Tracing a GitOps Reconcile

What actually happens between a `` `git commit` `` and running Pods?

about.md

Leigh Capili

- Flux Core Maintainer
- Principal Consultant @ ControlPlane
- 🇵🇭
- I grew up here in LA, live in Denver, CO

sidequests

- boba 🧋
- DDR, DanceRush, Pump it Up
- anything on a skateboard 🏂❄️
- making new friends

Why GitOps

GitOps brings the interface of our **distributed computers** to our place of **collaboration**

Nobody is logging into Kubernetes to talk to their teammates or make a decision.

GitOps gives us a staging area to work together and keep a central record for how the intellectual property of our businesses and projects evolve.

More Succinctly

1. write funny YAML of Deployment
2. `` `git commit -am 'yolo' && git push` ``
3. Pods go brrrrrr 🏎️ ☸️

Flux Controllers Architecture

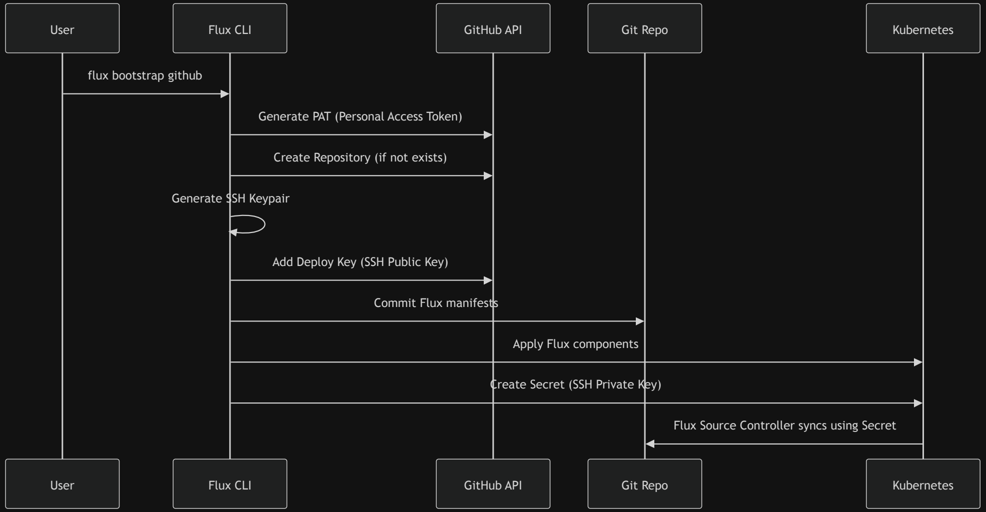
The Bootstrap Problem

How does Flux even get onto the cluster and start talking to Git?

1. Setup a Git Repo
2. Configure Access with some SSH Key or other auth
3. Install Flux onto your Kubernetes Cluster
4. Configure Flux with the appropriate Secrets and paths
5. Manage Flux somehow – maybe from the git repo itself?

Flux Bootstrap GitHub

```
flux bootstrap github \
  --owner=my-org \
  --repository=fleet-infra \
  --branch=main \
  --path=clusters/my-cluster \
  --personal
```


User
Flux CLI
GitHub API
Git Repo
Kubernetes
flux bootstrap github
Generate PAT (Personal Access Token)
Create Repository (if not exists)
Generate SSH Keypair
Add Deploy Key (SSH Public Key)
Commit Flux manifests
Apply Flux components
Create Secret (SSH Private Key)
Flux Source Controller syncs using Secret
User
Flux CLI
GitHub API
Git Repo
Kubernetes

Flux Operator

```yaml
apiVersion: fluxcd.controlplane.io/v1
kind: FluxInstance
metadata:
  name: flux
  namespace: flux-system
spec:
  distribution:
    version: "2.x"
    registry: "ghcr.io/fluxcd"
  components:
    - source-controller
    - kustomize-controller
    - helm-controller
    - notification-controller
  cluster:
    networkPolicy: true
    domain: "cluster.local"
  sync:
    kind: OCIRepository
    url: oci://ghcr.io/my-org/fleet-infra
    ref:
      tag: production-usa-west
```

Core Kubernetes SDKs

Flux controllers are built as native Kubernetes operators using the same libraries as Kubernetes itself:

- `` `k8s.io/client-go` ``: The primary Kubernetes Go client
- `` `k8s.io/api` `` & `` `k8s.io/apimachinery` ``: Core API types
- `` `sigs.k8s.io/controller-runtime` ``: manage reconciliations, workqueues, and caching.
- `` `sigs.k8s.io/kustomize/api` `` & `` `kyaml` ``: build YAML

Flux Core & Git Libraries

Flux is modular but shares powerful internal libraries:

- `` `github.com/fluxcd/pkg/...` ``: `` `runtime` ``, Server-Side Apply (`` `ssa` ``), caching, tarball creation, etc.
- `` `github.com/go-git/go-git/v5` ``: A highly extensible Git implementation in pure Go. It allows `` `source-controller` `` to securely clone repositories, manage auth, and stream commits into the cluster, without exec-ing a `` `git` `` shell command.

