**Step 2:** Create a folder to store your Sentinel policies.

**Step 3:** Create a folder per policy for development purposes.

**Step 4:** In your *policy specific folder*from step 3 create a file named *<policy\_name>.sentinel*

**Step 5:** Create the following path that will serve your *specific policy*.

Your folder structure should look like this:

*/POLICIES\_FOLDER/policy\_specific\_folder/test/<policy\_name\_folder>*

**Step 6:** Within the *<policy\_name\_folder>* create two files: *pass.json* and *fail.json*

These files should look similar to this.

**pass.json**

{

"modules": {

"tfplan-functions": {

"path": "../../../tfplan-functions.sentinel"

}

},

"mock": {

"tfplan/v2": "../../mock-tfplan-pass-v2.sentinel"

},

"test": {

"main": true

}

}

**fail.json**:

{

"modules": {

"tfplan-functions": {

"path": "../../../tfplan-functions.sentinel"

}

},

"mock": {

"tfplan/v2": "../../mock-tfplan-fail-v2.sentinel"

},

"test": {

"main": false

}

}

The explanation for the content of these files is the following. It consists of three parts:

* **modules:** (available since [Sentinel v0.15.0](https://discuss.hashicorp.com/t/sentinel-v0-15-0-introducing-modules/6579)) libraries can be imported and they are required for testing when using the third generation of Sentinel policies. ( I will come back to this later)
* **mock:** which contain a test scenario (pass/fail) which Sentinel will use to test against.
* **test:** the expected result of the main rule from your Sentinel rule which can be true or false whether its a pass/fail scenario.

**Creating Mocks**

**Step 1:** Create an account at **[app.terraform.io](https://app.terraform.io/)**, an organization, and a workspace.

**Step 2:** Back in your computer, create a file named *backend.tf*in the */POLICIES\_FOLDER*from the previous section.

Its content should look like this:

terraform {

backend "remote" {

hostname = "app.terraform.io"

organization = "*<some-organization-name>*"

workspaces {

name = "*<some-workspace>*"

}

}

}

Replace the placeholders for organization and name with your appropriate values. Replace hostname if you’re using PTFE. The *hostname* line can be removed if using

app.terraform.io

**Step 3:** Assuming you’re using the Terraform website, use the following command to associate your Terraform organization and workspace with your local environment:

terraform login

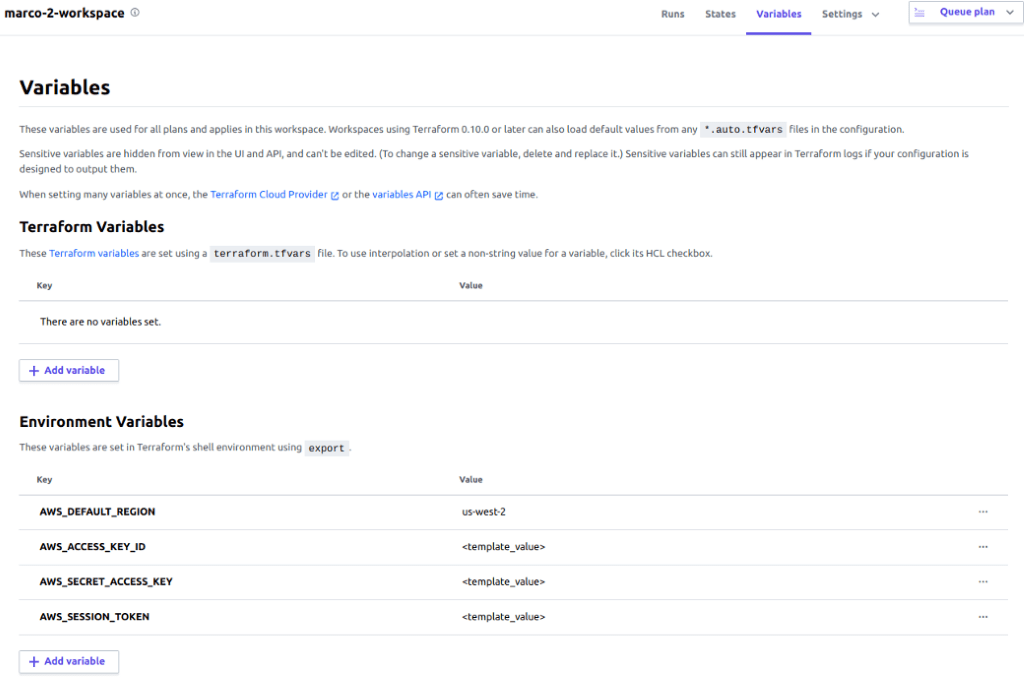
For PTFE use the following command, but verify the hostname in back-end configuration file:

AWSterraform login *<hostname>*

**Step 4 and tip:** Develop the Terraform code under */POLICIES\_FOLDER/policy\_specific\_folder* and after you are done copy the file (let’s assume its named *main.tf*) to your */POLICIES\_FOLDER.(Repeat this process when developing other policies.)*

***Note:****The Terraform code doesn’t have to be perfect, it can contain dummy values as place holders, just make sure to use the appropriate values for the items or sections you want to validate. Another tip, copy HashiCorp examples from the registry, notice some examples may have errors, but the CLI should help you identify. S3 bucket example from AWS*[*here*](https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/s3_bucket)*.*

**Step 5:** In the **Variables** tab of your workspace at **[app.terraform.io](https://app.terraform.io/),**fill out the Terraform Variables. Here is my example for aws.



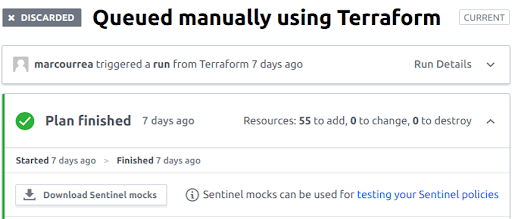
***Note:****AWS\_SESSION\_ TOKEN is not always needed, depends on your AWS organization settings.*

**Step 6:** Now that you have some Terraform code written, its time to plan or *(pretend to)*apply it.

terraform apply

Wait until the CLI requests validation and answer with anything different than **yes**assuming you’re using dummy values.

Then mocks should be available for download like in the image below by hitting the **Download Sentinel mocks** button



Place the extracted files from the downloaded zip in the following path:

*/POLICIES\_FOLDER/policy\_specific\_folder/*

Then create a *pass* scenario assuming your code is correct as follows:

cp mock-tfplan-v2.sentinel mock-tfplan-pass-v2.sentinel

Do the same thing for the *fail* scenario:

cp mock-tfplan-v2.sentinel mock-tfplan-fail-v2.sentinel

Modify the fail scenario attributes as required to make it trigger an unexpected behavior with your resource, like modifying an attribute with a wrong value or deleting an attribute that you’re expecting.

The creation of the pass and fail scenarios are based on HashiCorp’s documentation taken from [here](https://learn.hashicorp.com/tutorials/terraform/sentinel-testing).

You could modify the **acl**, assuming you are expecting “public-read” you could make it fail if you find a value different that is not “*public-read*”. Below its an excerpt of a mock, that would fail; assuming you’re validating the *acl* attribute expects “public-read”

resource\_changes = {

"aws\_s3\_bucket.demo": {

"address": "aws\_s3\_bucket.demo",

"change": {

"actions": [

"create",

],

"after": {

"acl": "not-public-read",

**Testing your mocks**

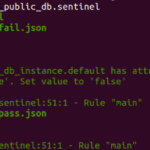
Position yourself in the following path:

*/POLICIES\_FOLDER/policy\_specific\_folder/*

There are two ways to test your mocks with the Sentinel CLI; one is with the apply option and the other one is with the test option. I will focus on the test option since with Sentinel V3 it’s easier just to execute that one. To print any warning messages use the option -verbose besides test, which looks like the following example:

sentinel test *-verbose* <policy\_name.sentinel>

Here is an output example of a policy I did to validate databases with publicly\_accessible property were not public:



Going back to the pass and fail JSON files, in the section for modules you have to import the framework functions (Sentinel V3). What I did is that I downloaded the file for **tfplan-functions.sentinel**and I added the reference within the modules section

"modules": {

   "tfplan-functions": {

     "path": "../../../tfplan-functions.sentinel"

    }

**Differences between Sentinel V2 and V3**

I noticed examples for Sentinel V2 found in HashiCorp’s documentation and Katacoda (the latter were retired) are rule-based.

*Rules must be avoided at all cost.*

The problem with rules is that you cannot store variables and it makes Sentinel difficult to work with as a regular programming language, functions are always preferred because they allow you to do things easier and you can store information in variables for later use.

Functions are always preferred.

The advantage of Sentinel V3 is that, since you can import modules (functions), you can speed up development.

The only rule that has to exist is the main rule.

# Main rule

main = rule {

    (message\_counter\_validator) else false

}

**Deciding whether a policy passes or fails**

This should be based on counting the error outputs. What I suggest is creating a list that stores the error messages and at the end of your policy, you count the number of error messages. *If the number of error messages is different than zero, the rule should be considered as failed*, that way you only prevent the creation of what is not allowed.

I must clarify that if you’re using the Sentinel V3 common functions, there is a built-in mechanism that simplifies the behavior mentioned above; I don’t completely understand all the functions yet, so I still have to demystify some things.

