Argo Guestbook (Individual Project)

Lernia DevOps Engineer (DevOps24), Lernia Alvik

Name: Martin Wallin

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.

Table of Contents

Introduction

The purpose of this assignment was to build upon the previous "Automate Guestbook" task and implement Continuous Deployment (CD) using ArgoCD. The goal was to create a robust pipeline where changes in source code or configuration are automatically mirrored in the production environment without manual intervention, following GitOps principles.

Background

In the Container Technology course, the foundation for the application was created, consisting of a frontend (Nginx), backend (Go), Redis, and PostgreSQL. In this project, the focus has shifted from manual management with `oc` commands to a declarative model where the Git repo constitutes the "single source of truth".

Method

Architecture and Naming Convention

The application is divided into microservices where each component has its own build process. To avoid conflicts in the shared OpenShift cluster, all resources (Deployments, Services, PVCs) have been prefixed with `mw-` (Martin Wallin), for example `mw-guestbook` and `mw-postgres`.

CI - Continuous Integration (GitHub Actions)

I have created separate workflows for each component (Frontend, Backend, Redis, Postgres). When code is pushed to the `main` branch, the following occurs:

1. **Build**: A new container image is built based on Red Hat UBI (Universal Base Image).
2. **Publish**: The image is tagged with both `latest` and the Git commit SHA hash (for traceability) and pushed to Quay.io.
3. **Manifest Update**: The workflow uses the tool `yq` to automatically update the Kubernetes manifest in the `k8s/` folder with the new specific SHA tag.
4. **Commit**: The change in the manifest is committed back to the repo by GitHub Actions.

CD - Continuous Deployment (ArgoCD)

ArgoCD is configured to monitor the `k8s/` folder in my GitHub repo.

- When the CI process updates an image tag in a yaml file, ArgoCD detects that the cluster status does not match Git.
- ArgoCD synchronizes automatically (Self-Heal & Auto-Sync) so that the cluster fetches the new image and restarts the pods.

Secret Management

Since ArgoCD reads from a public repo, sensitive data (like database passwords) cannot be stored there. I solved this through a separate workflow, `deploy-main.yml`, which runs before the ArgoCD sync. This workflow retrieves secrets from GitHub Repository Secrets and injects them into the cluster as an OpenShift Secret (`mw-guestbook-secrets`).

Results

Access and Functionality

The guestbook is deployed and accessible at the following URL:

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

I have verified that:

1. Posts are saved in the database and are visible even if I open the page in an incognito window (persistence works).
2. The cache (Redis) works and offloads the database during reading.

Scaling and Updates

The system handles the specified tests:

- When changing `replicas` in `k8s/backend-quay.yaml` from Git, ArgoCD automatically scales up the number of pods.
- Upon code changes, a new image is built, and ArgoCD rolls out the new version without downtime (Rolling Update).

Discussion

Reflection on the work

Moving from imperative deployment (scripts) to declarative (GitOps) required a new mindset, especially since all files are made available in a public GitHub repo. This entailed a security risk regarding the handling of Secrets, as ArgoCD in its default configuration reads everything from the repo. If I had placed the database passwords directly in the `k8s` directory, they would have been exposed openly. After investigating various options, I concluded that the smoothest solution for this task was to use GitHub Actions to inject the secrets from protected GitHub Secrets directly into the cluster, separate from ArgoCD's synchronization.

Another important insight was the necessity of using specific image tags (SHA) instead of `latest`. Since the `latest` tag does not change with new builds, ArgoCD might miss that an update has occurred. By letting the CI pipeline update the manifest with a unique commit hash, it is guaranteed that ArgoCD always detects the change and deploys exactly the version that was built.

Motivation of choices

- **UBI Images**: I chose Red Hat UBI 10 as the base image. Since this is a relatively new version, ready-made examples are often missing, and AI models sometimes have limited knowledge about it, which presented an exciting challenge. It provided a unique and educational exercise in developing containers in a "bleeding edge" environment without being able to rely on old guides.