Helm SDK

`` `helm-controller` `` manages charts declaratively:

- `` `helm.sh/helm/v3` `` & `` `v4` ``: Uses the upstream Helm SDK instead of relying on a CLI binary. This allows Flux to observe the internal lifecycle of a Helm release, efficiently diff values, and implement unique features beyond the capabilities of the Helm CLI.
- Drift detection
- Separate Storage Namespace
- Fine-grained status reporting and reactivity
- Complex lifecycle & remediation
- Healthcheck Context Cancel (new in Flux 2.8)
- Efficient Client machinery

Cloud & OCI Ecosystem SDKs

Flux needs to talk to cloud providers and registries securely:

- **Cloud Provider SDKs:** `` `aws-sdk-go-v2` ``, `` `azure-sdk-for-go` ``, `` `googleapis` `` are embedded for seamless cloud-native authentication, fetching from S3/Blob storage, and KMS integrations.
- `` `google/go-containerregistry` ``: Used heavily across Flux components to interact with OCI registries, pull/push artifacts, and parse image credentials.
- `` `oras.land/oras-go/v2` ``: The OCI Registry As Storage (ORAS) SDK. Used to push and pull arbitrary files (like K8s manifests or Helm charts) as OCI artifacts.

Security & Supply Chain SDKs

Security is a primary focus for Flux:

- `` `getsops/sops/v3` ``: Embedded natively in the `` `kustomize-controller` `` to allow users to store encrypted secrets in Git (KMS, PGP, Age) and decrypt them during cluster reconciliation.
- `` `sigstore/cosign/v3` `` & `` `sigstore-go` ``: Integrated to verify the signatures of OCI artifacts (images, Helm charts, manifests) before applying them.
- `` `notaryproject/notation-go` ``: Integrated for verifying artifacts signed with Notary Project, giving users a choice of verification standards.
- `` `cyphar/filepath-securejoin` ``: Prevents path traversal and symlink escape attacks when reading manifests from Git.

Controller-Runtime & Our Opinions

Flux uses heavily Kubernetes `controller-runtime`.

- Managers
- Caches
- Clients
- rate limiters for errors

We use our `pkg` library to implement our own opinions on how to create kubernetes controllers and CLI’s in a consistent way across the project

Native Go SDKs vs Fork/Exec

Why doesn’t Flux just run `` `os.Exec("git pull")` `` or `` `os.Exec("helm upgrade")` ``?

- **Performance:** Forking processes is expensive.
- **Security:** Tighter control over execution context.
- **Error Handling:** Structured Go errors instead of parsing stdout/stderr.
- **Zombies:** Avoiding orphaned processes.

Workqueue, Storage, and Inventory

- **Workqueue:** Events trigger reconciles. Rate limiting keeps the Kube API Server happy.
- **Storage:** Artifacts are kept locally. `` `source-controller` `` serves tarballs via a local HTTP server.
- **Inventory:** Keeping track of exactly what Flux applied so we can garbage collect resources safely when they are removed from Git.

2. API Machinery

How Flux applies changes to Kubernetes

Server-Side Apply (SSA)

The heart of how Flux updates resources.

- **SSA vs Client-Side Apply:** SSA delegates the merge logic to the API server.
- **Benefits:**
  - Field ownership (Flux vs other controllers).
  - Built-in conflict resolution.
  - Smaller patch payloads.

With Flux 2.8, we now support SSA by default using the Helm v4 SDK

Controller Setup

Let’s take a tour of kustomize-controller :)

Multi-Tenancy & Impersonation

Flux uses Kubernetes Impersonation to ensure it only has the permissions of the tenant it is reconciling for. This is a core concept that allows for soft multi-tenancy.

```go
// kustomize-controller/internal/controller/kustomization_controller.go
// Configure the Kubernetes client for impersonation.
var impersonatorOpts []runtimeClient.ImpersonatorOption
var mustImpersonate bool
if r.DefaultServiceAccount != "" || obj.Spec.ServiceAccountName != "" {
  mustImpersonate = true
  impersonatorOpts = append(impersonatorOpts,
    runtimeClient.WithServiceAccount(r.DefaultServiceAccount, obj.Spec.ServiceAccountName, obj.GetNamespace()))
}
if obj.Spec.KubeConfig != nil {
  mustImpersonate = true
  provider := r.getProviderRESTConfigFetcher(obj)
  impersonatorOpts = append(impersonatorOpts,
    runtimeClient.WithKubeConfig(obj.Spec.KubeConfig, r.KubeConfigOpts, obj.GetNamespace(), provider))
}
if r.ClusterReader != nil || len(statusReaders) > 0 {
  impersonatorOpts = append(impersonatorOpts,
    runtimeClient.WithPolling(r.ClusterReader, statusReaders...))
}
impersonation := runtimeClient.NewImpersonator(r.Client, impersonatorOpts...)
```

