Published in Panoptikum

Mario Kleinsasser Follow
Sep 10, 2020 · 7 min read · Listen

Save

# Using Terraformed Google GCP Data Transfer Service makes data safer!

Creating a backup even for cloud-based data storage should be obvious! But how-to create backup jobs between different GCP projects and multiple buckets automatically? Terraform can create GCP Data Transfer Service (DTS) jobs but there are a couple of pitfalls you might like to avoid. Here’s how!

Photo by Mitchell Luo on Unsplash

## Preface

Transferring data from local storage, for example, data within a virtual machine, to cloud storage (whatever cloud you choose) is quite a common task today. There are a lot of software products out there which are allowing and supporting such tasks out of the box, one example may be GitLab here with the "consolidated object storage" configuration. Moving data to the cloud this way is easy, but with this migration, the backup situation changes. To lower the abstraction level, we will stick with the GitLab example for this post.

If you are running GitLab on-premises (on your own), you will probably use the GitLab Omnibus installation, this is a one-package installation which brings everything you need and after the installation, you will find a convenient tool called *gitlab-backup* which you can use to do the all-in-one backup. This backup will backup everything which is on your local disk into a *tar.gz* file and therefore, in larger GitLab installations, this file can grow up to several *hundreds of gigabytes* whereas the uploads and artifacts can use up to fifty percent of the size of the tar.gz files.

Using *GitLab's consolidated object storage* can therefore reduce not only the backup file size by using cloud storage, but it will also reduce the backup time! Why? If the files are not on the local disk anymore (they are in cloud storage), **they are not picked up** by the *gitlab-backup* tool anymore! If you do not want to lose them, and I swear you won't lose them, you should take care to have a valid cloud storage configuration.

## Cloud Storage

As described in the preface chapter, a suitable and secure cloud storage configuration is essential for GitLab file data. It is always possible to configure *object versioning* for the used bucket — in this post we are talking about Google GCS buckets but all cloud providers offer such systems — and you should use versioning! Also, the use of *object life-cycle-policies* and object *retention-policies* is highly recommended! You can read the Google documentation about the possible settings. After some configuration, about we will talk later, you should have a reliable bucket (or multiple buckets) set up for your needs, but there is one case still open, which is covered neither by object-versioning and object-life-cycle-policies nor by object retention-policies!

> *What if your cloud storage access data (GCP service account key) is stolen and some suspect deleted everything? Your local data, your cloud data, your cloud backups (buckets and version), just everything?*

