Running a multi-tenant Kubernetes with GitOps

MORGRIDGE
INSTITUTE FOR RESEARCH
CORE COMPUTATION

FEARLESS SCIENCE

The Context

At Morgridge, we had the following set of requirements:

- Desire to run / host multiple on-premise services*,
- Have these services run by multiple teams – including both developers & operators,
- Provide service agility (no more "I'll build you a new VM next week!"),
- Interoperate with other sites doing operations, AND
- **Do this all on an academic budget** (staff time is very limited).

What did we end up with?

- The Tiger cluster at Morgridge,
- A multi-tenant Kubernetes cluster utilizing GitOps.

* Why on-prem? It's an interesting discussion but not the focus of the talk.

‘A multi-tenant Kubernetes cluster utilizing GitOps’

The Tiger cluster is formed by a bit of a patchwork:

- Three relatively powerful new hosts to form the core of the cluster.
- Six machines dedicated to supporting a single project
- A rack of old execute hosts (few hundred aging cores – great to spread services over but don’t particularly trust the disks).

We will not be setting any land-speed records with this hardware – but that’s not the intent.

Beyond basic built-in Kubernetes with Calico, cluster-wide services provided include:

- Integration with **SLATE**, ✅
- **DNS** integration (external-dns & CloudFlare), ✅
- Certificate management (**cert-manager** with both custom CA issuer and Let’s Encrypt), ✅
- Simple Persistent Volumes (local storage) ✅
- Databases on demand (via ElasticSearch and Postgres operators), ✅
- Sealed secrets for secret encryption, ✅
- **CVMFS** (via host mounts and a privileged Daemonset), ✅
- Centralized Logging, IN-PROGRESS ❌
- Persistent Volumes via **Rook** / **Ceph**. IN-PROGRESS ❌

Plus ‘flux’ and helm-operator for GitOps

Heads-up

In this talk, I'll be looking at centralized services appropriate for running in a single, non-distributed Kubernetes cluster.

Not tackling edge services or distributed Kubernetes.

‘A multi-tenant Kubernetes cluster utilizing GitOps’

Kubernetes + a list of additional services is probably clear to this audience. What do I mean by multi-tenant?

- I want to allow multiple teams to utilize the cluster.
- I want to provide the maximum functionality I can to the teams …
  - … but include reasonable safeguards.
  - I am willing to assume a high level of trust – I trust each team member to be non-malicious and deploy reasonable services. This is not mean to scale to arbitrary users.
- Flat trust model within the team: if you’re on the team, you can access everything within the team.
- For partitioning between the teams, my model is more “drywall” than “stone fortress”.

Multi-tenant Kubernetes

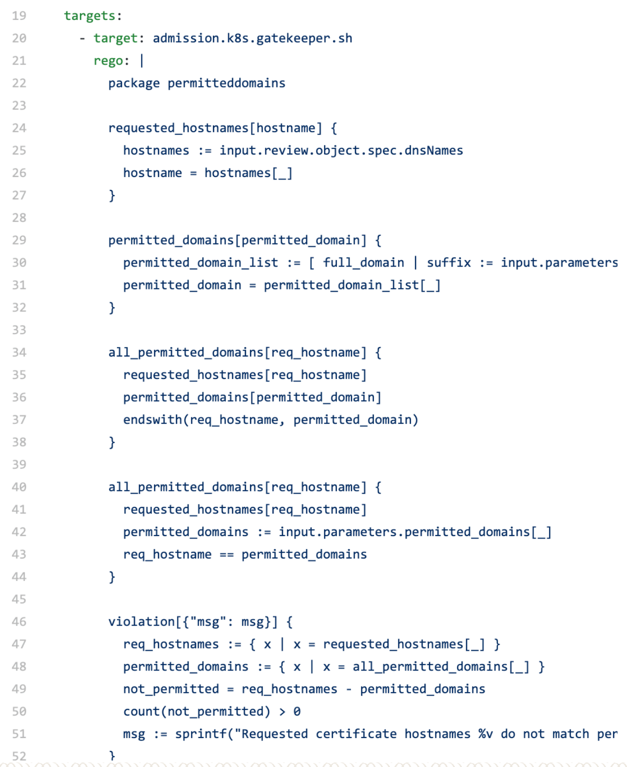
Each team gets two Kubernetes namespaces:

- One for development, one for production.
- Team members get read-write access to development and read-only access to production.
  - Team members are expected to use their write access judiciously in development – not meant to be the primary way to interact.
- Permissions to objects are managed via RBACs.
  - We have simple ClusterRoles we reuse to make RoleBindings in the appropriate namespace.
  - Each individual on the team gets their individual credentials (X.509 certificate – thanks to PRP for sharing their credential generation script with the SLATE team who shared it with us!).
- A default ingress and egress network policy is in place for all namespaces.
  - Pods must declare who they want to talk to.
- Additional security policies are enforced via Gatekeeper.

