DevOps Self-Service Centric Pipeline Integration

Secrets management as the glue of pipeline integration

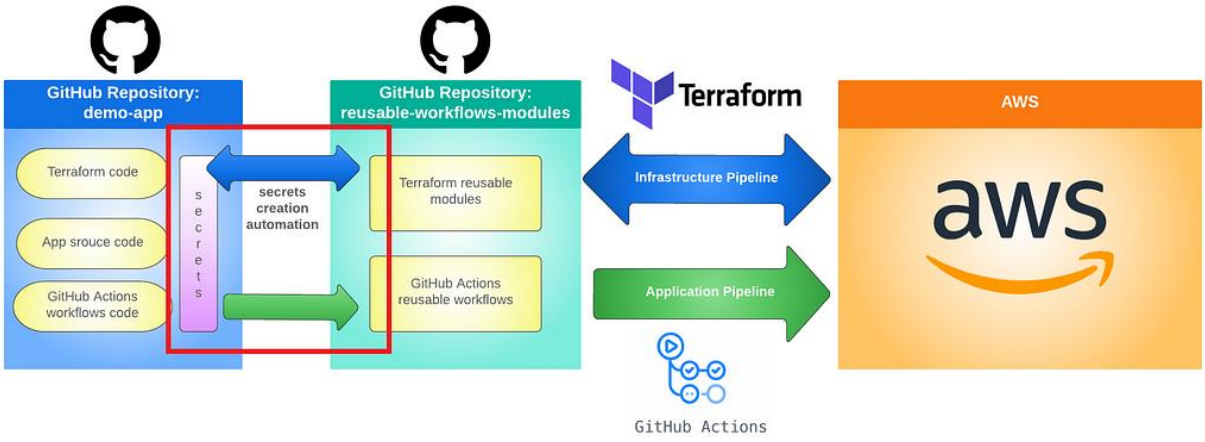

Diagram by author

Before we dive into pipeline integration, let's take a look at the journey we have come along toward DevOps self-service:

- Part 1: DevOps Self-Service Pipeline Architecture and Its 3–2–1 Rule
- Part 2: DevOps Self-Service Centric Terraform Project Structure
- Part 3: DevOps Self-Service Centric GitHub Actions Workflow Orchestration
- Part 4: DevOps Self-Service Centric Pipeline Security and Guardrails

We focused on one particular area in each of those four parts of the DevOps self-service series. The 3–2–1 rule explained in Part 1 of the pipeline architecture fleshed out our overall path to DevOps self-service.

In this story, we will focus on the “1” in the 3–2–1 rule, the pipeline integration (see the part highlighted in red in the diagram below). This secrets creation automation is the glue that holds the infrastructure pipeline and the application pipeline together.

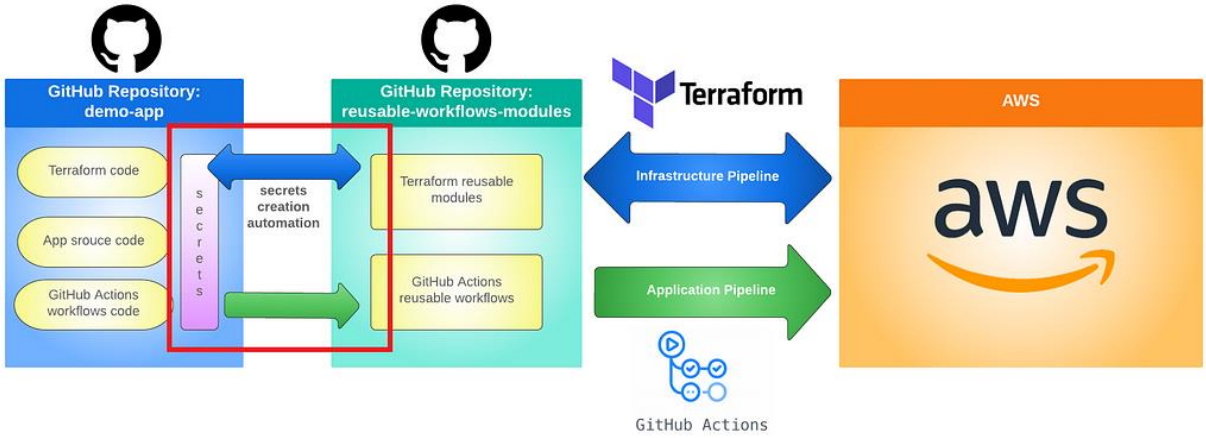

Diagram by author

What kind of secrets are we talking about, you ask.

