Flux Documentation

Flux development team

Feb 28, 2020

FLUX

CHAPTER

ONE

INTRODUCING FLUX

Continuous delivery is a term that encapsulates a set of best practices that surround building, deploying and monitoring applications. The goal is to provide a sustainable model for maintaining and improving an application.

Flux is a tool that automates the deployment of containers to Kubernetes. It fills the automation void that exists between building and monitoring.

1.1 Automated git->cluster synchronisation

Flux's main feature is the automated synchronisation between a version control repository and a cluster. If you make any changes to your repository, those changes are automatically deployed to your cluster.

This is a simple, but dramatic improvement on current state of the art.

- All configuration is stored within version control and is inherently up to date. At any point anyone could completely recreate the cluster in exactly the same state of configuration.
- Changes to the cluster are immediately visible to all interested parties.
- During a postmortem, the git log provides the perfect history for an audit.
- End to end, code to production pipelines become not only possible, but easy.

1.2 Automated deployment of new container images

Another feature is the automated deployment of containers. It will continuously monitor a range of container registries and deploy new versions where applicable.

This is really useful for keeping the repository and therefore the cluster up to date. It allows separate teams to have their own deployment pipelines since Flux is able to see the new image and update the cluster accordingly.

This feature can be disabled and images can be locked to a specific version.

1.3 Integrations with other devops tools

For configuration customization across environments and clusters, Flux comes with builtin support for *Kustomize* and *Helm*.

For advanced deployment patterns like Canary releases, A/B testing and Blue/Green deployments, Flux can be used together with Flagger.

CHAPTER

TWO

REQUIREMENTS AND LIMITATIONS

Flux has some requirements of the files it finds in your git repo.

- Flux can only deal with one such repo at a time. This limitation is technical and may go away.
- Flux only deals with YAML files at present. It tries to preserve comments and whitespace in YAMLs when updating them. You may see updates with incidental, harmless changes, like reindented blocks.
- All Kubernetes resource manifests should explicitly specify the namespace in which you want them to run. Otherwise, the conventional default (`"default"`) will be assumed.
- Flux will ignore directories that look like Helm charts, to avoid applying templated YAML manifests. A directory will be skipped if its contents include the files `Chart.yaml` and `values.yaml`, as these are the (only) mandatory components of a Helm chart.

It is *not* a requirement that the files are arranged in any particular way into directories. Flux will look in subdirectories for YAML files recursively, but does not infer any meaning from the directory structure.

Flux uses the Docker Registry API to collect metadata about the images running in the cluster. This comes with at least one limitation:

- Since Flux runs in a container in your cluster, it may not be able to resolve all hostnames that you or Kubernetes can resolve. In particular, it won't be able to get image metadata for images in a private image registry that's made available at `localhost`.

CHAPTER

THREE

GET STARTED

All you need is a Kubernetes cluster and a git repo. The git repo contains manifests (as YAML files) describing what should run in the cluster. Flux imposes *some requirements* on these files.

3.1 Installing Flux

Here are the instructions to *install Flux on your own cluster*.

If you are using Helm, we have a *separate section about this*.

CHAPTER

FOUR

REFERENCES

4.1 Blueprint

This page describes the goals of Flux, how it achieves them and significant architectural decisions. It is intentionally high level to prevent it from being out of date too quickly.

4.1.1 Flux's goals

The overall goal of Flux is to automate the deployment of services. A typical use case would be:

1. A developer makes changes
2. An operational cluster is now out of date and needs to be updated
3. Flux observes those changes and deploys them to the cluster
4. Flux maintains the current state of the cluster (e.g. in the event of failure)

Hence, the goal is to automate away the need for a developer to interact with an orchestrator (which is a common source of accidental failure) or with the systems that ensure that the orchestrator is in a working state.

Flux provides a CLI (`fluxctl`) to perform these operations manually. Flux is flexible enough to fit into any development process.

4.1.2 Implementation overview

The following describes how Flux achieves the goals.

Synchronisation of cluster state

The Flux team firmly believe that cluster state should be version controlled. This allows users to record the history of the cluster, fallback to previous versions and recreate clusters in exactly the same state when required.

But there is also another aspect. By tightly integrating the cluster with version control, the cluster becomes more tightly integrated with the deployment pipeline. This means that developers no longer have to interact directly with a cluster (with the inevitable consequences of a "fat-finger" mistake) which makes it far more stable and ideally immutable.

Flux achieves this by automatically synchronising the state of the cluster to match the code representing the cluster in the repository.

This simple idea then allows for a whole range of tools that can react to changes and simply write to a repository.

Monitoring for new images

Flux reads a list of running containers from the user git repository. For each image, it will query the container registry to obtain the most recently released tag.

Flux then compares the most recent image tag with that specified in the git repository. If they don't match, the repository is updated.

When services are in an "automated" mode, the service will periodically check to see whether there are any new images. If there are, then they are written to the repository.

When automation is disabled, images are not checked.

In order to access private registries, credentials may be required.

Deployment of images

Flux will only deploy different images. It will not re-deploy images with the same tag.

Once a list of new images have been established, it will alter the configuration of the cluster to deploy the new images.

Images can be "locked" to a specific version. "locked" images won't be updated by automated or manual means.

4.2 Daemon (`fluxd`)

4.2.1 Summary

Flux daemon (`fluxd`, aka Flux agent) allows automation of application deployments and version control of cluster configuration. Version controlling of cluster manifests provides reproducibility and a historical trail of events.

Responsibilities

Continuous Deployment

1. Flux daemon monitors user git repo Kubernetes manifests for changes, which it then deploys to the cluster.
2. Flux daemon monitors container registry for running container image updates. Detection of an image change (running container image tag vs container registry image tag) triggers k8s manifest update, which is committed to the user git repository, then deployed to the Kubernetes cluster.

Deployment approaches

1. Automate vs Deautomate

   Deployment happens automatically when a new image tag is detected. Deautomated deployment will not proceed until manually released (through the CLI tool `fluxctl`).

2. Lock vs Unlock

   Deployment is pinned to a particular image tag. New deployment will not proceed upon triggered release.

4.2.2 Setup and configuration

`fluxd` requires setup and offers customization though a multitude of flags.

4.2.3 More information

Setting up and configuring `fluxd` is discussed in *"Get started with Flux"*.

There is also more information on garbage collection, Git commit signing, and other elements in *references*.

4.3 `fluxctl`

`fluxctl` provides an API that can be used from the command line.

The `--help` for `fluxctl` is described below.

4.3.1 Installing `fluxctl`

Mac OS

If you are using a Mac and use Homebrew, you can simply run:

```
brew install fluxctl
```

Linux

Ubuntu (and others): snaps

Many Linux distributions support snaps these days, which makes it very easy to install `fluxctl` and stay up to date.

To install it, simply run:

```
sudo snap install fluxctl
```

If you would prefer to track builds from master, run

```
sudo snap install fluxctl --edge
```

instead.

Arch Linux

Install the `fluxctl-bin` package from the AUR:

```
git clone https://aur.archlinux.org/fluxctl-bin.git
cd fluxctl-bin
makepkg -si
```

Windows

Chocolatey

Chocolatey is a third party package manager for Windows.

If you haven't already installed chocolatey you will need to do this first.

fluxctl can then be installed from the public package repository:

```
choco install fluxctl
```

Binary releases

With every release of Flux, we release binaries of `fluxctl` for Mac, Linux and Windows. Download them from the Flux release page.

4.3.2 Connecting `fluxctl` to the daemon

By default, `fluxctl` will attempt to port-forward to your Flux instance, assuming it runs in the `"default"` namespace. You can specify a different namespace with the `--k8s-fwd-ns` flag:

```
fluxctl --k8s-fwd-ns=weave list-workloads
```

The namespace can also be given in the environment variable `FLUX_FORWARD_NAMESPACE`:

```
export FLUX_FORWARD_NAMESPACE=weave
fluxctl list-workloads
```

If you are not able to use the port forward to connect, you will need some way of connecting to the Flux API directly (NodePort, LoadBalancer, VPN, etc). **Be aware that exposing the Flux API in this way is a security hole, because it can be accessed without authentication.**

Once that is set up, you can specify an API URL with `--url` or the environment variable `FLUX_URL`:

```
fluxctl --url http://127.0.0.1:3030/api/flux list-workloads
```

Flux API service

Now you can easily query the Flux API:

```
fluxctl list-workloads --all-namespaces
```

Add an SSH deploy key to the repository

Flux connects to the repository using an SSH key. You have two options:

1. Allow Flux to generate a key for you

If you don't specify a key to use, Flux will create one for you. Obtain the public key through `fluxctl`:

```
$ fluxctl identity
ssh-rsa␣
↪AAAAB3NzaC1yc2EAAAADAQABAAAAgQDCN2ECqUFMR413CURbLBcG41fLY75SfVZCd3LCsJBClVlEcMk4lwXxA3X4jowpv2v4Jw2
↪/+ov0qGt/uRuexMN7WUx6c93VFGV7Pjd60Yilb6GSF8B39iEVq7GQUC1OZRgQnKZWLSQ==
```

Alternatively, you can see the public key in the `flux` log.

The public key will need to be given to the service hosting the Git repository. For example, in GitHub you would create an SSH deploy key in the repository, supplying that public key.

The `flux` logs should show that it has now connected to the repository and synchronised the cluster.

When using Kubernetes, this key is stored as a Kubernetes secret. You can restart `flux` and it will continue to use the same key.

2. Specify a key to use

Create a Kubernetes Secret from a private key:

```
kubectl create secret generic flux-git-deploy --from-file=identity=/full/path/to/
↪private_key
```

this will result in a secret that has the structure:

```
  apiVersion: v1
  data:
    identity: <base64 encoded RSA PRIVATE KEY>
  kind: Secret
  type: Opaque
  metadata:
    ...
```

Now add the secret to the `flux-deployment.yaml` manifest:

```
    ...
    spec:
      volumes:
      - name: git-key
        secret:
          secretName: flux-git-deploy
```

And add a volume mount for the container:

```
    ...
    spec:
      containers:
      - name: fluxd
        volumeMounts:
```

(continues on next page)

(continued from previous page)

```
        - name: git-key
          mountPath: /etc/fluxd/ssh
```

You can customise the paths and names of the chosen key with the arguments (examples with defaults): `--k8s-secret-name=flux-git-deploy`, `--k8s-secret-volume-mount-path=/etc/fluxd/ssh` and `--k8s-secret-data-key=identity`

Using an SSH key allows you to maintain control of the repository. You can revoke permission for `flux` to access the repository at any time by removing the deploy key.

```
fluxctl helps you deploy your code.

Connecting:

  # To a fluxd running in namespace "default" in your current kubectl context
  fluxctl list-workloads

  # To a fluxd running in namespace "weave" in your current kubectl context
  fluxctl --k8s-fwd-ns=weave list-workloads

  # To a Weave Cloud instance, with your instance token in $TOKEN
  fluxctl --token $TOKEN list-workloads

Workflow:
  fluxctl list-workloads                                                # Which␣
↪workloads are running?
  fluxctl list-images --workload=default:deployment/foo                 # Which␣
↪images are running/available?
  fluxctl release --workload=default:deployment/foo --update-image=bar:v2  # Release␣
↪new version.

Usage:
  fluxctl [command]

Available Commands:
  automate       Turn on automatic deployment for a workload.
  deautomate     Turn off automatic deployment for a workload.
  help           Help about any command
  identity       Display SSH public key
  install        Print and tweak Kubernetes manifests needed to install Flux in a␣
↪Cluster
  list-images    Show deployed and available images.
  list-workloads List workloads currently running in the cluster.
  lock           Lock a workload, so it cannot be deployed.
  policy         Manage policies for a workload.
  release        Release a new version of a workload.
  save           save workload definitions to local files in cluster-native format
  sync           synchronize the cluster with the git repository, now
  unlock         Unlock a workload, so it can be deployed.
  version        Output the version of fluxctl

Flags:
      --context string                The kubeconfig context to use
  -h, --help                          help for fluxctl
      --k8s-fwd-labels stringToString   Labels used to select the fluxd pod a port␣
↪forward should be created for. You can also set the environment variable FLUX_
↪FORWARD_LABELS (default [app=flux])
```

(continues on next page)

(continued from previous page)

```
      --k8s-fwd-ns string               Namespace in which fluxd is running, for␣
↪creating a port forward to access the API. No port forward will be created if a URL␣
↪or token is given. You can also set the environment variable FLUX_FORWARD_NAMESPACE␣
↪(default "default")
      --timeout duration                Global command timeout; you can also set the␣
↪environment variable FLUX_TIMEOUT (default 1m0s)
  -t, --token string                    Weave Cloud authentication token; you can␣
↪also set the environment variable WEAVE_CLOUD_TOKEN or FLUX_SERVICE_TOKEN
  -u, --url string                      Base URL of the Flux API (defaults to "https:/
↪/cloud.weave.works/api/flux" if a token is provided); you can also set the␣
↪environment variable FLUX_URL

Use "fluxctl [command] --help" for more information about a command.
```

Using `fluxctl install`

Installs Flux into your cluster, taking as input your Git details and namespace you want to target.

Example:

```
fluxctl install --git-url 'git@github.com:<your username>/flux-get-started' | kubectl␣
↪-f -
```

See here for a full tutorial which makes use of `fluxctl install`.

4.3.3 Workloads

What is a Workload?

This term refers to any cluster resource responsible for the creation of containers from versioned images - in Kubernetes these are objects such as Deployments, DaemonSets, StatefulSets and CronJobs.

Viewing Workloads

The first thing to do is to check whether Flux can see any running workloads. To do this, use the `list-workloads` subcommand:

```
$ fluxctl list-workloads
WORKLOAD                         CONTAINER   IMAGE                                    ␣
↪   RELEASE  POLICY
default:deployment/helloworld    helloworld  quay.io/weaveworks/helloworld:master-
↪a000001  ready
                                 sidecar     quay.io/weaveworks/sidecar:master-a000002
default:deployment/busybox       busybox     busybox:1.31.1                           ␣
↪   ready
default:deployment/nginx         nginx       nginx:stable-alpine                      ␣
↪   ready
```

Note that the actual images running will depend on your cluster.

