Gitlab CI/CD

Agenda

1. gitlab ci/cd
    - Overview
    - Using the test - template
    - Examples running stages
    - Predefined Vars
    - Rules
    - Example Defining and using artifacts
2. gitlab ci/cd - docker
    - Docker image automatisiert bauen - gitlab registry
    - Docker image automatisiert bauen - docker hub
3. Kubernetes (Refresher)
    - Aufbau von Kubernetes
4. gitlab / Kubernetes CI/CD - old.old.schol with kubectl without agent)
    - gitlab kubectl without agent
5. gitlab / Kubernetes (gitops)
    - gitlab Kubernetes Agent with gitops - mode
6. gitlab / Kubernetes (CI/CD - old-school mit kubectl aber agent)
    - Vorteile gitlab-agent
    - Step 1: Installation gitlab-agent for kubernetes
    - Step 2: Debugging KUBE_CONTEXT - Community Edition
    - Step 3: gitlab-ci.yml setup for deployment and sample manifest
    - Documentation
7. gitlab / Kubernetes (CI/CD - Auto Devops)
    - Was ist Auto DevOps
    - Debugging KUBE_CONTEXT - Community Edition
8. Tipps&Tricks
    - Passwörter in Kubernetes verschlüsselt speichern
9. Documentation
    - gitlab ci/cd predefined variables

gitlab ci/cd

Overview

Pipelines

- The foundation of ci/cd are the pipelines
- You can either have preconfigured pipelines (using Auto DevOps)
- Or you can
  - Adjust them yourself (from Auto Devops, templates)
  - Create one from scratch
- Pipelines are either defined by Auto Devops or:
  - By .gitlab-ci.yml - file in the root-level - folder of your project
- There is also an editor under CI/CD -> Editor

Type of pipelines: Basic Pipeline

- Image: https://docs.gitlab.com/ee/ci/pipelines/pipeline_architectures.html#basic-pipelines
- (each stage runs concurrently)
- Default behaviour

```
## Example:
stages:
  - build
  - test
  - deploy

image: alpine

build_a:
  stage: build
  script:
    - echo "This job builds something."

build_b:
  stage: build
  script:
    - echo "This job builds something else."

test_a:
  stage: test
  script:
    - echo "This job tests something. It will only run when all jobs in the"
    - echo "build stage are complete."

test_b:
  stage: test
  script:
    - echo "This job tests something else. It will only run when all jobs in the"
    - echo "build stage are complete too. It will start at about the same time as
test_a."

deploy_a:
```

```
  stage: deploy
  script:
    - echo "This job deploys something. It will only run when all jobs in the"
    - echo "test stage complete."

deploy_b:
  stage: deploy
  script:
    - echo "This job deploys something else. It will only run when all jobs in the"
    - echo "test stage complete. It will start at about the same time as deploy_a."
```

Type of pipelines: DAG (Directed Acyclic Graph) Pipelines

- Image:
- Deploy_a can run, although build_b->test_b is not even ready
- Because gitlab knows the dependencies by keyword: needs:

```
## Example:
stages:
  - build
  - test
  - deploy

image: alpine

build_a:
  stage: build
  script:
    - echo "This job builds something quickly."

build_b:
  stage: build
  script:
    - echo "This job builds something else slowly."

test_a:
  stage: test
  needs: [build_a]
  script:
    - echo "This test job will start as soon as build_a finishes."
    - echo "It will not wait for build_b, or other jobs in the build stage, to
finish."

test_b:
  stage: test
  needs: [build_b]
  script:
    - echo "This test job will start as soon as build_b finishes."
    - echo "It will not wait for other jobs in the build stage to finish."

deploy_a:
  stage: deploy
  needs: [test_a]
```

```
  script:
    - echo "Since build_a and test_a run quickly, this deploy job can run much
earlier."
    - echo "It does not need to wait for build_b or test_b."

deploy_b:
  stage: deploy
  needs: [test_b]
  script:
    - echo "Since build_b and test_b run slowly, this deploy job will run much later."
```

