Configuring Flux with KinD and Azure Container Registry (OCI)

This guide walks you through setting up Flux in a KinD (Kubernetes in Docker) cluster to sync OCI artifacts from Azure Container Registry (ACR).

Prerequisites

Before starting, ensure you have the following tools installed:

- Docker
- KinD
- kubectl
- Flux CLI
- Azure CLI

Step 1: Create KinD Cluster

Create a KinD cluster with a custom configuration to enable proper networking:

```
# Create kind-config.yaml
cat <<EOF > kind-config.yaml
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
name: flux-cluster
nodes:
- role: control-plane
  kubeadmConfigPatches:
  - |
    kind: InitConfiguration
    nodeRegistration:
      kubeletExtraArgs:
        node-labels: "ingress-ready=true"
  extraPortMappings:
  - containerPort: 80
    hostPort: 80
    protocol: TCP
  - containerPort: 443
    hostPort: 443
    protocol: TCP
EOF

# Create the cluster
kind create cluster --config=kind-config.yaml
```

Verify the cluster is running:

```
kubectl cluster-info --context kind-flux-cluster
kubectl get nodes
```

Step 2: Set Up Azure Container Registry

Create Azure Container Registry

```
# Login to Azure
az login

# Set variables
RESOURCE_GROUP="flux-demo-rg"
ACR_NAME="fluxdemoacr$(date +%s)"  # Unique name
LOCATION="eastus"

# Create resource group
az group create --name $RESOURCE_GROUP --location $LOCATION

# Create ACR
az acr create --resource-group $RESOURCE_GROUP --name $ACR_NAME --sku
Basic

# Get ACR login server
ACR_LOGIN_SERVER=$(az acr show --name $ACR_NAME --resource-group
$RESOURCE_GROUP --query "loginServer" --output tsv)
echo "ACR Login Server: $ACR_LOGIN_SERVER"
```

Create Service Principal for Flux Authentication

```
# Create service principal
SP_NAME="flux-acr-sp"
ACR_REGISTRY_ID=$(az acr show --name $ACR_NAME --resource-group
$RESOURCE_GROUP --query "id" --output tsv)

SP_PASSWD=$(az ad sp create-for-rbac --name $SP_NAME --scopes
$ACR_REGISTRY_ID --role acrpull --query "password" --output tsv)
SP_APP_ID=$(az ad sp list --display-name $SP_NAME --query "[].appId" --
output tsv)

echo "Service Principal App ID: $SP_APP_ID"
echo "Service Principal Password: $SP_PASSWD"
```

Step 3: Install Flux

Bootstrap Flux in the KinD Cluster

```
# Install Flux components
flux install

# Verify Flux installation
kubectl get pods -n flux-system
```

Wait for all Flux pods to be running before proceeding.

Step 4: Create Kubernetes Secret for ACR Authentication

```
# Create namespace for your application (if needed)
kubectl create namespace demo

# Create Docker registry secret for ACR
kubectl create secret docker-registry acr-secret \
  --namespace=flux-system \
  --docker-server=$ACR_LOGIN_SERVER \
  --docker-username=$SP_APP_ID \
  --docker-password=$SP_PASSWD
```

Step 5: Push OCI Artifacts to ACR

You can push either Kustomizations or plain Kubernetes manifests to ACR as OCI artifacts. Choose the approach that fits your needs:

Option A: Push a Kustomization to ACR as OCI Artifact

First, create a sample Kustomization:

```
# Create a sample application directory
mkdir -p sample-app
cd sample-app

# Create a simple deployment
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: demo
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
```

```
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.21
        ports:
        - containerPort: 80
EOF

# Create kustomization.yaml
cat <<EOF > kustomization.yaml
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
- deployment.yaml
EOF

cd ..
```

