Gitlab CI/CD

Agenda

1. gitlab ci/cd (Überblick)
   - Architecture
   - Overview/Pipelines
   - SaaS vs. On-Premise (Self Hosted)
2. Hintergründe
   - Warum before_script ?
   - GIT_STRATEGY usw.
3. gitlab ci/cd (Praxis I)
   - Using the test - template
   - Examples running stages
   - Predefined Vars
   - Variablen definieren
   - Variablen überschreiben/leeren
   - Rules
   - Example Defining and using artifacts
4. gitlab ci/cd (Praxis II)
   - Mehrzeile Kommandos in gitlab ci-cd ausführen
   - Kommandos auf Zielsystem mit ssh ausführen (auch multiline)
5. gitlab-ci/cd - Workflows
   - Workflows + only start by starting pipeline
   - Templates for branch and merge request workflow
6. gitlab-ci/cd - Variables
   - Variablen in Pipelines Web-Dialog anzeigen
7. gitlab - ci/cd - Pipelines strukturieren / Templates
   - Includes mit untertemplates
   - Parent/Child Pipeline
   - Multiproject Pipeline / Downstream
   - Vorgefertigte Templates verwenden
   - Arbeiten mit extend und anchor - Dinge wiederverwenden
8. gitlab - wann laufen jobs ?
   - Job nur händisch über Pipelines starten
   - Auch weiterlaufen, wenn Job fehlschlägt
9. gitlab ci/cd docker
   - Docker image automatisiert bauen - gitlab registry
10. gitlab ci/cd docker compose
    - Docker compose local testen
    - Docker compose über ssh
    - Docker compose classic über scp
11. Performance / Caching
    - Wann ist eine Pipeline langsam ?
    - Caching->cache in gitlab
12. Matrix = Loops
    - Matrix=Loops, Jobs mehrmals ausführen
13. Monitoring gitlab
    - Monitoring gitlab good or bad ?
14. Security
    - Container Scanning
15. Environments
    - Environments
16. Documentation
    - gitlab ci/cd predefined variables
    - .gitlab-ci.yml Reference
    - Referenz: global -> workflow
    - Referenz: global -> default

17. Documentation - Includes

    - includes
    - includes -> rules
    - includes -> rules -> variables
    - includes -> templates -> override-configuration
    - includes -> defaults

18. Documentation - Instances Limits

    - applicaton limits

Backlog I

1. gitlab ci/cd (Überblick)

    - Jenkins mit Gitlab vs. gitlab ci/cd

2. gitlab - setzen von Variablen

    - Variablen für angepasste Builds verwenden und scheduled pipeline

3. Exercises

    - build with maven and using artifacts

4. gitlab ci/cd - docker

    - Docker image automatisiert bauen - docker hub
    - Selbst gebauten Container manuell ausführen
    - Neues Image in gitlab ci/cd aus gitlab registry verwenden

5. Tipps&Tricks

    - Image/Container debuggen in mit gitlab ci/cd

Backlog II

1. Kubernetes (Refresher)

    - Aufbau von Kubernetes

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

    - gitlab kubectl without agent

3. gitlab / Kubernetes (gitops)

    - gitlab Kubernetes Agent with gitops - mode

4. 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

5. gitlab / Kubernetes (CI/CD - Auto Devops)

    - Was ist Auto DevOps
    - Debugging KUBE_CONTEXT - Community Edition

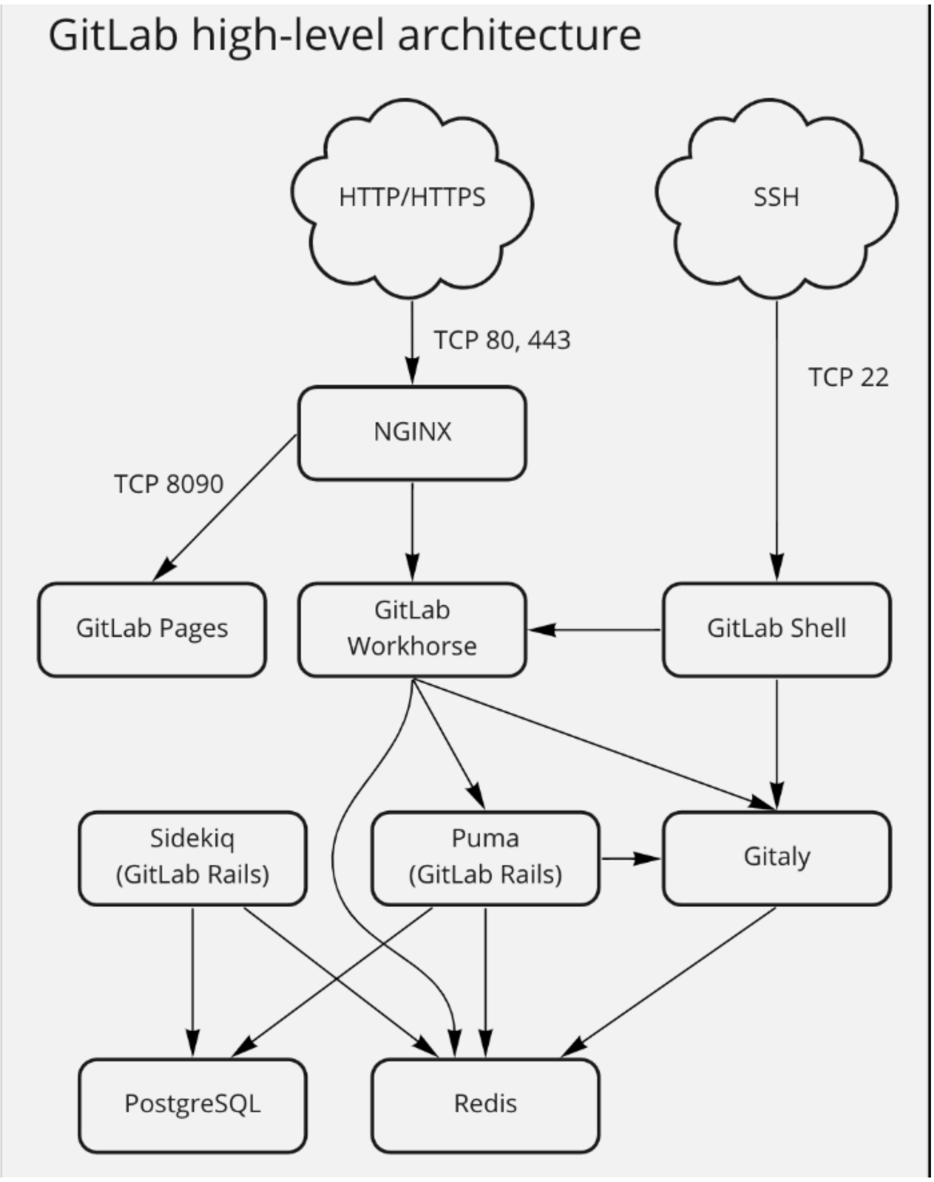
6. Tipps&Tricks

    - Passwörter in Kubernetes verschlüsselt speichern

gitlab ci/cd (Überblick)