Type of pipelines: Child- / Parent - Pipelines

- https://docs.gitlab.com/ee/ci/pipelines/pipeline_architectures.html#child--parent-pipelines
- in Example: two types of things that could be built independently.
  - Combines child and DAG in this case
  - Trigger is used to start the child - pipeline
- Include:
  - not to repeat yourself + eventually as template (using . - prefix)
- Rules:
  - are like conditions

```
## Example
## File 1: .gitlab-ci.yml
stages:
  - triggers

trigger_a:
  stage: triggers
  trigger:
    include: a/.gitlab-ci.yml
  rules:
    - changes:
        - a/*

trigger_b:
  stage: triggers
  trigger:
    include: b/.gitlab-ci.yml
  rules:
    - changes:
        - b/*
```

```
## File 2: a/.gitlab-ci.yml
stages:
  - build
  - test
  - deploy

image: alpine
```

```
build_a:
  stage: build
  script:
    - echo "This job builds something."

test_a:
  stage: test
  needs: [build_a]
  script:
    - echo "This job tests something."

deploy_a:
  stage: deploy
  needs: [test_a]
  script:
    - echo "This job deploys something."
```

```
## File 3: a/.gitlab-ci.yml
stages:
  - build
  - test
  - deploy

image: alpine

build_b:
  stage: build
  script:
    - echo "This job builds something else."

test_b:
  stage: test
  needs: [build_b]
  script:
    - echo "This job tests something else."

deploy_b:
  stage: deploy
  needs: [test_b]
  script:
    - echo "This job deploys something else."
```

Type of pipelines: Ref:

- https://docs.gitlab.com/ee/ci/pipelines/pipeline_architectures.html

Stages

- Stages run one after each other
- They default to: build, test, deploy (if you do not define any)
- If you want to have less, you have to define which
- Reference:

Jobs

- Jobs define what to do within the stages
- Normally jobs are run concurrently in each stage
- Reference:

Using the test - template

Example Walkthrough

```
## Schritt 1: Neues Repo aufsetzen

## Setup a new repo
## Setting:

## o Public, dann bekommen wir mehr Rechenzeit
## o  No deployment planned
## o No SAST
## o Add README.md

## Using naming convention
## Name it however you want, but have you tln - nr inside
## e.g.
## test-artifacts-tln1


## Schritt 2: Ein Standard-Template als Ausgangsbasis holen
## Get default ci-Template
CI-CD -> Pipelines -> Try Test-Template

## Testtemplate wird in file gitlab-ci.yaml angelegt.
## Es erscheint unter: CI-CD -> Editor

1x speichern und committen.

## Jetzt wird es in der Pipeline ausgeführt.
```

Examples running stages

Running default stages

- build, test, deploy are stages set by default

```
## No stages defined, so build, test and deploy are run

build-job:       # This job runs in the build stage, which runs first.
  stage: build
  script:
    - echo "Compiling the code..."
    - echo "Compile complete."

unit-test-job:   # This job runs in the test stage.
  stage: test    # It only starts when the job in the build stage completes
successfully.
```

```
  script:
    - echo "Running unit tests... This will take about 60 seconds."
    - sleep 1
    - echo "Code coverage is 90%"

deploy-job:      # This job runs in the deploy stage.
  stage: deploy  # It only runs when *both* jobs in the test stage complete
successfully.
  script:
    - echo "Deploying application..."
    - echo "Application successfully deployed."
```

only run some

```
## einfaches stages - keyword ergänzen und die stages die man haben will
stages:
  - build
  - deploy


build-job:       # This job runs in the build stage, which runs first.
  stage: build
  script:
    - echo "Compiling the code..."
    - echo "Compile complete."

## unit-test-job wurde gelöscht

deploy-job:      # This job runs in the deploy stage.
  stage: deploy  # It only runs when *both* jobs in the test stage complete
successfully.
  script:
    - echo "Deploying application..."
    - echo "Application successfully deployed."
```

- Danach sich die Pipelines anschauen (CI/CD -> Pipeline)

Predefined Vars

Example to show them

```
stages:
  - build

show_env:
  stage: build
  scripts:
    - env
    - pwd
```

Reference

- https://docs.gitlab.com/ee/ci/variables/predefined_variables.html

Rules

Ref:

- https://docs.gitlab.com/ee/ci/jobs/job_control.html#specify-when-jobs-run-with-rules

Example Defining and using artifacts

What is it ?

```
Jobs can output an archive of files and directories. This output is known as a job
artifact.
You can download job artifacts by using the GitLab UI or the API.
```

