5/1/2021

GitOps With FluxCD

By: Eng. Mohamed ElEmam Hussein
Email: Mohamed.ElEmam.Hussin@gmail.com

GitOps With FluxCD

What is GitOps?

GitOps is a way of implementing Continuous Deployment for cloud native applications. It focuses on a developer-centric experience when operating infrastructure, by using tools developers are already familiar with, including Git and Continuous Deployment tools.

The core idea of GitOps is having a Git repository that always contains declarative descriptions of the infrastructure currently desired in the production environment and an automated process to make the production environment match the described state in the repository. If you want to deploy a new application or update an existing one, you only need to update the repository - the automated process handles everything else. It's like having cruise control for managing your applications in production.

GitOps can be summarized as these two things:

1. An operating model for Kubernetes and other cloud native technologies, providing a set of best practices that unify Git deployment, management and monitoring for containerized clusters and applications.
2. A path towards a developer experience for managing applications; where end-to-end CICD pipelines and Git workflows are applied to both operations, and development.

Why should I use GitOps?

1. **Deploy Faster**

probably every Continuous Deployment technology promises to make deploying faster and allows you to deploy more often. What is unique about GitOps is that you don't have to switch tools for deploying your application. Everything happens in the version control system you use for developing the application anyways.

2. Easy and Fast Error Recovery

Your production environment is down! With GitOps you have a complete history of how your environment changed over time. This makes error recovery as easy as issuing a git revert and watching your environment being restored.

3. Easier Credential Management

GitOps allows you to manage deployments completely from inside your environment. For that, your environment only needs access to your repository and image registry. That's it. You don't have to give your developers direct access to the environment.

4. Self-documenting Deployments

Have you ever SSH'd into a server and wondered what's running there? With GitOps, every change to any environment must happen through the repository. You can always check out the master branch and get a complete description of what is deployed where plus the complete history of every change ever made to the system. And you get an audit trail of any changes in your system for free!

5. Shared Knowledge in Teams

Using Git to store complete descriptions of your deployed infrastructure allows everybody in your team to check out its evolution over time. With great commit messages everybody can reproduce the thought process of changing infrastructure and also easily find examples of how to set up new systems.

Principles of GitOps

To start managing your cluster with GitOps workflows, the following must be in place:

1. The entire system described declaratively.

With Gitops, Kubernetes is just one example of many modern cloud native tools that are "declarative" and that can be treated as code. Declarative means that configuration is guaranteed by a set of facts instead of by a set of instructions. With your application's declarations versioned in Git, you have a single source of truth. Your apps can then be easily deployed and rolled back to and from Kubernetes. And even more importantly, when disaster strikes, your cluster's infrastructure can also be dependably and quickly reproduced.

2. The canonical desired system state versioned in Git.

With the declaration of your system stored in a version control system, and serving as your canonical source of truth, you have a single place from which everything is derived and driven. This trivializes rollbacks; where you can use a `Git revert` to go back to your previous application state. With Git's excellent security guarantees, you can also use your SSH key to sign commits that enforce strong security guarantees about the authorship and provenance of your code.

3. Approved changes that can be automatically applied to the system.

Once you have the declared state kept in Git, the next step is to allow any changes to that state to be automatically applied to your system. What's significant about this is that you don't need cluster credentials to make a change to your system. With GitOps, there is a segregated environment of which the state definition lives outside. This allows you to separate what you do and how you're going to do it.

4. Software agents to ensure correctness and alert on divergence.

Once the state of your system is declared and kept under version control, software agents can inform you whenever reality doesn't match your expectations. The use of agents also ensures that your entire system is self-healing. And by self-healing, we don't just mean when nodes or pods fail—those are handled by Kubernetes—but in a broader sense, like in the case of human error. In this case, software agents act as the feedback and control loop for your operations.

How does GitOps work?

GitOps organizes the deployment process around code repositories as the central element. There are at least two repositories: the application repository and the environment configuration repository. The application repository contains the source code of the application and the deployment manifests to deploy the application. The environment configuration repository contains all deployment manifests of the currently desired infrastructure of an deployment environment. It describes what applications and infrastructural services (message broker, service mesh, monitoring tool, ...) should run with what configuration and version in the deployment environment.

Push-based vs. Pull-based Deployments

There are two ways to implement the deployment strategy for GitOps: Push-based and Pull-based deployments. The difference between the two deployment types is how it is ensured, that the deployment environment actually resembles the desired infrastructure. When possible, the Pull-based approach should be preferred as it is considered the more secure and thus better practice to implement GitOps.

Push-based Deployments

The Push-based deployment strategy is implemented by popular CI/CD tools such as Jenkins, CircleCI, or Travis CI. The source code of the application lives inside the application repository along with the Kubernetes YAMLs needed to deploy the app. Whenever the application code is updated, the build pipeline is triggered, which builds the container images and finally the environment configuration repository is updated with new deployment descriptors.

Pull-based Deployments

The Pull-based deployment strategy that used by GitOps is uses the same concepts as the push-based variant but differs in how the deployment pipeline works. Traditional CI/CD pipelines are triggered by an external event, for example when new code is pushed to an application repository. With the pull-based deployment approach, the ***operator*** is introduced. It takes over the role of the pipeline by continuously comparing the desired state in the environment repository with the actual state in the deployed infrastructure. Whenever differences are noticed, the operator updates the infrastructure to match the environment repository. Additionally the image registry can be monitored to find new versions of images to deploy.

Now let's have a look on kustomize ?!

What is Kustomize?

Deploying applications to Kubernetes can sometimes feel cumbersome. You deploy some Pods, backed by a Deployment, with accessibility defined in a Service. All of these resources require YAML files for proper definition and configuration.

On top of this, your application might need to communicate with a database, manage web content, or set logging verbosity. Further, these parameters may need to differ depending on the environment to which you are deploying. All of this can result in a sprawling codebase of YAML definitions, each with one- or two-line changes that are difficult to pinpoint.

Kustomize is an open-source configuration management tool developed to help address these concerns. Since Kubernetes 1.14, kubectl fully supports Kustomize and kustomization files.

How to use kustomize?

you will deploy a development version of sammy-app—a static web application hosted on Nginx. You will store your web content as data in a ConfigMap, which you will mount on a Pod in a Deployment. Each of these will require a separate YAML file, which you will now create.

First, make a folder for your application and all of its configuration files.

```
root@k8s-master01:~# mkdir ~/sample-app && cd ~/ sample-app
```

Now use your preferred text editor to create and open a file called configmap.yml:

```
root@k8s-master01:~/sample-app# vim configmap.yml
```

Add the following content:

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: sample-app
  namespace: default
data:
  body: >
    <html>
      <style>
        body {
          background-color: #222;
        }
        p {
          font-family:"Courier New";
          font-size:xx-large;
          color:#f22;
          text-align:center;
        }
      </style>
      <body>
        <p>DEVELOPMENT</p>
      </body>
    </html>
```

This specification creates a new ConfigMap object. You are naming it sample-app and saving some HTML web content inside data:.

Save and close the file.

Second create and open a second file called deployment.yml:

```
root@k8s-master01:~/sample-app# vim deployment.yml
```

Add the following content:

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-app
  namespace: default
  labels:
    app: sample-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: sample-app
  template:
    metadata:
      labels:
        app: sample-app
    spec:
      containers:
      - name: server
        image: nginx:1.17
        volumeMounts:
          - name: sample-app
            mountPath: /usr/share/nginx/html
        ports:
        - containerPort: 80
          protocol: TCP
        resources:
          requests:
            cpu: 100m
            memory: "128M"
          limits:
            cpu: 100m
            memory: "256M"
        env:
        - name: LOG_LEVEL
          value: "DEBUG"
      volumes:
      - name: sample-app
        configMap:
          name: sample-app
          items:
          - key: body
            path: index.html
```

This specification creates a new Deployment object. You are adding the name and label of sample-app, setting the number of replicas to 1, and specifying the object to use the Nginx version 1.17 container image. You are also setting the container's port to 80, defining cpu and memory requests and limitations, and setting your logging level to DEBUG.

Save and close the file.