Architecture

Overview



Components

```
GitLab Workhorse
================
-> smart reverse proxy
GitLab Workhorse is a smart reverse proxy for GitLab. It handles “large” HTTP requests such as file downloads, file uploads, Git
push/pull and Git archive downloads.

GitLab Shell
============
GitLab Shell handles Git SSH sessions for GitLab and modifies the list of authorized keys. GitLab Shell is not a Unix shell nor a
replacement for Bash or Zsh.
GitLab supports Git LFS authentication through SSH.
--> Alternative notwendig statt openssh

When you access the GitLab server over SSH, GitLab Shell then:

1. Limits you to predefined Git commands (git push, git pull, git fetch).
```

```
2. Calls the GitLab Rails API to check if you are authorized, and what Gitaly server your repository is on.
3. Copies data back and forth between the SSH client and the Gitaly server.

Sidekiq (GitLab Rails -> gitlab rails console)
======================

Simple, efficient background processing for Ruby.

Sidekiq uses threads to handle many jobs at the same time in the same process. It does not require Rails but will integrate
tightly with Rails to make background processing dead simple.

Puma (Gitlab Rails -> gitlab rails console)
===================
Puma is a fast, multi-threaded, and highly concurrent HTTP 1.1 server for Ruby applications. It runs the core Rails application
that provides the user-facing features of GitLab.

Gitaly
======
Dein Repo liegt auf einem bestimmten gitaly - Server
Gitaly provides high-level RPC access to Git repositories. It is used by GitLab to read and write Git data.

Gitaly is present in every GitLab installation and coordinates Git repository storage and retrieval. Gitaly can be:

A background service operating on a single instance Linux package installation (all of GitLab on one machine).
Separated onto its own instance and configured in a full cluster configuration, depending on scaling and availability
requirements.
```

Overview/Pipelines

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:
    - sleep 20
    - 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:

SaaS vs. On-Premise (Self Hosted)

```
Cons (On-Premise):

- Gitlab runner selber einrichten / eigene Gitlab - Runne
- Sicherheit der Daten
- Nicht Gezwungen, die neueste Version zu verwenden

Cons (SaaS)

- Ich muss die Änderungen für Updates zeitnah durchführen

Pros (SaaS):

- Automatische Upgrade auf die neueste Version
```

Hintergründe

Warum before_script ?

```
## Wir können einen hidden job definieren, der dann beim Job verwendet wird.
## Kommandos von before_script und script überschreiben sich nicht gegenseitig

.install_dependencies:
  before_script:
    - pip install --upgrade pip
    - pip install -r requirements.txt

my_test_job:
  extends: .install_dependencies
  script:
     - pytest
```

GIT_STRATEGY usw.

```
Es gibt tatsächlich ein GIT_DEPTH.
Zwei Probleme gibt es beim automatischen Clonen.
I. Es wird nur der aktuelle branch gezogen.
II. Die Tiefe steht standardmäßig auf 20.
Das könnten man hochsetzen


https://docs.gitlab.com/ee/ci/pipelines/settings.html#limit-the-number-of-changes-fetched-during-clone

Besser wäre wahrscheinlich
GIT_STRATEGY none
(bei den Variablen), dann greift nur das 2.)


Ausgabe, obwohl 2 Branches
 git branch -vr
  origin/main 5932a8c Update project1.gitlab-ci.yml


https://docs.gitlab.com/ee/ci/runners/configure_runners.html#git-fetch-extra-flags

variables:
  GIT_FETCH_EXTRA_FLAGS: --all
script:
  - ls -al cache/

---> RESULT: ated fresh repository.
fatal: fetch --all does not make sense with refspecs

So all does not work here, because we fetch a specific branch with refspec
```

gitlab ci/cd (Praxis I)

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
Build -> Pipeline - Editor

## Es erscheint der Editor mit einem Test-Template


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
  script:
    - env
    - pwd
```

Reference

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

Variablen definieren

Möglichkeit 1: TopLevel (Im Project)

```
## Settings -> CI/CD -> Variables
```

Beispiele:

```
## gitlab-ci.yml
variables:
  TEST_URL: http://schulung.t3isp.de # globalen Scope - in der gesamten Pipeline
                                     # Überschreibt NICHT -> ... Settings -> CI/CD -> Variables
  TEST_VERSION: "1.0" # global
  TEST_ENV: Prod # global
  TEST_VAR: "overwrite?"

stages:
  - build
  - test

show_env:
  stage: build

  variables:
    TEST_JOB: lowrunner # variable mit lokalem Scope - nur in Job
    TEST_URL: http://www.test.de # Auch das überschreibt NICHT -> ... Settings -> CI/CD -> Variables

  script:
  - echo $TEST_VAR
  - echo $TEST_MASKED
  - echo $TEST_URL
  - echo $TEST_URL > /tmp/urltest.txt
  - cat /tmp/urltest.txt
  - echo $TEST_CONTENT # tolle Sache
  - cat $TEST_CONTENT
  - echo $TEST_VERSION
  - echo $TEST_ENV
  - echo $TEST_JOB

test_env:
  stage: test

  script:
  - echo $TEST_URL
  - echo $TEST_CONTENT # tolle Sache
  - cat $TEST_CONTENT
  - echo $TEST_VERSION
  - echo $TEST_ENV
  - echo $TEST_JOB
```

Reihenfolge, welche Variablen welche überschreien (Ebenene)

- https://docs.gitlab.com/ee/ci/variables/#cicd-variable-precedence

Variablen überschreiben/leeren

gitlab-ci.yml

```
default:
  image: alpine


variables:
  VAR_GLOBAL: "meine globale var"

.base:
  script: test
  variables:
    VAR1: base var 1

job-test3:
  extends: .base
  variables: {} ## globale variable sollte danach eigentlich leer sein !!
  script:
   - echo $VAR1
   - echo "global->"$VAR_GLOBAL

job-test4:
  extends: .base
  variables: null
  script:
  - echo $VAR1
```

Rules

CI_PIPELINE_SOURCE

- https://gitlab.com/training.tn1/jochen-siegen1/-/jobs/4867098253

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 1: Creating an artifact

```
## .gitlab-ci.yml

stages:
  - build

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

Example 2: 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/*
```

Example 3: 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/*
```

Example 4: 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/
```

Example 5: 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 1: Passing artifacts between stages (enabled by default)

```
default:
  image: ubuntu:22.04

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

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 2: 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 3: 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:
    - ls -la
    - 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 (Praxis II)

Mehrzeile Kommandos in gitlab ci-cd ausführen

Step 1:

- Create new repo

Step 2: create good.sh next to README.md

- Create file good.sh in repo root-level

```
##!/bin/bash

echo "good things start now"
ls -la
date
```

Step 3: create gitlab-ci.yml with Pipeline Editor

```
stages:
  - build

workflow:
  rules:
  - if: $CI_PIPELINE_SOURCE == "web"

build-stage:
  stage: build
  variables:
    CMD: |
      echo hello-you;
      ls -la;
  script:
  - echo "execute script from git repo"
  - bash -s < good.sh
  - echo -n $CMD
  - echo "eval the command"
  - bash -c "$CMD"
  - |-
      bash -s << HEREDOC
        echo hello
        ls -la
      HEREDOC
  - |
      tr a-z A-Z << END_TEXT
        one two three
        four five six
      END_TEXT
  - |
      bash -s << HEREDOC
        echo hello
        ls -la
      HEREDOC
  - |-
      tr a-z A-Z << END_TEXT
        one two three
        four five six
      END_TEXT
  - >
      echo "First command line
      is split over two lines."

      echo "Second command line."
```

Run Pipeline (need to trigger manually)