Example: Creating an artifact

```
## .gitlab-ci.yml

stages:
  - build

create_txt:
  stage: build
  script:
    - echo "hello" > ergebnis.txt
  artifacts:
    paths:
      - ergebnis.txt
```

Example creating artifacts with wildcards and different name

```
## .gitlab-ci.yml
stages:
  - build
create_txt:
  stage: build
  script:
    - mkdir -p path/my-xyz
    - echo "hello" > path/my-xyz/ergebnis.txt
    - mkdir -p path/some-xyz
    - echo "some" > path/some-xyz/testtext.txt
  artifacts:
    name: meine-daten
    paths:
      - path/*xyz/*
```

Artifakte und Name aus Variable vergeben

- If your branch-name contains forward slashes
  - (for example feature/my-feature)

- it’s advised to use $CI_COMMIT_REF_SLUG instead of $CI_COMMIT_REF_NAME
  - for proper naming of the artifact.

```
## .gitlab-ci.yml
stages:
  - build
create_txt:
  stage: build
  script:
    - mkdir -p path/my-xyz
    - echo "hello" > path/my-xyz/ergebnis.txt
    - mkdir -p path/some-xyz
    - echo "some" > path/some-xyz/testtext.txt
  artifacts:
    name: "$CI_JOB_NAME-$CI_COMMIT_REF_NAME"
    paths:
      - path/*xyz/*
```

Alle files in einem Verzeichnis recursive

```
## .gitlab-ci.yml
stages:
  - build
create_txt:
  stage: build
  script:
    - mkdir -p path/my-xyz
    - echo "toplevel" > path/you-got-it.txt
    - echo "hello" > path/my-xyz/ergebnis.txt
    - mkdir -p path/some-xyz
    - echo "some" > path/some-xyz/testtext.txt
  artifacts:
    paths:
      - path/
```

Artifakte und Bedingungen

```
## nur artifact erstellen, wenn ein commit-tag gesetzt ist.
## Gibt es kein commit-tag ist diese Variable NICHT GESETZT.


### .gitlab-ci.yml
stages:
  - build

output_something:
  stage: build
  script:
    - echo "just writing something"
```

```
    - env
    - echo "CI_COMMIT_TAG:..$CI_COMMIT_TAG.."

create_txt:
  stage: build
  script:
    - mkdir -p path/my-xyz
    - echo "toplevel" > path/you-got-it.txt
    - echo "hello" > path/my-xyz/ergebnis.txt
    - mkdir -p path/some-xyz
    - echo "some" > path/some-xyz/testtext.txt
    - env
    - echo "TAG ? $CI_COMMIT_TAG"
  artifacts:
    paths:
      - path/

  rules:
    - if: $CI_COMMIT_TAG
```

- Test 1: committen und Pipeline beobachten
- Test 2: Tag über repository > Tags erstellen und nochmal Pipeline beobachten

Passing artifacts between stages (enabled by default)

```
image: ubuntu:20.04

## stages are set to build, test, deploy by default

build:
  stage: build
  script:
    - echo "in building..." >> ./control.txt
  artifacts:
    paths:
    - control.txt
    expire_in: 1 week

my_unit_test:
  stage: test
  script:
    - ls
    - cat control.txt
    - echo "now in unit testing ..." >> ./control.txt
  artifacts:
    paths:
    - control.txt
    expire_in: 1 week

deploy:
  stage: deploy
  script:
```

```
    - ls
    - cat control.txt
```

Passing artifacts between stages (enabled by default) - only writing it in stage: build

```
## only change in stage: build
image: ubuntu:20.04

## stages are set to build, test, deploy by default

build:
  stage: build
  script:
    - echo "in building..." >> ./control.txt
  artifacts:
    paths:
    - control.txt
    expire_in: 1 week

my_unit_test:
  stage: test
  script:
    - cat control.txt

deploy:
  stage: deploy
  script:
    - ls
    - cat control.txt
```

Passing artifacts (+ommitting test - stage)