Gatekeeper

The Open Policy Agent Gatekeeper is an admissions controller that enforces custom policies.

- **Admissions controller**: The API server will send Gatekeeper changes
- **Open Policy Agent**: A framework for writing policies; allows one to reason about structured documents.
  - Policies are written in the Rego language, which is a declarative logic programming language. It's strange and bizarre unless you're a fan of Datalog… or perhaps HTCondor ClassAds.
- PS: admissions controllers are tricky beasts. Hilarity will ensue if they timeout requests instead of approve / disapprove. Beware network upgrades…



```
targets:
  - target: admission.k8s.gatekeeper.sh
    rego: |
      package permitteddomains

      requested_hostnames[hostname] {
        hostnames := input.review.object.spec.dnsNames
        hostname = hostnames[_]
      }

      permitted_domains[permitted_domain] {
        permitted_domain_list := [ full_domain | suffix := input.parameters
        permitted_domain = permitted_domain_list[_]
      }

      all_permitted_domains[req_hostname] {
        requested_hostnames[req_hostname]
        permitted_domains[permitted_domain]
        endswith(req_hostname, permitted_domain)
      }

      all_permitted_domains[req_hostname] {
        requested_hostnames[req_hostname]
        permitted_domains := input.parameters.permitted_domains[_]
        req_hostname == permitted_domains
      }

      violation[{"msg": msg}] {
        req_hostnames := { x | x = requested_hostnames[_] }
        permitted_domains := { x | x = all_permitted_domains[_] }
        not_permitted = req_hostnames - permitted_domains
        count(not_permitted) > 0
        msg := sprintf("Requested certificate hostnames %v do not match per
      }
```

FEARLESS SCIENCE MORGRIDGE INSTITUTE FOR RESEARCH

Building walls

Sample gatekeeper policies:

- Limit the domain names that can be requested by an Service.
  - The 'OSG' team in its 'osgdev' namespace can only request hostnames matching '*.osgdev.chtc.io'.
- Similarly, we limit the certificates that can be requested by the teams. Team A cannot request a certificate for hostnames belonging to Team B.
- The Kubernetes multi-tenancy SIG has a wonderful list of things that should not be allowed in multi-tenant environments.
  - They even have sample Gatekeeper policies to enforce these limits! Sadly, we haven't worked our way down this list yet.

There's a number of rocks left unturned in PodSecurityPolicies.

- Given the functionality we desire, it's unlikely we will take away privileged mode from the teams until there's more extensive user namespace support in Kubernetes. Maybe 2021?

‘A multi-tenant Kubernetes cluster utilizing GitOps’

From Weaveworks: “GitOps is a way to do Kubernetes cluster management and application delivery. It works by using **Git as a single source of truth for declarative infrastructure** and applications.”

That is,

- Current correct state of the infrastructure is kept in git.
- All changes are recorded in git.
  - You can do a \`git log\` and \`git blame\` on your infrastructure!
- Changes are deployed from git by an agent, not by a human.
- If you want to know what your cluster is supposed to be doing, look at git.
- If I delete every object in your cluster, the exact state can be reproduced from git.

GitOps with Flux

We use the **flux** operator in order to implement GitOps.

- A cluster-wide operator does a `git pull` from GitHub every minute.
- It does a `kustomize build` on the repository to render the manifests in the repository as Kubernetes objects.
  - It then does `kubectl apply`, creating or changing existing objects as needed.
- Any object in the cluster it previously created not currently in git will be deleted.
- The cluster's state becomes the Git repo's state!

The cluster-wide operator only does the basics!

- Creates CRDs we use.
- Manages RBACs, user accounts, and bindings.
- Installs other cluster-wide operators (e.g., ElasticSearch)
- Creates namespaces for each team.
- **Deploys a namespaced flux instance for the team**.

Namespaced flux instances

The cluster instance creates **two flux instances per team**

- One for development and one for the production namespace.
- Recall: **no users have write access to production, only flux**.
- Each instance looks at a specific GitHub repo / directory.
  - We have chosen the model of having a "production" and "development" directory inside the same branch. Alternatives include one branch per environment.

Everything Goes To GitHub!

- Yes, even secrets … kind of.
- We use the BitNami SealedSecrets operator. This **encrypts** the contents of the secret with a public key. Only the private key - which lives on the cluster – can decrypt the secret.
  - This prevents anyone with read access to our Git repo from reading the secrets. Only the cluster admin (or the operator) can decrypt things.