- **GitHub Image Build**: I chose to use GitHub Actions to build my container images. Since ArgoCD needs a ready image to deploy, it is logical that the build step occurs before the deployment process is initiated. Although it would have been technically interesting to build images directly in the cluster (e.g., with OpenShift Builds) and then push to Quay.io, I judged that the complexity did not justify the value in this project. GitHub Actions offered a simpler and more integrated solution.

A positive effect of building my own database images was the opportunity to test the CI/CD flow more thoroughly. I implemented, among other things, a script for automatic "health checks" in Redis. Previously, I have experienced problems with Redis stability, likely due to incorrect configuration. By building the image myself, I could ensure functionality and verify that the CI/CD pipeline handled updates correctly.

My PostgreSQL image was also improved during the work. The original `run.sh` script assumed that the PVC was empty upon installation, which forced a deletion of the volume at every redeployment or change of secrets. I updated the script so that it handles existing data better and reconfigures the database if parameters change. Although I haven't had time to test all edge cases fully, the new solution seems more robust. Just like with Redis, the repeated builds of the PostgreSQL image gave valuable training in handling the CI/CD flow.

- **GitHub Actions + ArgoCD**: Although the main focus for deployment was on ArgoCD, it proved necessary to complement with GitHub Actions to create a complete and robust pipeline. Besides handling the CI part (building images), GitHub Actions solved two critical problems that ArgoCD alone does not handle as smoothly: secure injection of secrets and bootstrapping of the ArgoCD installation itself. After experiencing downtime in the cluster, I realized the value of being able to restore the entire environment – including the ArgoCD configuration – automatically, instead of manually configuring via the web interface. This hybrid setup provides a faster and more reliable recovery (Disaster Recovery).

Suggestions for improvement

The extensive work of acting as cluster administrator took time away from the development itself. Troubleshooting and fixing cluster configurations, as well as handling downtime (for example, when the school cluster did not restart correctly, which cost a whole working day), meant that some planned features had to be prioritized away:

1. **Permanent Login (Service Account)**: Currently, `deploy-main.yml` relies on an `OPENSHIFT_TOKEN` that expires after 24 hours. This creates problems during presentation and long-term operation, as the token must be updated manually. Since the fully automated deployment process is one of the project's greatest strengths, I would like to implement a more persistent login solution, for example via a Service Account with long-lived tokens.

2. **StatefulSets for Databases**: I have read about how one can scale up databases horizontally without risking data corruption, often by using StatefulSets instead of Deployments. Since I use my own images, it would have been very educational to implement and test this for both PostgreSQL and Redis to increase availability and performance.

3. **Security and HTTPS**: That the application lacks HTTPS support is a clear deficiency. With more time, I would have liked to configure TLS termination, which would likely have required adjustments in the frontend image or Ingress/Route configuration. Additionally, `NetworkPolicies` should be introduced to isolate the databases, so that only the backend service is allowed to communicate with them.

4. **Automated Tests and Reporting**: I have not had time to perform as extensive testing of the CI/CD flow as I would have wished. Although the system seems to work as intended, verification of edge cases is missing. I would have liked to implement more robust "health checks", clearer logging, and automatic reporting. In previous projects, I have used Telegram's API to send status reports directly to the mobile, which would have been a valuable addition here as well to quickly detect problems.

References

- **Project Repo (GitHub)**: https://github.com/SvartaStorken/argo_guestbook
  - Contains all source code, Kubernetes manifests, and documentation for this project.
- **Codebase for Images (GitLab)**: https://gitlab.com/SvartaStorken
  - Repo for previous assignments and source code that forms the basis for container images.
- **Container Registry (Quay.io)**: https://quay.io/repository/eksta_mannen/
  - Public registry where the project's built images are stored.
- **ArgoCD Documentation**: https://argo-cd.readthedocs.io/
  - Official documentation used for configuration and troubleshooting.
- **Course Material (GitHub)**: https://github.com/jonasbjork
  - Jonas Björk's repository with labs and instructions that formed the basis for the assignment.