- You can decide in which state you need the artifacts

```
## only change in stage: build
image: ubuntu:20.04

## stages are set to build, test, deploy by default

build:
  stage: build
  script:
    - echo "in building..." >> ./control.txt
  artifacts:
    paths:
    - control.txt
    expire_in: 1 week

my_unit_test:
```

```
  stage: test
  dependencies: []
  script:
    - echo "no control.txt here"
    - ls -la

deploy:
  stage: deploy
  script:
    - ls
    - cat control.txt
```

Using the gitlab - artifacts api

API - Reference:

- https://docs.gitlab.com/ee/api/job_artifacts.html

Reference:

- https://docs.gitlab.com/ee/ci/pipelines/job_artifacts.html

gitlab ci/cd - docker

Docker image automatisiert bauen - gitlab registry

Dockerfile - RootLevel

```
FROM ubuntu:latest
RUN touch MYFILE
```

.gitlab-ci.yml

```
stages:          # List of stages for jobs, and their order of execution
  - build

build-image:       # This job runs in the build stage, which runs first.
  stage: build
  image: docker:20.10.10
  services:
     - docker:20.10.10-dind
  script:
    - echo "user:"$CI_REGISTRY_USER
    - echo "pass:"$CI_REGISTRY_PASSWORD
    - echo "registry:"$CI_REGISTRY
    - echo $CI_REGISTRY_PASSWORD | docker login -u $CI_REGISTRY_USER $CI_REGISTRY --
password-stdin
    - docker build -t $CI_REGISTRY_IMAGE .
    - docker push $CI_REGISTRY_IMAGE
    - echo "BUILD for $CI_REGISTRY_IMAGE done"
```

Docker image automatisiert bauen - docker hub

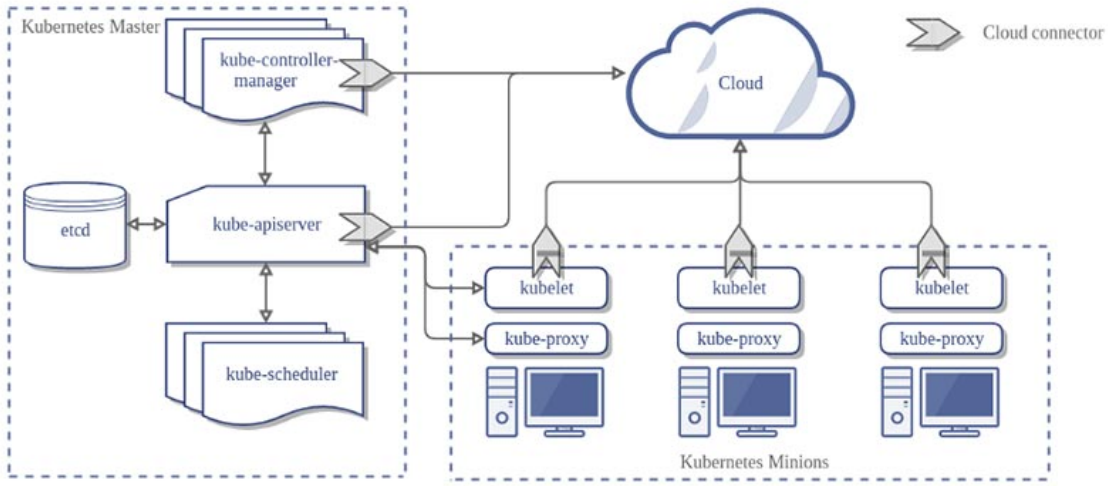
Docker Hub (gitlab-ci.yml)

```
variables:
  DOCKER_REGISTRY_USER: $DOCKER_REGISTRY_USER
  DOCKER_REGISTRY_PASSWORD: $DOCKER_REGISTRY_PASSWORD
  DOCKER_REGISTRY_IMAGE: dockertrainereu/jochen1:latest

stages:          # List of stages for jobs, and their order of execution
  - build

build-image:       # This job runs in the build stage, which runs first.
  stage: build
  image: docker:20.10.10
  services:
     - docker:20.10.10-dind
  script:
    - echo "user:"$DOCKER_REGISTRY_USER
    - echo "pass:"$DOCKER_REGISTRY_PASSWORD
    - echo $DOCKER_REGISTRY_PASSWORD | docker login -u $DOCKER_REGISTRY_USER --
password-stdin
    - docker build -t $DOCKER_REGISTRY_IMAGE .
    - docker push $DOCKER_REGISTRY_IMAGE
    - echo "BUILD for $DOCKER_REGISTRY_IMAGE done"
```

Kubernetes (Refresher)

Aufbau von Kubernetes

Schaubild



Komponenten / Grundbegriffe

Master (Control Plane)

Aufgaben

- Der Master koordiniert den Cluster
- Der Master koordiniert alle Aktivitäten in Ihrem Cluster
  - Planen von Anwendungen
  - Verwalten des gewünschten Status der Anwendungen
  - Skalieren von Anwendungen
  - Rollout neuer Updates.