This is not the application secrets that belong in the secrets management tool of your cloud provider, such as AWS Secrets Manager.

This is data from your Terraform outputs, which need to be consumed by your application pipeline.

After Terraform GitHub Actions workflow applies Terraform configuration, your resources are successfully provisioned in your cloud provider. With your infrastructure ready, you can kick off your application pipeline to deploy your code into your newly provisioned infrastructure.

In between these two steps, you often need to configure GitHub secrets to instruct your application workflows to interact with your cloud infrastructure. This may include specifying the S3 bucket name, so your SPA can be uploaded to the specified bucket, or for backend microservices to be deployed to AWS ECS, configuring a bunch of secrets such as cluster name, ECS service, task definition, container name, etc.

You may have been configuring these secrets manually after your infrastructure is provisioned. There's got to be a better way! That's where the pipeline integration through GitHub secrets creation automation comes into play.

Terraform is truly an amazing tool. Out of its 2800+ providers at official, partner, and community tiers, one provider interacts with GitHub resources — the GitHub provider. This provider will assist us in this secrets creation automation. Let's explore how to automate these secrets creation/updates in your infrastructure pipeline.

Step 1: Add GitHub Provider To Your Terraform Configuration

The snippet below calls the cloudposse/cloudfront-s3-cdn/aws Terraform module to provision an AWS CloudFront CDN with an S3 origin.

- Lines 6–9: add GitHub provider, which can coexist with the AWS provider from lines 1–3. Notice GitHub provider needs both `token` and `owner` configured. We will explain how to configure `token` in step 2 below. For `owner`, enter the owner of your GitHub repository.
- Lines 30–42: two GitHub environment secrets were created by calling the GitHub provider's resource `github_actions_environment_secret`.

One secret is for `S3_BUCKET_NAME`, the other `CLOUDFRONT_DISTRIBUTION_ID`. Notice we need to pass in `repository` and `environment` for this resource, which are fed in from the application pipeline. We will explore the details in step 2.

- Also, note that we pass the values for the secrets into `plaintext_value` on lines 34 and 41. For security, the contents of the `plaintext_value` field have been marked as `sensitive` in Terraform, but it is important to note this does not hide it from state files. In our case, since we are only storing the S3 bucket name and CloudFront distribution ID in the secrets, not any sensitive data such as a password, it is fine to store in `plaintext_value`.
  For sensitive values, it's recommended to populate the `encrypted_value`. Whether you use `plaintext_value` or `encrypted_value`, it's best practice to extract fields from a resource or a data source as the value of a secret. In our case, we are getting the secret values from the outputs of `module.static_site`.

Step 2: Pass Environment Variables From Terraform GitHub Actions Workflow Into Terraform Configuration File

As mentioned in step 1, we have three values to explore, which we deferred to this step. They are:

- `pipeline_token`
- `deploy_repo`

- `deploy_env`

How do you define these values in Terraform GitHub Actions workflow? Read on.

`PIPELINE_TOKEN`: You need this token for Terraform to call the GitHub provider to auto-create GitHub secrets, such as `S3_BUCKET_NAME`, based on the resources Terraform provisioned. To generate the token, do the following:

- Go to https://github.com/settings/tokens
- Press "Generate new token"
- Be sure to assign the token "`repo`" scope and "`read:public_key`" scope (see screenshot below)
- Name the token `PIPELINE_TOKEN`
- Copy the token value

To add this new token as your repository secret, navigate to Settings → Secrets → Actions → new repository secret. Create a new repository secret with the key `PIPELINE_TOKEN` and value what you just copied from above.