You can also filter workloads by container name, using the `--container|-c` option:

```
$ fluxctl list-workloads --container helloworld
WORKLOAD                        CONTAINER   IMAGE                                                          ␣
↪   RELEASE  POLICY
default:deployment/helloworld  helloworld  quay.io/weaveworks/helloworld:master-
↪a000001  ready
                               sidecar     quay.io/weaveworks/sidecar:master-a000002
```

Inspecting the Version of a Container

Once we have a list of workloads, we can begin to inspect which versions of the image are running.

```
$ fluxctl list-images --workload default:deployment/helloworld
WORKLOAD                        CONTAINER   IMAGE                          CREATED
default:deployment/helloworld  helloworld  quay.io/weaveworks/helloworld
                                            |   master-9a16ff945b9e        20 Jul 16␣
↪13:19 UTC
                                            |   master-b31c617a0fe3        20 Jul 16␣
↪13:19 UTC
                                            |   master-a000002             12 Jul 16␣
↪17:17 UTC
                                            '-> master-a000001             12 Jul 16␣
↪17:16 UTC
                               sidecar     quay.io/weaveworks/sidecar
                                            '-> master-a000002             23 Aug 16␣
↪10:05 UTC
                                                master-a000001             23 Aug 16␣
↪09:53 UTC
```

The arrows will point to the version that is currently running alongside a list of other versions and their timestamps.

When using `fluxctl` in scripts, you can remove the table headers with `--no-headers` for both `list-images` and `list-workloads` command to suppress the header:

```
$ fluxctl list-workloads --no-headers
default:deployment/helloworld  helloworld  quay.io/weaveworks/helloworld:master-
↪a000001  ready
                               sidecar     quay.io/weaveworks/sidecar:master-a000002
$ fluxctl list-images --workload default:deployment/helloworld --no-headers
default:deployment/helloworld  helloworld  quay.io/weaveworks/helloworld
```

Releasing a Workload

We can now go ahead and update a workload with the `release` subcommand. This will check whether each workload needs to be updated, and if so, write the new configuration to the repository.

```
$ fluxctl release --workload=default:deployment/helloworld --user=phil --message="New␣
↪version" --update-all-images
Submitting release ...
Commit pushed: 7dc025c
Applied 7dc025c61fdbbfc2c32f792ad61e6ff52cf0590a
WORKLOAD                       STATUS   UPDATES
default:deployment/helloworld  success  helloworld: quay.io/weaveworks/
↪helloworld:master-a000001 -> master-9a16ff945b9e
```

(continues on next page)

(continued from previous page)

```
$ fluxctl list-images --workload default:deployment/helloworld
WORKLOAD                         CONTAINER   IMAGE                            CREATED
default:deployment/helloworld    helloworld  quay.io/weaveworks/helloworld
                                             '-> master-9a16ff945b9e          20 Jul 16␣
↪13:19 UTC
                                                 master-b31c617a0fe3          20 Jul 16␣
↪13:19 UTC
                                                 master-a000002               12 Jul 16␣
↪17:17 UTC
                                                 master-a000001               12 Jul 16␣
↪17:16 UTC
                                 sidecar     quay.io/weaveworks/sidecar
                                             '-> master-a000002               23 Aug 16␣
↪10:05 UTC
                                                 master-a000001               23 Aug 16␣
↪09:53 UTC
```

Turning on Automation

Automation can be easily controlled from `fluxctl` with the `automate` subcommand.

```
$ fluxctl automate --workload=default:deployment/helloworld
Commit pushed: af4bf73
WORKLOAD                       STATUS   UPDATES
default:deployment/helloworld  success

$ fluxctl list-workloads --namespace=default
WORKLOAD                         CONTAINER   IMAGE                                  ␣
↪       RELEASE  POLICY
default:deployment/helloworld  helloworld  quay.io/weaveworks/helloworld:master-
↪9a16ff945b9e ready    automated
                                 sidecar     quay.io/weaveworks/sidecar:master-a000002
```

Automation can also be enabled by adding the annotation `fluxcd.io/automated: "true"` to the deployment.

We can see that the `list-workloads` subcommand reports that the helloworld application is automated. Flux will now automatically deploy a new version of a workload whenever one is available and commit the new configuration to the version control system.

Turning off Automation

Turning off automation is performed with the `deautomate` command:

```
$ fluxctl deautomate --workload=default:deployment/helloworld
Commit pushed: a54ef2c
WORKLOAD                       STATUS   UPDATES
default:deployment/helloworld  success

$ fluxctl list-workloads --namespace=default
WORKLOAD                         CONTAINER   IMAGE                                  ␣
↪       RELEASE  POLICY
default:deployment/helloworld  helloworld  quay.io/weaveworks/helloworld:master-
↪9a16ff945b9e ready
```

(continues on next page)

(continued from previous page)

```
                                sidecar     quay.io/weaveworks/sidecar:master-a000002
```

We can see that the workload is no longer automated.

Rolling back a Workload

Rolling back can be achieved by combining:

- *deautomate* to prevent Flux from automatically updating to newer versions, and
- *release* to deploy the version you want to roll back to.

```
$ fluxctl list-images --workload default:deployment/helloworld
WORKLOAD                        CONTAINER   IMAGE                           CREATED
default:deployment/helloworld   helloworld  quay.io/weaveworks/helloworld
                                            '-> master-9a16ff945b9e         20 Jul 16␣
↪13:19 UTC
                                                master-b31c617a0fe3         20 Jul 16␣
↪13:19 UTC
                                                master-a000002              12 Jul 16␣
↪17:17 UTC
                                                master-a000001              12 Jul 16␣
↪17:16 UTC
                                sidecar     quay.io/weaveworks/sidecar
                                            '-> master-a000002              23 Aug 16␣
↪10:05 UTC
                                                master-a000001              23 Aug 16␣
↪09:53 UTC

$ fluxctl deautomate --workload=default:deployment/helloworld
Commit pushed: c07f317
WORKLOAD                        STATUS   UPDATES
default:deployment/helloworld   success

$ fluxctl release --workload=default:deployment/helloworld --update-image=quay.io/
↪weaveworks/helloworld:master-a000001
Submitting release ...
Commit pushed: 33ce4e3
Applied 33ce4e38048f4b787c583e64505485a13c8a7836
WORKLOAD                        STATUS   UPDATES
default:deployment/helloworld   success  helloworld: quay.io/weaveworks/
↪helloworld:master-9a16ff945b9e -> master-a000001

$ fluxctl list-images --workload default:deployment/helloworld
WORKLOAD                        CONTAINER   IMAGE                           CREATED
default:deployment/helloworld   helloworld  quay.io/weaveworks/helloworld
                                            |   master-9a16ff945b9e         20 Jul 16␣
↪13:19 UTC
                                            |   master-b31c617a0fe3         20 Jul 16␣
↪13:19 UTC
                                            |   master-a000002              12 Jul 16␣
↪17:17 UTC
                                            '-> master-a000001              12 Jul 16␣
↪17:16 UTC
                                sidecar     quay.io/weaveworks/sidecar
```

(continues on next page)

(continued from previous page)

```
                                        '-> master-a000002              23 Aug 16␣
↪10:05 UTC
                                            master-a000001              23 Aug 16␣
↪09:53 UTC
```

Locking a Workload

Locking a workload will stop manual or automated releases to that workload. Changes made in the file will still be synced.

```
$ fluxctl lock --workload=deployment/helloworld
Commit pushed: d726722
WORKLOAD                        STATUS   UPDATES
default:deployment/helloworld   success
```

Releasing an image to a locked workload

It may be desirable to release an image to a locked workload while maintaining the lock afterwards. In order to not having to modify the lock policy (which includes author and reason), one may use `--force`:

```
fluxctl release --workload=default:deployment/helloworld --update-all-images --force
```

Unlocking a Workload

Unlocking a workload allows it to have manual or automated releases (again).

```
$ fluxctl unlock --workload=deployment/helloworld
Commit pushed: 708b63a
WORKLOAD                        STATUS   UPDATES
default:deployment/helloworld   success
```

Recording user and message with the triggered action

Issuing a deployment change results in a version control change/git commit, keeping the history of the actions. The Flux daemon can be started with several flags that impact the commit information:

Actions triggered by a user through the CLI `fluxctl` tool, can have the commit author information customized. This is handy for providing extra context in the notifications and history. Whether the customization is possible, depends on the Flux daemon (`fluxd`) `git-set-author` flag. If set, the commit author will be customized in the following way:

4.3.4 Image Tag Filtering

When building images it is often useful to tag build images by the branch that they were built against for example:

```
quay.io/weaveworks/helloworld:master-9a16ff945b9e
```

Indicates that the `helloworld` image was built against master commit `9a16ff945b9e`.

When automation is turned on Flux will, by default, use whatever is the latest image on a given repository. If you want to only auto-update your image against a certain subset of tags then you can do that using tag filtering.

So for example, if you want to only update the "helloworld" image to tags that were built against the "prod" branch then you could do the following:

```
fluxctl policy --workload=default:deployment/helloworld --tag-all='prod-*'
```

If your pod contains multiple containers then you tag each container individually:

```
fluxctl policy --workload=default:deployment/helloworld --tag='helloworld=prod-*' --
↪tag='sidecar=prod-*'
```

Manual releases without explicit mention of the target image will also adhere to tag filters. This will only release the newest image matching the tag filter:

```
fluxctl release --workload=default:deployment/helloworld --update-all-images
```

To release an image outside of tag filters, either specify the image:

```
fluxctl release --workload=default:deployment/helloworld --update-
↪image=helloworld:dev-abc123
```

or use `--force`:

```
fluxctl release --workload=default:deployment/helloworld --update-all-images --force
```

Please note that automation might immediately undo this.

Filter pattern types

Flux currently offers support for `glob`, `semver` and `regexp` based filtering.

Glob

The glob (`*`) filter is the simplest filter Flux supports, a filter can contain multiple globs:

```
fluxctl policy --workload=default:deployment/helloworld --tag-all='glob:master-v1.*.*'
```

Semver

If your images use semantic versioning you can filter by image tags that adhere to certain constraints:

```
fluxctl policy --workload=default:deployment/helloworld --tag-all='semver:~1'
```

or only release images that have a stable semantic version tag (X.Y.Z):

```
fluxctl policy --workload=default:deployment/helloworld --tag-all='semver:*'
```

Using a semver filter will also affect how Flux sorts images, so that the higher versions will be considered newer.

Semver has a concept of "pre-release" versions which have an extra label like `-beta` at the end. If you want to include these then write a policy with a hyphen; for example `>=1.2.3` will skip prereleases while `>=1.2.3-0` will include prereleases.

Regexp

If your images have complex tags you can filter by regular expression:

```
fluxctl policy --workload=default:deployment/helloworld --tag-all='regexp:^([a-zA-Z]+)
↪$'
```

Instead of `regexp` it is also possible to use its alias `regex`. Please bear in mind that if you want to match the whole tag, you must bookend your pattern with `^` and `$`.

Controlling image timestamps with labels

Some image registries do not expose a reliable creation timestamp for image tags, which could pose a problem for the automated roll-out of images.

To overcome this problem you can define one of the supported labels in your `Dockerfile`. Flux will prioritize labels over the timestamp it retrieves from the registry.

Supported label formats

- `org.opencontainers.image.created` date and time on which the image was built (string, date-time as defined by RFC 3339).
- `org.label-schema.build-date` date and time on which the image was built (string, date-time as defined by RFC 3339).

4.3.5 Actions triggered through `fluxctl`

`fluxctl` provides the following flags for the message and author customization:

```
  -m, --message string      attach a message to the update
      --user    string      override the user reported as initiating the update
```

Commit customization

1. Commit message

```
fluxctl --message="Message providing more context for the action" .....
```

2. Committer

   Committer information can be overriden with the appropriate fluxd flags:

```
--git-user
--git-email
```

   See daemon.md for more information.

3. Commit author

   The default for the author is the committer information, which can be overriden, in the following manner:

   a) Default override uses user's git configuration, ie `user.name` and `user.email` (.gitconfig) to set the commit author. If the user has neither user.name nor for user.email set up, the committer information will be used. If only one is set up, that will be used.

   b) This can be further overriden by the use of the `fluxctl --user` flag.

Examples

1. `fluxctl --user="Jane Doe <jane@doe.com>" .....` This will always succeed as git expects a new author in the format "some_string <some_other_string>".
2. `fluxctl --user="Jane Doe" .....` This form will succeed if there is already a repo commit, done by Jane Doe.
3. `fluxctl --user="jane@doe.com" .....` This form will succeed if there is already a repo commit, done by jane@doe.com.

Errors due to author customization

In case of no prior commit by the specified author, an error will be reported for 2) and 3):

```
git commit: fatal: --author 'unknown' is not 'Name <email>' and matches
no existing author
```

4.3.6 Using Annotations

Automation and image tag filtering can also be managed using annotations (`fluxctl` is using the same mechanism).

Automation can be enabled with `fluxcd.io/automated: "true"`. Image filtering annotations take the form `fluxcd.io/tag.<container-name>: <filter-type>:<filter-value>` or `filter.fluxcd.io/<container-name>: <filter-type>:<filter-value>`. Values of `filter-type` can be *glob*, *semver*, and *regexp*. Filter values use the same syntax as when the filter is configured using `fluxctl`.

