Confluent Terraform Provider

Scripting Confluent Cloud

# Objectives

We have created and configured resources like a Kafka Cluster or a connector through the Confluent Cloud Console (UI) or the Confluent CLI.   
This approach is appropriate for learning about the different configurations, but we need more for a production environment, such as

* Documentation
* Repeatable Process
* Governance of who can create or modify resources

Here is where the Confluent Terraform Provider comes in. It is the official and supported way to create and configure resources via a script.

In this lab, we will slowly build up a set of Terraform scripts that can serve as a template for creating a production-ready environment.

# Labs

## Preparations

### Note about security

You will need a Cloud API Key and Secret to access the Confluent Cloud and create and modify resources.

Under no circumstances should you upload the key and secret into public storage such as a GitHub repository, but you need to have both values somewhere in your environment. Here are some hints.

* Create a Terraform script, for example, **variables.tf**, and define two variables:

| variable "confluent\_cloud\_api\_key" {  description = "Confluent Cloud API Key (also referred as Cloud API ID)"  type = string  }  variable "confluent\_cloud\_api\_secret" {  description = "Confluent Cloud API Secret"  type = string  sensitive = true  } |
| --- |

* Then, create a cloud API key in your UI and download it.
  + Under the “Hamburger” Menu, navigate to “Cloud API Keys”.
  + Create a new key
  + Specify global access
  + Give the key a description, then download the key and secret

You can store the key and secret in different locations. Choose the option you find most appropriate for your environment.

1. Create two environment variables in the shell where you will invoke Terraform.

| export TF\_VAR\_confluent\_cloud\_api\_key="<cloud\_api\_key>"  export TF\_VAR\_confluent\_cloud\_api\_secret="<cloud\_api\_secret>" |
| --- |

1. If you have [direnv](https://direnv.net/) installed, you can store these variables in your .envrc file, guaranteeing they are enabled whenever you navigate your Terraform directory.
2. Alternatively, set the variables in your terraform.tfvars file, but do not add this file to your Git repository (you *are* versioning your scripts, right?)

| confluent\_cloud\_api\_key = "<cloud\_api\_key>"  confluent\_cloud\_api\_secret = "<cloud\_api\_secret>" |
| --- |

### Initialising your Terraform environment

You need to reference the Confluent Terraform provider in your script so that Terraform knows what to download. You can fix a particular version (the provider changes regularly) or always pick the latest version. For a production environment, the recommendation is to define the version and only upgrade it after testing.

After downloading the provider, initialise it using your API Key and Secret.

Create a new file with the following content, maybe called provider.tf (it has to have the .tf extension).

| terraform {  required\_providers {  confluent = {  source = "confluentinc/confluent"  version = "2.4.0"  }  }  }  provider "confluent" {  cloud\_api\_key = var.confluent\_cloud\_api\_key  cloud\_api\_secret = var.confluent\_cloud\_api\_secret  } |
| --- |

If you wan[t](https://registry.terraform.io/providers/confluentinc/confluent/latest) to try the latest version, you can check this website for the Terraform provider, which is updated regularly.

To download the provider code and verify your basic environment is set up correctly, run

terraform init

### 

### Note about lifecycle protection

The documentation recommends adding the following stanza to important resources like environments:

| lifecycle {  prevent\_destroy = true  } |
| --- |

While this is important in a production environment, in a bootcamp environment, this is in the way since you cannot easily destroy resources for testing. We recommend you ignore this directive for this bootcamp.

**You are now ready to proceed with the actual lab.**

## Create a new environment

By now, you will have realised that you spread your Terraform definitions across multiple files as long as they are in the same directory and end in .tf.

This allows you to dedicate each file to a particular purpose and keeps file sizes reasonably small. Of course, you can also merge all the files into one if you prefer.

Using an IDE like IntelliJ helps a lot here since it allows you to use code completion if you need to reference another resource from the same project.