Third Create and open a third YAML file called service.yml:

```
root@k8s-master01:~/sample-app# vim service.yml
```

Add the following content:

```
---
apiVersion: v1
kind: Service
metadata:
  name: sample-app
  labels:
    app: sample-app
spec:
  type: LoadBalancer
  ports:
  - name: sample-app-http
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: sample-app
```

Save and close the file.

To deploy an application in the form that Kustomize expects.

Your filesystem currently looks like this:

```
sample-app/
├── configmap.yml
├── deployment.yml
├── kustomization.yml
└── service.yml
```

To make this application deployable with Kustomize, you need to add one file, kustomization.yml. Do so now:

```
root@k8s-master01:~/sample-app# vim kustomization.yml
```

At a minimum, this file should specify what resources to manage when running kubectl with the -k option, which will direct kubectl to process the kustomization file.

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
- configmap.yml
- deployment.yml
- service.yml
```

Save and close the file.

Then to deploy

```
root@k8s-master01:~/sample-app# kubectl apply -k .
```

Instead of providing the -f option to kubectl to direct Kubernetes to create resources from a file, you provide -k and a directory (in this case, . denotes the current directory). This instructs kubectl to use Kustomize and to inspect that directory's kustomization.yml.

That's it !! This creates all three resources: the ConfigMap, Deployment, and Service. 😉

Managing Application Variance with Kustomize

Configuration files for Kubernetes resources can really start to sprawl once you start dealing with multiple resource types, especially when there are small differences between environments (like development versus production, for example). You might have a deployment-development.yml and deployment-production.yml instead of just a deployment.yml. The situation might be similar for all of your other resources, too.

Imagine what might happen when a new version of the Nginx Docker image is released, and you want to start using it. Perhaps you test the new version in deployment-development.yml and want to proceed, but then you forget to update deployment-production.yml with the new version. Suddenly, you're running a different version of Nginx in development than you are in production. Small configuration errors like this can quickly break your application.

Kustomize can greatly simplify these management issues. Remember that you now have a filesystem with your Kubernetes configuration files and a kustomization.yml:

```
sample-app/
├── configmap.yml
├── deployment.yml
├── kustomization.yml
└── service.yml
```

Imagine that you are now ready to deploy sample-app to production. You've also decided that the production version of your application will differ from its development version in the following ways:

- replicas will increase from 1 to 3.
- container resource requests will increase from 100m CPU and 128M memory to 250m CPU and 256M memory.
- container resource limits will increase from 100m CPU and 256M memory to 1 CPU and 1G memory.
- the LOG_LEVEL environment variable will change from DEBUG to INFO.
- ConfigMap data will change to display slightly different web content.

To begin, create some new directories to organize things in a more Kustomize-specific way:

```
root@k8s-master01:~/sample-app# mkdir base
```

This will hold your "default" configuration—your base. In your example, this is the development version of sample-app.

Now move your current configuration in sample-app/ into this directory:

```
root@k8s-master01:~/sample-app# mv configmap.yml deployment.yml service.yml

kustomization.yml base/
```

Then make a new directory for your production configuration. Kustomize calls this an overlay. Think of overlays as layers on top of the base—they always require a base to function:

```
root@k8s-master01:~/sample-app# mkdir -p overlays/production
```

Create another kustomization.yml file to define your production overlay:

```
root@k8s-master01:~/sample-app# vim overlays/production/kustomization.yml
```

Add the following content:

```
---
bases:
- ../../base
patchesStrategicMerge:
- configmap.yml
- deployment.yml
```

This file will specify a base for the overlay and what strategy Kubernetes will use to patch the resources. In this example, you will specify a strategic-merge-style patch to update the ConfigMap and Deployment resources.

Save and close the file.

And finally, add new deployment.yml and configmap.yml files into the overlays/production/ directory.

Create the new deployment.yml file first:

```
root@k8s-master01:~/sample-app# vim overlays/production/deployment.yml
```

Add the following to your file. The highlighted sections denote changes from your development configuration:

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-app
  namespace: default
spec:
  replicas: 3
  template:
    spec:
      containers:
      - name: server
        resources:
          requests:
            cpu: 250m
            memory: "256M"
          limits:
            cpu: 1
            memory: "1G"
        env:
```

```
        - name: LOG_LEVEL
          value: "INFO"
```

Notice the contents of this new deployment.yml. It contains only the TypeMeta fields used to identify the resource that changed (in this case, the Deployment of your application), and just enough remaining fields to step into the nested structure to specify a new field value, e.g., the container resource requests and limits.

Save and close the file.

Now create a new configmap.yml for your production overlay:

```
root@k8s-master01:~/sample-app# vim /overlays/production/configmap.yml
```

Add the following content:

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: sample-app
  namespace: default
data:
  body: >
    <html>
      <style>
        body {
          background-color: #222;
        }
        p {
          font-family:"Courier New";
          font-size:xx-large;
          color:#22f;
          text-align:center;
        }
      </style>
      <body>
        <p>PRODUCTION</p>
      </body>
    </html>
```

Here you have changed the text to display PRODUCTION instead of DEVELOPMENT. Note that you also changed the text color from a red hue #f22 to a blue hue #22f. Consider how difficult it could be to locate and track such minor changes if you were not using a configuration management tool like Kustomize.

Your directory structure now looks like this:

```
sample-app/
├── base
│   ├── configmap.yml
│   ├── deployment.yml
│   ├── kustomization.yml
│   └── service.yml
└── overlays
    └── production
        ├── configmap.yml
        ├── deployment.yml
        └── kustomization.yml
```

You are ready to deploy using your base configuration. First, delete the existing resources:

```
root@k8s-master01:~/sample-app# kubectl delete deployment/sample-app service/ sample-
app configmap/ sample-app
```

Deploy your base configuration to Kubernetes:

```
root@k8s-master01:~/sample-app# kubectl apply -k base/
```

Inspect your deployment and will see the expected base configuration, with the development version visible on the EXTERNAL-IP of the Service:

```
root@k8s-master01:~/sample-app# kubectl get pods,services -l app=sample-app
Output
NAME                              READY   STATUS    RESTARTS   AGE
pod/sample-app-5668b6dc75-rwbtq    1/1     Running   0          21s

NAME                 TYPE           CLUSTER-IP       EXTERNAL-IP            PORT(S)
AGE
service/sample-app    LoadBalancer   10.245.110.172   your_external_ip   80:31764/TCP
7m43s
```

Now Deploy your production configuration:

```
root@k8s-master01:~/sample-app# kubectl apply -k overlays/production/
```

Inspect your deployment again:

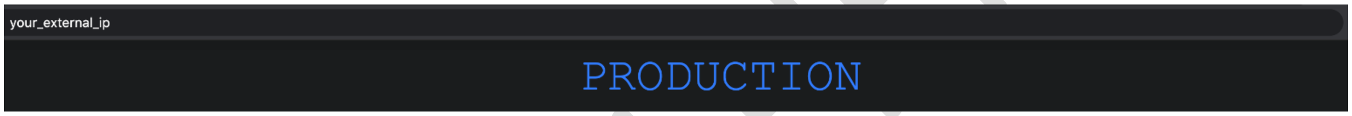
```
root@k8s-master01:~/sample-app# kubectl get pods,services -l app=sample-app
Output

NAME                              READY   STATUS    RESTARTS   AGE

pod/sample-app-86759677b4-h5ndw    1/1     Running   0          15s
```

```
pod/sample-app-86759677b4-t2dml    1/1     Running   0          17s

pod/sample-app-86759677b4-z56f8    1/1     Running   0          13s


NAME                    TYPE           CLUSTER-IP       EXTERNAL-IP          PORT(S)
AGE

service/sample-app    LoadBalancer    10.245.110.172    your_external_ip    80:31764/TCP
8m59s
```

Notice in the production configuration that there are 3 Pods in total instead of 1. You can view the Deployment resource to confirm that the less-apparent changes have taken effect, too:

```
root@k8s-master01:~/sample-app# kubectl get deployments -l app=sample-app -o yaml
```

Visit your_external_ip in a browser to view the production version of your site.



You are now using Kustomize to manage application variance. Thinking back to one of your original problems, if you now wanted to change the Nginx image version, you would only need to modify deployment.yml in the base, and your overlays that use that base will also receive that change through Kustomize. This greatly simplifies your development workflow, improves readability, and reduces the likelihood of errors.

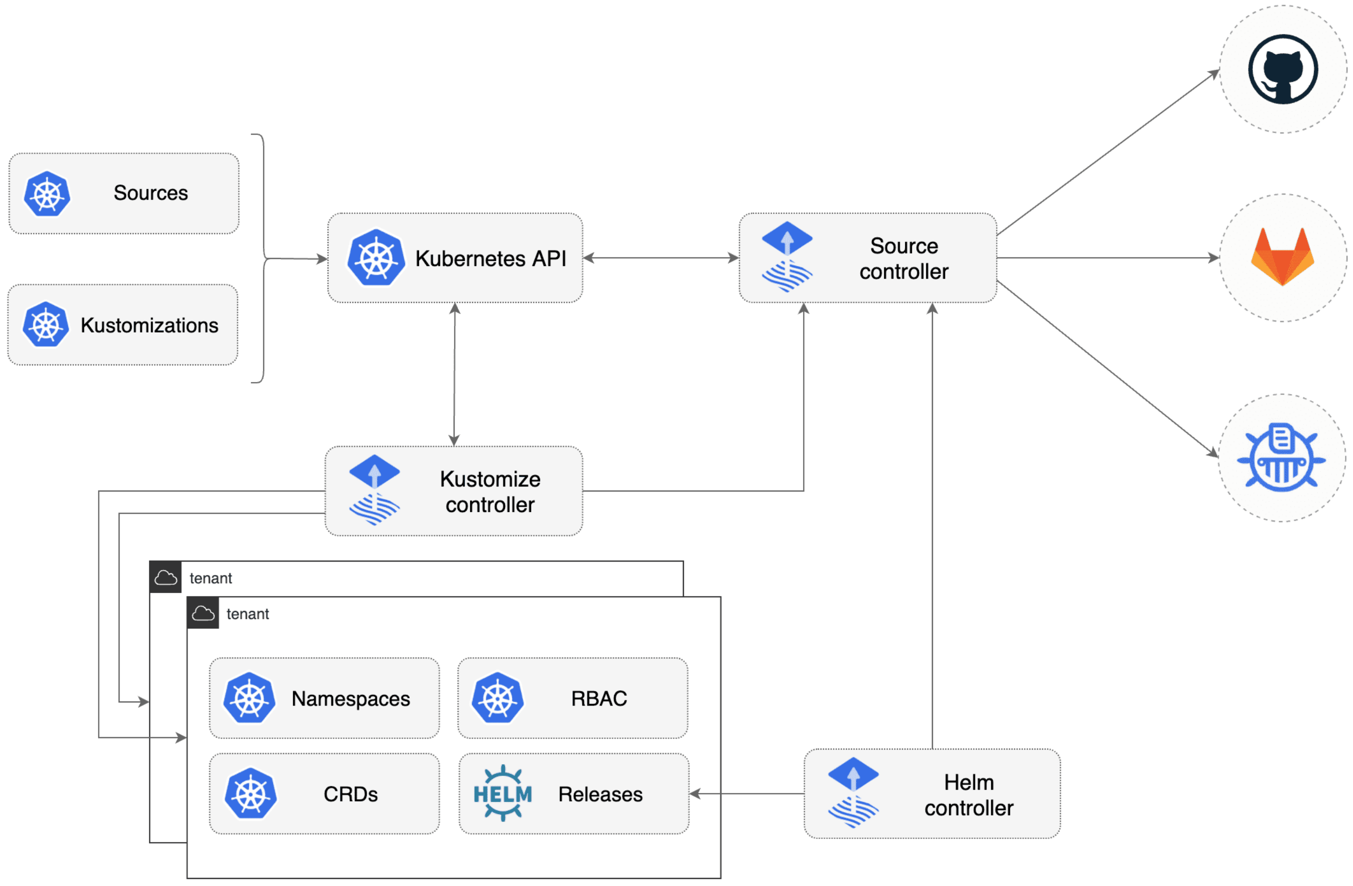
What is FluxCD?

Flux is described as a GitOps operator for Kubernetes that synchronises the state of manifests in a Git repository to what is running in a cluster. Among the three tools in this evaluation, it is by far the simplest one. In fact, it’s amazing to see how a GitOps workflow can be set up with only a few steps.

Flux is based on a set of Kubernetes API extensions (“custom resources”), which control how git repositories and other sources of configuration are applied into the cluster (“synced”). For example, you create a GitRepository object to mirror configuration from a Git repository, then a Kustomization object to sync that configuration.

Flux works with Kubernetes' role-based access control (RBAC), so you can lock down what any particular sync can change. It can send notifications to Slack and other like systems when configuration is synced and ready, and receive webhooks to tell it when to sync.

The flux command-line tool is a convenient way to bootstrap the system in a cluster, and to access the custom resources that make up the API.



Flux Toolkit components

1. **Source Controller:** Watching for any change in the source (like GitHub)
2. **Kustomize Controller:** Reconciles the cluster state from multiple sources (provided by source-controller)
3. **Helm Controller:** Watches for HelmRelease objects and generates HelmChart objects.
4. **Notification Controller:** handles events coming from external systems (GitHub, GitLab, Bitbucket, Harbor, Jenkins, etc) and notifies the GitOps toolkit controllers about source changes.
5. **Image reflector and automation controllers:** Work together to update a Git repository when new container images are available.

Why Use Flux?

Once deployed, Flux synchronizes the Kubernetes manifests stored in your source control system with your Kubernetes clusters. This in turn uses periodical polling updating the cluster when changes are identified.

This automated approach removes the need to run kubectl commands and monitor your clusters to see if they have deployed the correct configuration and workloads!

The key benefits of deploying Flux are as follows:

- Your source control becomes a single source of the truth.
- Your source control becomes a central place for all of your environments, and configurations which are defined in code.
- All changes are observable and verified.

Core Concepts

Sources

A *Source* defines the origin of a repository containing the desired state of the system and the requirements to obtain it (e.g. credentials, version selectors). For example, the latest 1.x tag available from a Git repository over SSH.

Reconciliation

Reconciliation refers to ensuring that a given state (e.g. application running in the cluster, infrastructure) matches a desired state declaratively defined somewhere (e.g. a Git repository).

Kustomization

The Kustomization custom resource represents a local set of Kubernetes resources (e.g. kustomize overlay) that Flux is supposed to reconcile in the cluster. The reconciliation runs every one minute by default, but this can be changed with .spec.interval. If you make any changes to the cluster using kubectl edit/patch/delete, they will be promptly reverted. You either suspend the reconciliation or push your changes to a Git repository.

Bootstrap

The process of installing the Flux components in a GitOps manner is called a bootstrap. The manifests are applied to the cluster, a GitRepository and Kustomization are created for the Flux components, then the manifests are pushed to an existing Git repository (or a new one is created). Flux can manage itself just as it manages other resources. The bootstrap is done using the flux CLI or using Terraform Provider.

Helm Operator

The Helm Operator is a Kubernetes Operator, allowing one to declaratively manage Helm chart releases. The desired state of a Helm release is described through a Kubernetes Custom Resource named HelmRelease and HelmRepositories. Based on the creation, mutation, or removal of a HelmRelease resource in the cluster, Helm actions are performed by the operator.

HelmRepositories: has the repo for Helm Chart

HelmRelease: its like Deployment of the Helm Chart

How Does Flux Work?

Before we jump into deploying Flux, let's familiarize ourselves with how the platform works!

Flux configuration files are written in YAML manifests declaratively. These configuration files define everything required to deploy your application to your Kubernetes clusters. The steps for deployment & changes are as follows:

1. The team describes the Kubernetes cluster configuration and defines this in a manifest that resides in their source control platform which is normally git.
2. The memcached pod stores the running configuration.
3. Flux periodically connects to your source control platform and compares the running configuration from the memcached pod vs the source control manifest.
4. If changes are detected, Flux runs several kubectl apply and delete commands to bring the cluster in sync. The new running configuration is then saved to the memcached pod.

Flux can also poll container registries and update the Kubernetes manifest with the latest versions. This is very powerful should you wish to automate the deployment of new containers.

Installation

Flux installation is as easy as deploying any other typical pod in a cluster. It officially supports installations based on Helm Charts and Kustomize. There's also `fluxctl`, a command line interface (CLI) tool, released as a binary, to help with the installation process and to interact with a running Flux daemon in the cluster to perform some management tasks.

The simplest way to install Flux in a cluster is by applying a small set of manifests that can be generated using the `fluxctl install` command with the appropriate arguments, including the URL of the Git repository that will be watched.

Note: assuming that your Kubernetes cluster is version 1.16 or newer and your kubectl binary is version 1.18 or newer.

From master node run the following command