Here's a simple but complete deployment file with annotations:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: podinfo
  namespace: demo
  labels:
    app: podinfo
  annotations:
    fluxcd.io/automated: "true"
    fluxcd.io/tag.podinfod: semver:~1.3
spec:
  selector:
    matchLabels:
      app: podinfo
  template:
    metadata:
      labels:
        app: podinfo
    spec:
      containers:
      - name: podinfod
        image: stefanprodan/podinfo:1.3.2
        ports:
        - containerPort: 9898
          name: http
        command:
        - ./podinfo
        - --port=9898
```

Things to notice:

1. The annotations are made in `metadata.annotations`, not in `spec.template.metadata`.
2. The `fluxcd.io/tag....` references the container name `podinfod`, this will change based on your container name. If you have multiple containers you would have multiple lines like that.
3. The value for the `fluxcd.io/tag....` annotation should includes the filter pattern type, in this case `semver`.

Annotations can also be used to tell Flux to temporarily ignore certain manifests using `fluxcd.io/ignore: "true"`. Read more about this in the *FAQ*.

4.4 Manifest generation through `.flux.yaml` configuration files

This feature lets you generate Kubernetes manifests with a program, instead of having to include them in your git repo as YAML files. For example, you can use `kustomize` to patch a common set of resources to suit a particular environment.

> Note: For a full, self-contained example of Flux generating manifests with `kustomize` you can go to https://github.com/fluxcd/flux-kustomize-example

Manifest generation is controlled by the flags given to `fluxd`, and `.flux.yaml` files in your git repo.

To enable it, you will need to

1. pass the command-line flag `--manifest-generation=true` to `fluxd`.
2. put at least one `.flux.yaml` file in the git repository.

Where to put `.flux.yaml`, and what should be in it, are described in the sections following.

4.4.1 Setting manifest generation up

The command-line flag `--git-path` (which can be given multiple values) marks a "target path" within the git repository in which to find manifests. If `--git-path` is not supplied, the top of the git repository is assumed to be the sole target path.

Without manifest generation, fluxd will recursively walk the directories under each target path, to look for YAML files.

With manifest generation **enabled**, fluxd will look for processing instructions in a file `.flux.yaml`, which can be located *at* the target path, or in a directory *above* it in the git repository.

- if a `.flux.yaml` file is found, it is used **instead** of looking for YAML files, and no other files are examined for that target path;
- if no `.flux.yaml` file is found, the usual behaviour of looking for YAML files is adopted for that target path.
- a `.flux.yaml` file containing the `scanForFiles` directive resets the behaviour to looking for YAML files. This is explained below.

The manifests from all the target paths – read from YAML files or generated – are combined before applying to the cluster. If duplicates are detected, an error is logged and fluxd will abandon the attempt to apply manifests to the cluster.

Here are some examples:

```
.
├── base
│   ├── demo-ns.yaml
│   ├── kustomization.yaml
│   ├── podinfo-dep.yaml
│   ├── podinfo-hpa.yaml
│   └── podinfo-svc.yaml
├── .flux.yaml
├── production
│   ├── flux-patch.yaml
│   ├── kustomization.yaml
│   └── replicas-patch.yaml
└── staging
    ├── flux-patch.yaml
    └── kustomization.yaml
```

In this case, say you started `fluxd` with `--git-path=staging`, it would find `.flux.yaml` in the top directory and use that to generate manifests. The other files and directories (if there were any) in `staging/` are not examined by fluxd, in favour of following the instructions given in the `.flux.yaml` file.

This layout could also be used with `--git-path=production`.

In this modified example, the `.flux.yaml` file has been moved under `staging/`:

```
.
├── base
│   ├── demo-ns.yaml
│   ├── kustomization.yaml
│   ├── podinfo-dep.yaml
│   ├── podinfo-hpa.yaml
│   └── podinfo-svc.yaml
├── production
```

(continues on next page)

(continued from previous page)

```
    ├── flux-patch.yaml
    ├── kustomization.yaml
    └── replicas-patch.yaml
└── staging
    ├── flux-patch.yaml
    ├── .flux.yaml
    └── kustomization.yaml
```

... since the `.flux.yaml` file is now under `staging/`, it will still take effect for `--git-path=staging`. However:

Using `--git-path=production` would **not produce a usable configuration**, because without an applicable `.flux.yaml`, the files under `production/` would be treated as plain Kubernetes manifests, which they are plainly not.

Note also that the configuration file would **not** take effect for `--git-path=.` (i.e., the top directory), because manifest generation will not look in subdirectories for a `.flux.yaml` file.

The `scanForFiles` directive

The `scanForFiles` directive indicates that the target path should be treated as though it had *no* `.flux.yaml` in effect. In other words, fluxd will look for YAML files under the directory, and update manifests directly by rewriting the YAML files.

Here's an example `.flux.yaml` with the `scanForFiles` directive:

```
version: 1
scanForFiles: {}
```

(The `{}` is an empty map, which acts as a placeholder value).

This is to account for the case in which you have a `.flux.yaml` higher in the directory tree, applying to several target paths beneath it, but want to have a directory wth regular YAMLs as well.

In the following example, the top-level `.flux.yaml` would take effect for `--git-path=staging` or `--git-path=production`.

But if you wanted `yamls/permissions.yaml` to be applied (as it is), you could put a `.flux.yaml` containing `scanForFiles` in that directory, and specify `--git-path=staging,yamls`.

```
.
├── .flux.yaml
├── base
│   ├── demo-ns.yaml
│   ├── kustomization.yaml
│   ├── podinfo-dep.yaml
│   ├── podinfo-hpa.yaml
│   └── podinfo-svc.yaml
├── production
│   ├── flux-patch.yaml
│   ├── kustomization.yaml
│   └── replicas-patch.yaml
├── yamls
│   ├── .flux.yaml # (with "scanForFiles" directive)
│   └── permissions.yaml
└── staging
```

(continues on next page)

(continued from previous page)

```
    ├── flux-patch.yaml
    └── kustomization.yaml
```

4.4.2 How to construct a .flux.yaml file

Aside from the special case of the `scanForFiles` directive, `.flux.yaml` files come in two varieties: "patch-updated", "command-updated". These refer to the way in which automated updates are applied to files in the repo:

- when patch-updated, fluxd will keep updates in its own patch file, which it applies to the generated manifests before applying to the cluster;
- when command-updated, you must supply commands to update the appropriate file or files.

Patch-updated will work with any kind of manifest generation, because the patch is entirely managed by `fluxd` and applied post-hoc to the manifests.

Command-updated is more general, but since you need to supply your own programs to find and update the right file, it is likely to be a lot more work.

Both patch-updated and command-updated configurations have the same way of specifying how to generate manifests, and differ only in how updates are recorded.

Generator configuration

Here is an example of a `.flux.yaml`:

```yaml
version: 1 # must be `1`
patchUpdated:
  generators:
  - command: kustomize build .
  patchFile: flux-patch.yaml
```

The `generators` field is an array of commands, all of which will be run in turn. Each command is expected to print a YAML stream to its stdout. The streams are concatenated and parsed as one big YAML stream, before being applied.

Much of the time, it will only be necessary to supply one command to be run.

The commands will be run with the target path being processed as a working directory – which is not necessarily the same directory in which the `.flux.yaml` file was found. *See below* for more details on the execution context in which commands are run.

Using patch-updated configuration

A patch-updated configuration generates manifests using commands, and records updates as a set of strategic merge patches in a file.

For example, when an automated image upgrade is run, fluxd will do this:

1. run the generator commands and parse the manifests;
2. find the manifest that needs to be updated, and calculate the patch to it that performs the update;
3. record that patch in the patch file.

When syncing, fluxd will generate the manifests as usual, then apply all the patches that have been recorded in the patch file.

This is how a patch-updated `.flux.yaml` looks in general:

```
version: 1
patchUpdated:
  generators:
  - command: generator_command
  patchFile: path/to/patch.yaml
```

The `generators` field is explained just above. The `patchFile` field gives a path, relative to the target path, in which to record patches. `fluxd` will create or update the file when needed, and commit any changes it makes to git.

> Note: at present, it is necessary to manually remove patches that refer to deleted manifests. See issue #2428

Using command-updated configuration

A command-updated configuration generates manifests in the same way, but records changes by running commands as given in the `.flux.yaml`.

This is how a command-updated `.flux.yaml` looks in general:

```
version: 1
commandUpdated:
  generators:
  - command: generator_command
  updaters:
  - containerImage:
      command: image_updater_program
    policy:
      command: policy_updater_program
```

The `updaters` section is particular to command-updated configuration. It contains an array of updaters, each of which gives a command for updating container images, and a command for updating policies (policy controls how automated updates should be applied to a resource; these appear as annotations in generated manifests).

When asked to update a resource, fluxd will run execute the appropriate variety of command for *each* entry in `updaters:`. For example, when updating an image, it will execute the command under `containerImage`, for each updater entry, in turn.

Usually updates come in batches – e.g., updating the same container image in several resources – so the commands will likely be run several times.

Execution context of commands

`generators` and `updaters` are run in a POSIX shell inside the fluxd container. This means that the executables mentioned in commands must be available in the running fluxd container.

Flux currently includes `kustomize`, `sops` and basic Unix shell tools. If the tools in the Flux image are not sufficient for your use case, you have some options:

- build your own custom image based on the Flux image that includes the tooling you need, and run that image instead of `fluxcd.io/flux`;
- copy files from an `initContainer` into a volume shared by the flux container, within the deployment.

In the future it may be possible to specify an container image for each command, rather than relying on the tooling being in the filesystem.

The working directory (also known as CWD) of the `command`s executed from a `.flux.yaml` file will be set to the target path, i.e., the `--git-path` entry.

For example, when using flux with `--git-path=staging` on a git repository with this structure:

```
├── .flux.yaml
├── staging/
│   └── [...]
└── production/
    └── [...]
```

... the commands in `.flux.yaml` will be executed with their working directory set to `staging`.

In addition, commands from `updaters` are given arguments via environment variables, when executed:

- `FLUX_WORKLOAD`: the workload to be updated. Its format is `<namespace>:<kind>/<name>` (e.g. `default:deployment/foo`). For convenience (to circumvent parsing) `FLUX_WORKLOAD` is also broken down into the following environment variables:
- `FLUX_WL_NS`
- `FLUX_WL_KIND`
- `FLUX_WL_NAME`
- `containerImage` updaters are provided with:
  - `FLUX_CONTAINER`: Name of the container within the workload whose image needs to be updated.
  - `FLUX_IMG`: Image name which the container needs to be updated to (e.g. `nginx`).
  - `FLUX_TAG`: Image tag which the container needs to be updated to (e.g. `1.15`).
- `policy` updaters are provided with:
  - `FLUX_POLICY`: the name of the policy to be added or updated in the workload. To make into an annotation name, prefix with `fluxcd.io/`
  - `FLUX_POLICY_VALUE`: value of the policy to be added or updated in the controller. If the `FLUX_POLICY_VALUE` environment variable is not set, it means the policy should be removed.

Please note that the default timeout for sync commands is set to one minute. If you run into errors like `error executing generator command: context deadline exceeded`, you can increase the timeout with the `--sync-timeout` fluxd command flag or the `sync.timeout` Helm chart option.

4.5 Garbage collection

Part of syncing a cluster with a git repository is getting rid of resources in the cluster that have been removed in the repository. You can tell `fluxd` to do this "garbage collection" using the command-line flag `--sync-garbage-collection`. It's important to know how it operates, and appreciate its limitations, before enabling it.

4.5.1 How garbage collection works

When garbage collection is enabled, syncing is done in two phases:

1. Apply all the manifests in the git repo (as delimited by the branch and path arguments), and give each resource a label marking it as having been synced from this source.
2. Ask the cluster for all the resources marked as being from this source, and delete those that were not applied in step 1.

In the above, "source" refers to the particular combination of git repo URL, branch, and paths that this `fluxd` has been configured to use, which is taken as identifying the resources under *this* `fluxd`'s control.

We need to be careful about identifying these accurately, since getting it wrong could mean *not* deleting resources that should be deleted; or (much worse), deleting resources that are under another `fluxd`'s control.

The definition of "source" affects how garbage collection behaves when you reconfigure `fluxd`. It is intended to be conservative: it ensures that `fluxd` will not delete resources that it did not create.

4.5.2 Limitations of this approach

In general, if you change an element of the source (the git repo URL, branch, and paths), there is a possiblility that resources no longer present in the new source will be missed (i.e., not deleted) by garbage collection, and you will need to delete them by hand.

4.6 Git commit signing and verification

4.6.1 Summary

Flux can be configured to sign commits that it makes to the user git repo when, for example, it detects an updated Docker image is available for a release with automatic deployments enabled. To complete this functionality it is also able to verify signatures of commits (and the sync tag in git) to prevent Flux from applying unauthorized changes on the cluster.

4.6.2 Commit signing

The signing of commits (and the sync tag) requires two flags to be set:

1. `--git-gpg-key-import` should be set to the path(s) Flux should look for GPG key(s) to import, this can be direct paths to keys and/or the paths to folders Flux should scan for files.
2. `--git-signing-key` should be set to the ID of the key Flux should use to sign commits, this can be the full fingerprint or the long ID, for example: `700D397C988079BFF0DDAFED6A7436E8790F8689` (or `6A7436E8790F8689`)

Once enabled Flux will sign both commits and the sync tag with given `--git-signing-key`.

Creating a GPG signing key

Note: This requires gnupg to be installed on your system.

1. Enter the following shell command to start the key generation dialog:

```
$ gpg --full-generate-key
```

2. The dialog will guide you through the process of generating a key. Pressing the `Enter` key will assign the default value, please note that when in doubt, in almost all cases, the default value is recommended.

   Select what kind of key you want and press `Enter`:

```
Please select what kind of key you want:
(1) RSA and RSA (default)
(2) DSA and Elgamal
(3) DSA (sign only)
(4) RSA (sign only)
Your selection? 1
```

3. Enter the desired key size (or simply press `Enter` as the default will be secure for almost any setup):

```
RSA keys may be between 1024 and 4096 bits long.
What keysize do you want? (2048)
```

4. Specify how long the key should be valid (or simply press `Enter`):

```
Please specify how long the key should be valid.
       0 = key does not expire
    <n>  = key expires in n days
    <n>w = key expires in n weeks
    <n>m = key expires in n months
    <n>y = key expires in n years
Key is valid for? (0)
```

1. Verify your selection of choices and accept (`y` and `Enter`)
2. Enter your user ID information, it is recommended to set the email address to the same address as the daemon uses for Git operations.
3. **Do not enter a passphrase**, as Flux will be unable to sign with a passphrase protected private key, instead, keep it in a secure place.
4. You can validate the public and private keypair were created with success by running:

```
$ gpg --list-secret-keys --keyid-format long <email address>
sec   rsa2048/6A7436E8790F8689 2019-03-28 [SC]
      700D397C988079BFF0DDAFED6A7436E8790F8689
uid                 [ultimate] Weaveworks Flux <support@weave.works>
ssb   rsa2048/ECA4FF5BD988B8E9 2019-03-28 [E]
```

Importing a GPG signing key

Any file found in the configured `--git-gpg-key-import` path(s) will be imported into GPG; therefore, by volume-mounting a key into that directory it will be made available for use by Flux.

1. Retrieve the key ID (second row of the `sec` column):

```
$  gpg --list-secret-keys --keyid-format long <email address>
sec   rsa2048/6A7436E8790F8689 2019-03-28 [SC]
      700D397C988079BFF0DDAFED6A7436E8790F8689
uid                 [ultimate] Weaveworks Flux <support@weave.works>
ssb   rsa2048/ECA4FF5BD988B8E9 2019-03-28 [E]
```

2. Export the public and private keypair from your local GPG keyring to a Kubernetes secret with `--export-secret-keys <key id>`:

```
$ gpg --export-secret-keys --armor 700D397C988079BFF0DDAFED6A7436E8790F8689 |
  kubectl create secret generic flux-gpg-signing-key --from-file=flux.asc=/dev/
↪stdin --dry-run -o yaml
apiVersion: v1
data:
  flux.asc: <base64 string>
kind: Secret
metadata:
  creationTimestamp: null
  name: flux-gpg-signing-key
```

3. Adapt your Flux deployment to mount the secret and enable the signing of commits:

```
spec:
  template:
    spec:
      volumes:
      - name: gpg-signing-key
        secret:
          secretName: flux-gpg-signing-key
          defaultMode: 0400
      containers:
      - name: flux
        volumeMounts:
        - name: gpg-signing-key
          mountPath: /root/gpg-signing-key/
          readOnly: true
        args:
        - --git-gpg-key-import=/root/gpg-signing-key
        - --git-signing-key=700D397C988079BFF0DDAFED6A7436E8790F8689 # key id
```

   or set the `gpgKeys.secretName` in your Helm `values.yaml` to `gpg-keys`, and `signingKey` to your `<key id>`.

4. To validate your setup is working, run `git log --show-signature` or `git verify-tag <configured label>` to assure Flux signs its git actions.