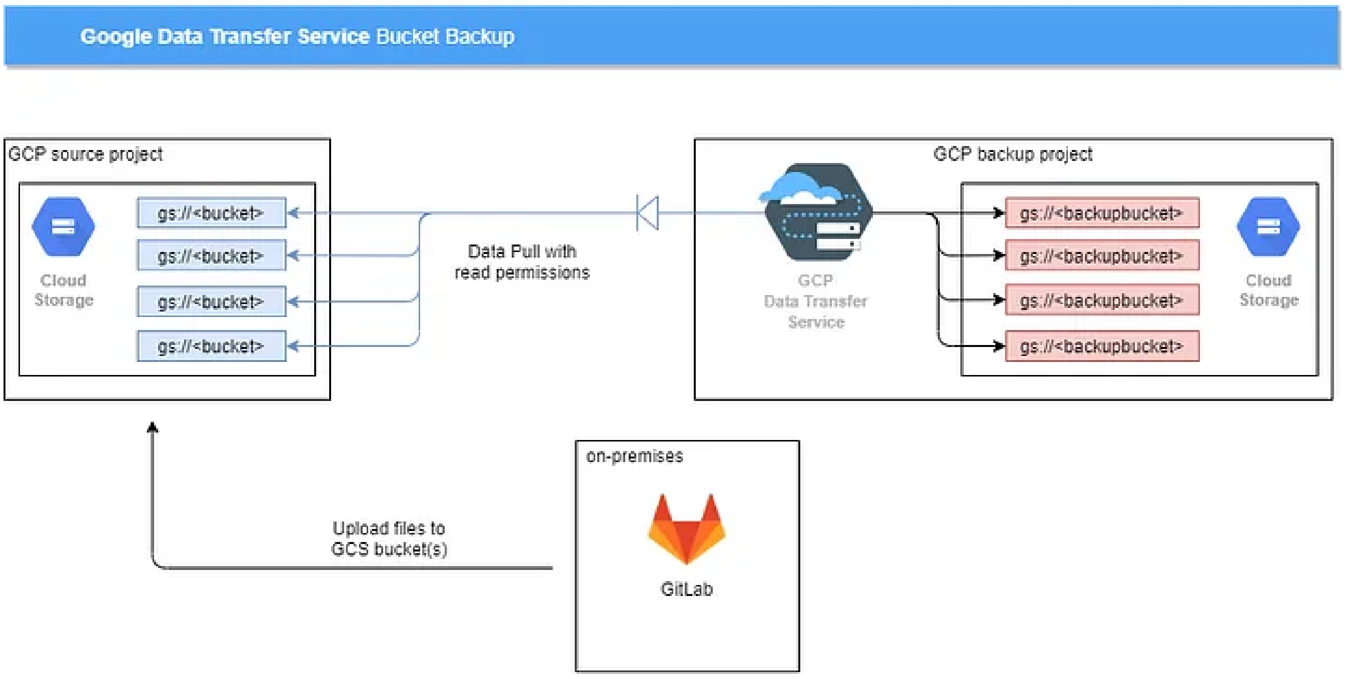
Image by Pete Linforth from Pixabay

## The Crime Scene

If someone gains access to your GitLab instance (as an example in this post) or someone gains knowledge about your service account key which is used by the GitLab installation, your data is in trouble! Ever software which is doing cloud backups or which is storing data in cloud storage, in general, *must* have access to credential used for the cloud storage as otherwise, it would not be possible to upload files there. Uploading in this case means *managing* data and this often allows *create, update (modify) and delete* operations. If the user credentials are allowed to delete data (which is often a must), your data can be deleted. In the cloud case, not only the *live objects* in the bucket but also *the archived object-versions* and probably *the bucket(s)* itself could be deleted. This situation should be avoided!

## The Idea

That's the point where Googles GCP Data Transfer Server (DTS) can help you and the idea behind it is, to decouple the backup job (DTS) from the source bucket. The DTS job is configured inside another GCP project which has no connection to the source bucket in the source GCP project. The only permissions which are needed are read permissions for the GCP service account user from the Backup project. The following diagram displays the details of the setup.



As shown above, the data can only be pulled from the source project buckets and transferred into the backup buckets of the destination/backup GCP project. Neither the GitLab installation nor the GCP source project will have access to the backup project.

## The Terraform Plan

Terraform plans can be applied in many different ways and how you do it mostly depends on your possibilities. We are using a Docker image that contains the *terraform* binary and some glue-code which enables the usage of the Docker image within a GitLab pipeline but that's another story which I will post later sometime. What you need is the ability to apply Terraform plans against two different GCP projects as shown above.

```
# ------------ artifacts --------------
resource "google_storage_bucket" "artifacts" {
  name          = "artifacts-<your google project id>"
  project       = "<your google project id>"
  location      = "EUROPE-WEST3"
  storage_class = "STANDARD"
  force_destroy = false

  versioning {
    enabled = true
  }

  lifecycle_rule {
    condition {
      age = "1"
      with_state = "ARCHIVED"
    }
    action {
      type = "Delete"
    }
  }
}

resource "google_storage_bucket_iam_member" "artifacts-objectViewer" {
  bucket = google_storage_bucket.artifacts.name
  role = "roles/storage.objectViewer"
  member = "serviceAccount:project-<your google backup project number>@storage-transfer-service.iam.gserviceaccount.com"
  depends_on= [google_storage_bucket.artifacts]
}

resource "google_storage_bucket_iam_member" "artifacts-legacyBucketReader" {
  bucket = google_storage_bucket.artifacts.name
  role = "roles/storage.legacyBucketReader"
  member = "serviceAccount:project-<your google backup project number>@storage-transfer-service.iam.gserviceaccount.com"
  depends_on= [google_storage_bucket.artifacts]
}

resource "google_storage_bucket_iam_member" "artifacts-gitlab" {
  bucket = google_storage_bucket.artifacts.name
  role = "roles/storage.objectAdmin"
  member = "serviceAccount:gitlab@<your google project id>.iam.gserviceaccount.com"
  depends_on= [google_storage_bucket.artifacts]
}
```