```
root@k8s-master01:~/sample-app#  curl -s https://toolkit.Fluxcd.io/install.sh | sudo bash
```

To confirm you have installed Flux and it's working run:

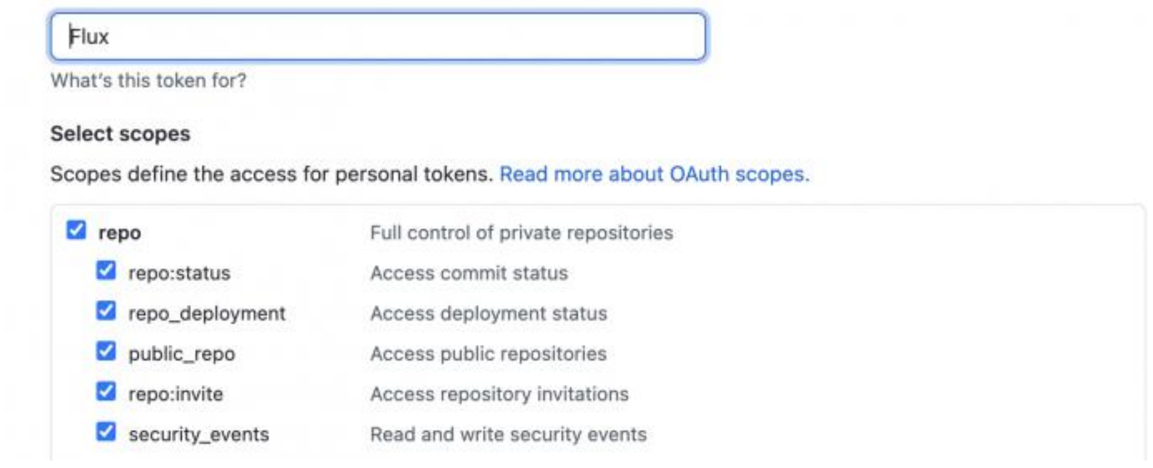
```
root@k8s-master01:~/sample-app#  flux -v
```

Bootstrapping

Before we use Flux to deploy to our cluster, we need to have a repository with our Kubernetes configuration. In this step, we are going to create a repo in GitHub and define our workspace and namespace within it.

First, let's create GitHub Token for Flux to use to deploy your new repo and link it to your cluster.

Generating a Github token is done on the Github site. Go to your profile and then select 'Developer Settings' on the left-hand side. From there select 'Personal access token' and then 'Generate new token'. Flux requires the permissions below:



Flux

What's this token for?

Select scopes

Scopes define the access for personal tokens. Read more about OAuth scopes.

✓ repo	Full control of private repositories
✓ repo:status	Access commit status
✓ repo_deployment	Access deployment status
✓ public_repo	Access public repositories
✓ repo:invite	Access repository invitations
✓ security_events	Read and write security events

Using the flux bootstrap command you can install Flux on a Kubernetes cluster and configure it to manage itself from a Git repository.

```
root@k8s-master01:~/sample-app# export GITHUB_USERNAME=EngMohamedElEmam
root@k8s-master01:~/sample-app# export CLUSTER_NAME=aks-dev
root@k8s-master01:~/sample-app# export GITHUB_TOKEN=<yourgithubtoken>
root@k8s-master01:~/sample-app# export ADMIN_REPOSITORY=azure

root@k8s-master01:~/sample-app# flux bootstrap github \
>  --owner=$GITHUB_USERNAME \
>  --repository=$ADMIN_REPOSITORY \
>  --path=kubernetes-infrastructure/clusters/aks-dev/workloads \
>  --personal \
>  --branch=master
```

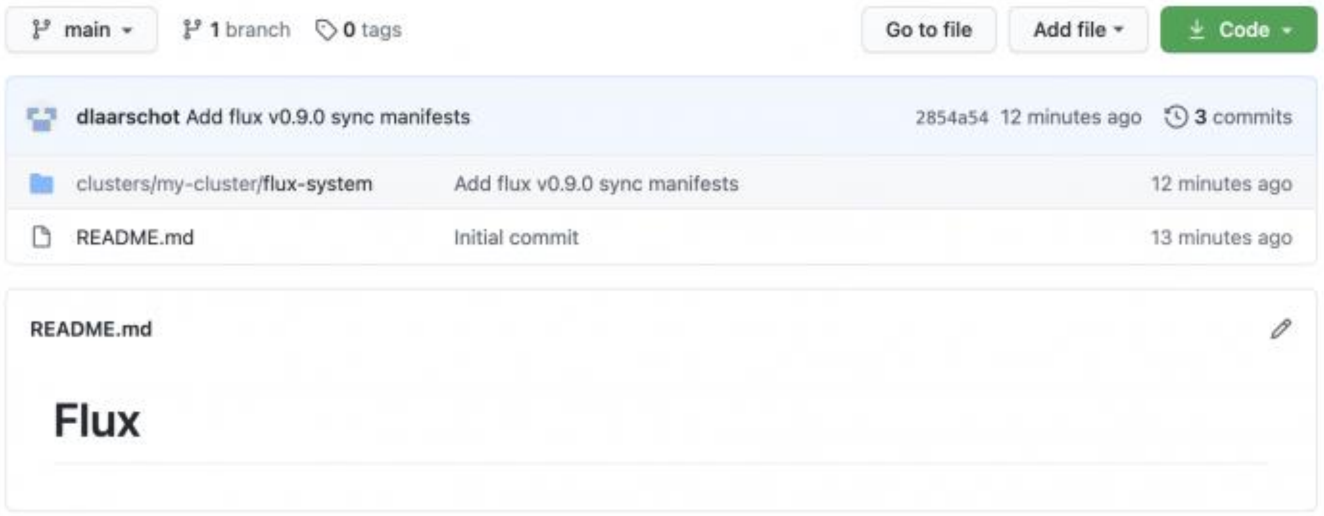
- **GITHUB_USERNAME:** your username
- **CLUSTER_NAME:** your cluster name
- **ADMIN_REPOSITORY:** your git repository name
- **--path:** this is the path/folder inside the git repository that will be bootstrapped to be managed by flux

You should get response like

```
root@k8s-master01:~/sample-app# flux bootstrap github \
>  --owner=$GITHUB_USERNAME \
>  --repository=$ADMIN_REPOSITORY \
>  --path=kubernetes-infrastructure/clusters/aks-dev/workloads \
>  --personal \
>  --branch=master
► connecting to github.com
✔ repository created
✔ repository cloned
✚ generating manifests
✔ components manifests pushed
► installing components in Flux-system namespace
namespace/Flux-system created
customresourcedefinition.apiextensions.k8s.io/alerts.notification.toolkit.Fluxcd.io
created
customresourcedefinition.apiextensions.k8s.io/buckets.source.toolkit.Fluxcd.io created
customresourcedefinition.apiextensions.k8s.io/gitrepositories.source.toolkit.Fluxcd.io
created
customresourcedefinition.apiextensions.k8s.io/helmcharts.source.toolkit.Fluxcd.io
created
customresourcedefinition.apiextensions.k8s.io/helmreleases.helm.toolkit.Fluxcd.io
created
```