Option C: Push Helm Charts as OCI Artifacts

Flux also supports Helm charts stored as OCI artifacts with custom values:

```
# Create a simple Helm chart
mkdir -p helm-app
cd helm-app

# Create Chart.yaml
cat <<EOF > Chart.yaml
apiVersion: v2
name: nginx-app
description: A simple nginx Helm chart
version: 0.1.0
appVersion: "1.21"
EOF

# Create templates directory
mkdir templates

# Create deployment template
cat <<EOF > templates/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "nginx-app.fullname" . }}
  namespace: {{ .Values.namespace | default "demo" }}
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      app: {{ include "nginx-app.name" . }}
```

```
    template:
      metadata:
        labels:
          app: {{ include "nginx-app.name" . }}
      spec:
        containers:
        - name: nginx
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
          ports:
          - containerPort: {{ .Values.service.port }}
          resources:
            {{- toYaml .Values.resources | nindent 12 }}
EOF

# Create service template
cat <<EOF > templates/service.yaml
apiVersion: v1
kind: Service
metadata:
  name: {{ include "nginx-app.fullname" . }}-service
  namespace: {{ .Values.namespace | default "demo" }}
spec:
  selector:
    app: {{ include "nginx-app.name" . }}
  ports:
    - protocol: TCP
      port: {{ .Values.service.port }}
      targetPort: {{ .Values.service.port }}
  type: {{ .Values.service.type }}
EOF

# Create _helpers.tpl
cat <<EOF > templates/_helpers.tpl
{{- define "nginx-app.name" -}}
{{- default .Chart.Name .Values.nameOverride | trunc 63 | trimSuffix "-"
}}
{{- end }}

{{- define "nginx-app.fullname" -}}
{{- if .Values.fullnameOverride }}
{{- .Values.fullnameOverride | trunc 63 | trimSuffix "-" }}
{{- else }}
{{- \$name := default .Chart.Name .Values.nameOverride }}
{{- if contains \$name .Release.Name }}
{{- .Release.Name | trunc 63 | trimSuffix "-" }}
{{- else }}
{{- printf "%s-%s" .Release.Name \$name | trunc 63 | trimSuffix "-" }}
{{- end }}
{{- end }}
{{- end }}
EOF

# Create default values.yaml
cat <<EOF > values.yaml
```

```
replicaCount: 2
namespace: demo

image:
  repository: nginx
  tag: "1.21"

service:
  type: ClusterIP
  port: 80

resources:
  limits:
    cpu: 100m
    memory: 128Mi
  requests:
    cpu: 100m
    memory: 128Mi

nameOverride: ""
fullnameOverride: ""
EOF

cd ..
```

Option B: Push Plain Kubernetes Manifests (No Kustomize)

If you prefer to use plain Kubernetes manifests without Kustomize:

```
# Create a plain manifests directory
mkdir -p plain-app
cd plain-app

# Create a simple deployment
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: demo
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
```

```
      - name: nginx
        image: nginx:1.21
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
  namespace: demo
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: ClusterIP
EOF

cd ..
```

Push to ACR using Flux CLI

```
# Login to ACR
az acr login --name $ACR_NAME

# Option A: Push the kustomization as OCI artifact
flux push artifact oci://$ACR_LOGIN_SERVER/flux/sample-app:v1.0.0 \
  --path="./sample-app" \
  --source="$(git config --get remote.origin.url)" \
  --revision="$(git rev-parse HEAD)"

# Option B: Push plain manifests as OCI artifact
flux push artifact oci://$ACR_LOGIN_SERVER/flux/plain-app:v1.0.0 \
  --path="./plain-app" \
  --source="$(git config --get remote.origin.url)" \
  --revision="$(git rev-parse HEAD)"

# Option C: Push Helm chart as OCI artifact
helm package helm-app
helm push nginx-app-0.1.0.tgz oci://$ACR_LOGIN_SERVER/helm
```

Step 6: Configure Flux OCIRepository

Create an OCIRepository resource to connect Flux to your ACR. Choose the configuration that matches your artifact type:

For Kustomization-based Artifacts

```
# Create oci-repository.yaml
cat <<EOF > oci-repository.yaml
apiVersion: source.toolkit.fluxcd.io/v1beta2
kind: OCIRepository
metadata:
  name: sample-app-oci
  namespace: flux-system
spec:
  interval: 5m
  url: oci://$ACR_LOGIN_SERVER/flux/sample-app
  ref:
    tag: v1.0.0
  secretRef:
    name: acr-secret
EOF
```

For Plain Manifest Artifacts

```
# Create oci-repository-plain.yaml
cat <<EOF > oci-repository-plain.yaml
apiVersion: source.toolkit.fluxcd.io/v1beta2
kind: OCIRepository
metadata:
  name: plain-app-oci
  namespace: flux-system
spec:
  interval: 5m
  url: oci://$ACR_LOGIN_SERVER/flux/plain-app
  ref:
    tag: v1.0.0
  secretRef:
    name: acr-secret
EOF
```