Komponenten des Masters

ETCD

- Verwalten der Konfiguration des Clusters (key/value - pairs)

KUBE-CONTROLLER-MANAGER

- Zuständig für die Überwachung der Stati im Cluster mit Hilfe von endlos loops.
- kommuniziert mit dem Cluster über die kubernetes-api (bereitgestellt vom kube-api-server)

KUBE-API-SERVER

- provides api-frontend for administration (no gui)
- Exposes an HTTP API (users, parts of the cluster and external components communicate with it)
- REST API

KUBE-SCHEDULER

- assigns Pods to Nodes.
- scheduler determines which Nodes are valid placements for each Pod in the scheduling queue (according to constraints and available resources)
- The scheduler then ranks each valid Node and binds the Pod to a suitable Node.
- Reference implementation (other schedulers can be used)

Nodes

- Nodes (Knoten) sind die Arbeiter (Maschinen), die Anwendungen ausführen
- Ref: https://kubernetes.io/de/docs/concepts/architecture/nodes/

Pod/Pods

- Pods sind die kleinsten einsetzbaren Einheiten, die in Kubernetes erstellt und verwaltet werden können.
- Ein Pod (übersetzt Gruppe) ist eine Gruppe von einem oder mehreren Containern
  - gemeinsam genutzter Speicher- und Netzwerkressourcen
  - Befinden sich immer auf dem gleich virtuellen Server

Control Plane Node (former: master) - components

Node (Minion) - components

General

- On the nodes we will rollout the applications

kubelet

```
Node Agent that runs on every node (worker)
Er stellt sicher, dass Container in einem Pod ausgeführt werden.
```

Kube-proxy

- Läuft auf jedem Node
- = Netzwerk-Proxy für die Kubernetes-Netzwerk-Services.
- Kube-proxy verwaltet die Netzwerkkommunikation innerhalb oder außerhalb Ihres Clusters.

Referenzen

- https://www.redhat.com/de/topics/containers/kubernetes-architecture

gitlab / Kubernetes CI/CD - old.old.schol with kubectl without agent)

gitlab kubectl without agent

Walkthrough

```
1. Create new repo on gitlab
2. CI/CD workflow aktivieren, in dem wir auf das Menü CI/CD klicken
-> Get started with GitLab CI/CD -> Use Template

3. file .gitlab-ci.yml anpassen
```

```
variables:
   KUBECONFIG_SECRET: $KUBECONFIG_SECRET

build-version:       # This job runs in the build stage, which runs first.
  stage: build
  image:
##    name: dtzar/helm-kubectl:3.7.1
    name: bitnami/kubectl:latest
    entrypoint: [""]
  script:
    - echo "Show use our repo"
    - cd $CI_PROJECT_DIR
    - ls -la
    - kubectl version --client
    - echo "kubeconfig aufsetzen"
    - mkdir -p ~/.kube
    - echo "$KUBECONFIG_SECRET" > ~/.kube/config
    - ls -la ~/.kube/config
    - cat ~/.kube/config
    - kubectl cluster-info
    - kubectl get pods
    - echo "Deploying..."
## - kubectl apply -f manifests/deploy.yml
    - sleep 2
    - echo "And now..."
    - kubectl get pods
```

```
## manifests anlegen in manifests/01-deploy.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
```

```
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

```
4. Zugangsdaten auf master-server auslesen und in den Zwischenspeicher kopieren

microk8s config

5. Im Repo und SETTINGS -> CI/CD -> Variables

variable
KUBECONFIG_SECRET
mit Inhalt aus 4. setzen

MASKED und PROTECTED Nicht aktivieren

Speicern

6. im repo folgende Datei anlegen.