```
$ git verify-tag <configured label>
gpg: Signature made vr 29 mrt 2019 15:28:34 CET
gpg:                using RSA key 700D397C988079BFF0DDAFED6A7436E8790F8689
gpg: Good signature from "Weaveworks Flux <support@weave.works>" [ultimate]
```

   Note: Flux *does not* recursively scan a given directory but does understand symbolic links to files.

Note: Flux will automatically add any imported key to the GnuPG trustdb. This is required as git will otherwise not trust signatures made with the imported keys.

4.6.3 Signature verification

The verification of commit signatures is enabled by importing all trusted public keys (`--git-gpg-key-import=<path>,<path2>`), and by setting the `--gpg-verify-signatures` flag. Once enabled Flux will verify all commit signatures, and the signature from the sync tag it is comparing revisions with.

In case a signature can not be verified, Flux will sync state up to the last valid revision it can find *before* the unverified commit was made, and lock on this revision.

Importing trusted GPG keys and enabling verification

1. Collect the public keys from all trusted git authors.
2. Create a `ConfigMap` with all trusted public keys:

```
$ kubectl create configmap generic flux-gpg-public-keys \
  --from-file=author.asc --from-file=author2.asc --dry-run -o yaml
apiVersion: v1
data:
  author.asc: <base64 string>
  author2.asc: <base64 string>
kind: ConfigMap
metadata:
  creationTimestamp: null
  name: flux-gpg-public-keys
```

3. Mount the config map in your Flux deployment, add the mount path to `--git-gpg-key-import`, and enable the verification of commits:

```
spec:
  template:
    spec:
      volumes:
      - name: gpg-public-keys
        configMap:
          name: flux-gpg-public-keys
          defaultMode: 0400
      containers:
      - name: flux
        volumeMounts:
        - name: gpg-public-keys
          mountPath: /root/gpg-public-keys
          readOnly: true
        args:
        - --git-gpg-key-import=/root/gpg-public-keys
        - --git-verify-signatures
```

Note: Flux *does not* recursively scan a given directory but does understand symbolic links to files.

Enabling verification for existing repositories, disaster recovery, and deleted sync tags

In case you have existing commits in your repository without a signature you may want to:

a. First enable signing by setting the `--git-gpg-key-import` and `--git-signing-key`, after Flux has synchronized the first commit with a signature, enable verification.

b. Sign the sync tag by yourself, with a key that is imported, to point towards the first commit with a signature (or the current `HEAD`). Flux will then start synchronizing the changes between the sync tag revision and `HEAD`.

```
$ git tag --force --local-user=<key id> -a -m "Sync pointer" <tag name> <revision>
$ git push --force origin <tag name>
```

Choosing a `--git-verify-signatures-mode`

`"none"` (default)

By default, Flux skips GPG verification of all commits.

`"all"`

This is the regular verification behavior, consistent with the original `--gitVerifySignatures` flag. It will perform GPG verification on every commit between the tip of the Flux branch and the Flux sync tag, including all parents. If your `master` branch contains only signed commits (a flow which GitHub supports), then this flow ought to work.

`"first-parent"`

However, there are some arguments for more limited signing behaviors, e.g. this parable and this thread). In particular, it can be useful to allow unsigned commits into `master`, and to point Flux at a `release` branch containing signed merges from `master`. A merge commit has two parents: the previous commit "in the branch," as well as the last commit in the merged branch. In this scenario, use the `"first-parent"` mode – only the merge commits "in the branch" should be GPG-verified, since the commits from master have no signature.

4.7 Automated deployment of new container images

Flux can be used to automate container image updates in your cluster. Flux periodically scans the pods running in your cluster and builds a list of all container images. Using the image pull secrets, it connects to the container registries, pulls the images metadata and stores the image tag list in memcached.

You can enable the automate image tag updates by annotating your deployments, statefulsets, daemonsets or cronjobs objects. You can also control what tags should be considered for an update by using glob, regex or semantic version expressions.

> **Note:** that Flux only works with immutable image tags (`:latest` is not supported). Every image tag must be unique, for this you can use the Git commit SHA or semver when tagging images.

4.7.1 Examples

What follows is a list of examples on how you can control the image update automation. If you're using Helm releases please see the Helm operator integration docs.

Turn on automation based on timestamp:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    fluxcd.io/automated: "true"
spec:
  template:
    spec:
      containers:
      - name: app
        image: docker.io/org/my-app:1.0.0
```

The above configuration will make Flux update the `app` container when you push a new image tag, be it `my-app:1.0.1` or `my-app:9e3bdaf`.

Restrict image updates with sem ver:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    fluxcd.io/automated: "true"
    fluxcd.io/tag.app: semver:~1.0
spec:
  template:
    spec:
      containers:
      - name: app
        image: docker.io/org/my-app:1.0.0
```

The above configuration will make Flux update the image when you push an image tag that matches the semantic version expression e.g `my-app:1.0.1` but not `my-app:1.2.0`.

Flux also support all the other ranges and operators available here in addition to the ~ range.

Restrict image to deploy prerelease version up until `myapp:1.0.0` but not `myapp:1.0.1`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    fluxcd.io/automated: "true"
    fluxcd.io/tag.app: "semver: >= 1.0.0-rc.0, <1.0.1"
spec:
  template:
    spec:
      containers:
      - name: app
        image: docker.io/org/my-app:1.0.0-rc.1
```

Restrict image updates with glob and regex expressions:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    fluxcd.io/automated: "true"
    fluxcd.io/tag.sidecar: regex:^stg.*
    fluxcd.io/tag.app: glob:dev-*
spec:
  template:
    spec:
      containers:
      - name: sidecar
        image: docker.io/org/my-proxy:stg-4s7bsgv
      - name: app
        image: docker.io/org/my-app:dev-9e3bdaf
```

The above configuration will make Flux update the `sidecar` when you push a tag for the `my-proxy` image that begins with `stg`. For the `app` container, Flux will update it when you push a tag for the `my-app` image that begins with `dev-`.

To target a specific container the annotation format is `fluxcd.io/tag.<CONTAINER>: <TYPE>:<EXPRESSION>`.

You can turn off the automation with `fluxcd.io/automated: "false"` or with `fluxcd.io/locked: "true"`.

4.8 Integration with the Helm operator

You can release charts to your cluster via "GitOps", by combining Flux and the Helm operator.

The essential mechanism is this: the declaration of a Helm release is represented by a custom resource, specifying the chart and its values. If you put such a resource in your git repo as a file, Flux will apply it to the cluster, and once it's in the cluster, the Helm Operator will make sure the release exists by installing or upgrading it.

4.8.1 Upgrading images in a `HelmRelease` using Flux

If the chart you're using in a `HelmRelease` lets you specify the particular images to run, you will usually be able to update them with Flux, the same way you can with Deployments and so on.

> **Note:** for automation to work, the repository *and* tag should be defined (either as a whole string, or under separate keys), as Flux determines image updates based on what it reads in the `.spec.values` of the `HelmRelease`.

Automated image detection

Flux interprets certain commonly used structures in the `values` section of a `HelmRelease` as referring to images, at least an `image` key needs to be specified. The following are understood (showing just the `values` section):

```
values:
  image: repo/image:version
```

```yaml
values:
  image: repo/image
  tag: version
```

```yaml
values:
  registry: docker.io
  image: repo/image
  tag: version
```

```yaml
values:
  image:
    repository: repo/image
    tag: version
```

```yaml
values:
  image:
    registry: docker.io
    repository: repo/image
    tag: version
```

These can appear at the top level (immediately under `values:`), or in a subsection (under a key, itself under `values:`). Other values may be mixed in arbitrarily. Here's an example of a values section that specifies two images:

```yaml
values:
  persistent: true

  # image that will be labeled "chart-image"
  image: repo/image1:version

  subsystem:
    # image that will be labeled "subsystem"
    image:
      repository: repo/image2
      tag: version
      imagePullPolicy: IfNotPresent
    port: 4040
```

Annotations

If Flux does not automatically detect your image, it is possible to map the image paths by alias with YAML dot notation annotations. An alias overrules a detected image.

The following annotations are available, and `repository.fluxcd.io` is required for any of these to take effect.

The following example `HelmRelease` specifies two images:

```yaml
metadata:
  annotations:
    # image and tag
    repository.fluxcd.io/app: appImage
    tag.fluxcd.io/app: appTag
    filter.fluxcd.io/app: 'glob: *'
    # nested image with registry and tag
    registry.fluxcd.io/submarine: sub.marinesystem.reg
```

(continues on next page)

(continued from previous page)

```
    repository.fluxcd.io/submarine: sub.marinesystem.img
    tag.fluxcd.io/submarine: sub.marinesystem.tag

spec:
  values:
    # image and tag
    appImage: repo/image1
    appTag: version
    sub:
      marinesystem:
        # nested image with registry and tag
        reg: domain.com
        img: repo/image2
        tag: version
```

Filters

You can use the same annotations in the `HelmRelease` as you would for a Deployment or other workload, to control updates and automation. For the purpose of specifying filters, the container name is either `chart-image` (if at the top level), the key under which the image is given (e.g., `"subsystem"` from the example above), or the alias you are using in your annotations.

Top level image example:

```
kind: HelmRelease
metadata:
  annotations:
    fluxcd.io/automated: "true"
    filter.fluxcd.io/chart-image: semver:~4.0
spec:
  values:
    image:
      repository: bitnami/mongodb
      tag: 4.0.3
```

Sub-section images example:

```
kind: HelmRelease
metadata:
  annotations:
    fluxcd.io/automated: "true"
    filter.fluxcd.io/prometheus: semver:~2.3
    filter.fluxcd.io/alertmanager: glob:v0.15.*
    filter.fluxcd.io/nats: regex:^0.6.*
spec:
  values:
    prometheus:
      image: prom/prometheus:v2.3.1
    alertmanager:
      image: prom/alertmanager:v0.15.0
    nats:
      image:
        repository: nats-streaming
        tag: 0.6.0
```

4.9 Monitoring Flux

The Flux daemon exposes `/metrics` endpoints which can be scraped for monitoring data in Prometheus format; exact metric names and help are available from the endpoints themselves.

The following metrics are exposed:

Flux sync state can be obtained by using the following PromQL expressions:

- `delta(flux_daemon_sync_duration_seconds_count{success='true'}[6m]) < 1` - for general flux sync errors - usually if that is true then there are some problems with infrastructure or there are manifests parse error or there are manifests with duplicate ids.
- `flux_daemon_sync_manifests{success='false'} > 0` - for git manifests errors - if true then there are either some problems with applying git manifests to kubernetes - e.g. configmap size is too big to fit in annotations or immutable field (like label selector) was changed.

CHAPTER

FIVE

GUIDES

5.1 Providing your own SSH key

Flux connects to the repository using an SSH key it retrieves from a Kubernetes secret, if the configured (`--k8s-secret-name`) secret has no `identity` key/value pair, it will generate new private key.

With this knowledge, providing your own SSH key is as simple as creating the configured secret in the expected format.

1. create a Kubernetes secret from your own private key:

   ```
   kubectl create secret generic flux-git-deploy --from-file=identity=/full/path/to/
   ↪private_key
   ```

   this will result in a secret that has the structure:

   ```
   apiVersion: v1
   data:
     identity: <base64 encoded RSA PRIVATE KEY>
   kind: Secret
   type: Opaque
   metadata:
     ...
   ```

2. *(optional)* if you created the secret with a non-default name (default: `flux-git-deploy`), set the `--k8s-secret-name` flag to the name of your secret (i.e. `--k8s-secret-name=foo`).

   Note: the SSH key must be configured to have R/W access to the repository. More specifically, the SSH key must be able to create and update tags. E.g. in Gitlab, that means it requires `Maintainer` permissions. The `Developer` permission can create tags, but not update them.

5.2 Using Git over HTTPS

Instead of making use of Flux' capabilities to generate an SSH private key, or supplying your own, it is possible to set environment variables and use these in your `--git-url` argument to provide your HTTPS basic auth credentials without having to expose them as a plain value in your workload.

Note: setting an HTTP(S) URL as `--git-url` will disable the generation of a private key and prevent the setup of the SSH keyring.

Note: the variables *must be escaped with `$()`* for Kubernetes to pass the values to the Flux container, e.g. `$(GIT_AUTHKEY)`. Read more about this Kubernetes feature.

1. Create a personal access token to be used as the `GIT_AUTHKEY`:

- GitHub
- GitLab
- BitBucket

2. Create a Kubernetes secret with two environment variables and their respective values (replace `<username>` and `<token/password>`):

```
kubectl create secret generic flux-git-auth --from-literal=GIT_AUTHUSER=<username>
↪ --from-literal=GIT_AUTHKEY=<token/password>
```

this will result in a secret that has the structure:

```
apiVersion: v1
data:
  GIT_AUTHKEY: <base64 encoded token/password>
  GIT_AUTHUSER: <base64 encoded username>
kind: Secret
type: Opaque
metadata:
  ...
```

3. Mount the Kubernetes secret as environment variables using `envFrom` and use them in your `--git-url` argument:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flux
...
spec:
  containers:
  - name: flux
    envFrom:
    - secretRef:
        name: flux-git-auth
    args:
    # Replace `github.com/...` with your git repository
    - --git-url=https://$(GIT_AUTHUSER):$(GIT_AUTHKEY)@github.com/fluxcd/flux-get-
↪started.git
    ...
```

5.3 Using a private Git host

If you're using your own git host – e.g., your own installation of gitlab, or bitbucket server – you will need to add its host key to `~/.ssh/known_hosts` in the Flux daemon container.