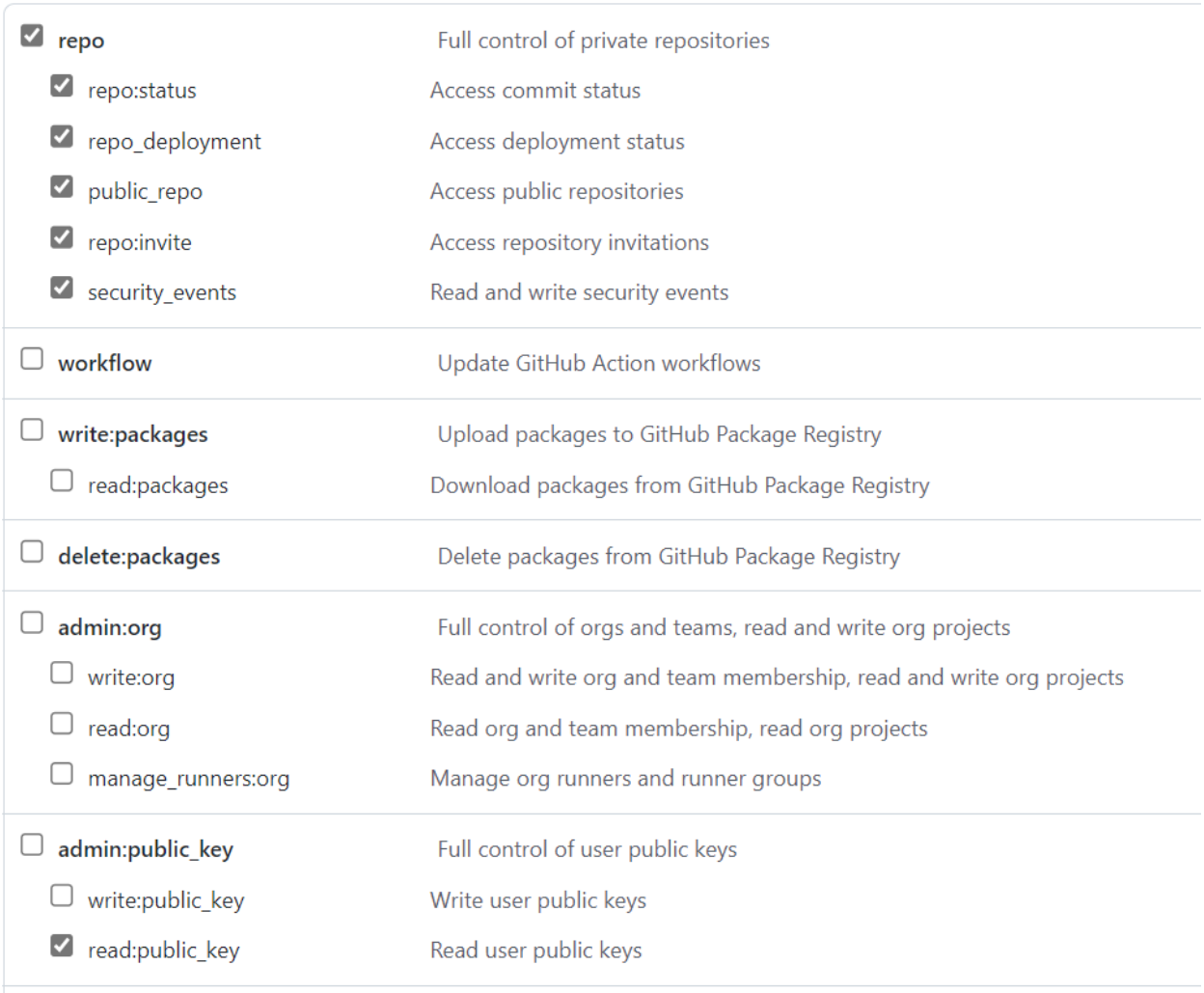

Image by author

`deploy_repo` and `deploy_env` are two environment variables configured in the Terraform GitHub Actions workflow; see lines 30-31 below. `deploy_repo` refers to the GitHub repository this workflow runs from, expressed in GitHub context with `github.event.repository.name`.