## manifests/deploy.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: echo-server-deployment-from-gitlab
  labels:
    app: echo-server
spec:
  selector:
    matchLabels:
      app: echo-server
  replicas: 1
  template:
    metadata:
      annotations:
      labels:
        app: echo-server
    spec:
      containers:
        - name: echo-server
          image: hashicorp/http-echo
          imagePullPolicy: IfNotPresent
          args:
```

```
            - -listen=:8080
            - -text="hello NEWNEW world"


7. in CI/CD - Menü -> Pipelines gucken, ob die Pipeline durchläuft und die
detaillierte Ausgabe anzeigen

8. Änderung in deploy.yml durchführen.

z.B. Text: hello NEWNEW world in hello OLDNEW world ändern.

9. Prüfen ob neuer Pod erstellt wird durch überprüfen der Ausgabe in Pipelines
```

```
deploy:
  image:
    name: bitnami/kubectl:latest
    entrypoint: ['']
  script:
    - kubectl config get-contexts
    - kubectl config use-context path/to/agent/repository:agent-name
    - kubectl get pods
```

A bit nicer:

```
https://sanderknape.com/2019/02/automated-deployments-kubernetes-gitlab/
```

gitlab / Kubernetes (gitops)

gitlab Kubernetes Agent with gitops - mode

Create a new project

```
  * Name: kubernetes-gitops-tn<nr>
  * e.g. k8s-gitops-tn1
  * Public
  * Readme.md
  * Disabled -> SAST
```

Setting up the config (gitops - Style) - sample not yet working

- Create an agent configuration file
- .gitlab/agents/name/
- We will use the following convention or name in the training:
  - gitlab-agent-tn-nr- - gitlab-agent-tn1
- https://docs.gitlab.com/ee/user/clusters/agent/install/index.html#create-an-agent-configuration-file
- Then in that folder we need to place a configuration - file - config.yaml - NOT !!! - config.yml
  - THE CONFIGURATION WILL NOT GET DETECTED
- Content see below:

```
## gitops:
## tln1 ersetzen, durch eigene teilnemer - nr. bei default_namespace
gitops:
  manifest_projects:
  - id: dummyhoney/kubernetes-gitops-tn1
    default_namespace: tln1
    paths:
      # Read all YAML/YML  files from this directory.
    - glob: '/manifests/**/*.{yaml,yml}'
    # Read all .yaml files from team2/apps and all subdirectories.
    # - glob: '/team2/apps/**/*.yaml'
    # If 'paths' is not specified or is an empty list, the configuration below is
used.
    # - glob: '/**/*.{yaml,yml,json}'
    reconcile_timeout: 3600s
    dry_run_strategy: none
    prune: true
    prune_timeout: 3600s
    prune_propagation_policy: foreground
    inventory_policy: must_match
```

- Reference: https://docs.gitlab.com/ee/user/clusters/agent/gitops.html

Connect the cluster under Infrastructure -> Kubernetes

- Select the agent and click Register
- Copy the token to clipboard

Install the agent in the cluster using your client (Linux)

```
## Install helm3 if not present yet.
## Ubuntu Style:
snap install --classic helm
## kubectl needs to be working properly: kubectl cluster-info
## Can you see the cluster ?
helm repo add gitlab https://charts.gitlab.io
helm repo update
## adjust namespace tn<teilnehmer> e.g. gitlab-agent-tn1
helm upgrade --install gitlab-agent gitlab/gitlab-agent \
    --namespace gitlab-agent-tn1 \
    --create-namespace \
    --set config.token=<config-token-you-copied to clipboard> \
    --set config.kasAddress=wss://kas.gitlab.com
```

Check if it has been registered

- Look into infrastructure - kubernetes

Creating sample manifests file

```
## manifests/project1/web/bitnami-nginx-deploy.yml

apiVersion: apps/v1
```

```
kind: Deployment
metadata:
  name: nginx-deployment-gitops
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Checking the logs

```
kubectl logs -n gitlab-agent-tn1 deploy/gitlab-agent
```

gitlab / Kubernetes (CI/CD - old-school mit kubectl aber agent)

Vorteile gitlab-agent

Disadvantage of solution before gitlab agent

- the requirement to open up the cluster to the internet, especially to GitLab
- the need for cluster admin rights to get the benefit of GitLab Managed Clusters
- exclusive support for push-based deployments that might not suit some highly regulated industries

Advantage

- Solved the problem of weaknesses.