Multi-Tenancy & Impersonation

Flux uses Kubernetes Impersonation to ensure it only has the permissions of the tenant it is reconciling for. This is a core concept that allows for soft multi-tenancy.

```go
// kustomize-controller/internal/controller/kustomization_controller.go
// Configure the Kubernetes client for impersonation.
var impersonatorOpts []runtimeClient.ImpersonatorOption
var mustImpersonate bool
if r.DefaultServiceAccount != "" || obj.Spec.ServiceAccountName != "" {
  mustImpersonate = true
  impersonatorOpts = append(impersonatorOpts,
    runtimeClient.WithServiceAccount(r.DefaultServiceAccount, obj.Spec.ServiceAccountName, obj.GetNamespace()))
}
if obj.Spec.KubeConfig != nil {
  mustImpersonate = true
  provider := r.getProviderRESTConfigFetcher(obj)
  impersonatorOpts = append(impersonatorOpts,
    runtimeClient.WithKubeConfig(obj.Spec.KubeConfig, r.KubeConfigOpts, obj.GetNamespace(), provider))
}
if r.ClusterReader != nil || len(statusReaders) > 0 {
  impersonatorOpts = append(impersonatorOpts,
    runtimeClient.WithPolling(r.ClusterReader, statusReaders...))
}
impersonation := runtimeClient.NewImpersonator(r.Client, impersonatorOpts...)
```

Multi-Tenancy & Impersonation

Flux uses Kubernetes Impersonation to ensure it only has the permissions of the tenant it is reconciling for. This is a core concept that allows for soft multi-tenancy.

```go
// kustomize-controller/internal/controller/kustomization_controller.go
// Configure the Kubernetes client for impersonation.
var impersonatorOpts []runtimeClient.ImpersonatorOption
var mustImpersonate bool
if r.DefaultServiceAccount != "" || obj.Spec.ServiceAccountName != "" {
  mustImpersonate = true
  impersonatorOpts = append(impersonatorOpts,
    runtimeClient.WithServiceAccount(r.DefaultServiceAccount, obj.Spec.ServiceAccountName, obj.GetNamespace()))
}
if obj.Spec.KubeConfig != nil {
  mustImpersonate = true
  provider := r.getProviderRESTConfigFetcher(obj)
  impersonatorOpts = append(impersonatorOpts,
    runtimeClient.WithKubeConfig(obj.Spec.KubeConfig, r.KubeConfigOpts, obj.GetNamespace(), provider))
}
if r.ClusterReader != nil || len(statusReaders) > 0 {
  impersonatorOpts = append(impersonatorOpts,
    runtimeClient.WithPolling(r.ClusterReader, statusReaders...))
}
impersonation := runtimeClient.NewImpersonator(r.Client, impersonatorOpts...)
```

Multi-Tenancy & Impersonation

Flux uses Kubernetes Impersonation to ensure it only has the permissions of the tenant it is reconciling for. This is a core concept that allows for soft multi-tenancy.

```go
// kustomize-controller/internal/controller/kustomization_controller.go
// Configure the Kubernetes client for impersonation.
var impersonatorOpts []runtimeClient.ImpersonatorOption
var mustImpersonate bool
if r.DefaultServiceAccount != "" || obj.Spec.ServiceAccountName != "" {
  mustImpersonate = true
  impersonatorOpts = append(impersonatorOpts,
    runtimeClient.WithServiceAccount(r.DefaultServiceAccount, obj.Spec.ServiceAccountName, obj.GetNamespace()))
}
if obj.Spec.KubeConfig != nil {
  mustImpersonate = true
  provider := r.getProviderRESTConfigFetcher(obj)
  impersonatorOpts = append(impersonatorOpts,
    runtimeClient.WithKubeConfig(obj.Spec.KubeConfig, r.KubeConfigOpts, obj.GetNamespace(), provider))
}
if r.ClusterReader != nil || len(statusReaders) > 0 {
  impersonatorOpts = append(impersonatorOpts,
    runtimeClient.WithPolling(r.ClusterReader, statusReaders...))
}
impersonation := runtimeClient.NewImpersonator(r.Client, impersonatorOpts...)
```

Resource Versions, Generations, and Drift Detection

- **Etcd Tracking:** How Kubernetes tracks changes to objects.
- **Generations:**
  - `` `.metadata.generation` ``: Incremented by the API server when the spec changes.
  - `` `.status.observedGeneration` ``: Updated by our controllers to signal "I have processed this version."
- **Drift Detection:** Flux calculates drift by using dry-run apply and checking if the server would make changes based on field ownership.