`deploy_env` refers to the environment this workflow runs for. This environment is selected from the manual trigger. If it's not selected through the manual trigger, it defaults to `dev`: `github.event.inputs.environment || 'dev'`.

Now that these environment variables are in place, how do we pass them to Terraform configuration files? Notice the Terraform plan step, lines 105–107. First, we convert `DEPLOY_REPO` and `DEPLOY_ENV` to lowercase via an expression such as `${DEPLOY_REPO,,}`, and then pass the lowercase values to Terraform as its environment variables, which always start with `TF_VAR_` as their prefix.

Now, let's look at how those environment variables injected from Terraform GitHub Actions workflow can be used in Terraform configuration files. First, you need to define them in `variables.tf` for your project. See the sample code snippet below:

```
variable "deploy_repo" {
  description = "application's repo name"
  type    = string
  default = "myrepo"
}

variable "deploy_env" {
  description = "deployment environment"
  type    = string
  default = "dev"
}

variable "pipeline_token" {
  description = "pipeline token"
  type    = string
  default = ""
}
```

When those Terraform environment variables are used in Terraform configuration files, such as `main.tf`, you can refer to those Terraform environment variables as normal variables in the form of `var.pipeline_token` (line 2), `var.deploy_repo` (line 9), and `var.deploy_env` (line 10).

Step 3: Verify GitHub Secrets in the Application Pipeline

Once your secrets are automated through the above step, you may want to verify whether the secret is created with the right value. As you may already know, GitHub secrets UI doesn't allow users to view the value of a secret once it's configured. If you need to verify the value of the secret, you can use the snippet below in your application CI or CD workflow, which uses `sed`, stream editor, a Unix command, to echo the secret value, with space separating each letter in the secret value.

```
- name: Verify secrets
  run: |
    echo ${{ secrets.S3_BUCKET_NAME }} | sed -e 's/\(.\)/\1 /g'
```

Note this is only for debugging and secrets that don't hold sensitive values. Once you finish verifying the secrets, remove or comment on this step from your workflow.

That's it! With GitHub secrets creation automation, we have accomplished true end-to-end pipeline integration between the infrastructure and application pipelines. Once the infrastructure is provisioned through the infrastructure pipeline, all secrets have already been automatically inserted into your GitHub repository, allowing the application pipeline to kick off and execute smoothly without the manual intervention of managing those secrets. This glue binds these two pipelines together, making it an end-to-end state-of-the-art pipeline experience for the developers.

Summary

Automating GitHub secrets from Terraform outputs helps the smooth transition between the infrastructure and application pipelines. This article dived into the details of

capturing Terraform outputs within Terraform GitHub Actions workflow and how to automate GitHub secrets creation/update using GitHub provider in Terraform configuration. We also looked at verifying if the secrets are created/updated as desired.

I hope you find this article helpful.

I welcome you to check out the rest of the four parts in my five-part "The Path to DevOps Self-Service" series:

DevOps Self-Service Pipeline Architecture and Its 3–2–1 Rule
A high-level architectural overview of the self-service pipeline
betterprogramming.pub

DevOps Self-Service Centric Terraform Project Structure
How to structure Terraform code and its reusable modules
betterprogramming.pub

DevOps Self-Service-Centric GitHub Actions' Workflow Orchestration
How to orchestrate GitHub Actions' workflows that are driven by image immutability
betterprogramming.pub

DevOps Self-Service Centric Pipeline Security and Guardrails
A list of hand-picked actions for security scans and guardrails for your pipelines, infrastructure, source code, base…
betterprogramming.pub

Happy coding!

References

https://registry.terraform.io/providers/integrations/github/latest/docs

https://github.com/cloudposse/terraform-aws-cloudfront-s3-cdn

https://registry.terraform.io/providers/integrations/github/latest/docs/resources/actions_environment_secret