Argo Guestbook (individuellt projekt)

Lernia DevOps Engineer (DevOps24), Lernia Alvik

Namn: Martin Wallin

Sammanfattning

Detta projekt redovisar implementeringen av en GitOps-baserad CI/CD-pipeline för en gästboksapplikation på OpenShift. Med hjälp av ArgoCD, GitHub Actions och Quay.io har en helautomatiserad process skapats där kodändringar automatiskt bygger nya container-images och uppdaterar klustret. Lösningen använder prefixet `mw-` för att isolera resurser och hanterar hemligheter säkert via GitHub Secrets.

Abstract

This project presents the implementation of a GitOps-based CI/CD pipeline for a guestbook application on OpenShift. Using ArgoCD, GitHub Actions, and Quay.io, a fully automated process has been established where code changes automatically trigger new container image builds and cluster updates. The solution employs the `mw-` prefix for resource isolation and manages secrets securely via GitHub Secrets.

Innehållsförteckning

Inledning

Syftet med denna inlämningsuppgift var att bygga vidare på den tidigare "Automate Guestbook"-uppgiften och implementera Continuous Deployment (CD) med hjälp av verktyget ArgoCD. Målet var att skapa en robust pipeline där ändringar i källkod eller konfiguration automatiskt speglas i produktionsmiljön utan manuell handpåläggning, enligt GitOps-principen.

Bakgrund

I kursen Containerteknik skapades grunden för applikationen bestående av en frontend (Nginx), backend (Go), Redis och PostgreSQL. I detta projekt har fokus flyttats från manuell hantering med `oc`-kommandon till en deklarativ modell där Git-repot utgör "single source of truth".

Metod

Arkitektur och Namnstandard

Applikationen är uppdelad i mikrotjänster där varje komponent har sin egen byggprocess. För att undvika konflikter i det gemensamma OpenShift-klustret har alla resurser (Deployments, Services, PVCs) prefixats med `mw-` (Martin Wallin), exempelvis `mw-guestbook` och `mw-postgres`.

CI - Continuous Integration (GitHub Actions)

Jag har skapat separata workflows för varje komponent (Frontend, Backend, Redis, Postgres). När kod pushas till `main`-branchen sker följande:

1. **Bygge**: En ny container-image byggs baserat på Red Hat UBI (Universal Base Image).
2. **Publicering**: Imagen taggas med både `latest` och Git-commitens SHA-hash (för spårbarhet) och pushas till Quay.io.
3. **Manifest-uppdatering**: Workflowet använder verktyget `yq` för att automatiskt uppdatera Kubernetes-manifestet i `k8s/`-mappen med den nya specifika SHA-taggen.
4. **Commit**: Ändringen i manifestet committas tillbaka till repot av GitHub Actions.

CD - Continuous Deployment (ArgoCD)

ArgoCD är konfigurerat att övervaka mappen `k8s/` i mitt GitHub-repo.

- När CI-processen uppdaterar en image-tagg i en yaml-fil upptäcker ArgoCD att klustrets status inte matchar Git.
- ArgoCD synkroniserar automatiskt (Self-Heal & Auto-Sync) så att klustret hämtar den nya imagen och startar om poddarna.

Hantering av Hemligheter

Eftersom ArgoCD läser från ett publikt repo kan känslig data (som databaslösenord) inte lagras där. Jag löste detta genom ett separat workflow, `deploy-main.yml`, som körs innan ArgoCD-synken. Detta workflow hämtar hemligheter från GitHub Repository Secrets och injicerar dem i klustret som en OpenShift Secret (`mw-guestbook-secrets`).

Resultat

Åtkomst och Funktion

Gästboken är driftsatt och nåbar på följande URL:

- **URL**: http://mw-guestbook-grupp2.apps.devops24.cloud2.se

Jag har verifierat att:

1. Inlägg sparas i databasen och syns även om jag öppnar sidan i ett inkognito-fönster (persistens fungerar).
2. Cachen (Redis) fungerar och avlastar databasen vid läsning.

Skalning och Uppdatering

Systemet klarar av de tester som specificerats:

- Vid ändring av `replicas` i `k8s/backend-quay.yaml` från Git, skalar ArgoCD automatiskt upp antalet poddar.
- Vid kodändring byggs en ny image, och ArgoCD rullar ut den nya versionen utan nedtid (Rolling Update).