Technical

- Connected to Websocket Stream of KAS-Server
- Registered with gitlab - project

Reference:

- https://about.gitlab.com/blog/2020/09/22/introducing-the-gitlab-kubernetes-agent/

Step 1: Installation gitlab-agent for kubernetes

Steps

```
### Step 1:

Create New Repository -
name: b-tln<nr>

With
README.md
```

```
### Step 2: config für agents anlegen

## .gitlab/agents/gitlab-tln<nr>/config.yaml # Achtung kein .yml wird sonst nicht
erkannt.
## mit folgendem Inhalt

ci_access:
  projects:
    - id: dummyhoney/b-tln<nr>
```

```
### Step 3:
## agent registrieren / Cluster connecten

Infrastruktur > Kubernetes Clusters -> Connect a cluster (Agent)

Jetzt solltest du den Agent auswählen können und klickt auf Register
```

```
### Step 4:
## Du erhältst die Anweisungen zum Installieren und wandelst das ein bisschen ab,
## für das Training:

## Den token verwendest du aus der Anzeige
## tln1 ersetzt durch jeweils (2x) durch Deine Teilnehmer-Nr.
helm upgrade --install gitlab-agent gitlab/gitlab-agent --namespace tln<nr> --create-
namespace --set config.token=<token-from-screen>
```

Step 2: Debugging KUBE_CONTEXT - Community Edition

Why ?

```
kubectl does not work, because KUBECONFIG is not set properly
```

What does not work ?

```
## This overwrites auto devops completely
##.gitlab-ci.yml
deploy:
  image:
    name: bitnami/kubectl:latest
    entrypoint: [""]
  script:
    - kubectl cluster-info
```

Test Context

```
## This overwrites auto devops completely
##.gitlab-ci.yml
deploy:
  image:
    name: bitnami/kubectl:latest
    entrypoint: [""]
  script:
    - set
    - kubectl config get-contexts
## this will be the repo and the name of the agent
## Take it from the last block
## you will see it from the pipeline
    - kubectl config use-context dummyhoney/tln1:gitlab-tln1
    - kubectl config set-context --current --namespace tln1
    - kubectl get pods
    - ls -la
    - id
```

Fix by setting KUBE_CONFIG

```
## This is a problem in the community edition (CE)
## We need to fix it like so.
## Adjust it to your right context
## IN Settings -> CI/CD -> Variables
KUBE_CONFIG dummyhoney/spring-autodevops-tln1:gitlab-devops-tn1
```

Step 3: gitlab-ci.yml setup for deployment and sample manifest

Schritt 1: manifests - Struktur einrichten

```
## vi manifests/prod/01-pod.yml

apiVersion: v1
kind: Pod
metadata:
  name: nginx-static-web2
  labels:
    webserver: nginx
spec:
  containers:
  - name: web
    image: bitnami/nginx
```

Schritt 2: gitlab-ci.yml mit kubectl apply --recursive -f

```
## CI-CD -> Editor oder .gitlab-ci.yml im Wurzelverzeichnis
## only change in stage: build
image:
  name: bitnami/kubectl
```

```
  entrypoint: [""]

deploy:
  stage: deploy
  script:
    - set
    - kubectl config get-contexts
    - kubectl config use-context dummyhoney/b-tln1:gitlab-tln1
    - kubectl config set-context --current --namespace tln1
    - ls -la
    - kubectl apply --recursive -f manifests/prod
```

Schritt 3: pipeline anschauen

- War es erfolgreich - kein Fehler ?