Let’s create a new environment:

* Create a new environment with a new name, maybe “terraform”.
  + Add a file called “environment.tf”, following the [documentation](https://registry.terraform.io/providers/confluentinc/confluent/latest/docs/resources/confluent_environment).  
    Remember to leave out the lifecycle protection.
  + You need to reference the desired stream governance package at this stage. Choose the package “ESSENTIALS”.
* Run “terraform plan” to see if your file contains no syntax or dependency errors.
* If Terraform is satisfied, you can run “terraform apply” to commit the changes. You will have to type in “yes” to confirm that you want to make the changes.
* Once Terraform has run successfully, check your progress in the Confluent Cloud UI. You should see your new environment there.  
    
  Note that the schema registry has not been created yet for you; this will happen when you create your first cluster.

## Create a cluster

We need a **standard** cluster for these labs. Follow the [documentation](https://registry.terraform.io/providers/confluentinc/confluent/latest/docs/resources/confluent_kafka_cluster#example-usage) on the syntax. You can pick a cloud provider and region you prefer.

* Create a cluster via Terraform.
* Rerun “terraform apply”

Notice that you need to reference the environment. This defines the dependency graph, telling Terraform in which order it needs to create the resources.

Also, notice as you create your cluster, the environment is reused, not recreated. Terraform is idempotent; you can run the same setup multiple times. If nothing changes in your configuration files, nothing will change in the target environment.

## Create a service user with cluster permissions

The next step is to create a service user, and you will need to create topics.

* Create a service account following the [documentation](https://registry.terraform.io/providers/confluentinc/confluent/latest/docs/resources/confluent_service_account).
  + The user is referenced as “app-manager” in the rest of the lab.

To create the topics, we need to give the service account the correct permissions. In this case, we simplify our life by giving the user you created ClusterAdmin rights.

* Assign the CloudClusterAdmin role to your service user following this [documentation](https://registry.terraform.io/providers/confluentinc/confluent/latest/docs/resources/confluent_role_binding).
  + Reference the user as "User:${confluent\_service\_account.app-manager.id}"
  + The role name is "CloudClusterAdmin"
  + The crn\_pattern is the resource\_name of your cluster name

We also need an API Key and Secret. You can print these out later using the Terraform “output” resource or write them to a local file using the “local\_file” resource, but we just need the API Key and Secret internally.

* Create an API Key and Secret following this [documentation](https://registry.terraform.io/providers/confluentinc/confluent/latest/docs/resources/confluent_api_key).
  + You need to specify the owner, managed\_resource and environment.
  + As usual, ignore the lifecycle for this bootcamp.

Once you have finished your coding, rerun Terraform to see the result.

* Rerun “terraform apply”

## Create some topics

Now that you have an API Key, we can create some topics.

* Create two topics following this [documentation](https://registry.terraform.io/providers/confluentinc/confluent/latest/docs/resources/confluent_kafka_topic).
  + Ensure to add the configuration for the partitions\_count since you cannot change that anymore without dropping the topic
  + Using the config argument, create one compacted topic. Again, you cannot change “cleanup.policy” after you create the topic.
* Rerun “terraform apply”

## Reference the schema registry

The next step is to get a reference to the schema registry that has been created for you in your environment.

* Reference the Schema Registry using a data source following this [documentation](https://registry.terraform.io/providers/confluentinc/confluent/latest/docs/data-sources/confluent_schema_registry_cluster).
* To avoid dependency issues between the schema registry cluster and the environment and Kafka cluster, add a “depends\_on” clause that references your Kafka cluster. This will ensure that, if you recreate the whole environment, the Schema Registry is created before you reference it. Otherwise, you might encounter what is known as a “race condition”, and the creation will ultimately fail.  
  Here is an example:

| data "confluent\_schema\_registry\_cluster" "essentials" {  environment {  id = confluent\_environment.stream\_bootcamp.id  }  depends\_on = [  confluent\_kafka\_cluster.bootcamp  ]  } |
| --- |

