CI/CD Project
Rust web app with multi-stage builds

By: Martin Popovski 186086
Faculty of Computer Science and Engineering, UKIM
Professors: Milos Jovanovik, Pance Ribarski
Date: 15.02.2023

1. Web app

Source code: https://github.com/martinkozle/cicd-project/blob/main/src/main.rs
Actix web framework for Rust: https://actix.rs/

The web app has only 2 end points for the purpose of this project. One is `/var/{name}` which fetches a variable from the variables table. The other one is `/var/{name}/{value}` which sets the value of a variable in the variables table. The purpose is to have a read and write operation to the database so I can showcase multiple replicas accessing the same database.
The app gets the database url from the DATABASE_URL environment variable. For development a local database is run with docker-compose-db.yaml and its url is set in a .env file, while for deployment it is set in the helm chart.
The web app is exposed to port 8080.

2. SQLx

SQLx Rust SQL toolkit: https://github.com/launchbadge/sqlx
Migrations performed using SQLx CLI: https://github.com/launchbadge/sqlx/tree/main/sqlx-cli
Docker Hub image: https://hub.docker.com/r/martinkozle/cicd-project-sqlx

SQLx performs compile-time checks on the queries. The macro creates a temporary transaction to the database with random data and uses the database response to validate the query, if the query isn't valid then the build will fail. This is why in a CI environment SQLX_OFFLINE=true environment variable needs to be set in order to not do live transactions to the database, which won't exist in CI and will fail the build. Instead you need to use `cargo sqlx prepare` which will save query metadata to sqlx-data.json so the SQL queries can be validated in offline mode CI environment without a live database.
There is only one migration which creates the variables table which has to be run on the database before the web app is started. In order to run this migration in deployment I created Dockerfile.sqlx which installs (builds) sqlx-cli and copies the built binary to a debian:bullseye-slim based final image.

3. Web App Dockerfile

Source code: https://github.com/martinkozle/cicd-project/blob/main/Dockerfile
Docker Hub image: https://hub.docker.com/r/martinkozle/cicd-project

The Dockerfile is set up as a multi-stage build where for the build stage I use the rust:1.67 image as the base image.
First, Cargo.toml is copied over and the dependencies are installed, then the source code is copied over so that the app can be built for release. This is done like this in 2 parts so that source code changes don't trigger a rebuild of all of the dependencies.
Then, we have 2 diverging stages, the test stage continues from the build stage, installs Clippy, a linter for Rust, and runs the linter as well as the tests. My app doesn't have any tests, so it passes with 0/0 tests, good enough for the purpose of the project.
The final stage uses debian:buster-slim as a base which is a very light weight base image, it copies over the release binary from the build stage which can then be run when deployed.

4. CI

Source code:
https://github.com/martinkozle/cicd-project/blob/main/.github/workflows/ci.yml
For Continuous Integrations I used GitHub Workflows.
It consists of 8 steps:

1. Checkout. Standard step for all GitHub workflows
2. Build Test. Builds the docker image with target test
3. Test. Runs the test docker image which runs the linter and tests, these have to pass
4. Login to Docker Hub. Uses secrets set in the GitHub repository to log in to my Docker Hub account using a personal access token as a password
5. Build Publish. Builds the final docker image, build stage should already be cached, so just the release stage
6. Publish. Publishes/pushes the docker image to Docker Hub
7. Build cicd-project-sqlx. Builds the docker image for migrations (Dockerfile.sqlx)
8. Publish cicd-project-sqlx. Publishes/pushes the docker image to Docker Hub

5. Helm chart

Source code: https://github.com/martinkozle/cicd-project/tree/main/charts/cicd-project
The Helm chart is composed of 2 deployments, 2 services, 1 persistent volume claim and 1 ingress. I will explain them in order of dependentcies.

Values

I have 3 groups of values defined. Database related environment variables. Image values, like repository and tag. Service values, like base name and port.
In hindsight I could have organized and named the values better, and maybe even add more values because some things are hard coded in the charts.

_helpers.tpl

Other than the default template helper functions, I created one of my own, called `cicd-project.databaseUrl`. It takes the database environment variables and generates the database URL that the web app and the migration container use.
I wanted to do this so I can learn a little bit about writing template functions in Helm.

Persistent Volume Claim

The volume is mounted as a ReadWriteOnce, meaning it can only be mounted by a single node, this will be our database node. It requests 1GiB of storage.

Database Deployment

Uses the postgres:15 image with database environment variables for user, password and default database. Mounts the persistent volume claim so that it can persist between restarts.