Other GitOps notes

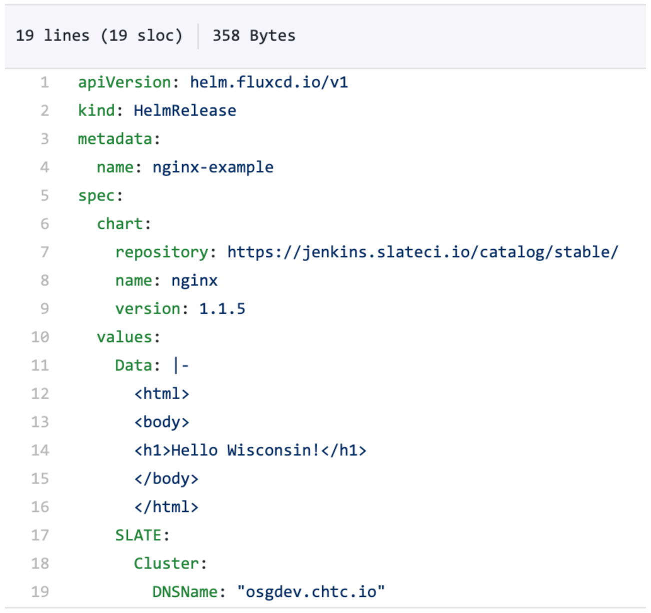
Flux will take care of image updates.

- It assumes you have an immutable tag model. Can be instructed to update the pod's container image version(s) when a new DockerHub image appears.
- Since we use GitOps – flux needs the ability to patch the Git repo with the new tag.
- We can filter the tags we're interested in – and decide to only automate deploys in ops.

The **helm-operator** will take a CRD and issue the appropriate helm commands.

- Your `config.yaml` from Helm is stored in git (see right),
- Deployed to Kubernetes by flux,
- And upgradces executed by the helm-operator.

Helm upgrades are not always fool-proof: I've had to manually kick things periodically.



19 lines (19 sloc) | 358 Bytes

```yaml
apiVersion: helm.fluxcd.io/v1
kind: HelmRelease
metadata:
  name: nginx-example
spec:
  chart:
    repository: https://jenkins.slateci.io/catalog/stable/
    name: nginx
    version: 1.1.5
  values:
    Data: |-
      <html>
      <body>
      <h1>Hello Wisconsin!</h1>
      </body>
      </html>
    SLATE:
      Cluster:
        DNSName: "osgdev.chtc.io"
```

A Day in the Life

What does (should?) this look like to an OSG developer?

1. Developer works on their laptop until they get a container they like.
2. Dockerfile is pushed to GitHub; GitHub Action setup to build and push to DockerHub.
3. Developer writes pod manifest and commits it to the OSG repository.
   - Maybe includes some credentials saved as a secret and a service description.
4. Wait for DNS, certificate, secret, then pod to be deployed onto the cluster. (2-3 minutes).
5. Change application code. Push changes to GitHub. Wait until its deployed on the cluster (5-7 minutes).
6. Once they're happy, import the manifest to the production environment!

When this all works, it's indistinguishable from magic.

Lessons Learned / Some Thoughts

1. Kubernetes provides us with agility that we never had before. We are still in the honeymoon phase. Has provided a new (and wonderful!) set of boundaries between development and operations.
2. I find GitOps essential when working as a team.
   - **Without GitOps**, I believe we don't have the discipline needed for multiple people to operate a service. When using `kubectl` directly, there's still too much about the desired state that's in the keyboard operator's head. Everyone becomes to scared to touch a service someone else deployed.
   - Life without GitOps = cowboy mode?
3. It is still early days for multi-tenant Kubernetes. I'm fairly happy with the balance we've struck between security and capabilities. Because it's custom, we got exactly what we want.
   - But at what cost in effort? E.g., OKD3 probably doesn't meet our needs. OKD4 is closer – will this all be perfect and wonderful in OKD5?

Open Questions

1. We want to federate with other clusters. How can we do that without losing GitOps? **Is Admiralty the answer**?
   - Current federation use case is being able to failover services to another datacenter during outages.
2. **Helm vs Kustomize**? Current thoughts: Helm when you want to build an application to share with others, Kustomize when it's bespoke / in-house.
3. How far can we limit privileges while still running complex services like HTCondor? We don't want to become a Kubernetes startup company! We will be watching efforts like KEP-127 (support for user namespace) to get rid of run-as-root.

MORGRIDGE
INSTITUTE FOR RESEARCH
CORE COMPUTATION

morgridge.org

This material is based upon work supported by the National Science Foundation under Grant No. 2030508. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

FEARLESS SCIENCE