.Diagnostics.HasError() {

return

}

for \_, item := range state.Buckets {

awsStringBucket := strings.Replace(item.Name.String(), "\"", "", -1)

svc := s3.New(r.client)

params := &s3.HeadBucketInput{

Bucket: aws.String(awsStringBucket),

}

\_, err := svc.HeadBucket(params)

if err != nil {

fmt.Println("Error getting bucket information:", err)

os.Exit(1)

}

}

// Set refreshed state

diags = resp.State.Set(ctx, &state)

resp.Diagnostics.Append(diags...)

if resp.Diagnostics.HasError() {

return

}

}

**Update function**

When executing the terraform apply command, and the Read function above identifies the changes made in the Terraform configuration files. It triggers the Update function to align the real-world provisioning with the desired and supplied configuration.

In this example, since it is not possible to change bucket names, the update function only works for changing the tag values. The same will happen in the final testing section of this blog post.

func (r \*orderResource) Update(ctx context.Context, req resource.UpdateRequest, resp \*resource.UpdateResponse) {

// Retrieve values from plan

var plan orderResourceModel

diags := req.Plan.Get(ctx, &plan)

resp.Diagnostics.Append(diags...)

if resp.Diagnostics.HasError() {

return

}

plan.ID = types.StringValue(strconv.Itoa(1))

for index, item := range plan.Buckets {

// Create an S3 service client

svc := s3.New(r.client)

awsStringBucket := strings.Replace(item.Name.String(), "\"", "", -1)

// Add tags

var tags []\*s3.Tag

tagValue := strings.Replace(item.Tags.String(), "\"", "", -1)

tags = append(tags, &s3.Tag{

Key: aws.String("tfkey"),

Value: aws.String(tagValue),

})

\_, err := svc.PutBucketTagging(&s3.PutBucketTaggingInput{

Bucket: aws.String(awsStringBucket),

Tagging: &s3.Tagging{

TagSet: tags,

},

})

if err != nil {

fmt.Println("Error adding tags to the bucket:", err)

return

}

plan.Buckets[index] = orderItemModel{

Name: types.StringValue(strings.Replace(awsStringBucket, "\"", "", -1)),

Date: types.StringValue(time.Now().Format(time.RFC850)),

Tags: types.StringValue(strings.Replace(tagValue, "\"", "", -1)),

}

}

plan.LastUpdated = types.StringValue(time.Now().Format(time.RFC850))

diags = resp.State.Set(ctx, plan)

resp.Diagnostics.Append(diags...)

if resp.Diagnostics.HasError() {

return

}

}

**Delete function**

The Delete function serves the terraform destroy command. It simply triggers deprovisioning of all the resources as saved in the state file. Like all other operations, we have also implemented this using AWS SDK.

func (r \*orderResource) Delete(ctx context.Context, req resource.DeleteRequest, resp \*resource.DeleteResponse) {

// Retrieve values from state

var state orderResourceModel

diags := req.State.Get(ctx, &state)

resp.Diagnostics.Append(diags...)

if resp.Diagnostics.HasError() {

return

}

for \_, item := range state.Buckets {

svc := s3.New(r.client)

input := &s3.DeleteBucketInput{

Bucket: aws.String(strings.Replace(item.Name.String(), "\"", "", -1)),

}

\_, err := svc.DeleteBucket(input)

if err != nil {

log.Fatalf("failed to delete bucket, %v", err)

}

}

}

**Configure function**

The Configure function performs the same duties as it does for DataSource described above.

func (r \*orderResource) Configure(\_ context.Context, req resource.ConfigureRequest, resp \*resource.ConfigureResponse) {

if req.ProviderData == nil {

return

}

client, ok := req.ProviderData.(\*session.Session)

if !ok {

resp.Diagnostics.AddError(

"Unexpected Data Source Configure Type",

fmt.Sprintf("Expected \*session.Session, got: %T. Please report this issue to the provider developers.", req.ProviderData),

)

return

}

r.client = client

}

**How to use this Terraform custom provider**

Create a separate Terraform project directory and to the main.tf file add the configuration below.

Adhering to the Schema defined in our resource implementation of customs3\_s3\_bucket, we have provided a bucket name and tags in the required format.

terraform {

required\_providers {

customs3 = {

source = "hashicorp.com/edu/custom-s3"

}

}

}

provider "customs3" {

region = "<REGION>"

access\_key = "<ACCESS KEY>"

secret\_key = "<SECRET KEY>"

}

resource "customs3\_s3\_bucket" "example" {

buckets = [{

name = "test-bucket-2398756"

tags = "mybucket"

}]

}

Run the terraform plan command and observe the output. Make sure that no errors are thrown, as shown below.

terraform plan

╷

│ Warning: Provider development overrides are in effect

│

│ The following provider development overrides are set in the CLI configuration:

│ - hashicorp.com/edu/hashicups-pf in /Users/ldt/go/bin

│ - hashicorp.com/edu/hashicups in /Users/ldt/go/bin

│ - hashicorp.com/edu/custom-s3 in /Users/ldt/go/bin

│

│ The behavior may therefore not match any released version of the provider and applying changes may cause the

│ state to become incompatible with published releases.