Diskussion

Reflektion över arbetet

Att gå från imperativ driftsättning (skript) till deklarativ (GitOps) krävde ett nytt tankesätt, särskilt eftersom alla filer görs tillgängliga i ett publikt GitHub-repo. Detta medförde en säkerhetsrisk gällande hanteringen av hemligheter (Secrets), då ArgoCD i sin standardkonfiguration läser allt från repot. Om jag hade lagt databaslösenorden direkt i `k8s`-katalogen hade de exponerats öppet. Efter att ha undersökt olika alternativ kom jag fram till att den smidigaste lösningen för denna uppgift var att använda GitHub Actions för att injicera hemligheterna från skyddade GitHub Secrets direkt in i klustret, separat från ArgoCDs synkronisering.

En annan viktig insikt var nödvändigheten av att använda specifika image-taggar (SHA) istället för `latest`. Eftersom `latest`-taggen inte ändras vid nya byggen kan ArgoCD missa att en uppdatering skett. Genom att låta CI-pipelinen uppdatera manifestet med en unik commit-hash garanteras att ArgoCD alltid upptäcker förändringen och deployar exakt den version som byggdes.

Motivering av val

- **UBI Images**: Jag valde Red Hat UBI 10 som bas-image. Eftersom detta är en relativt ny version saknas det ofta färdiga exempel och AI-modeller har ibland begränsad kontext kring den, vilket innebar en spännande utmaning. Det gav en unik och lärorik övning i att utveckla containers i en "bleeding edge"-miljö utan att kunna förlita sig på gamla guider.

- **GitHub Actions + ArgoCD**: Även om huvudfokus för deployment låg på ArgoCD, visade det sig nödvändigt att komplettera med GitHub Actions för att skapa en komplett och robust pipeline. Förutom att hantera CI-delen (bygga images), löste GitHub Actions två kritiska problem som ArgoCD ensamt inte hanterar lika smidigt: säker injicering av hemligheter och bootstrapping av själva ArgoCD-installationen. Efter att ha upplevt driftstörningar i klustret insåg jag värdet av att kunna återställa hela miljön – inklusive ArgoCD-konfigurationen – automatiskt, istället för att manuellt konfigurera via webbgränssnittet. Detta hybrid-upplägg ger en snabbare och mer pålitlig återställning (Disaster Recovery).

Förbättringsförslag

För att göra lösningen ännu mer produktionsmässig skulle jag vilja implementera följande:

1. **Permanent Inloggning (Service Account)**: Just nu förlitar sig `deploy-main.yml` på en `OPENSHIFT_TOKEN` som löper ut efter 24 timmar. En bättre lösning vore att skapa ett Service Account i OpenShift med begränsade rättigheter och använda dess token. Detta skulle göra pipelinen stabil över tid.

2. **StatefulSets för Databaser**: Nu körs Postgres och Redis som vanliga `Deployments`. I en riktig produktion bör dessa vara `StatefulSets`. Det ger stabila nätverksidentiteter (t.ex. `mw-postgres-0`)

och garanterar att lagringen (PVC) följer med podden om den startas om på en annan nod, vilket är kritiskt för dataintegritet.

3. **Säkerhet och HTTPS**: Trafiken är idag okrypterad (HTTP). Genom att konfigurera OpenShift Route med TLS (Edge eller Passthrough) skulle vi få HTTPS. Dessutom borde `NetworkPolicies` införas för att isolera databaserna så att endast backend-podden får prata med dem, inte övriga klustret.
4. **Automatiserade Tester**: CI-pipelinen bygger koden men kör inga tester. Att lägga till ett steg med `go test` för backend och linting för frontend innan bygget skulle öka kvalitetssäkringen markant.

Källförteckning

- **Projekt-repo**: https://github.com/SvartaStorken/argo_guestbook
- **Container Registry**: https://quay.io/repository/eksta_mannen/
- **ArgoCD Dokumentation**: https://argo-cd.readthedocs.io/

Bilagor

Se `README.md` i GitHub-repot för fullständig teknisk dokumentation, diagram och konfigurationsfiler.