First, run a check that you can clone the repo. The following assumes that your git server's hostname (e.g., `githost`) is in `$GITHOST` and the URL you'll use to access the repository (e.g., `user@githost:path/to/repo`) is in `$GITREPO`.

```
$ # Find the fluxd daemon pod:
$ kubectl get pods --all-namespaces -l name=flux
NAMESPACE   NAME                     READY     STATUS    RESTARTS   AGE
weave       flux-85cdc6cdfc-n2tgf    1/1       Running   0          1h
```

(continues on next page)

(continued from previous page)

```
$ kubectl exec -n weave flux-85cdc6cdfc-n2tgf -ti -- \
    env GITHOST="$GITHOST" GITREPO="$GITREPO" PS1="container$ " /bin/sh

container$ git clone $GITREPO
Cloning into <repository name>...
No ECDSA host key is known for  <GITHOST> and you have requested strict checking.
Host key verification failed.
fatal: Could not read from remote repository

container$ # ^ that was expected. Now we'll try with a modified known_hosts
container$ ssh-keyscan $GITHOST >> ~/.ssh/known_hosts
container$ git clone $GITREPO
Cloning into '...'
...
```

If `git clone` doesn't succeed, you'll need to check that the SSH key has been installed properly first, then come back. `ssh -vv $GITHOST` from within the container may help debug it.

If it *did* work, you will need to make it a more permanent arrangement. Back in that shell, create a ConfigMap for the cluster. To make sure the ConfigMap is created in the namespace of the Flux deployment, the namespace is set explicitly:

```
container$ kubectl create configmap flux-ssh-config --from-file=$HOME/.ssh/known_
↪hosts -n $(cat /var/run/secrets/kubernetes.io/serviceaccount/namespace)
configmap "flux-ssh-config" created
```

To use the ConfigMap every time the Flux daemon restarts, you'll need to mount it into the container. The example deployment manifest includes an example of doing this, commented out. Uncomment those two blocks:

```
      - name: ssh-config
        configMap:
          name: flux-ssh-config
```

```
        - name: ssh-config
          mountPath: /root/.ssh
```

It assumes you used `flux-ssh-config` as name of the ConfigMap and then reapply the manifest.

Another alternative is to create the ConfigMap from a template. This could be something like:

```
apiVersion: v1
data:
  known_hosts: |
    # github
    192.30.253.112 ssh-rsa␣
↪AAAAB3NzaC1yc2EAAAABIwAAAQEAq2A7hRGmdnm9tUDbO9IDSwBK6TbQa+PXYPCPy6rbTrTtw7PHkccKrpp0yVhp5HdEIcKr6p
↪yMf+Se8xhHTvKSCZIFImWwoG6mbUoWf9nzpIoaSjB+weqqUUmpaaasXVal72J+UX2B+2RPW3RcT0eOzQgqlJL3RKrTJvdsjE3J
↪w4yCE6gbODqnTWlg7+wC604ydGXA8VJiS5ap43JXiUFFAaQ==
    # github
    192.30.253.113 ssh-rsa␣
↪AAAAB3NzaC1yc2EAAAABIwAAAQEAq2A7hRGmdnm9tUDbO9IDSwBK6TbQa+PXYPCPy6rbTrTtw7PHkccKrpp0yVhp5HdEIcKr6p
↪yMf+Se8xhHTvKSCZIFImWwoG6mbUoWf9nzpIoaSjB+weqqUUmpaaasXVal72J+UX2B+2RPW3RcT0eOzQgqlJL3RKrTJvdsjE3J
↪w4yCE6gbODqnTWlg7+wC604ydGXA8VJiS5ap43JXiUFFAaQ==
    # private gitlab
    gitlab._________ ssh-rsa AAAAB3N...
```

(continues on next page)

(continued from previous page)

```
kind: ConfigMap
metadata:
  name: flux-ssh-config
  namespace: <OPTIONAL NAMESPACE (if not default)>
```

You will need to explicitly tell `fluxd` to use that service account by uncommenting and possible adapting the line `# serviceAccountName: flux` in the file `flux-deployment.yaml` before applying it.

5.4 Upgrading to Flux v1

Flux v1 is a major improvement over the previous versions, and is different enough that you need to do a bit of work to upgrade it.

In previous releases of Flux, much of the work was done by the service. This meant that to get a useful system, you had to run both the daemon and the service in your cluster. In version 1, the daemon does all of the mechanical work by itself.

5.4.1 Differences between old Flux and Flux v1

In version 1 the daemon is more self-sufficient and easier to set up. It is also more capable, and in particular, it now synchronises your cluster with the manifests you keep in git – enabling you to use git (and GitHub) workflows to manage your cluster.

5.4.2 Upgrade process

In summary, you will need to:

1. Remove the old Flux resources from your cluster
2. Delete any deployment keys
3. Run the new Flux resources
4. Install a new deploy key

First, it will help in a few places to have an old fluxctl around. Download it from GitHub:

```
curl -o fluxctl_030 https://github.com/fluxcd/flux/releases/download/0.3.0/fluxctl_
↪linux_amd64
# or if using macOS,
# curl -o fluxctl_030 https://github.com/fluxcd/flux/releases/download/0.3.0/fluxctl_
↪darwin_amd64
chmod a+x ./fluxctl_030
```

5.4.3 If you are running Flux in "standalone" mode

Set the environment variable FLUX_URL to point to the Flux service you are running, as described in the old deployment docs. The particular URL will differ, depending on how you have told Kubernetes to expose the Flux service.

Before making any changes, get the config so that it can be consulted later:

```
./fluxctl_030 get-config --fingerprint=md5 | tee old-config.yaml
```

Remove old Flux resources

> *Important! If you have Flux resources committed to git*
>
> The first thing to do here is to remove any manifests for running Flux you have stored in git, before deleting them in the cluster (below). If you don't remove these, running the new Flux daemon will restore the old configuration.

You can delete the Flux resources by referring to the manifest files used to create them. If you don't have the files on hand, you can try using the example deployment as a stand-in:

```
git clone --branch 0.3.0 git@github.com:fluxcd/flux flux-0.3.0
kubectl delete --ignore-not-found -R -f ./flux-0.3.0/deploy
```

That's something of a sledgehammer! But it should cover most cases.

Delete deployment keys

It's good practice to remove any unused deployment keys. If you're using GitHub, look at the settings for the repository you were pointing Flux at, and delete the key Flux was using. To check you are removing the correct key, you can see the fingerprint of the key used by Flux in the file `old-config.yaml` that was created earlier.

Configure and run new Flux resources

> First, it is important to understand that Flux manages more of your cluster resources now. It will automatically apply the manifests that appear in your config repo, either by creating or by updating them. In other words, it tries to keep the cluster running whatever is represented in the repo. (Though it doesn't delete things, yet).

To run Flux without connecting to Weave Cloud, adapt the manifests provided in the Flux repo with the git parameters (URL, path, and branch) from `old-config.yaml`, and then apply them with `kubectl`. Consider adding these adapted manifests to your own config repo.

The daemon now has an API itself, so you can point fluxctl directly at it (the example manifests include a Kubernetes service so you can do just that).

You may find that you need to set FLUX_URL again, to take account of the new deployment. See the setup instructions for guidance.

To see the SSH key created by Flux, download the latest fluxctl from the release page and run:

```
fluxctl identity
```

You will need to add this as a deploy key, which is also described in the setup instructions linked above.

5.4.4 Troubleshooting

The `kubectl delete` commands didn't delete anything

It's possible that the Flux resources are in an unusual namespace or given a different name. As a last resort, you can hunt down the resources by name and delete them. Use kubectl to look for likely suspects.

```
kubectl get serviceaccount,service,deployment --all-namespaces
```

Have a look for deployments and services with "flux" in the name.

I deleted the Flux resources but when I install Flux v1 they come back

The most likely explanation is that you have manifests for the resources in your config repo. When Flux v1 starts, it does a sync – and if there are manifests for the old Flux still in git, it will create those as resources.

If that's the case, you will need to remove the manifests from git before running Flux v1.

CHAPTER

SIX

TUTORIALS

6.1 Get started with Flux

This short guide shows a self-contained example of Flux and just takes a couple of minutes to get set up. By the end you will have Flux running in your cluster and it will be deploying any code changes for you.

> **Note:** If you would like to install Flux using Helm, refer to the Helm section.

6.1.1 Prerequisites

You will need to have Kubernetes set up. For a quick local test, you can use `minikube`, `kubeadm` or `kind`. Any other Kubernetes setup will work as well though.

If working on e.g. GKE with RBAC enabled, you will need to add a `ClusterRoleBinding`:

```
kubectl create clusterrolebinding "cluster-admin-$(whoami)" \
--clusterrole=cluster-admin \
--user="$(gcloud config get-value core/account)"
```

6.1.2 Set up Flux

In our example we are going to use flux-get-started. If you want to use that too, be sure to create a fork of it on GitHub.

First, please *install fluxctl*.

Create the `flux` namespace:

```
kubectl create ns flux
```

Then, install Flux in your cluster (replace `YOURUSER` with your GitHub username):

```
export GHUSER="YOURUSER"
fluxctl install \
--git-user=${GHUSER} \
--git-email=${GHUSER}@users.noreply.github.com \
--git-url=git@github.com:${GHUSER}/flux-get-started \
--git-path=namespaces,workloads \
--namespace=flux | kubectl apply -f -
```

`--git-path=namespaces,workloads`, is meant to exclude Helm manifests. Again, if you want to get started with Helm, please refer to the Helm section.

Wait for Flux to start:

```
kubectl -n flux rollout status deployment/flux
```

6.1.3 Giving write access

At startup Flux generates a SSH key and logs the public key. Find the SSH public key by installing *fluxctl* and running:

```
fluxctl identity --k8s-fwd-ns flux
```

In order to sync your cluster state with git you need to copy the public key and create a deploy key with write access on your GitHub repository.

Open GitHub, navigate to your fork, go to **Setting > Deploy keys**, click on **Add deploy key**, give it a `Title`, check **Allow write access**, paste the Flux public key and click **Add key**. See the GitHub docs for more info on how to manage deploy keys.

(Or replace `YOURUSER` with your GitHub ID in this url: `https://github.com/YOURUSER/flux-get-started/settings/keys/new` and paste the key there.)

> **Note:** the SSH key must be configured to have R/W access to the repository. More specifically, the SSH key must be able to create and update tags. E.g. in Gitlab, that means it requires `Maintainer` permissions. The `Developer` permission can create tags, but not update them.

6.1.4 Committing a small change

In this example we are using a simple example of a webservice and change its configuration to use a different message.

Replace `YOURUSER` in `https://github.com/YOURUSER/flux-get-started/blob/master/workloads/podinfo-dep.yaml` with your GitHub ID), open the URL in your browser, edit the file, change the `PODINFO_UI_MESSAGE` env var to `Welcome to Flux` and commit the file.

By default, Flux git pull frequency is set to 5 minutes. You can tell Flux to sync the changes immediately with:

```
fluxctl sync --k8s-fwd-ns flux
```

6.1.5 Confirm the change landed

To access our webservice and check out its welcome message, simply run:

```
kubectl -n demo port-forward deployment/podinfo 9898:9898 &
curl localhost:9898
```

Notice the updated `message` value in the JSON reply.

6.1.6 Conclusion

As you can see, the actual steps to set up Flux, get our app deployed, give Flux access to it and see modifications land are very straight-forward and are a quite natural work-flow.

As a next step, you might want to dive deeper into *how to control Flux*, or go through our hands-on tutorial about driving Flux, e.g. automations, annotations and locks.

6.2 Get started with Flux using Helm

If you are using Helm already, this guide is for you. By the end you will have Helm installing Flux in the cluster and deploying any code changes for you.

If you are looking for more generic notes for how to install Flux using Helm, we collected them in the chart's README.

6.2.1 Prerequisites

You will need to have Kubernetes set up. To get up and running fast, you might want to use `minikube` or `kubeadm`. Any other Kubernetes setup will work as well though.

Download Helm v3:

- On MacOS:

```
brew install kubernetes-helm
```

- On Linux:
  - Download the latest release, unpack the tarball and put the binary in your `$PATH`.

If you are using Helm v2 you have to create a service account and a cluster role binding for Tiller:

```
kubectl -n kube-system create sa tiller

kubectl create clusterrolebinding tiller-cluster-rule \
    --clusterrole=cluster-admin \
    --serviceaccount=kube-system:tiller
```

Deploy Tiller in `kube-system` namespace (Helm v2 only):

```
helm init --skip-refresh --upgrade --service-account tiller --history-max 10
```

> **Note:** This is a quick guide and by no means a production ready Tiller setup, please look into 'Securing your Helm installation' and be aware of the `--history-max` flag before promoting to production.

6.2.2 Install Flux

Add the Flux repository:

```
helm repo add fluxcd https://charts.fluxcd.io
```

Apply the Helm Release CRD:

```
kubectl apply -f https://raw.githubusercontent.com/fluxcd/helm-operator/master/deploy/
↪flux-helm-release-crd.yaml
```

In this next step you install Flux using `helm`. Simply

1. Fork `fluxcd/flux-get-started` on GitHub and replace the `fluxcd` with your GitHub username in here
2. Create the flux namespace:

```
kubectl create namespace flux
```

3. Install Flux and the Helm operator by specifying your fork URL:

   Just make sure you replace `YOURUSER` with your GitHub username in the command below:

   - Using a public git server from `bitbucket.com`, `github.com`, `gitlab.com`, `dev.azure.com`, or `vs-ssh.visualstudio.com`:

     ```
     helm upgrade -i flux fluxcd/flux \
     --set git.url=git@github.com:YOURUSER/flux-get-started \
     --namespace flux

     helm upgrade -i helm-operator fluxcd/helm-operator \
     --set git.ssh.secretName=flux-git-deploy \
     --namespace flux
     ```

     Note: By default the helm-operator chart will install with support for both Helm v2 (which requires Tiller) and v3. You can target specific versions by setting the `helm.versions` value, e.g. `--set helm.versions=v3`.

   - Using a private git server:

     When deploying from a private repo, the known_hosts of the git server needs to be configured into a kubernetes configmap so that `StrictHostKeyChecking` is respected. See the README of the chart for further installation instructions in this case.

Allow some time for all containers to get up and running. If you're impatient, run the following command and see the pod creation process.

```
watch kubectl -n flux get pods
```

You will notice that `flux` and `flux-helm-operator` will start turning up in the `flux` namespace.