For Helm Chart Artifacts

```
# Create helm-repository.yaml
cat <<EOF > helm-repository.yaml
apiVersion: source.toolkit.fluxcd.io/v1beta2
kind: HelmRepository
metadata:
  name: nginx-app-repo
  namespace: flux-system
spec:
  interval: 5m
  type: oci
  url: oci://$ACR_LOGIN_SERVER/helm
  secretRef:
```

```
    name: acr-secret
EOF
```

Apply the OCIRepository (choose one based on your artifact type):

```
# For Kustomization artifacts
kubectl apply -f oci-repository.yaml

# OR for plain manifest artifacts
kubectl apply -f oci-repository-plain.yaml

# OR for Helm chart artifacts
kubectl apply -f helm-repository.yaml
```

Step 7: Create Deployment Configuration

Choose the appropriate deployment method based on your artifact type:

Option A: Kustomization for Kustomize-based Artifacts

Create a Kustomization resource that references the OCI artifact:

```
# Create kustomization-deploy.yaml
cat <<EOF > kustomization-deploy.yaml
apiVersion: kustomize.toolkit.fluxcd.io/v1
kind: Kustomization
metadata:
  name: sample-app
  namespace: flux-system
spec:
  interval: 5m
  targetNamespace: demo
  sourceRef:
    kind: OCIRepository
    name: sample-app-oci
  path: "./"
  prune: true
  wait: true
  timeout: 2m
EOF
```

Apply the Kustomization:

```
kubectl apply -f kustomization-deploy.yaml
```

Option B: Kustomization for Plain Manifest Artifacts

For plain Kubernetes manifests, you can still use a Kustomization resource but without kustomize.yaml:

```
# Create kustomization-plain-deploy.yaml
cat <<EOF > kustomization-plain-deploy.yaml
apiVersion: kustomize.toolkit.fluxcd.io/v1
kind: Kustomization
metadata:
  name: plain-app
  namespace: flux-system
spec:
  interval: 5m
  targetNamespace: demo
  sourceRef:
    kind: OCIRepository
    name: plain-app-oci
  path: "./"
  prune: true
  wait: true
  timeout: 2m
EOF
```

Apply the Kustomization:

```
kubectl apply -f kustomization-plain-deploy.yaml
```

Note: Even when using plain Kubernetes manifests, Flux still uses the Kustomization controller for deployment. The difference is that your artifacts don't contain a `kustomization.yaml` file - Flux will simply apply all YAML files found in the specified path.

Option C: HelmRelease for Helm Chart Artifacts

For Helm charts stored as OCI artifacts, use a HelmRelease resource with custom values:

```
# Create custom-values.yaml for your environment-specific configuration
cat <<EOF > custom-values.yaml
replicaCount: 3
namespace: demo

image:
  repository: nginx
  tag: "1.22"

service:
  type: LoadBalancer
  port: 8080

resources:
  limits:
```

```
      cpu: 200m
      memory: 256Mi
    requests:
      cpu: 100m
      memory: 128Mi
EOF

# Create helm-release.yaml
cat <<EOF > helm-release.yaml
apiVersion: helm.toolkit.fluxcd.io/v2
kind: HelmRelease
metadata:
  name: nginx-app
  namespace: flux-system
spec:
  interval: 5m
  targetNamespace: demo
  chart:
    spec:
      chart: nginx-app
      version: "0.1.0"
      sourceRef:
        kind: HelmRepository
        name: nginx-app-repo
  values:
    replicaCount: 3
    namespace: demo
    image:
      repository: nginx
      tag: "1.22"
    service:
      type: LoadBalancer
      port: 8080
    resources:
      limits:
        cpu: 200m
        memory: 256Mi
      requests:
        cpu: 100m
        memory: 128Mi
  # Alternative: reference external values file
  # valuesFrom:
  # - kind: ConfigMap
  #   name: nginx-app-values
  #   valuesKey: values.yaml
EOF
```

Apply the HelmRelease:

```
kubectl apply -f helm-release.yaml

# Optional: Create ConfigMap from values file for external values
```

```
# kubectl create configmap nginx-app-values --from-
file=values.yaml=custom-values.yaml -n flux-system
```

Step 8: Verify Deployment

Check that Flux has successfully deployed your application:

```
# Check Flux sources
flux get sources oci
flux get sources helm

# Check Kustomizations
flux get kustomizations

# Check HelmReleases
flux get helmreleases

# Check if the demo namespace was created
kubectl get namespaces

# Check if the nginx deployment was created
kubectl get deployments -n demo

# Check pods
kubectl get pods -n demo

# For plain manifest artifacts, also check services
kubectl get services -n demo

# For Helm releases, check Helm-specific resources
helm list -n demo
```