╵

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated

with the following symbols:

+ create

Terraform will perform the following actions:

# customs3\_s3\_bucket.example will be created

+ resource "customs3\_s3\_bucket" "example" {

+ buckets = [

+ {

+ date = (known after apply)

+ name = "test-bucket-2398756"

+ tags = "mybucket"

},

]

+ id = (known after apply)

+ last\_updated = (known after apply)

}

Plan: 1 to add, 0 to change, 0 to destroy.

────────────────────────────────────────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these

actions if you run "terraform apply" now.

If everything goes fine, we should correctly get a message saying “1 resource needs to be added”. Go ahead and apply this Terraform configuration.

terraform apply

╷

│ Warning: Provider development overrides are in effect

│

│ The following provider development overrides are set in the CLI configuration:

│ - hashicorp.com/edu/hashicups-pf in /Users/ldt/go/bin

│ - hashicorp.com/edu/hashicups in /Users/ldt/go/bin

│ - hashicorp.com/edu/custom-s3 in /Users/ldt/go/bin

│

│ The behavior may therefore not match any released version of the provider and applying changes may cause the

│ state to become incompatible with published releases.

╵

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated

with the following symbols:

+ create

Terraform will perform the following actions:

# customs3\_s3\_bucket.example will be created

+ resource "customs3\_s3\_bucket" "example" {

+ buckets = [

+ {

+ date = (known after apply)

+ name = "test-bucket-2398756"

+ tags = "mybucket"

},

]

+ id = (known after apply)

+ last\_updated = (known after apply)

}

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?

Terraform will perform the actions described above.

Only 'yes' will be accepted to approve.

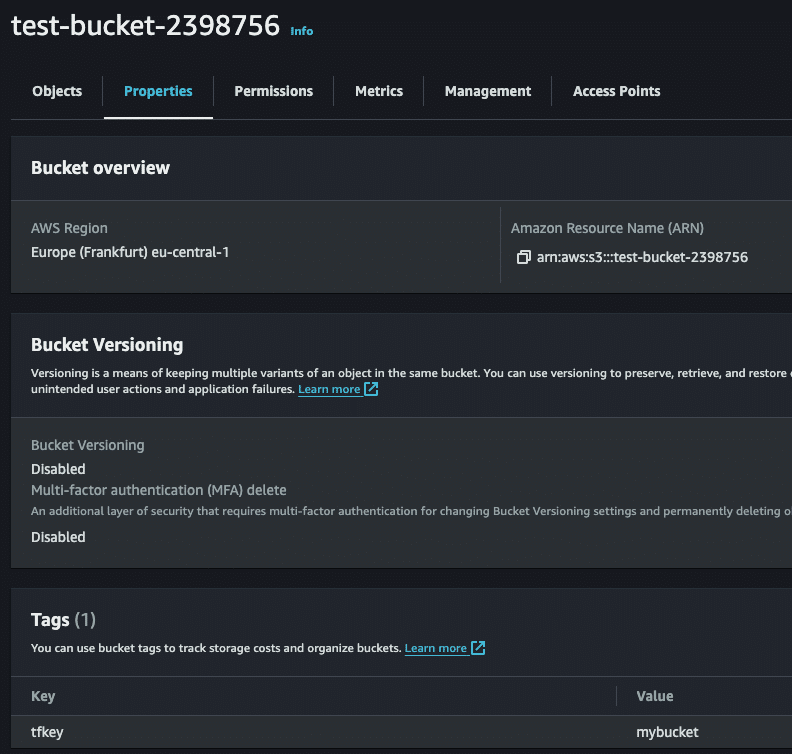
Enter a value: yes

customs3\_s3\_bucket.example: Creating...

customs3\_s3\_bucket.example: Creation complete after 2s [id=1]

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

The apply output above confirms that one S3 bucket with our desired configuration has been created. Verify it by logging into the AWS console.



Let’s try to utilize the Update function from Resource implementation. We cannot change the name of the bucket, but we can change the value of the tag.

Change the tag value as per your wish, and apply the Terraform configuration again.

**Note:** The key assigned to the tag value is hardcoded in the code.

The terminal output after running the apply command, after the change in tag value:

terraform apply

╷

│ Warning: Provider development overrides are in effect

│

│ The following provider development overrides are set in the CLI configuration:

│ - hashicorp.com/edu/hashicups-pf in /Users/ldt/go/bin

│ - hashicorp.com/edu/hashicups in /Users/ldt/go/bin

│ - hashicorp.com/edu/custom-s3 in /Users/ldt/go/bin

│

│ The behavior may therefore not match any released version of the provider and applying changes may cause the

│ state to become incompatible with published releases.

╵

customs3\_s3\_bucket.example: Refreshing state... [id=1]

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated

with the following symbols:

~ update in-place

Terraform will perform the following actions:

# customs3\_s3\_bucket.example will be updated in-place

~ resource "customs3\_s3\_bucket" "example" {

~ buckets = [

~ {

~ date = "Thursday, 25-Jan-24 23:36:08 CET" -> (known after apply)

name = "test-bucket-2398756"

~ tags = "mybucket" -> "yourbucket"

},

]

~ id = "1" -> (known after apply)

~ last\_updated = "Thursday, 25-Jan-24 23:36:08 CET" -> (known after apply)

}

Plan: 0 to add, 1 to change, 0 to destroy.