```
customresourcedefinition.apiextensions.k8s.io/helmrepositories.source.toolkit.Fluxcd.io
created
customresourcedefinition.apiextensions.k8s.io/kustomizations.kustomize.toolkit.Fluxcd.i
o created
customresourcedefinition.apiextensions.k8s.io/providers.notification.toolkit.Fluxcd.io
created
customresourcedefinition.apiextensions.k8s.io/receivers.notification.toolkit.Fluxcd.io
created
serviceaccount/helm-controller created
serviceaccount/kustomize-controller created
serviceaccount/notification-controller created
serviceaccount/source-controller created
clusterrole.rbac.authorization.k8s.io/crd-controller-Flux-system created
clusterrolebinding.rbac.authorization.k8s.io/cluster-reconciler-Flux-system created
clusterrolebinding.rbac.authorization.k8s.io/crd-controller-Flux-system created
service/notification-controller created
service/source-controller created
service/webhook-receiver created
deployment.apps/helm-controller created
deployment.apps/kustomize-controller created
deployment.apps/notification-controller created
deployment.apps/source-controller created
networkpolicy.networking.k8s.io/allow-scraping created
networkpolicy.networking.k8s.io/allow-webhooks created
networkpolicy.networking.k8s.io/deny-ingress created
◎ verifying installation
✔ install completed
► configuring deploy key
✔ deploy key configured
► generating sync manifests
✔ sync manifests pushed
► applying sync manifests
◎ waiting for cluster sync
✔ bootstrap finished
```

A quick check of our GitHub repo confirms the required configuration manifest files have been created!



dlaarschot Add flux v0.9.0 sync manifests 2854a54 12 minutes ago 3 commits

clusters/my-cluster/flux-system	Add flux v0.9.0 sync manifests	12 minutes ago
README.md	Initial commit	13 minutes ago

README.md

Flux

Let's check to see if the Flux pods are running. Using kubectl, run the following command:

```
root@k8s-master01:~/sample-app# kubectl get pods -n flux-system
NAME                                      READY   STATUS    RESTARTS   AGE
helm-controller-7d59896db7-fg5r5          1/1     Running   0          2d2h
kustomize-controller-5946d7c748-56bmj     1/1     Running   0          2d2h
notification-controller-cfb8b87c8-gbljd   1/1     Running   0          2d2h
source-controller-546f697b96-9sdz9        1/1     Running   0          2d2h
```

Using the Flux command, we can check to see the currently deployed service on the cluster. Run:

```
root@k8s-master01:~/sample-app# flux get kustomizations
```

This command will display all the services running on the cluster. Yours should look like the below:

```
root@k8s-master01:~/sample-app# flux get kustomizations
NAME            READY   MESSAGE
REVISION                                        SUSPENDED

Flux-system     True    Applied revision: main/08d4eb338ce75292b0669eef84b64e7db2b161cf
main/08d4eb338ce75292b0669eef84b64e7db2b161cf   False
```

Now Flux is running! It's time to deploy an application.

Here is a deployment with flux for NGINX.

Repo directory looks like:

```
kubernetes-infrastructure/clusters/aks-dev/workloads/NGINX
├── helmrelease.yaml
├── helmrepository.yaml
├── kustomization.yml
└── service.yml
```

Helmrelease.yaml

```yaml
apiVersion: helm.toolkit.fluxcd.io/v2beta1
kind: HelmRelease
metadata:
  name: nginx
  namespace: nginx
spec:
  interval: 5m
  chart:
    spec:
      chart: nginx
      version: '8.8.1'
      sourceRef:
        kind: HelmRepository
        name: nginx
        namespace: flux-system
      interval: 1m
```

Helmrepository.yaml

```yaml
apiVersion: source.toolkit.fluxcd.io/v1beta1
kind: HelmRepository
metadata:
  name: nginx
  namespace: flux-system
spec:
  interval: 1m
  url: https://charts.bitnami.com/bitnami
```

kustomization.yml

```yaml
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
- helmrelease.yaml
- helmrepository.yaml
- service.yaml
```

Service.yml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx
  namespace: nginx
```

```
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
    name: http
  - port: 443
    targetPort: 443
    protocol: TCP
    name: https
  selector:
    app: nginx
```

We need to understand what's going on and how the deployment will be done by Flux!!

1- **Source Controller** will check for the target source repo that we already bootstrapped
2- **Source Controller** Will find the folder NGINX with all the manifests
3- **Source Controller** Will clone all the files to the cluster
4- **Kustomize controller** will start to install all the defined resources
5- **Kustomize controller** will use Helm controller to add the helm chart repo from helm repositories manifest
6- **Kustomize controller** will use Helm controller to deploy the target deployment in the helmrelease manifest
7- **Kustomize controller** will deploy service manifest
8- **Notification Controller:** handles all the events and notifications between all controllers

In Helmrelease manifest you can add any additional values for your helm deployment and you will find it in helm repo for the deployment

For example: Values for my NGINX deployment exists in this file

https://github.com/bitnami/charts/blob/master/bitnami/nginx/values.yaml

If I want to add some values my Helmrelease file will be looks like this

Helmrelease.yaml

```
apiVersion: helm.toolkit.fluxcd.io/v2beta1
kind: HelmRelease
metadata:
  name: nginx
  namespace: nginx
spec:
  interval: 5m
  chart:
    spec:
      chart: nginx
      version: '8.8.1'
      sourceRef:
        kind: HelmRepository
        name: nginx
        namespace: flux-system
      interval: 1m

  values:
    image:
      tag: 1.19.9
      pullPolicy: Always
    podLabels:
      app: "nginx"
    autoscaling:
    enabled: enabled
    minReplicas: 1
    maxReplicas: 5
    targetCPU: 50
    targetMemory: 50
```

I've added under **values** section specific image and pull policy and pods label and autoscaling and there's many others you can add for each deployment

With command flux help you will see all the commands that you should use with flux you should see something similar

```
root@k8s-master01:~/sample-app# flux -h
Command line utility for assembling Kubernetes CD pipelines the GitOps way.

Usage:
  flux [command]

Examples:
  # Check prerequisites
  flux check --pre

  # Install the latest version of Flux
  flux install --version=master

  # Create a source for a public Git repository
```

```
flux create source git webapp-latest \
  --url=https://github.com/stefanprodan/podinfo \
  --branch=master \
  --interval=3m

# List GitRepository sources and their status
flux get sources git

# Trigger a GitRepository source reconciliation
flux reconcile source git flux-system

# Export GitRepository sources in YAML format
flux export source git --all > sources.yaml

# Create a Kustomization for deploying a series of microservices
flux create kustomization webapp-dev \
  --source=webapp-latest \
  --path="./deploy/webapp/" \
  --prune=true \
  --interval=5m \
  --validation=client \
  --health-check="Deployment/backend.webapp" \
  --health-check="Deployment/frontend.webapp" \
  --health-check-timeout=2m

# Trigger a git sync of the Kustomization's source and apply changes
flux reconcile kustomization webapp-dev --with-source

# Suspend a Kustomization reconciliation
flux suspend kustomization webapp-dev

# Export Kustomizations in YAML format
flux export kustomization --all > kustomizations.yaml

# Resume a Kustomization reconciliation
flux resume kustomization webapp-dev

# Delete a Kustomization
flux delete kustomization webapp-dev

# Delete a GitRepository source
flux delete source git webapp-latest

# Uninstall Flux and delete CRDs
flux uninstall
```

```
Available Commands:
  bootstrap   Bootstrap toolkit components
  check       Check requirements and installation
  completion  Generates completion scripts for various shells
  create      Create or update sources and resources
  delete      Delete sources and resources
  export      Export resources in YAML format
  get         Get the resources and their status
  help        Help about any command
  install     Install or upgrade Flux
  logs        Display formatted logs for Flux components
  reconcile   Reconcile sources and resources
  resume      Resume suspended resources
  suspend     Suspend resources
  uninstall   Uninstall Flux and its custom resource definitions

Flags:
      --context string      kubernetes context to use
  -h, --help                help for flux
      --kubeconfig string   absolute path to the kubeconfig file
  -n, --namespace string    the namespace scope for this operation (default "flux-
system")
      --timeout duration    timeout for this operation (default 5m0s)
      --verbose             print generated objects
  -v, --version             version for flux

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

Most important

- **Flux get kustomization**
- **Flux get helmrelease**
- **Flux get sources**

```
root@k8s-master01:~/sample-app# flux get
The get sub-commands print the statuses of Flux resources.