6.2.3 Giving write access

For the real benefits of GitOps, Flux will need access to your git repository to update configuration if necessary. To facilitate that you will need to add a deploy key to your fork of the repository.

This is pretty straight-forward as Flux generates a SSH key and logs the public key at startup. Find the SSH public key by installing *fluxctl* and running:

```
fluxctl identity --k8s-fwd-ns flux
```

In order to sync your cluster state with git you need to copy the public key and create a deploy key with write access on your GitHub repository.

Open GitHub, navigate to your fork, go to **Setting > Deploy keys**, click on **Add deploy key**, give it a `Title`, check **Allow write access**, paste the Flux public key and click **Add key**.

(Or replace `YOURUSER` with your GitHub ID in this url: `https://github.com/YOURUSER/flux-get-started/settings/keys/new` and paste the key there.)

Once Flux has confirmed access to the repository, it will start deploying the workloads of `flux-get-started`. After a while you will be able to see the Helm releases deployed by Flux (which are deployed into the `demo` namespace) listed like so:

```
helm list --namespace demo
```

6.2.4 Committing a small change

`flux-get-started` is a simple example in which three services (mongodb, redis and ghost) are deployed. Here we will simply update the version of mongodb to a newer version to see if Flux will pick this up and update our cluster.

The easiest way is to update your fork of `flux-get-started` and change the `image` argument.

Replace `YOURUSER` in `https://github.com/YOURUSER/flux-get-started/edit/master/releases/mongodb.yaml` with your GitHub ID, open the URL in your browser, edit the file, change the `tag:` line to the following:

```
  values:
    image:
      repository: bitnami/mongodb
      tag: 4.0.14
```

Commit the change to your `master` branch. It will now get automatically deployed to your cluster.

You can check out the Flux logs with:

```
kubectl -n flux logs deployment/flux -f
```

The default sync frequency for Flux using the Helm chart is five minutes. This can be tweaked easily. By observing the logs you can see when the change landed in the cluster.

Confirm the change landed by checking the `demo` namespace that Flux is deploying to:

```
kubectl describe -n demo deployment/mongodb | grep Image
```

6.2.5 Conclusion

As you can see, the actual steps to set up Flux, get our app deployed, give Flux access to it and see modifications land are very straight-forward and are a quite natural workflow.

6.2.6 A more advanced setup

For a more advanced Helm setup, take a look at the `fluxcd/helm-operator-get-started` repository.

6.3 How to bootstrap Flux using Kustomize

This guide shows you how to use Kustomize to bootstrap Flux on a Kubernetes cluster.

6.3.1 Prerequisites

You will need to have Kubernetes set up. For a quick local test, you can use `minikube` or `kubeadm`. Any other Kubernetes setup will work as well though.

A note on GKE with RBAC enabled

If working on e.g. GKE with RBAC enabled, you will need to add a cluster role binding:

```
kubectl create clusterrolebinding "cluster-admin-$(whoami)" \
    --clusterrole=cluster-admin \
    --user="$(gcloud config get-value core/account)"
```

6.3.2 Prepare Flux installation

First you'll need a git repository to store your cluster desired state. In our example we are going to use `fluxcd/flux-get-started`. If you want to use that too, be sure to create a fork of it on GitHub.

Create a directory, and add a `kustomization.yaml` file that uses the Flux deploy YAMLs as a base:

```
cat > fluxcd/kustomization.yaml <<EOF
namespace: flux
bases:
  - github.com/fluxcd/flux//deploy
patchesStrategicMerge:
  - patch.yaml
EOF
```

> **Note:** If you want to install a specific Flux release, add the version number to the base URL: `github.com/fluxcd/flux//deploy?ref=v1.14.1`

Create a patch file for Flux deployment and set the `--git-url` parameter to point to the config repository (replace `YOURUSER` with your GitHub username):

```
export GHUSER="YOURUSER"
cat > fluxcd/patch.yaml <<EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flux
  namespace: flux
spec:
  template:
    spec:
      containers:
        - name: flux
          args:
            - --manifest-generation=true
            - --memcached-hostname=memcached.flux
            - --memcached-service=
            - --ssh-keygen-dir=/var/fluxd/keygen
            - --git-branch=master
            - --git-path=namespaces,workloads
            - --git-user=${GHUSER}
            - --git-email=${GHUSER}@users.noreply.github.com
            - --git-url=git@github.com:${GHUSER}/flux-get-started
EOF
```

We set `--git-path=namespaces,workloads` to exclude Helm manifests. If you want to get started with Helm, please refer to the "Get started with Flux using Helm" tutorial.

Overwriting the default namespace

Overwriting the default (`flux`) namespace is possible by defining your own namespace and accordingly setting the `namespace:` key in the `kustomization.yaml` file.

Create your own namespace definition:

```
cat > fluxcd/namespace.yaml <<EOF
apiVersion: v1
kind: Namespace
metadata:
  name: <namespace>
EOF
```

Adapt your `fluxcd/kustomization.yaml` to include your own namespace resource and change the namespace:

```
namespace: <namespace>
resources:
  - namespace.yaml
bases:
  - github.com/fluxcd/flux//deploy
patchesStrategicMerge:
  - patch.yaml
```

6.3.3 Install Flux with Kustomize

In the next step, deploy Flux to the cluster (you'll need kubectl **1.14** or newer):

```
kubectl apply -k fluxcd
```

Wait for Flux to start:

```
kubectl -n flux rollout status deployment/flux
```

6.3.4 Setup GitHub write access

At startup Flux generates a SSH key and logs the public key. Find the SSH public key by installing *fluxctl* and running:

```
fluxctl identity --k8s-fwd-ns flux
```

In order to sync your cluster state with git you need to copy the public key and create a deploy key with write access on your GitHub repository.

Open GitHub, navigate to your fork, go to **Setting > Deploy keys**, click on **Add deploy key**, give it a `Title`, check **Allow write access**, paste the Flux public key and click **Add key**. See the GitHub docs for more info on how to manage deploy keys.

6.3.5 Committing a small change

In this example we'll be making a configuration change to a web application and display a different message in the UI.

Replace `YOURUSER` in `https://github.com/YOURUSER/flux-get-started/blob/master/workloads/podinfo-dep.yaml` with your GitHub ID), open the URL in your browser, edit the file, change the `PODINFO_UI_MESSAGE` env var to `Welcome to Flux` and commit the file.

By default, Flux git pull frequency is set to 5 minutes. You can tell Flux to sync the changes immediately with:

```
fluxctl sync --k8s-fwd-ns flux
```

6.3.6 Confirm the change landed

To access our webservice and check out its welcome message, simply run:

```
kubectl -n demo port-forward deployment/podinfo 9898:9898 &
curl localhost:9898
```

Notice the updated `message` value in the JSON reply.

6.3.7 Next steps

Try out flux-kustomize-example for using Flux with Kustomize to manage a staging and production clusters while minimizing duplicated declarations.

Try out fluxcd/multi-tenancy for using Flux with Kustomize to manage a multi-tenant cluster.

6.4 Automations, locks and annotations

In this tutorial we want to get a better feel for what we can do with Flux. We won't spend too much time with getting it up and running, so let's get that out of the way first.

In our example we are going to use the `flux-get-started` example deployment. So as your first step, please head to our example deployment and click on the "Fork" button.

6.4.1 Setup

First, please *install fluxctl*.

Then, run (replace `YOURUSER` with your GitHub username):

```
export GHUSER="YOURUSER"
fluxctl install \
--git-user=${GHUSER} \
--git-email=${GHUSER}@users.noreply.github.com \
--git-url=git@github.com:${GHUSER}/flux-get-started \
--git-path=namespaces,workloads \
--namespace=flux | kubectl apply -f -
```

Alternative: Using Helm for the setup

If you have never used Helm, you first need to

- Download/install Helm
- Set up Tiller. First create a service account and a cluster role binding for Tiller:

```
kubectl -n kube-system create sa tiller
kubectl create clusterrolebinding tiller-cluster-rule \
  --clusterrole=cluster-admin \
  --serviceaccount=kube-system:tiller
```

  Deploy Tiller in the `kube-system` namespace:

```
helm init --skip-refresh --upgrade --service-account tiller --history-max 10
```

> **Note:** This is a quick guide and by no means a production ready Tiller setup, please look into 'Securing your Helm installation' and be aware of the `--history-max` flag before promoting to production.

Now you can take care of the actual installation. First add the Flux chart repository:

```
helm repo add fluxcd https://charts.fluxcd.io
```

Apply the Helm Release CRD:

```
kubectl apply -f https://raw.githubusercontent.com/fluxcd/flux/helm-0.10.1/deploy-
↪helm/flux-helm-release-crd.yaml
```

Install Flux and its Helm Operator by specifying your fork URL. Just make sure you replace `YOURUSER` with your GitHub username in the command below:

```
helm upgrade -i Flux \
--set helmOperator.create=true \
--set helmOperator.createCRD=false \
--set git.url=git@github.com:YOURUSER/flux-get-started \
--namespace default \
fluxcd/flux
```

> **Note:** In this tutorial we keep things simple, so we deploy Flux into the `default` namespace. Normally you would pick a separate namespace for it. `fluxctl` has the *--k8s-fwd-ns <NAMESPACE> option* for specifying the right namespace.

Connecting to your git config

The first step is done. Flux is now and up running (you can confirm by running `kubectl get pods --all-namespaces`).

In the second step we will use `fluxctl` to talk to Flux in the cluster and interact with the deployments. (It enables you to drive all of Flux, so have a look at the output of `fluxctl -h` to get a better idea.)

> **Note:** Another option (without installing `fluxctl` is to take a look at the resulting annotation changes and make the changes in Git. This is GitOps after all. :-)

Tell fluxctl in which namespace is Flux installed:

```
export FLUX_FORWARD_NAMESPACE=flux
```

To enable Flux to sync your config, you need to add the deployment key to your fork.

Get your Flux deployment key by running:

```
fluxctl identity
```

Copy/paste the key and add it to `https://github.com/YOUR-USER-ID/flux-get-started/settings/keys/new` and enable write access for it.

Wait for sync to happen or run:

```
fluxctl sync
```

6.4.2 Driving Flux

After syncing, Flux will find out which workloads there are, which images are available and what needs doing. To find out which workloads are managed by Flux, run:

```
fluxctl list-workloads -a
```

Notice that `podinfo` is on `v1.3.2` and in state `automated`.

To check which images are avaible for podinfo run:

```
fluxctl list-images -w demo:deployment/podinfo
```

Now let's change the policy for `podinfo` to target `1.4.*` releases:

```
fluxctl policy -w demo:deployment/podinfo --tag-all='1.4.*'
```

On the command-line you should see a message just like this one:

```
WORKLOAD                  STATUS   UPDATES
demo:deployment/podinfo   success
Commit pushed:  4755a3b
```

If you now go back to `https://github.com/YOUR-USER-ID/flux-get-started` in your browser, you will notice that Flux has made a commit on your behalf. The policy change is now in Git, which is great for transparency and for defining expected state.

It should look a little something like this:

```
--- a/workloads/podinfo-dep.yaml
+++ b/workloads/podinfo-dep.yaml
@@ -8,8 +8,8 @@ metadata:
     app: podinfo
   annotations:
     fluxcd.io/automated: "true"
-    fluxcd.io/tag.init: regexp:^3.*
-    fluxcd.io/tag.podinfod: semver:~1.3
+    fluxcd.io/tag.init: glob:1.4.*
+    fluxcd.io/tag.podinfod: glob:1.4.*
 spec:
   strategy:
     rollingUpdate:
```

If you have a closer look at the last change which was committed, you'll see that the image filtering pattern has been changed. (Our docs explain how to use `semver`, `glob`, `regex` filtering.)

Again, wait for the sync to happen or run

```
fluxctl sync
```

To check which image is current, run:

```
fluxctl list-images -w demo:deployment/podinfo
```

In our case this is `1.4.2` (it could be a later image too). Let's say an engineer found that `1.4.2` was faulty and we have to go back to `1.4.1`. That's easy.

Lock deployment with a message describing why:

```
fluxctl lock -w demo:deployment/podinfo -m "1.4.2 does not work for us"
```

The resulting diff should look like this:

```
--- a/workloads/podinfo-dep.yaml
+++ b/workloads/podinfo-dep.yaml
@@ -10,6 +10,7 @@ metadata:
     app: podinfo
   annotations:
     fluxcd.io/automated: "true"
     fluxcd.io/tag.init: glob:1.4.*
     fluxcd.io/tag.podinfod: glob:1.4.*
+    fluxcd.io/locked: 'true'
 spec:
   strategy:
     rollingUpdate:
```

Rollback to `1.4.1`. Flag `--force` is needed because the workload is locked:

```
fluxctl release --force --workload demo:deployment/podinfo -i stefanprodan/podinfo:1.
↪4.1
```

The response should be:

```
Submitting release ...
CONTROLLER               STATUS   UPDATES
demo:deployment/podinfo  success  podinfod: stefanprodan/podinfo:1.4.2 -> 1.4.1
Commit pushed:  426d723
Commit applied: 426d723
```

and the diff for this is going to look like this:

```
--- a/workloads/podinfo-dep.yaml
+++ b/workloads/podinfo-dep.yaml
@@ -33,7 +33,7 @@ spec:
         - "1"
       containers:
       - name: podinfod
-        image: stefanprodan/podinfo:1.3.2
+        image: stefanprodan/podinfo:1.4.1
         imagePullPolicy: IfNotPresent
         ports:
         - containerPort: 9898
```

And that's it. At the end of this tutorial, you have automated, locked and annotated deployments with Flux.

Another tip, if you should get stuck anywhere: check what Flux is doing. You can do that by simply running:

```
kubectl logs -n flux deploy/flux -f
```

If you should have any questions, find us on Slack in the #flux channel, get an invite to it here.

CHAPTER

SEVEN

FREQUENTLY ASKED QUESTIONS

7.1 General questions

Also see

- *the introduction* for Flux's design principles
- *the troubleshooting section*

7.1.1 What does Flux do?

Flux automates the process of deploying new configuration and container images to Kubernetes.

7.1.2 How does it automate deployment?

It synchronises all manifests in a repository with a Kubernetes cluster. It also monitors container registries for new images and updates the manifests accordingly.

7.1.3 How is that different from a bash script?

The amount of functionality contained within Flux warrants a dedicated application/service. An equivalent script could easily get too large to maintain and reuse.

Anyway, we've already done it for you!