Schritt 4: Sichtprüfen mit kubectl über Client (lokaler Rechner/Desktop)

```
kubectl get pods | grep web2
```

Documentation

- https://docs.gitlab.com/ee/user/clusters/agent/ci_cd_workflow.html

gitlab / Kubernetes (CI/CD - Auto Devops)

Was ist Auto DevOps

Debugging KUBE_CONTEXT - Community Edition

Why ?

```
kubectl does not work, because KUBECONFIG is not set properly
```

What does not work ?

```
## This overwrites auto devops completely
##.gitlab-ci.yml
deploy:
  image:
    name: bitnami/kubectl:latest
    entrypoint: [""]
  script:
    - kubectl cluster-info
```

Test Context

```
## This overwrites auto devops completely
##.gitlab-ci.yml
deploy:
  image:
    name: bitnami/kubectl:latest
    entrypoint: [""]
```

```
  script:
    - set
    - kubectl config get-contexts
## this will be the repo and the name of the agent
## Take it from the last block
## you will see it from the pipeline
    - kubectl config use-context dummyhoney/tln1:gitlab-tln1
    - kubectl config set-context --current --namespace tln1
    - kubectl get pods
    - ls -la
    - id
```

Fix by setting KUBE_CONFIG

```
## This is a problem in the community edition (CE)
## We need to fix it like so.
## Adjust it to your right context
## IN Settings -> CI/CD -> Variables
KUBE_CONFIG dummyhoney/spring-autodevops-tln1:gitlab-devops-tn1
```

Tipps&Tricks

Passwörter in Kubernetes verschlüsselt speichern

2 Komponenten

- Sealed Secrets besteht aus 2 Teilen
  - kubeseal, um z.B. die Passwörter zu verschlüsseln
  - Dem Operator (ein Controller), der das Entschlüsseln übernimmt

Schritt 1: Walkthrough - Client Installation (als root)

```
## Binary für Linux runterladen, entpacken und installieren
## Achtung: Immer die neueste Version von den Releases nehmen, siehe unten:
## Install as root
cd /usr/src
wget https://github.com/bitnami-labs/sealed-
secrets/releases/download/v0.17.5/kubeseal-0.17.5-linux-amd64.tar.gz
tar xzvf kubeseal-0.17.5-linux-amd64.tar.gz
install -m 755 kubeseal /usr/local/bin/kubeseal
```

Schritt 2: Walkthrough - Server Installation mit kubectl client

```
## auf dem Client
## cd
## mkdir manifests/seal-controller/ #
## cd manifests/seal-controller
## Neueste Version
wget https://github.com/bitnami-labs/sealed-
secrets/releases/download/v0.17.5/controller.yaml
kubectl apply -f controller.yaml
```

Schritt 3: Walkthrough - Verwendung (als normaler/unpriviligierter Nutzer)

```
kubeseal --fetch-cert

## Secret - config erstellen mit dry-run, wird nicht auf Server angewendet (nicht an
Kube-Api-Server geschickt)
kubectl create secret generic basic-auth --from-literal=APP_USER=admin --from-
literal=APP_PASS=change-me --dry-run=client -o yaml > basic-auth.yaml
cat basic-auth.yaml

## öffentlichen Schlüssel zum Signieren holen
kubeseal --fetch-cert > pub-sealed-secrets.pem
cat pub-sealed-secrets.pem

kubeseal --format=yaml --cert=pub-sealed-secrets.pem < basic-auth.yaml > basic-auth-
sealed.yaml
cat basic-auth-sealed.yaml

## Ausgangsfile von dry-run löschen
rm basic-auth.yaml

## Ist das secret basic-auth vorher da ?
kubectl get secrets basic-auth

kubectl apply -f basic-auth-sealed.yaml

## Kurz danach erstellt der Controller aus dem sealed secret das secret
kubectl get secret
kubectl get secret -o yaml
```

```
## Ich kann dieses jetzt ganz normal in meinem pod verwenden.
## Step 3: setup another pod to use it in addition
## vi 02-secret-app.yml
apiVersion: v1
kind: Pod
metadata:
  name: secret-app
spec:
  containers:
    - name: env-ref-demo
      image: nginx
      envFrom:
      - secretRef:
          name: basic-auth
```

Hinweis: Ubuntu snaps

```
Installation über snap funktioniert nur, wenn ich auf meinem Client
ausschliesslich als root arbeite
```

Wie kann man sicherstellen, dass nach der automatischen Änderung des Secretes, der Pod bzw. Deployment neu gestartet wird ?

- https://github.com/stakater/Reloader

Ref:

- Controller: https://github.com/bitnami-labs/sealed-secrets/releases/

Documentation

gitlab ci/cd predefined variables

- https://docs.gitlab.com/ee/ci/variables/predefined_variables.html