**Importing functions from Sentinel V3**

# Import common-functions/tfplan-functions/tfplan-functions.sentinel

# with alias “plan”

import “tfplan-functions” as plan

**Tips and Tricks**

**Finding resources**

Import the module tfplan-functions:

# Import common-functions/tfplan-functions/tfplan-functions.sentinel

# with alias "plan"

**import “tfplan-functions” as plan**

usage:

plan.find\_resources(resource\_name)

Assume any line you find below in the examples comes from this loop:

for resource\_collection as r, rc {

// Some code

}

**Tags**

* **Single-word**

rc.change.after.tags.SingleWord

* **Space-separated words**

rc.change.after.tags[“Space Separated Word”]

**Casting to string**

You need to import any of the \**-functions.sentinel*modules.

For example, *tfplan-functions.sentinel*

# Import common-functions/tfplan-functions/tfplan-functions.sentinel

# with alias “plan”

**import “tfplan-functions” as plan**

Usage:

plan.to\_string(item)

**Validating and Storing Values gracefully**

To avoid a runtime error when storing items do the following:

* **Collections:** *use curly braces*

var\_name = plan.find\_resources(“aws\_some\_resource”) else {}

* **Attributes within a resource:** when an attribute like tags doesn’t exist

var\_name = rc.change.after.tags else "default-error"

then use an if clause to validate if tags is different than “default-error”

if(var\_name != "default-error"){

// some code here

}

* Variables: regular variables that hold information. As mentioned before they do **NOT** work in *rules*; **use** them in *functions or elsewhere*.

function\_name = func(resource\_collection){

// store stuff anywhere in here

// consider scopes when thinking of variables.

variable\_name = definition...

}

variable\_name2 = definition...

**For loops**

The b*reak* and *continue* statements do exist in Sentinel.

* Break example:

for resource\_collection as r, rc {

// Some code

 if(/\*some condition\*/){

   // some code

      break

 }

}

* Continue example:

for resource\_collection as r, rc {

// Some code

  if(/\*some condition\*/){

   // some code

       continue

 }

}

**Selecting items within the Terraform structure**

* Attributes:

For example tags:

tags *=* {

a = "b"

"a b" = "c"

}

Select them is with dot notation because it has an equals sign (‘=’) before the block:

rc.change.after.tags.a

* Block of code:

example1 {

example2 {

 attribute\_name = "attribute\_value"

         }

}

Selection when attributes *don’t* have ‘=’ signs:

example1[0].example2[0].attribute\_name

**Finding Blocks within resources**

Import the module tfplan-functions:

# Import common-functions/tfplan-functions/tfplan-functions.sentinel

# with alias “plan”

**import “tfplan-functions” as plan**

For example, when your Terraform code has a resource with a code of block that is repeated multiple times like **ip\_set\_descriptors**within in **aws\_waf\_ipset**examplefound [here](https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/waf_ipset).

You can find the multiple repetitions from **ip\_set\_descriptors**by using the following line, where the first parameter *rc* comes from our loop item, and the second one is the repeated code item:

plan.find\_blocks(rc, "ip\_set\_descriptors")

**Converting a JSON value to an actual JSON Structure**

Let’s use the following example of an S3 bucket policy; as you can see there is a policy attribute that contains a JSON- like structure in a policy.

resource "aws\_s3\_bucket\_policy" "b" {

 bucket = aws\_s3\_bucket.b.id

**policy** *= <<POLICY*

*{*

*"Version": "2012-10-17",*

*"Id": "MYBUCKETPOLICY",*

*"Statement": [*

*{*

*"Sid": "IPAllow",*

*"Effect": "Deny",*

*"Principal": "\*",*

*"Action": "s3:\*",*

*"Resource": "arn:aws:s3:::my\_tf\_test\_bucket/\*",*

*"Condition": {*

*"IpAddress": {"aws:SourceIp": "8.8.8.8/32"}*

*}*

*}*

*]*

*}*

*POLICY*

}

To convert the string contained in policy to a JSON structure perform the following:

json\_holder = json.unmarshal(rc.change.after.policy)

*Note: Don’t forget to import the JSON module/library.*

The line above should store the value of the JSON structure from the policy definition.

**Extracting JSON’s attribute values**

Notice in the previous example how Statement contains its values within braces, which means we need to call Statement in the way showed below. Notice it uses braces and an index within them.

json\_holder.Statement[0]

Note: if you want to extract only the value of Statement, the braces and the index can be omitted, but later you won’t be able to dig deeper into the item.

**Digging deeper into the JSON structure**

To select an item within Statement, do the following:

* **For direct attributes, like “***Sid”***:**

json\_holder.Statement[0].Sid

* **For deeper items like “***aws:SourceIP”*

json\_holder.Statement[0].Condition.*IpAddress*[“*aws:SourceIp*“]