google-source-bucket-terraform.tf hosted with ❤ by GitHub view raw

The excerpt in the Gist above is used to create the buckets in the first project. Here we define that we would like to have a versioned bucket and we would like to have all archived object versions to be deleted after one day. Why? This offers us the possibility to solve one main problem of every sync-based backup solution namely the “Missing Recycle Bin”-problem. The problem arises when you are doing your backup one a day for example at 10 pm. Let’s say a file is stored at 8 am and accidentally deleted at 3 pm.

> *Then, for the backup task, the file was never there as shown in the diagram below.*

If we are using a versioned bucket, with at least one day retention time, we had built a recycle bin for 24 hours. Nice! Googles DTS will only sync objects which are *“alive”* which means that no archived versions (deleted, changed,…) of objects will be synced. This is important to know because, in our second project, we would have to configure the bucket to retain the data for a longer time period. And, the most important thing of all is, that we allow the Google Service Account for the Data Transfer Service to read all files. Google creates a special service account user for DTS automatically and it always has the structure of “*project-<your google backup project number>@storage-transfer-service.iam.gserviceaccount.com”*. Be aware, that you are not using the GCP *“project-id”* here! You need your GCP’s project number! You can view the project number via Googles Cloud shell:

```
# gcloud projects list
PROJECT_ID    NAME                PROJECT_NUMBER
my-project    Happy Project       12345678
```

```
# ------- artifacts --------
resource "google_storage_bucket" "artifacts-<your source google project id>-backup" {
  name          = "artifacts-<your source google project id>-backup"
  project       = "<your google backup project number>"
  location      = "EUROPE-WEST4"
  storage_class = "STANDARD"
  force_destroy = false

  versioning {
    enabled = true
  }

  lifecycle_rule {
    condition {
      age = "90"
      with_state = "ARCHIVED"
    }
    action {
      type = "Delete"
    }
  }
}

resource "google_storage_bucket_iam_member" "artifacts-<your source google project id>-backup-objectAdmin" {
  bucket     = google_storage_bucket.artifacts-<your source google project id>-backup.name
  role       = "roles/storage.objectAdmin"
  member     = "serviceAccount:${data.google_storage_transfer_project_service_account.default.email}"
  depends_on = [google_storage_bucket.artifacts-<your source google project id>-backup]
}

resource "google_storage_bucket_iam_member" "artifacts-<your source google project id>-backup-legacyBucketReader" {
  bucket     = google_storage_bucket.artifacts-<your source google project id>-backup.name
  role       = "roles/storage.legacyBucketReader"
  member     = "serviceAccount:${data.google_storage_transfer_project_service_account.default.email}"
  depends_on = [google_storage_bucket.artifacts-<your source google project id>-backup]
```

As you can see in the above excerpt from the second GCP project (the backup project), we are also using a versioned bucket, but we delete the archived versions after 90 days.

> *Even if a suspect changes or data (ransomware for example) we have a versioned copy of a live version for 90 days.*

It is important here to have a good naming scheme for your terraform resources. If you are using a central backup Google project, you might have a lot of buckets there and buckets must have a globally unique name.