7.1.4 Why should I automate deployment?

Automation is a principle of lean development. It reduces waste, to provide efficiency gains. It empowers employees by removing dull tasks. It mitigates against failure by avoiding silly mistakes.

7.1.5 I thought Flux was about service routing?

That's where we started a while ago. But we discovered that automating deployments was more urgent for our own purposes.

Staging deployments with clever routing is useful, but it's a later level of operational maturity.

There are some pretty good solutions for service routing: Envoy, Istio for example. We may return to the matter of staged deployments.

7.1.6 Are there prerelease builds I can run?

There are builds from CI for each merge to master branch. See fluxcd/flux-prerelease.

7.2 Technical questions

7.2.1 Does it work only with one git repository?

At present, yes it works only with a single git repository containing Kubernetes manifests. You can have as many git repositories with application code as you like, to be clear – see *below*.

There's no principled reason for this, it's just a consequence of time and effort being in finite supply. If you have a use for multiple git repo support, please comment in https://github.com/fluxcd/flux/issues/1164.

In the meantime, for some use cases you can run more than one Flux daemon and point them at different repos. If you do this, consider trimming the RBAC permissions you give each daemon's service account.

This Flux (daemon) operator project may be of use for managing multiple daemons.

7.2.2 Do I have to put my application code and config in the same git repo?

Nope, but they can be if you want to keep them together. Flux doesn't need to know about your application code, since it deals with container images (i.e., once your application code has already been built).

7.2.3 Is there any special directory layout I need in my git repo?

Nope. Flux doesn't place any significance on the directory structure, and will descend into subdirectories in search of YAMLs. Although kubectl works with JSON files, Flux will ignore JSON. It avoids directories that look like Helm charts.

If you have YAML files in the repo that *aren't* for applying to Kubernetes, use `--git-path` to constrain where Flux starts looking.

See also *requirements.md* for a little more explanation.

7.2.4 Why does Flux need a git ssh key with write access?

There are a number of Flux commands and API calls which will update the git repo in the course of applying the command. This is done to ensure that git remains the single source of truth.

For example, if you use the following `fluxctl` command:

```
fluxctl release --controller=deployment/foo --update-image=bar:v2
```

The image tag will be updated in the git repository upon applying the command.

For more information about Flux commands see *the fluxctl docs*.

7.2.5 Can I run Flux with readonly Git access?

Yes. You can use the `--git-readonly` command line argument. The Helm chart exposes this as `git.readonly`.

This will prevent Flux from trying to write to your repository. You should also provide a readonly SSH key; e.g., on GitHub, leave the `Allow write access` box unchecked when you add the deploy key.

7.2.6 Does Flux automatically sync changes back to git?

No. It applies changes to git only when a Flux command or API call makes them.

7.2.7 Will Flux delete resources when I remove them from git?

Flux has an garbage collection feature, enabled by passing the command-line flag `--sync-garbage-collection` to `fluxd`.

The garbage collection is conservative: it is designed to not delete resources that were not created by `fluxd`. This means it will sometimes *not* delete resources that *were* created by `fluxd`, when reconfigured. Read more about garbage collection *here*.

7.2.8 How do I give Flux access to an image registry?

Flux transparently looks at the image pull secrets that you attach to workloads and service accounts, and thereby uses the same credentials that Kubernetes uses for pulling each image. In general, if your pods are running, then Kubernetes has pulled the images, and Flux should be able to access them too.

There are exceptions:

- One way of supplying credentials in Kubernetes is to put them on each node; Flux does not have access to those credentials.
- In some environments, authorisation provided by the platform is used instead of image pull secrets:
  - Google Container Registry works this way; Flux will automatically attempt to use platform-provided credentials when scanning images in GCR.
  - Amazon Elastic Container Registry (ECR) has its own authentication using IAM. If your worker nodes can read from ECR, then Flux will be able to access it too.

To work around exceptional cases, you can mount a docker config into the Flux container. See the argument `--docker-config` in *the daemon arguments reference*.

For ECR, Flux requires access to the EC2 instance metadata API to obtain AWS credentials. Kube2iam, Kiam, and potentially other Kuberenetes IAM utilities may block pod level access to the EC2 metadata APIs. If this is the case, Flux will be unable to poll ECR for automated workloads.

- If you are using Kiam, you need to whitelist the following API routes:

```
--whitelist-route-regexp=(/latest/meta-data/placement/availability-zone|/latest/
↪dynamic/instance-identity/document)
```

- If you are using kube2iam, ensure the values of –iptables and –in-interface are configured correctly for your virtual network provider.

See also *Why are my images not showing up in the list of images?*

7.2.9 How often does Flux check for new images?

- Flux scans image registries for metadata as quickly as it can, given rate limiting; and,
- checks if any automated workloads needs updates every five minutes, by default.

The latter default is quite conservative, so you can try lowering it (it's set with the flag `--automation-interval`).

Please don't *increase* the rate limiting numbers (`--registry-rps` and `--registry-burst`) – it's possible to get blacklisted by image registries if you spam them with requests.

If you are using GCP/GKE/GCR, you will likely want much lower rate limits. Please see fluxcd/flux#1016 for specific advice.

7.2.10 How often does Flux check for new git commits (and can I make it sync faster)?

Short answer: every five minutes; and yes.

There are two flags that control how often Flux syncs the cluster with git. They are

- `--git-poll-interval`, which controls how often it looks for new commits
- `--sync-interval`, which controls how often it will apply what's in git, to the cluster, absent new commits.

Both of these have five minutes as the default. When there are new commits, it will run a sync then and there, so in practice syncs happen more often than `--sync-interval`.

If you want to be more responsive to new commits, then give a shorter duration for `--git-poll-interval`, so it will check more often.

It is less useful to shorten the duration for `--sync-interval`, since that just controls how often it will sync *without* there being new commits. Reducing it below a minute or so may hinder Flux, since syncs can take tens of seconds, leaving not much time to do other operations.

7.2.11 How do I use my own deploy key?

Flux uses a k8s secret to hold the git ssh deploy key. It is possible to provide your own.

First delete the secret (if it exists):

```
kubectl delete secret flux-git-deploy
```

Then create a new secret named `flux-git-deploy`, using your private key as the content of the secret (you can generate the key with `ssh-keygen -q -N "" -f /full/path/to/private_key`):

```
kubectl create secret generic flux-git-deploy --from-file=identity=/full/path/
to/private_key
```

Now restart `fluxd` to re-read the k8s secret (if it is running):

```
kubectl delete $(kubectl get pod -o name -l name=flux)
```

If you have installed flux through Helm, make sure to pass `--set git.secretName=flux-git-deploy` when installing/upgrading the chart.

7.2.12 How do I use a private git host (or one that's not github.com, gitlab.com, bitbucket.org, dev.azure.com, or vs-ssh.visualstudio.com)?

As part of using git+ssh securely from the Flux daemon, we make sure `StrictHostKeyChecking` is on in the SSH config. This mitigates against man-in-the-middle attacks.

We bake host keys for `github.com`, `gitlab.com`, `bitbucket.org`, `dev.azure.com`, and `vs-ssh.visualstudio.com` into the image to cover some common cases. If you're using another service, or running your own git host, you need to supply your own host key(s).

How to do this is documented in *"Using a private Git host"*.

7.2.13 Why does my CI pipeline keep getting triggered?

There's a couple of reasons this can happen.

The first is that Flux pushes commits to your git repo, and if that repo is configured to go through CI, usually those commits will trigger a build. You can avoid this by supplying the flag `--ci-skip` so that Flux's commit will append `[ci skip]` to its commit messages. Many CI systems will treat that as meaning they should not run a build for that commit. You can use `--ci-skip-message`, if you need a different piece of text appended to commit messages.

The other thing that can trigger CI is that Flux pushes a tag to the upstream git repo whenever it has applied new commits. This acts as a "high water mark" for Flux to know which commits have already been seen. The default name for this tag is `flux-sync`, but it can be changed with the flags `--git-sync-tag` and `--git-label`. The simplest way to avoid triggering builds is to exclude this tag from builds – how to do that will depend on how your CI system is configured.

Here's the relevant docs for some common CI systems:

- CircleCI
- TravisCI
- GitLab
- Bitbucket Pipelines
- Azure Pipelines

7.2.14 Can I restrict the namespaces that Flux can see or operate on?

Flux will only operate on the namespaces that its service account has access to; so the most effective way to restrict it to certain namespaces is to use Kubernetes' role-based access control (RBAC) to make a service account that has restricted access itself. You may need to experiment to find the most restrictive permissions that work for your case.

You will need to use the command-line flag `--k8s-allow-namespace` to enumerate the namespaces that Flux attempts to scan for workloads.

7.2.15 Can I change the namespace Flux puts things in by default?

Yes. The `fluxd` image has a "kubeconfig" file baked in, which specifies a default namespace of `"default"`. That means any manifest not specifying a namespace (in `.metadata.namespace`) will be given the namespace `"default"` when applied to the cluster.

You can override this by mounting your own "kubeconfig" file into the container from a configmap, and using the `KUBECONFIG` environment entry to point to it. The example deployment shows how to do this, in commented out sections – it needs extra bits of config in three places (the `volume`, `volumeMount`, and `env` entries).

The easiest way to create a suitable "kubeconfig" will be to adapt the file that is baked into the image. Save that locally as `my-kubeconfig`, edit it to change the default namespace, then create the configmap, in the same namespace you run Flux in, with something like:

```
kubectl create configmap flux-kubeconfig --from-file=config=./my-kubeconfig
```

Be aware that the expected location (`$HOME/.kube/`) of the `kubeconfig` file is *also* used by `kubectl` to cache API responses, and mounting from a configmap will make it read-only and thus effectively disable the caching. For that reason, take care to mount your configmap elsewhere in the filesystem, as the example shows.

7.2.16 Can I temporarily make Flux ignore a manifest?

Yes. The easiest way to do that is to use the following annotation in the manifest, and commit the change to git:

```
    fluxcd.io/ignore: true
```

To stop ignoring these annotated resources, you simply remove the annotation from the manifests in git.

Flux will ignore any resource that has the annotation *either* in git, or in the cluster itself; sometimes it may be easier to annotate a *running resource in the cluster* as opposed to committing a change to git.

Mixing both kinds of annotations (in git, and in the cluster), can make it a bit hard to figure out how/where to undo the change (cf flux#1211). If the annotation exists in either the cluster or in git, it will be respected, so you may need to remove it from both places.

7.2.17 How can I prevent Flux overriding the replicas when using HPA?

When using a horizontal pod autoscaler you have to remove the `spec.replicas` from your deployment definition. If the replicas field is not present in Git, Flux will not override the replica count set by the HPA.

7.2.18 Can I disable Flux registry scanning?

You can completely disable registry scanning by using the `--registry-disable-scanning` flag. This allows deploying Flux without Memcached.

If you only want to scan certain images, don't set `--registry-disable-scanning`. Instead, you can tell Flux what images to include or exclude by supplying a list of glob expressions to the `--registry-include-image` and `--registry-exclude-image` flags:

- `--registry-exclude-image` takes patterns to be excluded; the default is to exclude the Kubernetes base images (`k8s.gcr.io/*`); and,
- `--registry-include-image` takes patterns to be included; no patterns (the default) means "include everything". If you provide a pattern, *only* images matching the pattern will be included (less any that are explicitly excluded).

To restrict scanning to only images from organisations `example` and `example-dev`, you might use:

```
--registry-include-image=*/example/*,*/example-dev/*
```

To exclude images from quay.io, use:

```
--registry-exclude-image=quay.io/*
```

Here are the Helm install equivalents (note the `\,` separator):

```
--set registry.includeImage="*/example/*\,*/example-dev/*" --set registry.
↪excludeImage="quay.io/*"
```

7.2.19 Does Flux support Kustomize/Templating/My favorite manifest factorization technology?

Yes!

Flux supports technology-agnostic manifest factorization through `.flux.yaml` configuration files placed in the Git repository. To enable it supply the command-line flag `--manifest-generation=true` to `fluxd`.

See *.flux.yaml configuration files documentation* for further details.

CHAPTER

EIGHT

TROUBLESHOOTING

Also see the issues labeled with `FAQ`, which often explain workarounds.

8.1 Flux is taking a long time to apply manifests when it syncs

If you notice that Flux takes tens of seconds or minutes to get through each sync, while you can apply the same manifests very quickly by hand, you may be running into this issue: fluxcd/flux#1422.

Briefly, the problem is that mounting a volume into `$HOME/.kube` effectively disables `kubectl`'s caching, which makes it much much slower. You may have used such a volume mount to override `$HOME/.kube/config`, possibly unknowingly – the Helm chart did this for you, prior to fluxcd/flux#1435.

The remedy is to mount the override to some other place in the filesystem, and use the environment entry `KUBECONFIG` to point `kubectl` at it. This is what the Helm chart now does, so fixing it may be as easy as reapplying the chart if that's what you're using.

This is also documented in the *FAQ*.

8.2 `fluxctl` returns a 500 Internal Server Error

This usually indicates there's a bug in the Flux daemon somewhere – in which case please tell us about it!

8.3 Flux answers everything with `git repo is not configured`

This means Flux can't read from and write to the git repo. Check that

- ... you've supplied a git repo URL. If it's of the form `https://github.com/user/repo` then you will need to use the SSH-style URL, `git@github.com:user/repo` instead.
- ... the deploy key has read/write access to the repo. In GitHub, deploy keys are installed in the settings for a repository. To get the deploy key Flux is using, use `fluxctl identity`.
- ... that the host where your git repo lives is in `~/.ssh/known_hosts` in the fluxd container. We prime the container *image* with host keys for `github.com`, `gitlab.com`, `bitbucket.org`, `dev.azure.com`, and `vs-ssh.visualstudio.com`, but if you're using your own git server, you'll need to add its host key. See *"Using a private Git host"*.

8.4 I'm using GCR/GKE and I keep seeing "Quota exceeded" in logs

GCP (in general) has quite conservative API rate limiting, and Flux's default settings can bump API usage over the limits. See fluxcd/flux#1016 for advice.

8.5 Flux doesn't seem to be able to use my imagePullSecrets

If you're using `kubectl` v1.13.x to create them, then it may be due to this problem. In short, there was a breaking change to how `kubectl` creates secrets, that found its way into the Kubernetes 1.13.0 release. It has been corrected in kubectl v1.13.2, so using that version or newer to create secrets should fix the problem.

