

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.

Innehållsförteckning

- 1. [Inledning](#)
- 2. [Bakgrund](#)
- 3. [Metod](#)
- 4. [Resultat](#)
- 5. [Diskussion](#)
- 6. [Källförteckning](#)

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 arbetsflöden för varje komponent (Frontend, Backend, Redis, Postgres). När kod pushas till `main`-grenen sker följande:

1. **Bygge:** En ny container-image byggs baserat på Red Hat UBI (Universal Base Image).

2. **Publicerings**: Imagen taggas med både `latest` och Git-commitens SHA-hash (för spårbarhet) och pushas till Quay.io.
3. **Manifest-uppdatering**: Arbetsflödet 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 synkroniseras automatiskt (Self-Heal & Auto-Sync) så att klustret hämtar den nya imagen och startar om poddarna.

Hantering av Hemligheter

Eftersom ArgoCD läser från ett publik repo kan känslig data (som databaslösenord) inte lagras där. Jag löste detta genom ett separat arbetsflöde, `deploy-main.yaml`, som körs innan ArgoCD-synken. Detta arbetsflöde 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 publik 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 kunskap om 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 Image Build:** Jag valde att använda GitHub Actions för att bygga mina container-images. Eftersom ArgoCD behöver en färdig image att deploya, är det logiskt att byggsteget sker innan deployment-processen initieras. Även om det hade varit tekniskt intressant att bygga images direkt i klustret (t.ex. med OpenShift Builds) och sedan pusha till Quay.io, bedömde jag att komplexiteten inte motiverade värdet i detta projekt. GitHub Actions erbjöd en enklare och mer integrerad lösning.

En positiv effekt av att bygga egna databas-images var möjligheten att testa CI/CD-flödet mer ingående. Jag implementerade bland annat ett skript för automatiska "health checks" i Redis. Tidigare har jag upplevt problem med Redis-stabilitet, sannolikt på grund av felaktig konfiguration. Genom att bygga imagen själv kunde jag säkerställa funktionaliteten och verifiera att CI/CD-pipelinen hanterade uppdateringar korrekt.

Även min PostgreSQL-image förbättrades under arbetets gång. Det ursprungliga run.sh-skriptet förutsatte att PVC:n var tom vid installation, vilket tvingade fram en radering av volymen vid varje omdeployering eller ändring av hemligheter. Jag uppdaterade skriptet så att det hanterar befintlig data bättre och konfigurerar om databasen om parametrarna ändras. Även om jag inte hunnit testa alla kantfall fullt ut, verkar den nya lösningen mer robust. Precis som med Redis gav de upprepade byggna av PostgreSQL-imagen värdefull träning i att hantera CI/CD-flödet.

- **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

Det omfattande arbetet med att agera klusteradministratör tog tid från själva utvecklingen. Att felsöka och åtgärda klusterkonfigurationer, samt hantera driftstörningar (exempelvis när skoklustret inte startade om korrekt, vilket kostade en hel arbetsdag), innebar att vissa planerade funktioner fick prioriteras bort:

1. **Permanent Inloggning (Service Account):** För närvarande använder `deploy-main.yml` en `OPENSHIFT_TOKEN` som löper ut efter 24 timmar. Detta skapar problem vid redovisning och långsiktig drift, då token måste uppdateras manuellt. Eftersom den helautomatiserade deployment-processen är en av projektets största styrkor, skulle jag vilja implementera en mer beständig inloggningslösning, exempelvis via ett Service Account med långlivade tokens.
2. **StatefulSets för Databaser:** Jag har läst om hur man kan skala upp databaser horisontellt utan att riskera datakorruption, ofta genom att använda StatefulSets istället för Deployments. Eftersom jag använder egna images hade det varit mycket lärorikt att implementera och testa detta för både PostgreSQL och Redis för att öka tillgängligheten och prestandan.
3. **Säkerhet och HTTPS:** Att applikationen saknar HTTPS-stöd är en tydlig brist. Med mer tid hade jag velat konfigurera TLS-terminering, vilket sannolikt hade krävt justeringar i frontend-imagen eller Ingress/Route-konfigurationen. Dessutom borde `NetworkPolicies` införas för att isolera databaserna, så att endast backend-tjänsten tillåts kommunicera med dem.
4. **Automatiserade Tester och Rapportering:** Jag har inte hunnit utföra så omfattande tester av CI/CD-flödet som jag hade önskat. Även om systemet verkar fungera som tänkt, saknas verifiering av kantfall. Jag hade velat implementera mer robusta "health checks", tydligare loggning och automatisk rapportering. I tidigare projekt har jag använt Telegrams API för att skicka statusrapporter direkt till mobilen, vilket hade varit ett värdefullt tillskott även här för att snabbt upptäcka problem.

Källförteckning

- **Projekt-repo (GitHub):** https://github.com/SvartaStorken/argo_guestbook
 - Innehåller all källkod, Kubernetes-manifest och dokumentation för detta projekt.
- **Kodbas för Images (GitLab):** <https://gitlab.com/SvartaStorken>
 - Repo för tidigare uppgifter samt källkod som ligger till grund för container-images.
- **Container Registry (Quay.io):** https://quay.io/repository/eksta_mannen/
 - Publikt register där projektets byggda images lagras.
- **ArgoCD Dokumentation:** <https://argo-cd.readthedocs.io/>
 - Officiell dokumentation som använts för konfiguration och felsökning.
- **Kursmaterial (GitHub):** <https://github.com/jonasbjork>
 - Jonas Björks repository med labbar och instruktioner som legat till grund för uppgiften.