Database Service

Exposes the database port to the cluster.

Web Deployment

Deploys 2 replicas of the web app image. Before the web app container starts, 2 init containers must complete.
First a postgres:15-alpine image which waits until the database deployment is up. I chose the alpine version here because I am using the image just for the pg_isready cli.
The second init container is migrate. It uses the SQLx image I built in order to run migrations on the database. It is okay if all of the replicas execute this one because if the table already exists it still passes with exit code 0, so that the web app container is allowed to start. The template function is used to set the DATABASE_URL environment variable.
The web app container uses imagePullPolicy Always so that it is up to date with the latest image version. The template function is used here again to set the DATABASE_URL environment variable. I also wrote a readiness probe for this container which just pings the exposed port every 10 seconds.

Web Service

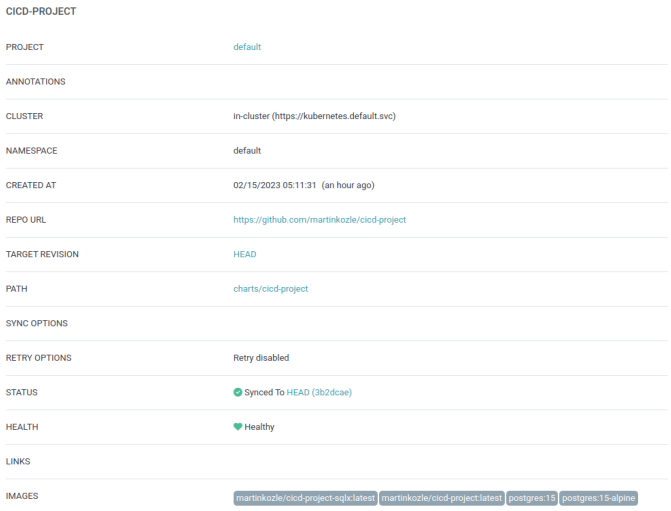
Exposes the web app 8080 port. Uses NodePort instead of ClusterIP for this one in order to support load balancing.

Ingress

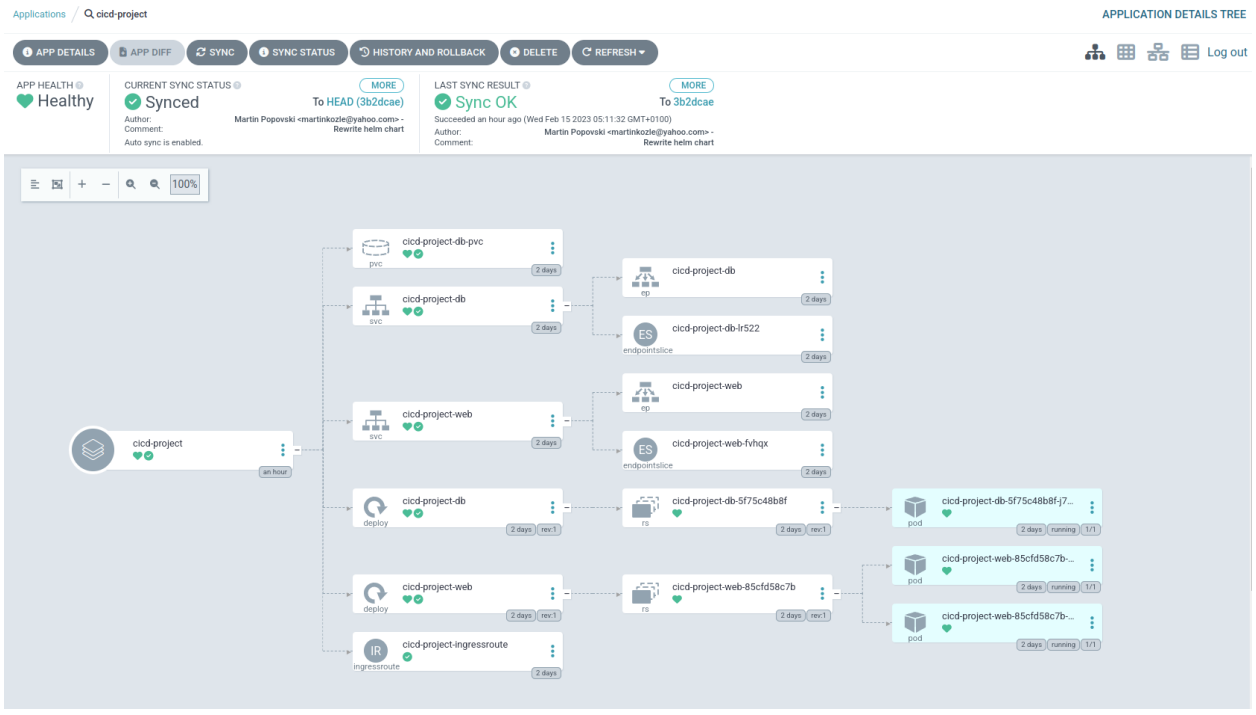
For the ingress I first tried with NGINX, but I couldn't get the load balancing to work, it kept sending all of the requests to just the first replica, it was pretty complicated to set up as well. Then I tried traefik, which ended up being a lot simpler to set up with load balancing. I exposed the web service to `localhost` so that I can try it out on my local machine. In an actual production I would need to set it to a subdomain on my domain. Traefik also supports setting specific routes, so on a single subdomain you can host multiple services.