Reference

Reference:

- https://docs.gitlab.com/ee/ci/yaml/script.html#split-long-commands
- https://stackoverflow.com/questions/3790454/how-do-i-break-a-string-in-yaml-over-multiple-lines/21699210#21699210

Kommandos auf Zielsystem mit ssh ausführen (auch multiline)

Preparation on Server

Step 1: Create public/private key

```
## on destinationn server
ssh-keygen
cd .ssh
ls -la
```

Step 2: Add id_rsa.pub /Public key to authorized_keys

```
cat id_rsa.pub >> authorized_keys
```

Step 3: Add id_rsa (private key) to GITLAB ci/cd -> Settings -> CI/CD as new Variable

```
cat id_rsa
## copy content and add as content to new variable SERVER_SSH_KEY
## DO not set variable as protected
```

create good.sh in root-folder of repo (git)

```
##!/bin/bash

echo "good things start now"
ls -la
date
```

Create gitlab-ci.yml

```
workflow:
  rules:
    - if: '$CI_PIPELINE_SOURCE == "web"'

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

deploy-job:

   stage: deploy
   image: ubuntu

   variables:
    CMD: |
      echo hello-you;
      ls -la;

   before_script:
    - apt -y update
    - apt install -y openssh-client
    - eval $(ssh-agent -s)
    - echo "$SERVER_SSH_KEY" | tr -d '\r' | ssh-add -
    - ls -la
    - mkdir -p ~/.ssh
    - chmod 700 ~/.ssh
    - ssh-keyscan $SERVER_IP >> ~/.ssh/known_hosts
    - chmod 644 ~/.ssh/known_hosts
##- echo $SERVER_SSH_KEY

   script:
     - echo 'V1 - commands in line'
     ############### ! Important
     # For ssh to exit on error, start your commands with first command set -e
     # - This will exit the executed on error with return - code > 0
     # - AND will throw an error from ssh and in pipeline
     ###############
     - ssh root@$SERVER_IP -C "set -e; ls -la; cd $SERVER_WEBDIR; ls -la;"
     - echo 'V2 - content of Variable $CMD'
     - ssh root@$SERVER_IP -C $CMD
     - echo 'V3 - script locally - executed remotely'
     - ssh root@$SERVER_IP < good.sh
     - echo 'V4 - script in heredoc'
     - |
      ssh root@$SERVER_IP bash -s << HEREDOC
        echo "hello V4"
        ls -la
      HEREDOC
     - echo 'V5 - copy script and execute'
     - scp good.sh root@$SERVER_IP:/usr/local/bin/better.sh
     - ssh root@$SERVER_IP -C "chmod u+x /usr/local/bin/better.sh; better.sh"
```

gitlab-ci/cd - Workflows

Workflows + only start by starting pipeline

What for ?

- Configure how pipelines behaves

Only start pipeline by starting it with pipeline run (manually)

```
## only: web geht hier nicht, aber das steht eigentlich für:
## '$CI_PIPELINE_SOURCE == "web"'
stages:
  - build

workflow:
  rules:
    - if: '$CI_PIPELINE_SOURCE == "web"'

build-stage:
  stage: build
  script:
    - echo "hello build"
```

More information about possible values for $CI_PIPELINE_SOURCE

- https://docs.gitlab.com/ee/ci/jobs/job_control.html#common-if-clauses-for-rules

Templates for branch and merge request workflow

```
merge_request_event
https://docs.gitlab.com/ee/ci/pipelines/merge_request_pipelines.html

merge_request_pipeline
Alternatively, you can configure your pipeline to run every time you make changes to the source branch for a merge request. This
type of pipeline is called a merge request pipeline.

https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Workflows/MergeRequest-Pipelines.gitlab-ci.yml
(not default)

branch_pipeline
You can configure your pipeline to run every time you commit changes to a branch. This type of pipeline is called a branch
pipeline.
(default)

https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Workflows/Branch-Pipelines.gitlab-ci.yml
```

gitlab-ci/cd - Variables

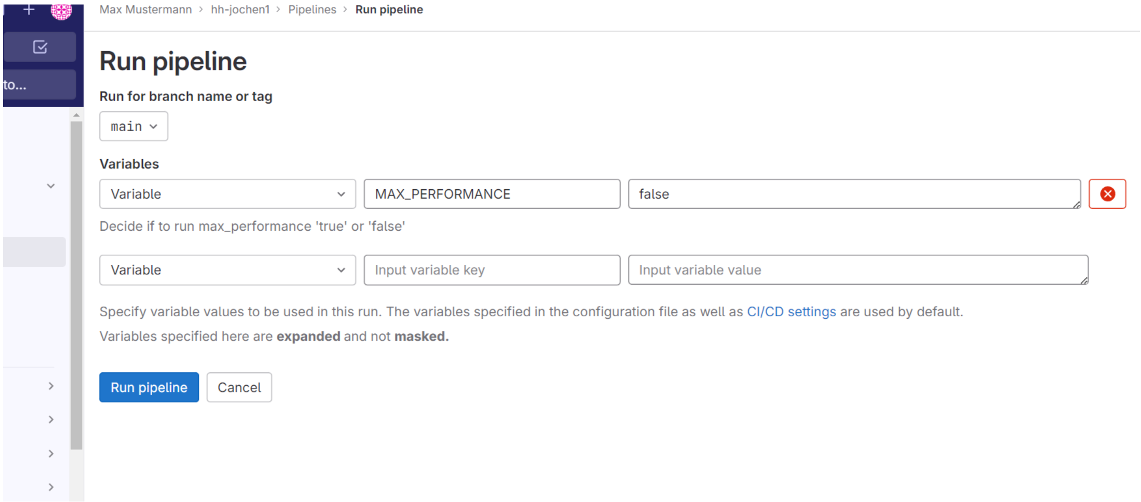
Variablen in Pipelines Web-Dialog anzeigen

So wird es nicht angezeigt:

```
variables:
  MAX_PERFORMANCE: 'true'
```

So wird es angezeigt:

```

variables:
   MAX_PERFORMANCE:
     description: Decide if to run max_performance 'true' or 'false'
     value: 'false'
```


Max Mustermann › hh-jochen1 › Pipelines › Run pipeline

Run pipeline

Run for branch name or tag

main

Variables

Variable | MAX_PERFORMANCE | false

Decide if to run max_performance 'true' or 'false'

Variable | Input variable key | Input variable value

Specify variable values to be used in this run. The variables specified in the configuration file as well as CI/CD settings are used by default.