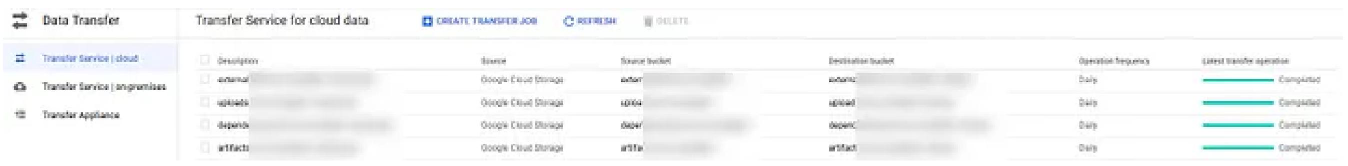
## Terraform Googles DTS

The last thing to do is to create Google DTS jobs with Terraform. As mentioned above, it is important that you name your Terraform resources correctly. In certain circumstances, you will have a lot of jobs inside one Google project and you don't want to lose an overview of them.

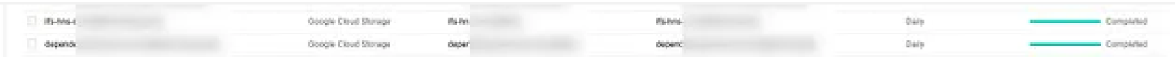
```
# ------- artifacts --------
resource "google_storage_transfer_job" "artifacts-<your source google project id>-backup" {
  description = "artifacts-<your source google project id> backup job"
  project     = local.project

  transfer_spec {

    transfer_options {
      delete_objects_unique_in_sink = true
    }

    gcs_data_source {
      bucket_name = "artifacts-<your source google project id>"
    }

    gcs_data_sink {
      bucket_name = google_storage_bucket.artifacts-<your source google project id>-backup.name
    }
  }

  schedule {
    schedule_start_date {
      year  = 2020
      month = 9
      day   = 1
    }
    schedule_end_date {
      year  = 2100
      month = 12
      day   = 31
    }
    start_time_of_day {
      hours   = 21
      minutes = 30
      seconds = 0
      nanos   = 0
    }
  }

  depends_on = [google_storage_bucket.artifacts-<your source google project id>-backup]
}
```

google-data-transfer-service.tf hosted with ❤ by GitHub view raw

The above Terraform Gist shows an example Google DTS Terraform file. The setup of a Google DTS job is easy as long as the permissions of the Google SA account used by the DTS-system is correct. Below you can see a screenshot of the Google Cloud Console where you see configured Google DTS jobs.



| Description | Source | Source bucket | Destination bucket | Operation frequency | Latest transfer operation |
|---|---|---|---|---|---|
| external[redacted] | Google Cloud Storage | exter[redacted] | externa[redacted] | Daily | Completed |
| uploads[redacted] | Google Cloud Storage | uploa[redacted] | upload[redacted] | Daily | Completed |
| depends[redacted] | Google Cloud Storage | deper[redacted] | depenc[redacted] | Daily | Completed |
| artifacts[redacted] | Google Cloud Storage | artifa[redacted] | artifact[redacted] | Daily | Completed |
| ifs-hns-[redacted] | Google Cloud Storage | ifs-hn[redacted] | ifs-hns-[redacted] | Daily | Completed |
| depends[redacted] | Google Cloud Storage | deper[redacted] | depenc[redacted] | Daily | Completed |



## Takeaways

First of all, I would like to say thank you to Hermann Wagner who works as an Application Modernization Specialist at Google Austria. 🤗In the last one and a half years we analyzed a lot of services together and also tried some new ones like Google Anthos, Traffic Director, and many more.

As you have read in this story, you can automate the process of bucket creations and using Google's DTS to create backups automatically with Terraform. The main points who have to have an eye on are:

- Take about the correct bucket permissions (in the source and destination buckets)
- Don’t forget that the Google Service Account used by DTS is a special one which is already created for you
- Try to implement a naming scheme for your Terraform resources as you probably will have many of them
- Think about the correct settings for object-versioning, object-life-cycling, and object-retention times

That’s it for today! Hopefully, you have enjoyed this story and it helps you to set up a useful Google DTS based bucket backup!

*Last edited on 10th September 2020*

Google Terraform Backup Gitlab