6. Continuous Deployment

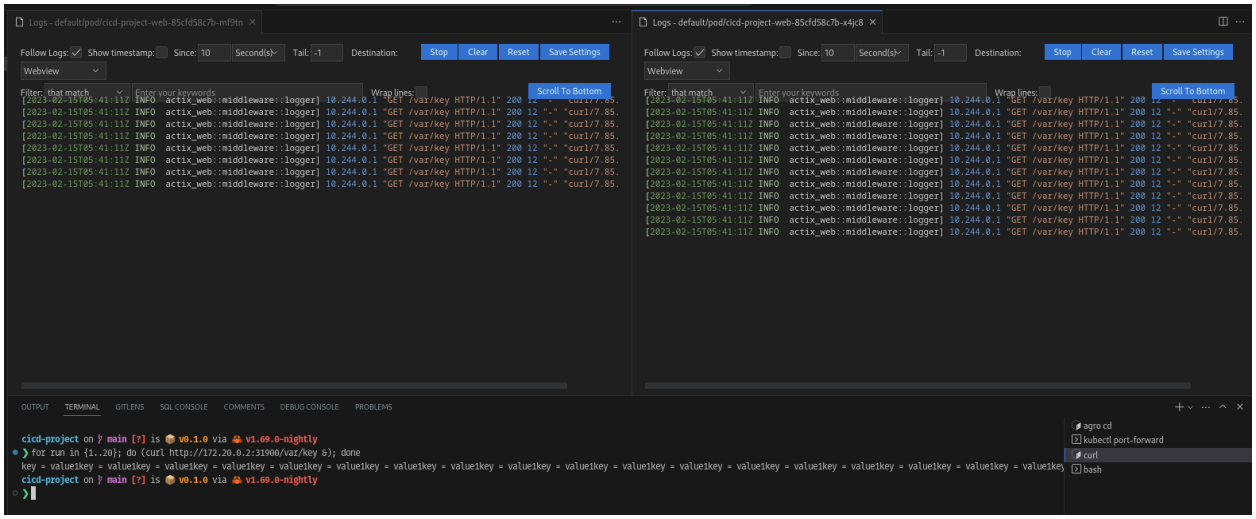
For the CD I used Argo CD. I created a new Application with the GitHub repository URL that I used.



And here is a screenshot of the final deployment graph:



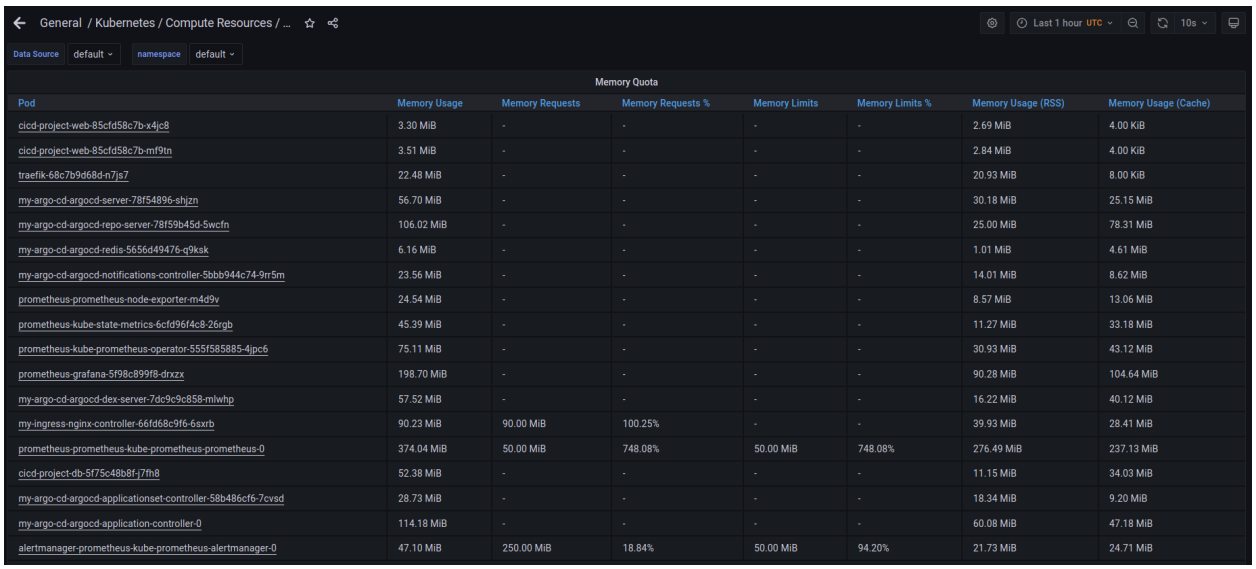
To test the load balancing I opened the logs of the 2 replicas side by side and I ran 20 async curl requests to the end point, and I got 8 requests to the first replica and 12 requests to the second replica.