Usage:
  flux get [command]

Available Commands:
  alert-providers Get Provider statuses
  alerts          Get Alert statuses
  helmreleases    Get HelmRelease statuses
  images          Get image automation object status
  kustomizations  Get Kustomization statuses
  receivers       Get Receiver statuses
  sources         Get source statuses

Flags:
  -A, --all-namespaces   list the requested object(s) across all namespaces
  -h, --help             help for get

Global Flags:
      --context string      kubernetes context to use
      --kubeconfig string   absolute path to the kubeconfig file
  -n, --namespace string    the namespace scope for this operation (default "flux-
system")
      --timeout duration    timeout for this operation (default 5m0s)
      --verbose             print generated objects

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

Flux get sources

This command will display all Get source statuses running on the cluster. Yours should look like the below:

```
root@k8s-master01:~/sample-app# flux get sources
NAME                             READY   MESSAGE
REVISION                                          SUSPENDED
gitrepository/flux-system        True    Fetched revision:
emam/54366fab33f32c27d3765bd6fff4dcfc5d246b1b
emam/54366fab33f32c27d3765bd6fff4dcfc5d246b1b     False
NAME                   READY   MESSAGE          REVISION                   SUSPENDED
helmrepository/NGINX         True    Fetched revision:
2ac309a6f610144c36036f651e68ec157d720322      2ac309a6f610144c36036f651e68ec157d720322
False

NAME                                                       READY   MESSAGE
REVISION        SUSPENDED
helmchart/NGINX-NGINX         True    Fetched revision: 3.0.3 3.0.3           False
```

Flux get kustomization

This command will display all the Applied revision on the cluster. Yours should look like the below:

```
root@k8s-master01:~/sample-app# flux get kustomizations
```

```
NAME             READY   MESSAGE
REVISION                                              SUSPENDED

Flux-system      True    Applied revision: main/08d4eb338ce75292b0669eef84b64e7db2b161cf
main/08d4eb338ce75292b0669eef84b64e7db2b161cf    False

NAME                     READY   MESSAGE
REVISION                                              SUSPENDED

NGINX        True    Applied revision: emam/54366fab33f32c27d3765bd6fff4dcfc5d246b1b
emam/54366fab33f32c27d3765bd6fff4dcfc5d246b1b    False
```

Flux get helmrelease

This command will display all the Applied helmrelease on the cluster. Yours should look like the below:

```
root@k8s-master01:~/sample-app# flux get helmrelease
NAMESPACE                NAME                     READY   MESSAGE
REVISION        SUSPENDED
NGINX        NGINX        True    Release reconciliation succeeded        3.0.3
False
```

Monitoring with Prometheus

Flux uses kube-prometheus-stack to provide a monitoring stack made out of:

- **Prometheus Operator** - manages Prometheus clusters atop Kubernetes
- **Prometheus** - collects metrics from the Flux controllers and Kubernetes API
- **Grafana** dashboards - displays the Flux control plane resource usage and reconciliation stats
- **kube-state-metrics** - generates metrics about the state of the Kubernetes objects

Install the kube-prometheus-stack

install the monitoring stack with flux, first register the toolkit Git repository on your cluster:

```
root@k8s-master01:~# flux create source git monitoring \

> --interval=30m \
> --url=https://github.com/fluxcd/flux2 \
> --branch=main
```

Then apply the manifests/monitoring/kube-prometheus-stack kustomization:

```
root@k8s-master01:~# flux create kustomization monitoring-stack \
> --interval=1h \
> --prune=true \
> --source=monitoring \
> --path="./manifests/monitoring/kube-prometheus-stack" \
> --health-check="Deployment/kube-prometheus-stack-operator.monitoring" \
> --health-check="Deployment/kube-prometheus-stack-grafana.monitoring"
```

The above Kustomization will install the kube-prometheus-stack in the monitoring namespace.

Install Flux Grafana dashboards

Note that the Flux controllers expose the /metrics endpoint on port 8080. When using Prometheus Operator you need a PodMonitor object to configure scraping for the controllers.

Apply the manifests/monitoring/monitoring-config containing the PodMonitor and the ConfigMap with Flux's Grafana dashboards:

```
root@k8s-master01:~# flux create kustomization monitoring-config \
> --interval=1h \
> --prune=true \
> --source=monitoring \
> --path="./manifests/monitoring/monitoring-config"
```

You can access Grafana using port forwarding:

```
root@k8s-master01:~# kubectl -n monitoring port-forward svc/kube-prometheus-stack-
grafana 3000:80
```

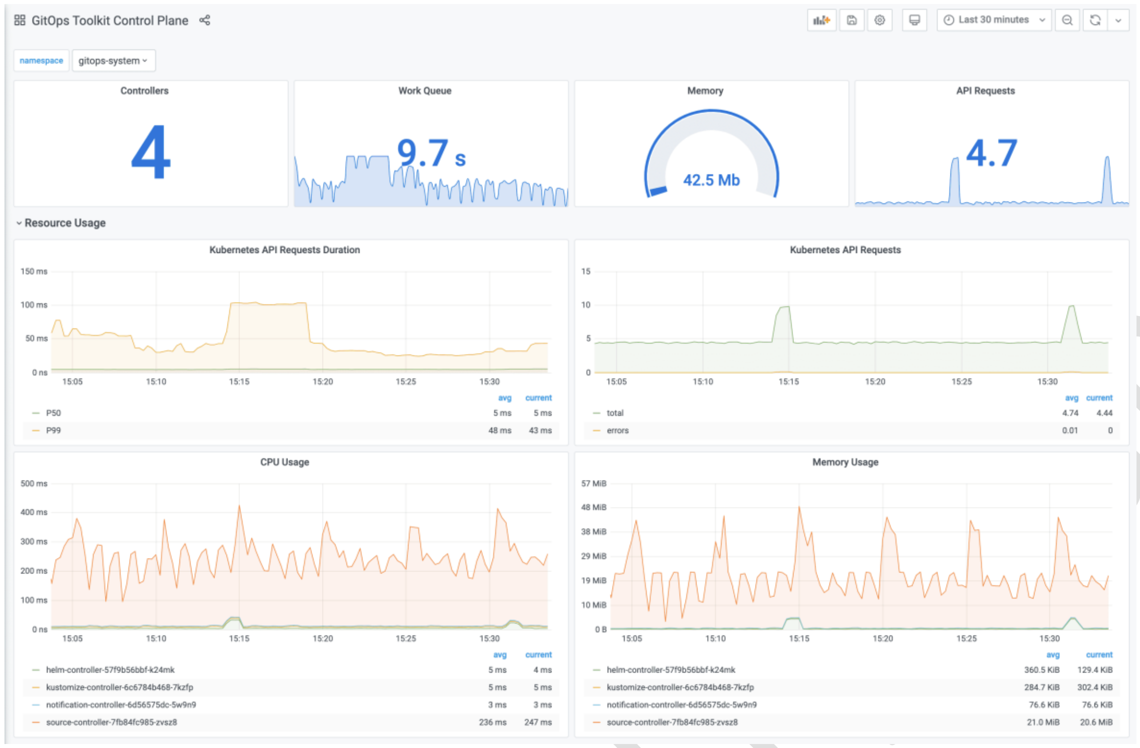
To log in to the Grafana dashboard, you can use the default credentials from the kube-prometheus-stack chart:

username: admin

password: prom-operator

Flux dashboards

Control plane dashboard http://localhost:3000/d/flux-control-plane:



	avg	current
P50	5 ms	5 ms
P99	48 ms	43 ms

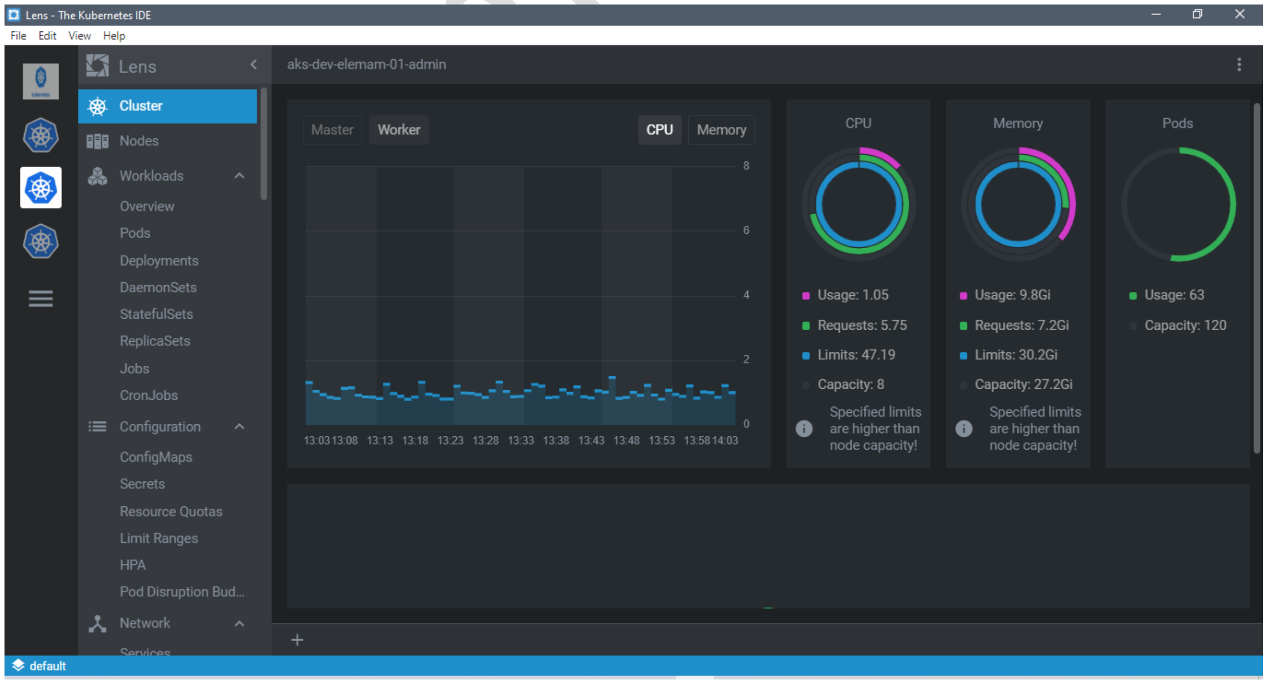
	avg	current
total	4.74	4.44
errors	0.01	0

	avg	current
helm-controller-57f9b56bbf-k24mk	5 ms	4 ms
kustomize-controller-6c6784b468-7kzfp	5 ms	5 ms
notification-controller-6d56575dc-5w9n9	3 ms	3 ms
source-controller-7fb84fc985-zvsz8	236 ms	247 ms

	avg	current
helm-controller-57f9b56bbf-k24mk	360.5 KiB	129.4 KiB
kustomize-controller-6c6784b468-7kzfp	284.7 KiB	302.4 KiB
notification-controller-6d56575dc-5w9n9	76.6 KiB	76.6 KiB
source-controller-7fb84fc985-zvsz8	21.0 MiB	20.6 MiB

Monitoring with Lens

Can use Lens to monitor our cluster and flux as well



CPU
- Usage: 1.05
- Requests: 5.75
- Limits: 47.19
- Capacity: 8
- Specified limits are higher than node capacity!

Memory
- Usage: 9.8Gi
- Requests: 7.2Gi
- Limits: 30.2Gi
- Capacity: 27.2Gi
- Specified limits are higher than node capacity!

Pods
- Usage: 63
- Capacity: 120

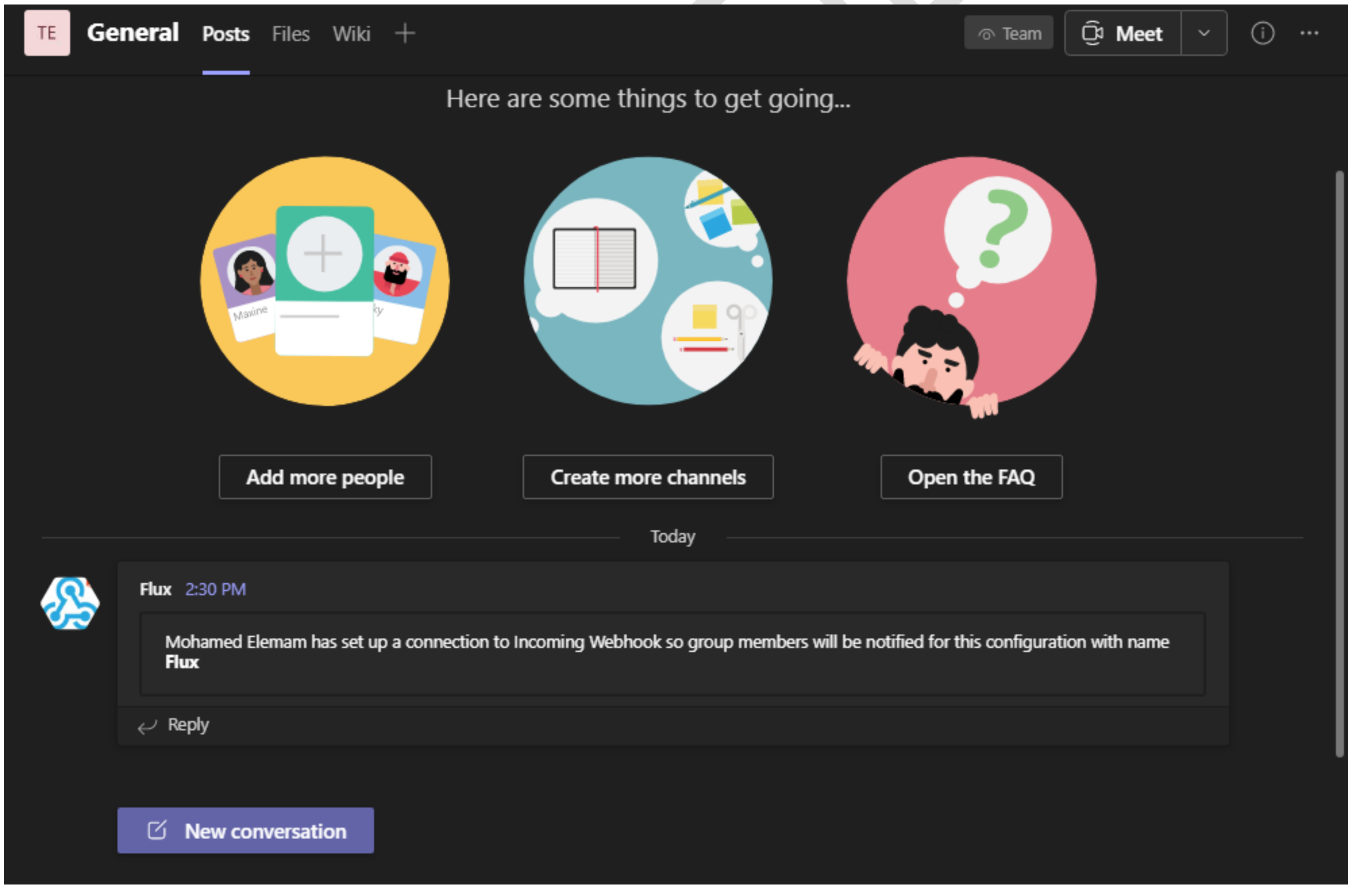
Setup Notifications

When operating a cluster, different teams may wish to receive notifications about the status of their GitOps pipelines. For example, the on-call team would receive alerts about reconciliation failures in the cluster, while the dev team may wish to be alerted when a new version of an app was deployed and if the deployment is healthy

The Flux controllers emit Kubernetes events whenever a resource status changes. You can use the notification-controller to forward these events to Slack, Microsoft Teams, Discord and others. The notification controller is part of the default Flux installation.

Define a provider

First create webhook for Teams or any other provider you want



Mohamed Elemam has set up a connection to Incoming Webhook so group members will be notified for this configuration with name **Flux**

Then create a secret with your Teams incoming webhook:

```
root@k8s-master01:~# kubectl -n flux-system create secret generic slack-url \
--from-literal=address= https://elemam.webhook.office.com/webhookb2/cdd37ad5-b62a-4888-a760-80708f7a1ff1@defa54f3-4877-425b-8965-eb5fbfd108ce/IncomingWebhook/<Your-Webhook>
```

Note that the secret must contain an address field, it can be a Slack, Microsoft Teams, Discord or Rocket webhook URL.

Then create a notification provider for Slack by referencing the above secret:

```
apiVersion: notification.toolkit.fluxcd.io/v1beta1
kind: Provider
metadata:
  name: teams
  namespace: flux-system
spec:
  type: teams
  channel: general
  secretRef:
    name: teams-url
```

The provider type can be slack, ms teams, discord, rocket, googlechat, webex, sentry or generic.