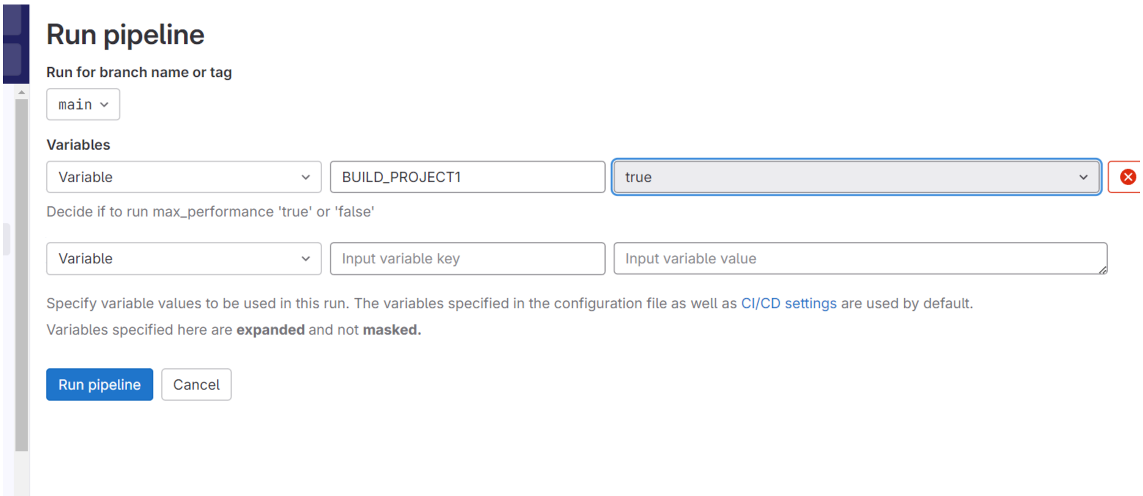
Variables specified here are **expanded** and not **masked.**

Mit Optionen

So hinterlegen

```
variables:
   MAX_PERFORMANCE:
     description: Decide if to run max_performance 'true' or 'false'
     value: 'false'
     options:
       - 'false'
       - 'true'
```

So sieht es aus:



Run pipeline

Run for branch name or tag

main

Variables

Variable | BUILD_PROJECT1 | true

Decide if to run max_performance 'true' or 'false'

Variable | Input variable key | Input variable value

Specify variable values to be used in this run. The variables specified in the configuration file as well as CI/CD settings are used by default.

Variables specified here are **expanded** and not **masked.**

gitlab - ci/cd - Pipelines strukturieren / Templates

Includes mit untertemplates

Prerequisites

```
1x main .gitlab-ci.yml

1x project1/project1.gitlab-ci.yml
1x project2/project2.gitlab-ci.yml
```

Step 1a: gitlab-ci.yml (simple)

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

include:
   - project1/project1.gitlab-ci.yml
   - project2/project2.gitlab-ci.yml
```

Step 1b: gitlab-ci.yml (start with pipeline start and variable setting

```
workflow:
  rules:
    - if: '$CI_PIPELINE_SOURCE == "web"'

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

include:
   - local: project1/project1.gitlab-ci.yml
     rules:
       - if: $BUILD_PROJECT1 == "true"

   - local: project2/project2.gitlab-ci.yml
     rules:
       - if: $BUILD_PROJECT2 == "true"

dummy-build:
   stage: build
   script:
     - echo "dummy build"
   rules:
     - if: $BUILD_PROJECT1 != "true" && $BUILD_PROJECT2 != "true"
```

Step 2: project1/project1.gitlab-ci.yml

```
stages:
  - build
```

```
project1.build-job:
  stage: build
  script:
    - echo "in project1 .. building"
```

Step 3: project2/project2.gitlab-ci.yml

```
stages:
  - build

project2.build-job:
  stage: build
  script:
    - echo "in project2 .. building"
```

Parent/Child Pipeline

Variante 1: gitlab-ci.yml (no subfolders)

```
project1:
  trigger:
    include: project1/project1.gitlab-ci.yml
    strategy: depend
  rules:
    - changes: [project1/*]
project2:
  trigger:
    include: project2/project2.gitlab-ci.yml
    strategy: depend
  rules:
    - changes: [project2/*]
```

Variante 2: gitlab-ci.yml (with subfolders)

```
project1:
  trigger:
    include: project1/project1.gitlab-ci.yml
    strategy: depend
  rules:
    - changes: [project1/**/*]
project2:
  trigger:
    include: project2/project2.gitlab-ci.yml
    strategy: depend
  rules:
    - changes: [project2/**/*]
```

Variante 3: gitlab-ci.yml (with subfolders ....)

- Not able to be started on run pipeline (manually)
- But, when it is triggered on changes

```
workflow:
  rules:
    - if: '$CI_PIPELINE_SOURCE == "web"'
      when: never
    - when: always

project1:
  trigger:
    include: project1/project1.gitlab-ci.yml
    strategy: depend
  rules:
    - changes: [project1/**/*]
project2:
  trigger:
    include: project2/project2.gitlab-ci.yml
    strategy: depend
  rules:
    - changes: [project2/**/*]
```

project1/project1.gitlab-ci.yml

```
stages:
  - build
```

```
project1.build-job:
  stage: build
  script:
    - echo "in project1 .. building"
    - echo $CI_PIPELINE_SOURCE
```

project2/project2.gitlab-ci.yml

```
stages:
  - build

project2.build-job:
  stage: build
  script:
    - echo "in project2 .. building"
    - echo $CI_PIPELINE_SOURCE
```

Alternative mit anderen stages in child

```
stages:
  - project1-build
  - project1-test
  - project1-deploy


project1.build-job:
  stage: project1-build
  script:
    - echo "in project1 .. building"
    - echo $CI_PIPELINE_SOURCE

project1.test-job:
  stage: project1-test
  script:
    - echo "in project1 .. test"
    - echo $CI_PIPELINE_SOURCE

project1.deploy-job:
  stage: project1-deploy
  script:
    - echo "in project1 .. deploy"
    - echo $CI_PIPELINE_SOURCE
```

Refs:

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

Multiproject Pipeline / Downstream

Practical Example (Variant 1)

Trigger - job in .gitlab-ci.yml

```
trigger_job:
  trigger:
    project: training.tn1/jochentest-multi1 # project/repo sonst geht es nicht (muss komplett angegeben werden)
    strategy: depend
```

New repo -> training.tn1/jochentest-multi1

```
.gitlab-ci.yml
```

```
## This is how my other project looks like
workflow:
    rules:
      - if: '$CI_PIPELINE_SOURCE == "web"'
      - if: '$CI_PIPELINE_SOURCE == "pipeline"' # ein projekt gestart innerhalb multiproject - pipeline

default:
  image: alpine

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

build-job:       # This job runs in the build stage, which runs first.
  stage: build
  script:
    - echo "Started"
    - echo "Show us the pipeline source $CI_PIPELINE_SOURCE"
```

Practical Example (Variant 2): Deploy after all Build triggers are done

```
stages:
  - build
  - deploy

project1:
  stage: build
  trigger:
    include: project1/project1.gitlab-ci.yml
    strategy: depend
 # rules:
 #   - changes: [project1/**/*]
project2:
  stage: build
  trigger:
    include: project2/project2.gitlab-ci.yml
    strategy: depend
  rules:
    - changes: [project2/**/*]

trigger_job:
  stage: build
  trigger:
    project: training.tn11/jochentest-multi2 # project/repo sonst geht es nicht (muss komplett angegeben werden)
    strategy: depend

deploy_job:
  stage: deploy
  image: alpine
  script:
    - echo "i am good to go"
    - sleep 30
```

Reference

- https://docs.gitlab.com/ee/ci/pipelines/downstream_pipelines.html?tab=Multi-project+pipeline

Vorgefertigte Templates verwenden

Step 1: Browser Template in Pipeline Editor (Top-Bottom) to find the one you want

Step 2: Include template in your gitlab-ci.yml - config

```

workflow:
  rules:
    - if: '$CI_PIPELINE_SOURCE == "web"'