* Add a new service account for your schema management.
* Add a new role binding for your new service account
  + Use the role “ResourceOwner”
  + Use the crn\_pattern "${confluent\_schema\_registry\_cluster.<<your cluster name>>.resource\_name}/subject=\*". <<your cluster name>> in the documentation is “essentials”.
* Create an API Key for the schema registry user
* Add the API Key and Secret to Terraform's output (see documentation [here](https://developer.hashicorp.com/terraform/language/values/outputs)). Alternatively, you can create a local file based on [this](https://registry.terraform.io/providers/hashicorp/local/latest/docs/resources/file) documentation.  
  We will need the API Key and Secret when we run a producer.
* Rerun “terraform apply”

Note that you could be more specific with your service user by only permitting them to read from the schema, for example. The schema then would need to be uploaded with a separate service user, which is often true for production but goes beyond the scope of this bootcamp.

## Create a service user for producing data

We could use the CloudClusterAdmin manager to produce data, but this user has admin rights over the whole cluster, which violates the least privilege principle.

* Create a new service user
* Give the service user DeveloperWrite permissions over the two topics you just created.
* Create an API Key and Secret and either print it out (output) or write it to a local file (local\_file).
* Rerun “terraform apply”
* Use the API Key to run a producer against your first topic without the schema registry.
* Use an application (for example producer.RegionProducer from [kafka\_transactions](https://github.com/sknop/kafka_transactions)) that accesses the schema registry to access the second (compacted) topic.
* Use the Confluent Cloud UI or a consumer app (or both) to consume the messages you created.
  + If you run an application, you will find that your API Key might not have the correct permission to read from the topics and access the consumer groups you need. Do this via your Terraform script, not manually.

## Create a managed connector

Finally, create a managed connector. We will be using the Datagen connector again since it has no external dependencies.

* Create another service user for your connector
* Create another topic
* Give this user permissions via ACLs, following [these](https://registry.terraform.io/providers/confluentinc/confluent/latest/docs/resources/confluent_kafka_acl) instructions.
  + You need DESCRIBE permissions on the cluster.
  + You will also need WRITE permissions on your new topic
* Create the connector using [these](https://registry.terraform.io/providers/confluentinc/confluent/latest/docs/resources/confluent_connector) instructions.
* Verify that the connector is producing through the UI and via a consumer.

# References

[Confluent Terraform provider overview](https://registry.terraform.io/providers/confluentinc/confluent/latest/docs)

[Environment](https://registry.terraform.io/providers/confluentinc/confluent/latest/docs/resources/confluent_environment)

[Kafka cluster](https://registry.terraform.io/providers/confluentinc/confluent/latest/docs/resources/confluent_kafka_cluster)

[Service Account](https://registry.terraform.io/providers/confluentinc/confluent/latest/docs/resources/confluent_service_account)

[API Key](https://registry.terraform.io/providers/confluentinc/confluent/latest/docs/resources/confluent_api_key)

[Kafka topic](https://registry.terraform.io/providers/confluentinc/confluent/latest/docs/resources/confluent_kafka_topic)

[Schema Registry](https://registry.terraform.io/providers/confluentinc/confluent/latest/docs/resources/confluent_schema_registry_cluster)

[ACLs](https://registry.terraform.io/providers/confluentinc/confluent/latest/docs/resources/confluent_kafka_acl)

[Connectors](https://registry.terraform.io/providers/confluentinc/confluent/latest/docs/resources/confluent_connector)

# Expected Outcomes

Create a whole production environment in the Confluent Cloud using Terraform.

Your cluster should be fully secured, and be provisioned with topics and API Keys without the need to create any additional resources manually.

Prove that the cluster is indeed accessible by successfully running producers and consumers using the API Keys you created.

# Check your understanding

This colour marks advanced questions.

* Why use Terraform to create your infrastructure?
  + What alternatives would you use if Terraform was not available?
* Why should you not upload your API Key into a public repository?
* Which resource can you not create via Terraform?