I also tested in the browser, one replica can set a variable value in the database and then the other one will read the updated variable on the next request.

7. Logging and monitoring

Logs of all of the pods can be seen in Argo CD. While for monitoring I installed Prometheus Operator (https://prometheus-operator.dev/) in the cluster. It comes with Grafana, so I port forwarded the port and I logged in the web UI and I can see various monitoring stats of the system, the cluster and the running pods. For example we can see that the Rust web app uses very little memory in deployment (3-4MiB):



Pod	Memory Usage	Memory Requests	Memory Requests %	Memory Limits	Memory Limits %	Memory Usage (RSS)	Memory Usage (Cache)
cicd-project-web-85cfd58c7b-x4jc8	3.30 MiB	-	-	-	-	2.69 MiB	4.00 KiB
cicd-project-web-85cfd58c7b-mf9tn	3.51 MiB	-	-	-	-	2.84 MiB	4.00 KiB
traefik-68c7b9d68d-n7js7	22.48 MiB	-	-	-	-	20.93 MiB	8.00 KiB
my-argo-cd-argocd-server-78f54896-shjzn	56.70 MiB	-	-	-	-	30.18 MiB	25.15 MiB
my-argo-cd-argocd-repo-server-78f59b45d-5wcfn	106.02 MiB	-	-	-	-	25.00 MiB	78.31 MiB
my-argo-cd-argocd-redis-5656d49476-q9ksk	6.16 MiB	-	-	-	-	1.01 MiB	4.61 MiB
my-argo-cd-argocd-notifications-controller-5bbb944c74-9rr5m	23.56 MiB	-	-	-	-	14.01 MiB	8.62 MiB
prometheus-prometheus-node-exporter-m4d9v	24.54 MiB	-	-	-	-	8.57 MiB	13.06 MiB
prometheus-kube-state-metrics-6cfd96f4c8-26rgb	45.39 MiB	-	-	-	-	11.27 MiB	33.18 MiB
prometheus-kube-prometheus-operator-555f585885-4jpc6	75.11 MiB	-	-	-	-	30.93 MiB	43.12 MiB
prometheus-grafana-5f98c899f8-drxzx	198.70 MiB	-	-	-	-	90.28 MiB	104.64 MiB
my-argo-cd-argocd-dex-server-7dc9c9c858-mlwhp	57.52 MiB	-	-	-	-	16.22 MiB	40.12 MiB
my-ingress-nginx-controller-66fd68c9f6-6sxrb	90.23 MiB	90.00 MiB	100.25%	-	-	39.93 MiB	28.41 MiB
prometheus-prometheus-kube-prometheus-prometheus-0	374.04 MiB	50.00 MiB	748.08%	50.00 MiB	748.08%	276.49 MiB	237.13 MiB
cicd-project-db-5f75c48b8f-j7fh8	52.38 MiB	-	-	-	-	11.15 MiB	34.03 MiB
my-argo-cd-argocd-applicationset-controller-58b486cf6-7cvsd	28.73 MiB	-	-	-	-	18.34 MiB	9.20 MiB
my-argo-cd-argocd-application-controller-0	114.18 MiB	-	-	-	-	60.08 MiB	47.18 MiB
alertmanager-prometheus-kube-prometheus-alertmanager-0	47.10 MiB	250.00 MiB	18.84%	50.00 MiB	94.20%	21.73 MiB	24.71 MiB