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

include:
  template: Jobs/Test.gitlab-ci.yml

run-deploy:
  stage: deploy
  script:
    - echo "deploy started"

```

Arbeiten mit extend und anchor - Dinge wiederverwenden

Hinweis:

- Dinge, die wiederverwendet werden sollen, müssen vorher definiert sein, in der Datei
- d.h. .base vor myjob
- .default_scripts bzw &default_scripts vor Verwendung als *default_scripts

gitlab-ci.yml

```
.base:
  variables:
    TEST_CASE: "true"
    VERSION: "1.0"
```

```
.default_scripts: &default_scripts
  - echo "from _default_scripts"
  - echo "next default step"

myjob:
  variables:
    TEST_CASE: 'bad'
  extends: .base
  script:
    - *default_scripts
    - echo "in MYJOB"
    - ls -la
    - echo $TEST_CASE
    - echo $VERSION
```

gitlab - wann laufen jobs ?

Job nur händisch über Pipelines starten

```
## gitlab-ci.yml
stages:          # List of stages for jobs, and their order of execution
  - build
  - test

build-job:       # This job runs in the build stage, which runs first.
  stage: build
  only:
    - web
  image: maven
  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.
  only:
    - web
  script:
    - echo "Running unit tests... This will take about 60 seconds."
    - sleep 60
    - echo "Code coverage is 90%"
```

Auch weiterlaufen, wenn Job fehlschlägt

Walkthrough

```
stages:
  - build
  - deploy

default:
  image: alpine

build_job1:
  stage: build
  allow_failure: true
  script:
    - xls

build_job2:
  stage: build
  script:
    - ls -la
    - sleep 120

build_job3:
   stage: build
   script:
     - echo "ich bin job3"

deploy_job:
  stage: deploy
  script:
    - echo "i am good to go"
    - sleep 30
```

gitlab ci/cd docker

Docker image automatisiert bauen - gitlab registry

good.sh

```
##!/bin/bash
date
```

Dockerfile - RootLevel

```
FROM ubuntu:22.04
##
RUN  apt-get update && \
     apt-get install -y openssh-client  && \
     rm -rf /var/lib/apt/lists/*
COPY good.sh /usr/local/bin/better.sh
```

Variante 1: .gitlab-ci.yml (Version with docker-dind (docker-in-docker)

```
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 images
    - docker push $CI_REGISTRY_IMAGE
    - echo "BUILD for $CI_REGISTRY_IMAGE done"
```

Variante 2: kaniko (rootless from google)

```
build:
  stage: build
  image:
    name: gcr.io/kaniko-project/executor:v1.9.0-debug
    entrypoint: [""]
  script:
    - /kaniko/executor
      --context "${CI_PROJECT_DIR}"
      --dockerfile "${CI_PROJECT_DIR}/Dockerfile"
      --destination "${CI_REGISTRY_IMAGE}:${CI_COMMIT_TAG}"
  rules:
    - if: $CI_COMMIT_TAG
```

gitlab ci/cd docker compose

Docker compose local testen

Evolutions-Phase 1: Testen eines docker compose file lokal auf unserem Zielsysem

```
## public/private key muss eingerichtet sein
ssh root@<ziel-ip>
```

Docker installieren

```
sudo apt-get update
sudo apt-get -y install \
    ca-certificates \
    curl \
    gnupg \
    lsb-release

sudo mkdir -p /etc/apt/keyrings
```

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg

echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

sudo apt-get update
sudo apt-get install -y docker-ce docker-ce-cli containerd.io docker-compose-plugin
```

```
### Läuft der Dienst (dockerd)
systemctl status docker
```

```
mkdir cms
cd cms
nano docker-compose.yaml
```

```
services:
  db:
    # We use a mariadb image which supports both amd64 & arm64 architecture
    image: mariadb:10.6.4-focal
    # If you really want to use MySQL, uncomment the following line
    #image: mysql:8.0.27
    command: '--default-authentication-plugin=mysql_native_password'
    volumes:
      - db_data:/var/lib/mysql
    restart: always
    environment:
      - MYSQL_ROOT_PASSWORD=somewordpress
      - MYSQL_DATABASE=wordpress
      - MYSQL_USER=wordpress
      - MYSQL_PASSWORD=wordpress
    expose:
      - 3306
      - 33060
  wordpress:
    image: wordpress:latest
    volumes:
      - wp_data:/var/www/html
    ports:
      - 80:80
    restart: always
    environment:
      - WORDPRESS_DB_HOST=db
      - WORDPRESS_DB_USER=wordpress
      - WORDPRESS_DB_PASSWORD=wordpress
      - WORDPRESS_DB_NAME=wordpress
volumes:
  db_data:
  wp_data:
```

```
docker compose up -d
```

Docker compose über ssh

Evolutions-Phase 2: Anwenden eines Docker Compose files über ssh

Vorbereitung: Im Repo cms/docker-compose.yaml einfügen

```
services:
  db:
    # We use a mariadb image which supports both amd64 & arm64 architecture
    image: mariadb:10.6.4-focal
    command: '--default-authentication-plugin=mysql_native_password'
    volumes:
      - db_data:/var/lib/mysql
    restart: always
    environment:
      - MYSQL_ROOT_PASSWORD=somewordpress
      - MYSQL_DATABASE=wordpress
      - MYSQL_USER=wordpress
      - MYSQL_PASSWORD=wordpress
    expose:
      - 3306
      - 33060
  wordpress:
    image: wordpress:latest
    volumes:
      - wp_data:/var/www/html
    ports:
      - 80:80
    restart: always
    environment:
      - WORDPRESS_DB_HOST=db
      - WORDPRESS_DB_USER=wordpress
      - WORDPRESS_DB_PASSWORD=wordpress
      - WORDPRESS_DB_NAME=wordpress
volumes:
  db_data:
  wp_data:
```

Schritt 1: gitlab-ci.yaml

```
workflow:
  rules:
    - if: '$CI_PIPELINE_SOURCE == "web"'

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

deploy-job:
   stage: deploy
   image: ubuntu

   before_script:
    - apt-get -y update
    - apt-get install -y openssh-client ca-certificates curl gnupg lsb-release
    - mkdir -p /etc/apt/keyrings
    # We want the newest version from docker
    # version from ubuntu repo does not work (docker compose) - version too old
    - curl -fsSL https://download.docker.com/linux/ubuntu/gpg | gpg --dearmor -o /etc/apt/keyrings/docker.gpg
    - echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg]
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | tee /etc/apt/sources.list.d/docker.list > /dev/null
    - apt-get update -y
    - apt-get install -y docker-ce docker-ce-cli containerd.io docker-compose-plugin
    - eval $(ssh-agent -s)
    - echo "$SERVER_SSH_KEY" | tr -d '\r' | ssh-add -
    - ls -la
    - mkdir -p ~/.ssh
    - chmod 700 ~/.ssh
    - ssh-keyscan $SERVER_IP >> ~/.ssh/known_hosts
    - chmod 644 ~/.ssh/known_hosts
    # - echo $SERVER_SSH_KEY
    # eventually not needed
    #- echo $SERVER_SSH_KEY >  ~/.ssh/id_rsa
    #- chmod 600 ~/.ssh/id_rsa

   script:
     - echo 'Deploying wordpress'
     - cd cms
     - export DOCKER_HOST="ssh://root@$TOMCAT_SERVER_IP"
     - docker info
     - docker container ls
     - docker compose up -d
```

Docker compose classic über scp

Evolutions-Phase 3:

Vorbereitung:

- SSH_SERVER_KEY (private key), SERVER_IP als Variablen in Settings -> CI/CD angelegt werden.