Lifecycle of Status Conditions and Events

- **Transitions:** `` `Reconciling` `` -> `` `Ready` `` (or `` `Error` ``).
- **Events:** Emitting Kubernetes Events so users (`` `kubectl get events` ``) know what is happening.

```
// Setting status conditions
conditions.MarkReconciling(obj, meta.ProgressingReason, "Reconciliation in progress")
// ... do work ...
conditions.MarkTrue(obj, meta.ReadyCondition, meta.SucceededReason, "Applied revision: %s", revision)
```

3. Observability & Performance

Keeping an eye on the controllers

Prometheus Metrics and Notification Controller

- **Metrics:** Queue depths, reconcile durations, apply errors, client-side API requests.
- **Notification Controller:** Translates GitOps events into Slack, Discord, or Webhook pings.
- We export state metrics so you can provide fast feedback to devs on their deployments and alert on stuck reconciliations.

Flux Operator’s Dashboard

- Visualizes Flux and the Kube API Server behavior.
- Makes it easy to digest the constant stream of GitOps events.
- Shows you the health of your `` `Kustomizations` `` and `` `HelmReleases` `` at a glance.

Tuning Options

We scale Flux on massive clusters with tens of thousands of Flux resources. Flux horizontally scales across thousands of clusters.

- Tune concurrent reconciles (`` `--concurrent=25` ``).
- `` `--requeue-dependency=10s` ``
- Scale CPU and RAM per controller
- `` `--min-retry-delay=2s` ``
- `` `--max-retry-delay=30m` ``
- **Sync intervals vs Webhooks:** Use Git webhooks to trigger immediate reconciles instead of polling every 1 minute.
- Label Configs and Secrets with `` `reconcile.fluxcd.io/watch=Enabled` ``
- `` `--feature-gates=CancelHealthCheckOnNewRevision` ``

Flux defaults were chosen to work on Raspberry Pi’s. Flux Operator has improved defaults and T-Shirt sizing for enterprise use-cases.

Supply Chain Provenance

SLSA Build Level 3 & Artifact Signing

Flux takes its own supply chain very seriously:

- **Signed Container Images:** All Flux CLI and controller images are signed using Sigstore Cosign and GitHub OIDC.
- **SBOMs:** A Software Bill of Materials (SPDX format via Syft) is published with each release and embedded in images.
- **SLSA Level 3:** The build, release, and provenance portions of the Flux supply chain meet SLSA Build Level 3.
- **Buildkit Attestations:** Provenance attestations (SLSA schema v0.2) include build timestamps, environment metadata, and source code details.

Kubernetes Security Posture

Pod Standards & RBAC

Flux controllers adhere strictly to Kubernetes security best practices:

- **Restricted Pod Security Standard:** All Linux capabilities dropped, read-only root FS, run as non-root (UID 65534), and runtime default seccomp profiles.
- **Explicit RBAC:** Separation of concerns. Only `kustomize-controller` and `helm-controller` are granted `cluster-admin` by default, as they manage cluster state.
- **Soft Multi-Tenancy:** In multi-tenant environments, Flux uses the Kubernetes Impersonation API. It reconciles tenant repos under specific service accounts, enforcing tenant RBAC policies.
- **Cross-Namespace Policies:** Strict flags (`--no-cross-namespace-refs`) to prevent cross-namespace reference exploits for sensitive data like Secrets.

Building Your Own GitOps Tools

The GitOps Toolkit (gotk)

Flux is built on a set of APIs and controllers called the **GitOps Toolkit**. You can use these packages to extend Flux!

- **Learn more at:** fluxcd.io/flux/gitops-toolkit
- **Custom Source Types:** Want to fetch some YAML from Postgres? Implement the `` `ExternalArtifact` `` API and write your own custom Source Controller.
- **Custom Appliers:** Want to deploy resources with Terraform or Ansible? Consume Flux's `` `Artifact` `` API and apply them using the `` `pkg/runtime` `` and `` `pkg/ssa` `` libraries.
- The GitOps Toolkit allows anyone to build controllers in the Flux style with consistent observability and mechanics.

4. Flux in the Future

Flux in the Future

1. Flux Operator `` `ResourceSet` `` templates with dynamic inputs
2. Flux UI
3. Gitless GitOps
4. Passwordless Machine Identity
5. Agentic GitOps using the Flux Operator MCP

Thank You!

Questions?

- Find me on GitHub: @stealthybox
- Flux Documentation: fluxcd.io
- CNCF Slack: `` `#flux` ``