Step 9: Monitor and Troubleshoot

View Flux Events and Logs

```
# Get events from Flux
flux events

# Check OCIRepository status
kubectl describe ocirepository sample-app-oci -n flux-system

# Check Kustomization status
kubectl describe kustomization sample-app -n flux-system

# View controller logs
kubectl logs -n flux-system deployment/source-controller
kubectl logs -n flux-system deployment/kustomize-controller
kubectl logs -n flux-system deployment/helm-controller
```

Common Issues and Solutions

1. **Authentication Issues**

```
# Verify secret exists
kubectl get secret acr-secret -n flux-system

# Check secret content
kubectl get secret acr-secret -n flux-system -o yaml
```

2. **OCI Repository Not Found**

```
# List artifacts in ACR
az acr repository list --name $ACR_NAME

# Check specific artifact
az acr repository show-tags --name $ACR_NAME --repository flux/sample-
app
# For plain manifests
az acr repository show-tags --name $ACR_NAME --repository flux/plain-
app
# For Helm charts
az acr repository show-tags --name $ACR_NAME --repository helm/nginx-
app
```

3. **Network Issues in KinD**

```
# Test DNS resolution from within cluster
kubectl run test-pod --image=busybox -it --rm -- nslookup
$ACR_LOGIN_SERVER
```

Step 10: Updating OCI Artifacts

To update your application, push a new version and update the corresponding resource:

```
# Make changes to your application files (sample-app, plain-app, or helm-
app directory)

# Push new version - choose the appropriate path based on your artifact
type
# For Kustomization artifacts:
flux push artifact oci://$ACR_LOGIN_SERVER/flux/sample-app:v1.1.0 \
  --path="./sample-app" \
  --source="$(git config --get remote.origin.url)" \
```

```
  --revision="$(git rev-parse HEAD)"

# For plain manifest artifacts:
flux push artifact oci://$ACR_LOGIN_SERVER/flux/plain-app:v1.1.0 \
  --path="./plain-app" \
  --source="$(git config --get remote.origin.url)" \
  --revision="$(git rev-parse HEAD)"

# For Helm chart artifacts:
# Update Chart.yaml version first, then:
helm package helm-app
helm push nginx-app-0.2.0.tgz oci://$ACR_LOGIN_SERVER/helm

# Update resources to use new version - choose based on your artifact type
# For Kustomization artifacts:
kubectl patch ocirepository sample-app-oci -n flux-system \
  --type='merge' -p='{"spec":{"ref":{"tag":"v1.1.0"}}}'

# For plain manifest artifacts:
kubectl patch ocirepository plain-app-oci -n flux-system \
  --type='merge' -p='{"spec":{"ref":{"tag":"v1.1.0"}}}'

# For Helm chart artifacts:
kubectl patch helmrelease nginx-app -n flux-system \
  --type='merge' -p='{"spec":{"chart":{"spec":{"version":"0.2.0"}}}}'

# Or update Helm values without changing chart version:
kubectl patch helmrelease nginx-app -n flux-system \
  --type='merge' -p='{"spec":{"values":{"replicaCount":5}}}'
```

Cleanup

To clean up the resources:

```
# Delete the KinD cluster
kind delete cluster --name flux-cluster

# Delete Azure resources
az group delete --name $RESOURCE_GROUP --yes --no-wait

# Delete service principal
az ad sp delete --id $SP_APP_ID
```

Additional Resources

- Flux OCI Documentation
- Azure Container Registry Documentation
- KinD Documentation
- Flux GitHub Repository

Notes

- Replace placeholder values (like `$ACR_NAME`, `$RESOURCE_GROUP`) with your actual values
- Ensure your Azure subscription has sufficient permissions to create ACR and service principals
- The service principal needs `acrpull` permissions to access the registry
- OCI artifacts in ACR are billed as storage, so monitor usage in production environments
- **Kustomize vs Plain Manifests vs Helm Charts:**
  - Use **Kustomization artifacts** (Option A) when you want to leverage Kustomize features like patches, transformers, and overlays
  - Use **Plain manifest artifacts** (Option B) when you have simple Kubernetes YAML files and don't need Kustomize functionality
  - Use **Helm chart artifacts** (Option C) when you want templating, parameterization, and package management capabilities
  - Both Kustomization and plain manifest options use Flux's Kustomization controller for deployment
  - Helm charts use Flux's Helm controller and support custom values for different environments
  - Helm charts offer the most flexibility for configuration management across multiple environments