Schritt 1: cms/docker-compose.yaml in gitlab anlegen

```
services:
  db:
    # We use a mariadb image which supports both amd64 & arm64 architecture
    image: mariadb:10.6.4-focal
    command: '--default-authentication-plugin=mysql_native_password'
    volumes:
      - db_data:/var/lib/mysql
    restart: always
    environment:
      - MYSQL_ROOT_PASSWORD=somewordpress
      - MYSQL_DATABASE=wordpress
      - MYSQL_USER=wordpress
      - MYSQL_PASSWORD=wordpress
    expose:
      - 3306
      - 33060
  wordpress:
    image: wordpress:latest
    volumes:
      - wp_data:/var/www/html
    ports:
      - 80:80
    restart: always
    environment:
      - WORDPRESS_DB_HOST=db
      - WORDPRESS_DB_USER=wordpress
      - WORDPRESS_DB_PASSWORD=wordpress
      - WORDPRESS_DB_NAME=wordpress
volumes:
  db_data:
  wp_data:
```

Schritt 2: gitlab-ci.yaml

```
workflow:
  rules:
    - if: '$CI_PIPELINE_SOURCE == "web"'

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

deploy-job:
   stage: deploy
   image: ubuntu

   before_script:
    - apt-get -y update
    - apt-get install -y openssh-client
    - eval $(ssh-agent -s)
    - echo "$SERVER_SSH_KEY" | tr -d '\r' | ssh-add -
    - mkdir -p ~/.ssh
    - chmod 700 ~/.ssh
    - ssh-keyscan $SERVER_IP >> ~/.ssh/known_hosts
    - chmod 644 ~/.ssh/known_hosts
    #- echo $SERVER_SSH_KEY
    # eventually not neededrsa

   script:
     - echo 'Deploying wordpress'

     - |
       ssh root@$SERVER_IP bash -s << HEREDOC
        cd
        mkdir -p cms
       HEREDOC
     - scp cms/docker-compose.yaml root@$SERVER_IP:~/cms/
     - ssh root@$SERVER_IP -C "cd; cd cms; docker compose up -d"
```

Schritt 3: Pipeline manuell über pipeline menü starten

Performance / Caching

Wann ist eine Pipeline langsam ?

Repo ist sehr gross

- was kann ich tun: depth niedriger setzen, statt 20
- Brauche ich das Repo in dem Job ?-> ansonsten GIT_STRATEGY: none

Image ist zu gross

- Image verschlanken, wenn es keine Standard-Image ist, im Image-Bauprozes

Caching

- Wenn caching verwendet und wenn ja richtig ?

Scheduling in der Pipelines

- sehr lange Laufzeiten, weil einzelne Stages sehr lange brauchen (standardverhalten: jeder Stage muss komplett fertig sein, damit der nächste Stage starten kann)
- Lösung: Acyclice Pipelines: z.B. build_a -> test_a -> deploy_a kann parallel laufen zu build_b -> test_b -> deploy_b

Job abspecken / zu langsam

- Muss ich wirklich alles in diesem Job so machen oder kann ich Sachen weglassen (weil redundant, nicht notwendig)

Brauche ich artifakte in jedem Job, ansonsten deaktivieren

```
## 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:
    - ls -la
    - echo "no control.txt here"
    - ls -la

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

Caching->cache in gitlab

Key facts

- Caches are reused in pipelines
- Use to cache depenedencies (libraries a.s.o)
- Use artifacts, if data is created by a job
- the cache is stored in the same place where GitLab Runner is installed. If the distributed cache is configured, S3 works as storage.

Example

Prepare

- Import this repo: https://gitlab.com/gitlab-examples/nodejs.git

Iteration 1: Use a cache

```
image: node:latest # (1)

## This folder is cached between builds
cache:
  paths:
    - node_modules/ # (2)

test_async:
  script:
    - npm install # (3)
    - node ./specs/start.js ./specs/async.spec.js

test_db:
  script:
    - npm install # (4)
    - node ./specs/start.js ./specs/db-postgres.spec.js
```

Iteration 2: Modify .gitlab-ci.yaml

```
image: node:16.3.0 # (1)

stages:
  - setup
  - test

variables:
  npm_config_cache: "$CI_PROJECT_DIR/.npm"

## Define a hidden job to be used with extends
## Better than default to avoid activating cache for all jobs
.dependencies_cache:
  cache:
    key:
      files:
        - package-lock.json
    paths:
      - .npm
    policy: pull

setup:
  stage: setup
  script:
    - npm install
  extends: .dependencies_cache
  cache:
    policy: pull-push
  artifacts:
    expire_in: 1h
    paths:
      - node_modules

test_async:
  stage: test
  script:
    - node ./specs/start.js ./specs/async.spec.js

test_db:
  stage: test
  script:
    - node ./specs/start.js ./specs/db-postgres.spec.js
```

Iteration 3: Modify .gitlab-ci.yaml

- package-lock.json bauen (npm install) und als artifact zur Verfügung stellen

```
image: node:16.3.0 # (1)

stages:
  - setup
  - test

variables:
  npm_config_cache: "$CI_PROJECT_DIR/.npm"

## Define a hidden job to be used with extends
## Better than default to avoid activating cache for all jobs
.dependencies_cache:
  cache:
    key:
      files:
        - package-lock.json
    paths:
      - .npm
    policy: pull

setup:
  stage: setup
  script:
    - npm install
  extends: .dependencies_cache
  cache:
    policy: pull-push
  artifacts:
    expire_in: 1h
    paths:
      - node_modules
      - package-lock.json

test_async:
  stage: test
  script:
    - node ./specs/start.js ./specs/async.spec.js

test_db:
  stage: test
  script:
    - node ./specs/start.js ./specs/db-postgres.spec.js
```

```
## Artifact runterladen, Inhalt aus package-lock.json rauskopieren und Datei in Repo erstellen
## package-lock.json und Inhalt einfügen
```

```
## Änderung von .gitlab-ci.yaml -> artifaction package-json.lock rausgenommen und ein npm ci gemacht in setup
image: node:16.3.0 # (1)

stages:
  - setup
  - test

variables:
  npm_config_cache: "$CI_PROJECT_DIR/.npm"

## Define a hidden job to be used with extends
## Better than default to avoid activating cache for all jobs
.dependencies_cache:
  cache:
    key:
      files:
        - package-lock.json
    paths:
      - .npm
    policy: pull

setup:
  stage: setup
  script:
    - npm ci
  extends: .dependencies_cache
  cache:
    policy: pull-push
  artifacts:
    expire_in: 1h
    paths:
      - node_modules

test_async:
  stage: test
  script:
    - node ./specs/start.js ./specs/async.spec.js

test_db:
  stage: test
  script:
    - node ./specs/start.js ./specs/db-postgres.spec.js
```

Reference:

- https://dev.to/drakulavich/gitlab-ci-cache-and-artifacts-explained-by-example-2opi

Matrix = Loops

Matrix=Loops, Jobs mehrmals ausführen

Example 1

```
stages:
  - build
  - test

build:
  stage: build
  script:
    - echo "Building $DISTRIBUTION on $ARCH"
  parallel:
    matrix:
      - DISTRIBUTION: [rhel8, ubuntu20]
        ARCH: [x64,x86]

test:
  stage: test
  script:
    - echo "Testing $DISTRIBUTION on $ARCH"
  parallel:
    matrix:
      - DISTRIBUTION: [rhel8, ubuntu20]
        ARCH: [x64,x86]
```

Example 1a