8.6 Why are my images not showing up in the list of images?

Sometimes, instead of seeing the various images and their tags, the output of `fluxctl list-images` shows nothing. There's a number of reasons this can happen:

- Flux just hasn't fetched the image metadata yet. This may be the case if you've only just started using a particular image in a workload.
- Flux can't get suitable credentials for the image repository. At present, it looks at `imagePullSecrets` attached to workloads, service accounts, platform-provided credentials on GCP, AWS or Azure, and a Docker config file if you mount one into the `fluxd` container (see the *command-line usage*).
- When using images in ECR, from EC2, the `NodeInstanceRole` for the worker node running `fluxd` must have permissions to query the ECR registry (or registries) in question. `eksctl` and `kops` (with `.iam.allowContainerRegistry=true`) both make sure this is the case.
- When using images from ACR in AKS, the HostPath `/etc/kubernetes/azure.json` should be mounted into the Flux Pod. Set `registry.acr.enabled=True` in the helm chart or alter the Deployment:

```
spec:
  containers:
    image: docker.io/fluxcd/flux
    ...
    volumeMounts:
    - name: acr-credentials
      mountPath: /etc/kubernetes/azure.json
      readOnly: true
  volumes:
  - name: acr-credentials
    hostPath:
      path: /etc/kubernetes/azure.json
      type: ""
```

  If you encounter permission errors, you can alternatively create a secret `acr-credentials` based on the `azure.json` file and set `registry.acr.secretName=acr-credentials`.

- Flux excludes images with no suitable manifest (linux amd64) in manifestlist
- Flux doesn't yet understand image refs that use digests instead of tags; see fluxcd/flux#885.

If none of these explanations seem to apply, please file an issue.

8.7 Why do my image tags appear out of order?

You may notice that the ordering given to image tags does not always correspond with the order in which you pushed the images. That's because Flux sorts them by the image creation time; and, if you have retagged an older image, the creation time won't correspond to when you pushed the image. (Why does Flux look at the image creation time? In general there is no way for Flux to retrieve the time at which a tag was pushed from an image registry.)

This can happen if you explicitly tag an image that already exists. Because of the way Docker shares image layers, it can also happen *implicitly* if you happen to build an image that is identical to an existing image.

If this appears to be a problem for you, one way to ensure each image build has its own creation time is to label it with a build time; e.g., using OpenContainers pre-defined annotations.

8.8 What is the "sync tag"; or, why do I see a `flux-sync` tag in my git repo?

Flux keeps track of the last commit that it's applied to the cluster, by pushing a tag (controlled by the command-line flags `--git-sync-tag` and `--git-label`) to the git repository. This gives it a persistent high water mark, so even if it is restarted from scratch, it will be able to tell where it got to.

Technically, it only needs this to be able to determine which image releases (including automated upgrades) it has applied, and that only matters if it has been asked to report those with the `--connect` flag. Future versions of Flux may be more sparing in use of the sync tag.

8.9 Flux fails with an error log similar to *couldn't get resource list for example.com/version: the server is currently unable to handle the request*

This means your Kubernetes cluster fails to respond to list queries for resources in *example.com/version*.

If the error is transient, Flux will work once the error recedes.

However, the error won't normally go away since most of the time it's caused by a misconfiguration of your cluster.

For instance, you can run into this problem:

- When a Kubernetes Webhook server is removed without removing its Webhook definition.
- When a custom resource definition (CRD) is not available due to a `FailedDiscoveryCheck` error.

We recommend trying to address the root cause by fixing your cluster configuration. In the examples above, you would need to remove the Webhook definition or add the CRD.

However, fixing your cluster configuration may not always be possible. The problem is common enough that Flux provides a flag called `--k8s-unsafe-exclude-resource`. The name says it all, you should only use it if you know what you are doing.

`--k8s-unsafe-exclude-resource` will tell Flux to avoid querying the cluster for those resources. This in turn means that Flux won't take into account those excluded cluster resources when syncing. This can cause excluded resources:

- to be unexpectedly overwritten by their corresponding definition in Git during a sync (even if they are annotated with `flux.weave.works/ignore: "true"` on the cluster-side).
- not to be garbage-collected.

The rule of thumb is that you can use `--k8s-unsafe-exclude-resource` on resources not matching any manifests in your Git repository.

CHAPTER

NINE

CONTRIBUTING

9.1 Get started developing

This guide shows a workflow for making a small (actually, tiny) change to Flux, building and testing that change locally.

9.1.1 TL;DR

From a very high level, there are at least 3 ways you can develop on Flux once you have your environment set up:

1. The "minimalist" approach (only requires and `kubectl`):
    1. `make`
    2. copy the specific image tag (e.g. `docker.io/fluxcd/flux:master-a86167e4`) for what you just built and paste it into `/deploy/flux-deployment.yaml` as the image you're targeting to deploy
    3. deploy the resources in `/develop/*.yaml` manually with `kubectl apply`
    4. make a change to the code
    5. see your code changes have been deployed
    6. repeat
2. Use `freshpod` to deploy changes to the `/deploy` directory resources:
    1. `make`
    2. make a change to the code
    3. see your changes have been deployed
    4. repeat
3. Remote cluster development approach:
    1. ensure local `kubectl` access to a remote Kubernetes cluster
    2. have an available local memcached instance
    3. make a change to the code
    4.
```
go run cmd/fluxd/main.go \
    --memcached-hostname localhost  \
    --memcached-port 11211 \
    --memcached-service "" \
```

(continues on next page)

(continued from previous page)

```
    --git-url git@github.com:fluxcd/flux-get-started \
    --k8s-in-cluster=false
```

This guide covers approaches 1 and 2 using `minikube`. `freshpod` is superseded by `Skaffold` and is generally the future. That said, `freshpod` is very simple to use and reason about (and is still well supported by `minikube`) which is why it's used in this guide.

9.1.2 Run `fluxcd/flux-getting-started`

We're going to make some changes soon enough, but just to get a good baseline please follow the *"Get started with Flux"* tutorial and run the `fluxcd/flux-getting-started` repo through its normal paces.

Now that we know everything is working with `flux-getting-started`, we're going to try and do nearly the same thing as `flux-getting-started`, except instead of using official releases of flux, we're going to build and run what we have locally.

9.1.3 Prepare your environment

1. Install the prerequisites. This guide is written from running Linux, but the same instructions will generally apply to OSX. Although everything you need has been known to work independently in Windows from time to time, results may vary.
   - `minikube`
   - `kubectl`
   - `docker`
   - `go`
2. Configure your environment so you can run tests. Run:

   ```
   make test
   ```

3. We want to make sure we're starting fresh. Tell `minikube` to clear any previously running clusters:

   ```
   minikube delete
   ```

4. The `minikube` addon called freshpod that will be very useful to us later. You'll see. It's gonna be cool.

   ```
   minikube addons enable freshpod
   ```

5. This part is really important. You're going to set some environment variables which will intercept any images pulled by docker. Run `minikube docker-env` to see what we're talking about. You'll get an output that shows you what the script is doing. Thankfully, it's not terribly complicated - it just sets some environment variables which will allow `minikube` to man-in-the-middle the requests Kubernetes makes to pull images. It will look something like this:

   ```
   export DOCKER_TLS_VERIFY="1"
   export DOCKER_HOST="tcp://192.168.99.128:2376"
   export DOCKER_CERT_PATH="/home/fluxrulez/.minikube/certs"
   export DOCKER_API_VERSION="1.35"
   # Run this command to configure your shell:
   # eval $(minikube docker-env)
   ```

   So, as the script suggests, run the following command:

```
eval $(minikube docker-env)
```

Now, be warned. These are local variables. This means that if you run this `eval` in one terminal and then switch to another for later when we build the Flux project, you're gonna hit some issues. For one, you'll know it isn't working because Kubernetes will tell you that it can't pull the image when you run `kubectl get pods`:

```
NAME                          READY   STATUS         RESTARTS   AGE
flux-7f6bd57699-shx9v         0/1     ErrImagePull   0          35s
```

9.1.4 Prepare the repository

1. Fork the repo on GitHub.
2. Clone `git@github.com:<YOUR-GITHUB-USERNAME>/flux.git` replacing `<YOUR-GITHUB-USERNAME>` with your GitHub username.

   In the same terminal you ran `eval $(minikube docker-env)`, run `GO111MODULE=on go mod download` followed by `make` from the root directory of the Flux repo. You'll see docker's usual output as it builds the image layers. Once it's done, you should see something like this in the middle of the output:

   ```
   Successfully built 606610e0f4ef
   Successfully tagged docker.io/fluxcd/flux:latest
   Successfully tagged docker.io/fluxcd/flux:master-a86167e4
   ```

   This confirms that a new docker image was tagged for your image.
3. Open up `deploy/flux-deployment.yaml` and update the image at `spec.template.spec.containers[0].image` to be simply `docker.io/fluxcd/flux`. While we're here, also change the `--git-url` to point towards your fork. It will look something like this in the YAML:

   ```
   spec:
     template:
       spec:
         containers:
         - name: flux
           image: docker.io/fluxcd/flux
           imagePullPolicy: IfNotPresent
           args:
             - --git-url=git@github.com:<YOUR-GITHUB-USERNAME>/flux-getting-started
             - --git-branch=master
   ```
4. We're ready to apply your newly-customized deployment! Since `kubectl` will apply all the Kubernetes manifests it finds (recursively) in a folder, we simply need to pass the directory to `kubectl apply`:

   ```
   kubectl apply --filename ./deploy
   ```

   You should see an output similar to:

   ```
   serviceaccount/flux created
   clusterrole.rbac.authorization.k8s.io/flux created
   clusterrolebinding.rbac.authorization.k8s.io/flux created
   deployment.apps/flux created
   secret/flux-git-deploy created
   deployment.apps/memcached created
   service/memcached created
   secret/flux-git-deploy configured
   ```

Congrats you just deployed your local Flux to your default namespace. Check that everything is running:

```
kubectl get pods --selector=name=flux
```

You should get an output that looks like:

```
NAME                    READY   STATUS    RESTARTS   AGE
flux-6f7fd5bbc-hpq85    1/1     Running   0          38s
```

If (instead) you see that Ready is showing `0/1` and/or the status is `ErrImagePull` double back on the instructions and make sure you did everything correctly and in order.

5. Pull the logs for your "fresh off of master" copy of Flux that you just deployed locally to `minikube`:

```
kubectl logs --selector=name=flux
```

You should see an output that looks something like this:

```
ts=2019-02-28T18:58:45.091531939Z caller=warming.go:268 component=warmer info=
↪"refreshing image" image=docker.io/fluxcd/flux tag_count=60 to_update=60 of_
↪which_refresh=0 of_which_missing=60
ts=2019-02-28T18:58:46.233723421Z caller=warming.go:364 component=warmer␣
↪updated=docker.io/fluxcd/flux    successful=60 attempted=60
ts=2019-02-28T18:58:46.234086642Z caller=images.go:17 component=sync-loop msg=
↪"polling images"
ts=2019-02-28T18:58:46.234125646Z caller=images.go:27 component=sync-loop msg="no␣
↪automated services"
ts=2019-02-28T18:58:46.749598558Z caller=warming.go:268 component=warmer info=
↪"refreshing image" image=memcached    tag_count=66 to_update=66 of_which_
↪refresh=0 of_which_missing=66
ts=2019-02-28T18:58:51.017452675Z caller=warming.go:364 component=warmer␣
↪updated=memcached successful=66 attempted=66
ts=2019-02-28T18:58:51.020061586Z caller=images.go:17 component=sync-loop msg=
↪"polling images"
ts=2019-02-28T18:58:51.020113243Z caller=images.go:27 component=sync-loop msg="no␣
↪automated services"
```

9.1.5 Make some changes

1. Now for the part you've been waiting for! We're going to make a cosmetic change to our local copy of Flux. Navigate to git/operations.go. In it, you will find a private function to this package that goes by the name `execGitCmd`. Paste the following as the (new) first line of the function:

```
fmt.Println("executing git command␣
↪ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ")
```

2. Run `make` again. Once this finishes you can check on your running pods with:

```
kubectl get pods --selector=name=flux
```

Keep your eye on the `AGE` column. It should be just a few seconds old if you check out the `AGE` column:

```
NAME                    READY   STATUS    RESTARTS   AGE
flux-6f7fd5bbc-6j9d5    1/1     Running   0          10s
```

This pod was deployed even though we didn't run any `kubectl` commands or interact with Kubernetes directly because of the `freshpod minikube` addon that we enabled earlier. Freshpod saw that a new Docker image was tagged for `docker.io/fluxcd/flux:latest` and it went ahead and redeployed that pod for us.

Consider that simply applying the `flux-deployment.yaml` file again wouldn't do anything since the actual image we're targeting (which is actually `docker.io/fluxcd/flux` with no `:latest` tag, but it's the same difference) hasn't changed. The Kubernetes api server will get that JSON request from `kubectl` and go: "right... so nothing has changed in the file so I have nothing to do... IGNORE!".

There is another way to do this, of course. Remember that before when we ran `make` that we did *also* get an image tagged with the `:<branch>-<commit hash>` syntax (in our specific example above it was `:master-a86167e4`). We could, in theory, grab that tag every time we `make`, and then paste it into `spec.template.spec.containers[0].image` of our deployment. That's tedious and error prone. Instead, `freshpod` cuts this step out for us and accomplishes the same end goal.

3. Check the logs again (with `kubectl logs --selector=name=flux`) to find that your obnoxious chain of `Z`s is present.

9.1.6 Congratulations!

You have now modified Flux and deployed that change locally. From here on out, you simply need to run `make` after you save your changes and wait a few seconds for your new pod to be deployed to `minikube`. Keep in mind, that (as in the situation where you run `make` without saving any changes) if the Docker image you pointed to in the Kubernetes deployment for Flux is not Successfully tagged, `freshpod` won't have anything new to deploy. Other than that, you should be good to go!

9.2 Building

You'll need a working `go` environment version >= 1.11 (official releases are built against `1.13`). It's also expected that you have a Docker daemon for building images.

Clone the repository. The project uses Go Modules, so if you explicitly define `$GOPATH` you should clone somewhere else.

Then, from the root directory:

```
make
```

This makes Docker images, and installs binaries to `$GOBIN` (if you define it) or `$(go env GOPATH)/bin`.

> **Note:** the default target architecture is amd64. If you would like to try to build Docker images and binaries for a different architecture you will have to set ARCH variable:
>
> ```
> $ make ARCH=<target_arch>
> ```

9.2.1 Running tests

```
# Unit tests
make test

# End-to-end tests
make e2e
```