When type generic is specified, the notification controller will post the incoming event in JSON format to the webhook address. This way you can create custom handlers that can store the events in Elasticsearch, CloudWatch, Stackdriver, etc.

Then Create an alert definition for all repositories and kustomizations:

```
apiVersion: notification.toolkit.fluxcd.io/v1beta1
kind: Alert
metadata:
  name: on-call-webapp
  namespace: flux-system
spec:
  providerRef:
    name: slack
  eventSeverity: info
  eventSources:
    - kind: GitRepository
      name: '*'
    - kind: Kustomization
```

```
      name: '*'
```

Apply the above files or commit them to the repository.

To verify that the alert has been acknowledge by the notification controller do:

```
root@k8s-master01:~# kubectl -n flux-system get alerts
NAME                READY   STATUS        AGE
on-call-webapp      True    Initialized   1m
```

- Multiple alerts can be used to send notifications to different channels or Slack organizations.
- The event severity can be set to info or error. When the severity is set to error, the kustomize controller will alert on any error encountered during the reconciliation process. This includes kustomize build and validation errors, apply errors and health check failures.
- When the verbosity is set to info, the controller will alert if:
  - a Kubernetes object was created, updated or deleted
  - heath checks are passing
  - a dependency is delaying the execution
  - an error occurs

Automate image updates to Git

Now we will configuring container image scanning and deployment rollouts with Flux.

For a container image you can configure Flux to:

- scan the container registry and fetch the image tags
- select the latest tag based on the defined policy (semver, calver, regex)
- replace the tag in Kubernetes manifests (YAML format)
- checkout a branch, commit and push the changes to the remote Git repository
- apply the changes in-cluster and rollout the container image

For production environments, this feature allows you to automatically deploy application patches (CVEs and bug fixes), and keep a record of all deployments in Git history.

Production CI/CD workflow

- DEV: push a bug fix to the app repository
- DEV: bump the patch version and release e.g. v1.0.1
- CI: build and push a container image tagged as registry.domain/org/app:v1.0.1
- CD: pull the latest image metadata from the app registry (Flux image scanning)
- CD: update the image tag in the app manifest to v1.0.1 (Flux cluster to Git reconciliation)
- CD: deploy v1.0.1 to production clusters (Flux Git to cluster reconciliation)

For staging environments, this features allow you to deploy the latest build of a branch, without having to manually edit the app deployment manifest in Git.

Staging CI/CD workflow

- DEV: push code changes to the app repository main branch
- CI: build and push a container image tagged as ${GIT_BRANCH}-${GIT_SHA:0:7}-$(date +%s)
- CD: pull the latest image metadata from the app registry (Flux image scanning)
- CD: update the image tag in the app manifest to main-2d3fcbd-1611906956 (Flux cluster to Git reconciliation)
- CD: deploy main-2d3fcbd-1611906956 to staging clusters (Flux Git to cluster reconciliation)

Image automation

Image Automation controller isn't installed by default (since it's an alpha feature) et you need to reconfigure Flux. The flux bootstrap command is idempotent and you can run it again with the new parameters:

```
root@k8s-master01:~# --components-extra=image-reflector-controller,image-automation-controller
```

We are going to create three versions of our image.

```
root@k8s-master01:~# docker tag  EngMohamedElEmam/nginx  EngMohamedElEmam/nginx:v1.0.0
root@k8s-master01:~# docker tag  EngMohamedElEmam/nginx  EngMohamedElEmam/nginx:v1.1.0
root@k8s-master01:~# docker tag  EngMohamedElEmam/nginx  EngMohamedElEmam/nginx:v2.0.0
```

Use the CRD ImageRepository to track your image:

```
apiVersion: image.toolkit.fluxcd.io/v1alpha1
kind: ImageRepository
metadata:
  name: nginx
  namespace: flux-system
```

```
spec:
  image:  EngMohamedElEmam/nginx
  interval: 1m0s
```

Then, use the CRD ImagePolicy to specify which version you want to track.

```
apiVersion: image.toolkit.fluxcd.io/v1alpha1
kind: ImagePolicy
metadata:
  name: nginx
  namespace: flux-system
spec:
  imageRepositoryRef:
    name: nginx
  policy:
    semver:
      range: '>= 1.0.0 <2.0.0'
```

I choose to track every versions between 1.0.0 and 2.0.0. Only EngMohamedElEmam/nginx:1.0.0 and EngMohamedElEmam/nginx:1.1.0 should be concerned by the update, EngMohamedElEmam/nginx:2.0.0 shouldn't.

Those CRD can set the tracking up but they will not trigger an update into the Git repository.

You can check that the tracking works as expected:

```
root@k8s-master01:~# flux get image policy nginx
NAME            READY   MESSAGE
        LATEST IMAGE
nginx   True    Latest image tag for ' EngMohamedElEmam/nginx' resolved to: 1.1.0
EngMohamedElEmam/nginx:1.1.0
root@k8s-master01:~#
root@k8s-master01:~# flux get image repository nginx
NAME            READY   MESSAGE                                 LAST SCAN
        SUSPENDED
nginx   True    successful scan, found 11 tags  2021-02-03T23:45:03Z    False
```

We see the last image I push and match my semver rule : 1.1.0.

Let's configure Flux to update our Git repository when a new version is found by our ImagePolicy. Flux needs write permission to your Git repository to trigger the reconciliation process.

Two things here. First create the new resource ImageUpdateAutomation then modify your application to tell Flux which image would need to be updated. To do so, you need to add a marker to the container spec: # {"$imagepolicy": "POLICY_NAMESPACE:POLICY_NAME"}.

```
spec:
  containers:
  - name: container
    image:  EngMohamedElEmam/nginx:1.0.0 # {"$imagepolicy": "flux-system:nginx"}
```

```
apiVersion: image.toolkit.fluxcd.io/v1alpha1
kind: ImageUpdateAutomation
metadata:
  name: nginx
  namespace: flux-system
spec:
  checkout:
    branch: main
    gitRepositoryRef:
      name: nginx
  commit:
    authorEmail: fluxcdbot@particule.io
    authorName: fluxcdbot
    messageTemplate: 'update image'
  interval: 1m0s
  update:
    strategy: Setters
```

Check that the Deploy Key in your Git repository has write permission. If that is not the case, you can get the public key used and reconfigure your Github repository:

```
root@k8s-master01:~# kubectl -n flux-system get secret flux-system -o json | jq
'.data."identity.pub"' -r | base64 -d
```

Apply, wait and watch:

```
root@k8s-master01:~# kubectl get deploy nginx -o json | jq -r
'.spec.template.spec.containers[0].image'
 EngMohamedElEmam/nginx:1.1.0
```

Our Deployment has been updated with the new image and our Git repository has been also updated.

```
commit 75ba609fa4fc7b6bf9c3eb742acf1743a1cb4d7b (HEAD -> main, origin/main,
origin/HEAD)
Author: fluxcdbot <fluxcdbot@particule.io>
Date:   Thu Feb 4 09:27:40 2021 +0000

    update image

diff --git a/clusters/fluxcd-demo/fullapp.yaml b/clusters/fluxcd-demo/fullapp.yaml
index 636eda1..b1b224d 100644
--- a/clusters/fluxcd-demo/fullapp.yaml
+++ b/clusters/fluxcd-demo/fullapp.yaml
@@ -1,4 +1,3 @@
----
 apiVersion: apps/v1
 kind: Deployment
 metadata:
@@ -16,10 +15,10 @@ spec:
         app: nginx
     spec:
      containers:
-       image:  EngMohamedElEmam/nginx # {"$imagepolicy": "flux-system:nginx"}
+       image:  EngMohamedElEmam/nginx:1.1.0 # {"$imagepolicy": "flux-system:nginx"}
```

At the end we can push a new image 2.0.0 and see that it would not trigger an update since it doesn't match our ImagePolicy.

```
root@k8s-master01:~# docker push  EngMohamedElEmam/nginx:2.0.0
```

Resources

https://fluxcd.io/docs/

https://fluxcd.io/blog/

https://github.com/fluxcd/flux2