Do you want to perform these actions?

Terraform will perform the actions described above.

Only 'yes' will be accepted to approve.

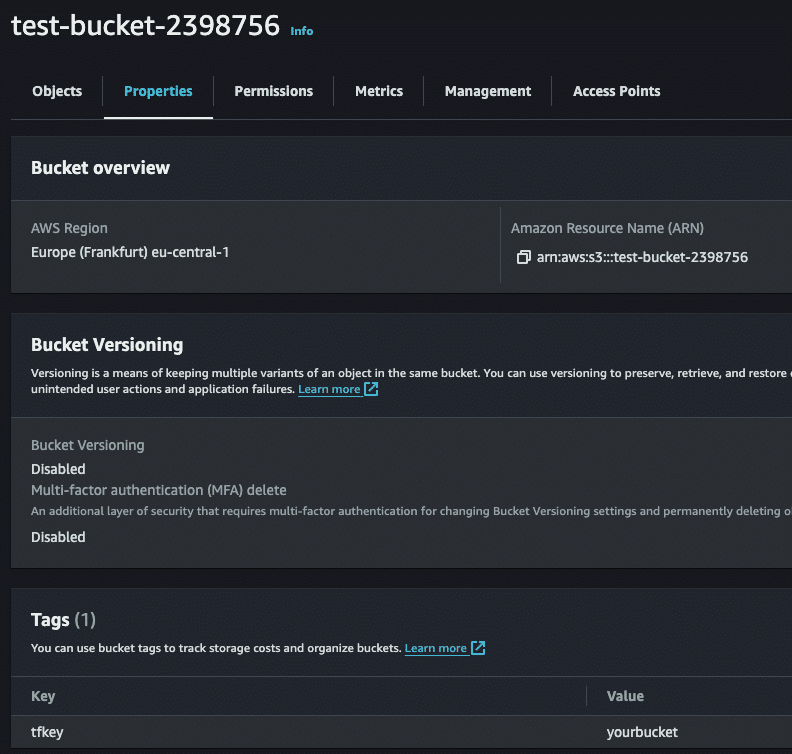
Enter a value: yes

customs3\_s3\_bucket.example: Modifying... [id=1]

customs3\_s3\_bucket.example: Modifications complete after 0s [id=1]

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

Let’s verify it.



Before we run the destroy command, take a moment to look at the state file, the schema, and other attributes.

{

"version": 4,

"terraform\_version": "1.6.6",

"serial": 23,

"lineage": "7507e650-e757-9858-d037-b3bb3ae18107",

"outputs": {},

"resources": [

{

"mode": "managed",

"type": "customs3\_s3\_bucket",

"name": "example",

"provider": "provider[\"hashicorp.com/edu/custom-s3\"]",

"instances": [

{

"schema\_version": 0,

"attributes": {

"buckets": [

{

"date": "Thursday, 25-Jan-24 23:40:32 CET",

"name": "test-bucket-2398756",

"tags": "yourbucket"

}

],

"id": "1",

"last\_updated": "Thursday, 25-Jan-24 23:40:32 CET"

},

"sensitive\_attributes": []

}

]

}

],

"check\_results": null

}

Finally, destroy this bucket by running the terraform destroy command. This should use the Delete function we implemented in Resources.

terraform destroy

╷

│ Warning: Provider development overrides are in effect

│

│ The following provider development overrides are set in the CLI configuration:

│ - hashicorp.com/edu/hashicups-pf in /Users/ldt/go/bin

│ - hashicorp.com/edu/hashicups in /Users/ldt/go/bin

│ - hashicorp.com/edu/custom-s3 in /Users/ldt/go/bin

│

│ The behavior may therefore not match any released version of the provider and applying changes may cause the

│ state to become incompatible with published releases.

╵

customs3\_s3\_bucket.example: Refreshing state... [id=1]

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated

with the following symbols:

- destroy

Terraform will perform the following actions:

# customs3\_s3\_bucket.example will be destroyed

- resource "customs3\_s3\_bucket" "example" {

- buckets = [

- {

- date = "Thursday, 25-Jan-24 23:40:32 CET" -> null

- name = "test-bucket-2398756" -> null

- tags = "yourbucket" -> null

},

] -> null

- id = "1" -> null

- last\_updated = "Thursday, 25-Jan-24 23:40:32 CET" -> null

}

Plan: 0 to add, 0 to change, 1 to destroy.

Do you really want to destroy all resources?

Terraform will destroy all your managed infrastructure, as shown above.

There is no undo. Only 'yes' will be accepted to confirm.

Enter a value: yes

customs3\_s3\_bucket.example: Destroying... [id=1]

customs3\_s3\_bucket.example: Destruction complete after 1s

Destroy complete! Resources: 1 destroyed.