```
stages:
  - build
  - test

.parallel-matrix:
  parallel:
    matrix:
      - DISTRIBUTION: [rhel8, ubuntu20]
        ARCH: [x64,x86]

build:
  stage: build
  extends: .parallel-matrix
  script:
    - echo "Building $DISTRIBUTION on $ARCH"


test:
  stage: test
  extends: .parallel-matrix
  script:
    - echo "Testing $DISTRIBUTION on $ARCH"
```

Example 2

```
stages:
- matrix

.parallel-matrix:
  parallel:
    matrix:
      - CLOUD:
        - aws
        - azure
        - gcp
        ARCH:
        - kubernetes
        - service-mesh

Matrix:
  stage: matrix
  image: alpine:latest
  extends: .parallel-matrix
  script:
  - echo "Hello from $ARCH from $CLOUD"
```

Reference:

- https://yashwanth-l.medium.com/gitlab-ci-parallel-with-matrix-7bd3acca8f70

Monitoring gitlab

Monitoring gitlab good or bad ?

Error-Tracking

- With Sentry, needs to be implement with sdk in application

Alerts

- Can be triggered by Prometheus or http-Endpoint

Incidents

- Incidents manuell anlegen

Service Desk

- Emails können hier hingeschickt werden und tauchen als Incidents auf.

Security

Container Scanning

Walkthrough (Variant 1)

```
include:
  - template: Security/Container-Scanning.gitlab-ci.yml
```

```
container_scanning:
  variables:
     CS_IMAGE: registry.gitlab.com/training.tn11/jochentest1:latest
```

Walkthrough (Variant 2) - including building image

```
include:
  - template: Jobs/Build.gitlab-ci.yml
  - template: Security/Container-Scanning.gitlab-ci.yml

container_scanning:
  variables:
    CS_DEFAULT_BRANCH_IMAGE: $CI_REGISTRY_IMAGE/$CI_DEFAULT_BRANCH:$CI_COMMIT_SHA
```

Ref:

- https://docs.gitlab.com/ee/user/application_security/container_scanning/

Environments

Environments

Variant 1: Add Environment in Operate -> Environments

```
1. In Operate -> Deployment  neues Environment Testing anlegen
keine URL angeben - nur name

2. In Project -> Settings->CI/CD-> Variables

Variable 1:
CODE_BRANCH   testing   (Environment: Testing)
CODE_BRANCH   main      (Environment: Production)

3. Kleines Code-Schnipsel
(normalerweise wird es in deploy verwenden
und hat dort noch Zusatzintegrationen)
```

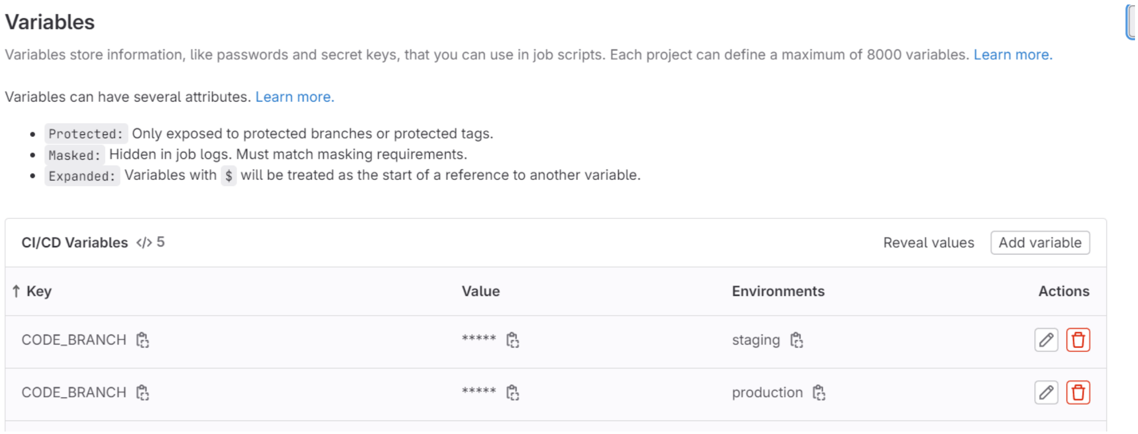
```
stages:
  - build


build-prod:
    stage: build
    environment:
        name: production
        action: prepare
    script:
       echo "That is the code branch "$CODE_BRANCH

build-testing:
    stage: build
    environment:
        name: testing
        action: prepare
    script:
       echo "That is the code branch "$CODE_BRANCH
```

Variant 2: Add environment statically through gitlab-ci.yaml

```
## add enviroment to .gitlab-ci.yaml
stages:
  - deploy

deploy_staging:
  stage: deploy
  script:
    - echo "deploy to staging"
    - echo "$CODE_BRANCH"
  environment:
     name: staging
  when: manual

deploy_production:
  stage: deploy
  script:
    - echo "deploy to production"
    - echo "$CODE_BRANCH"
  environment:
    name: production
  when: manual
```

```
## Start pipeline - if not already done after saving
```

```
## Now enter 2 variables in Settings -> CI/CD Variables
For CODE_BRANCH / production
For CODE_BRANCH / staging
```



Variables

Variables store information, like passwords and secret keys, that you can use in job scripts. Each project can define a maximum of 8000 variables. Learn more.

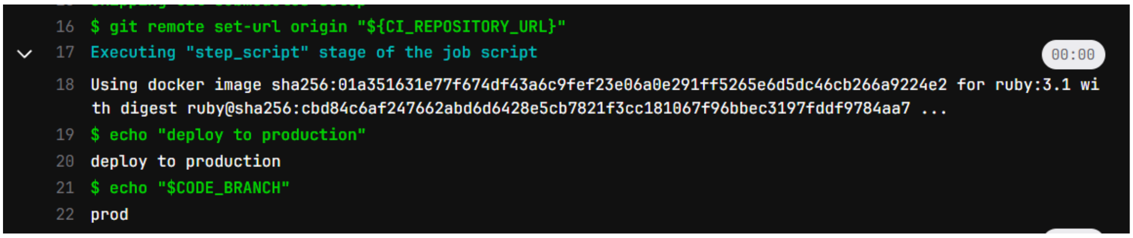
Variables can have several attributes. Learn more.

- `Protected:` Only exposed to protected branches or protected tags.
- `Masked:` Hidden in job logs. Must match masking requirements.
- `Expanded:` Variables with `$` will be treated as the start of a reference to another variable.

CI/CD Variables </> 5

↑ Key	Value	Environments	Actions
CODE_BRANCH	*****	staging	
CODE_BRANCH	*****	production	

```
## Start jobs manually in pipeline
## and watch log for CODE_BRANCH
```



```
$ git remote set-url origin "${CI_REPOSITORY_URL}"
Executing "step_script" stage of the job script
Using docker image sha256:01a351631e77f674df43a6c9fef23e06a0e291ff5265e6d5dc46cb266a9224e2 for ruby:3.1 with digest ruby@sha256:cbd84c6af247662abd6d6428e5cb7821f3cc181067f96bbec3197fddf9784aa7 ...
$ echo "deploy to production"
deploy to production
$ echo "$CODE_BRANCH"
prod
```

Documentation

gitlab ci/cd predefined variables

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

.gitlab-ci.yml Reference

- https://docs.gitlab.com/ee/ci/yaml/

Referenz: global -> workflow

- https://docs.gitlab.com/ee/ci/yaml/#workflow

Referenz: global -> default

- https://docs.gitlab.com/ee/ci/yaml/#default

Documentation - Includes

includes

- https://docs.gitlab.com/ee/ci/yaml/includes.html

includes -> rules

- https://docs.gitlab.com/ee/ci/yaml/includes.html#use-rules-with-include

includes -> rules -> variables

- https://docs.gitlab.com/ee/ci/yaml/#rulesvariables

includes -> templates -> override-configuration

- https://docs.gitlab.com/ee/ci/yaml/includes.html#override-included-configuration-values

includes -> defaults

- https://docs.gitlab.com/ee/ci/yaml/includes.html#use-default-configuration-from-an-included-configuration-file

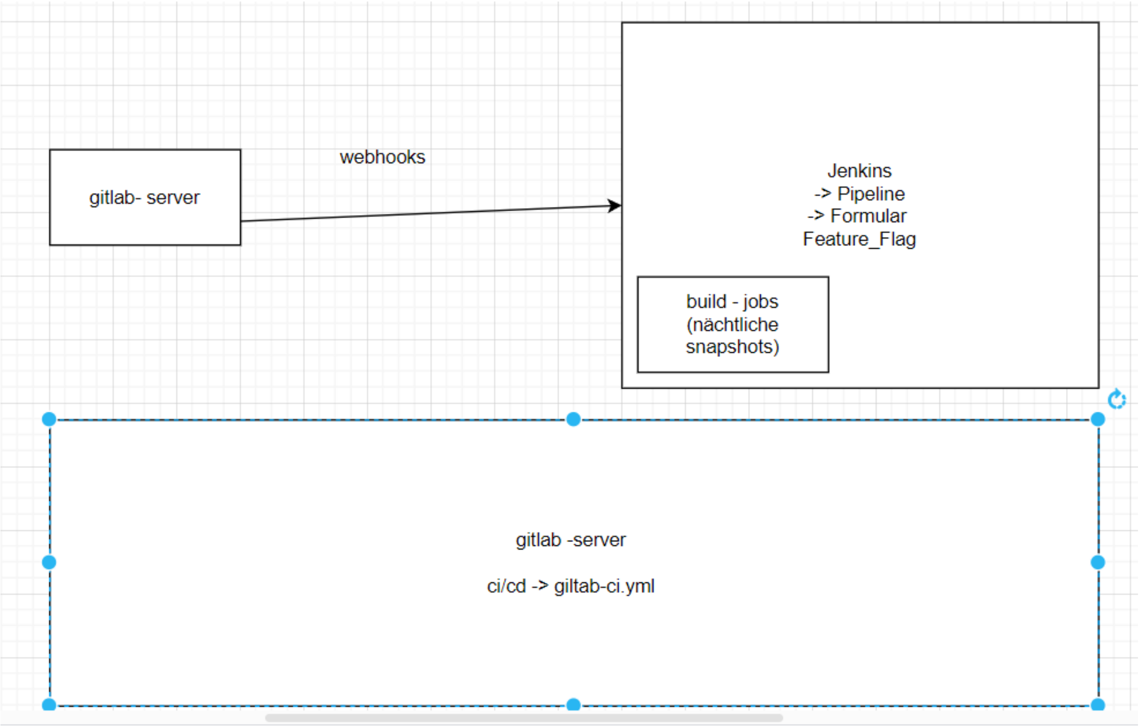
Documentation - Instances Limits

applicaton limits

- https://docs.gitlab.com/ee/administration/instance_limits.html

gitlab ci/cd (Überblick)

Jenkins mit Gitlab vs. gitlab ci/cd



gitlab - setzen von Variablen

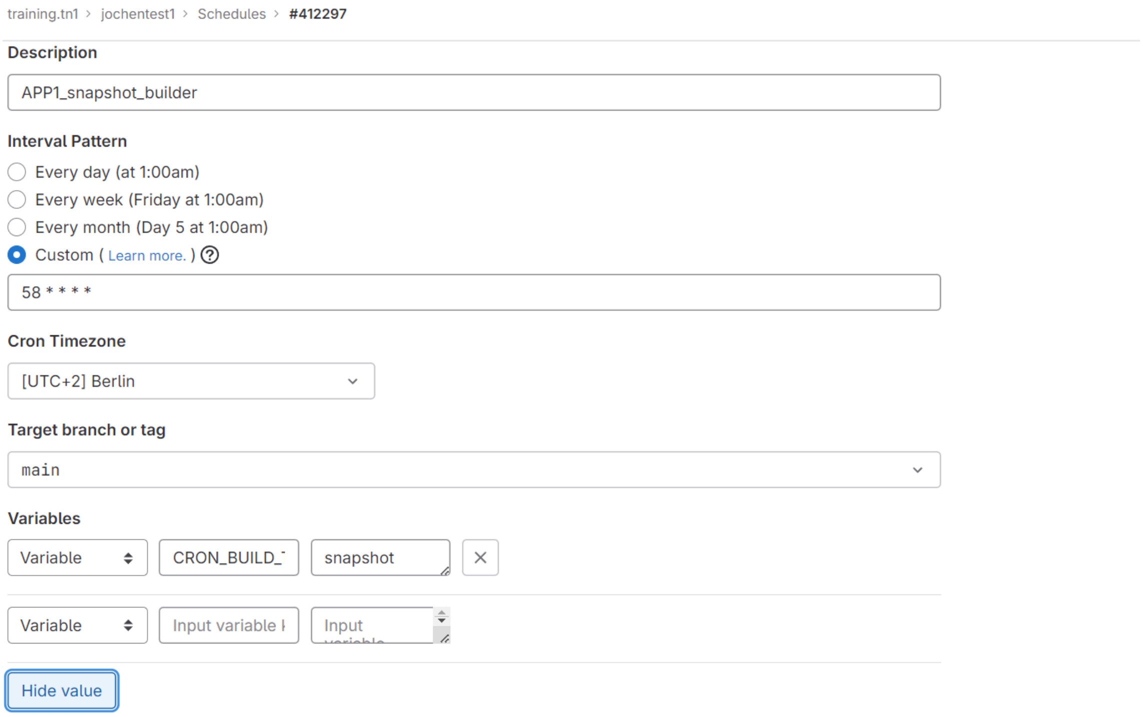
Variablen für angepasste Builds verwenden und scheduled pipeline

in: gitlab-ci.yml

```
workflow:
  rules:
    - if: $CI_PIPELINE_SOURCE == "web"

image: alpine
## correct: eigenes image oder maven

stages:
  - build

job-from-scheduler:
  stage: build

  rules:
    - if: $CRON_BUILD_TYPE == "snapshot"
      variables:                              # Override DEPLOY_VARIABLE defined
        MVN_BUILD_GOAL: "snapshot"  # at the job level.
    - if: $CRON_BUILD_TYPE == "release"
      variables:

        MVN_BUILD_GOAL: "release"                 # Define a new variable.
  script:
    - echo "Run script with $DEPLOY_VARIABLE as an argument"
    - echo "Run another script if $IS_A_FEATURE exists"
    - echo "mvn $MVN_BUILD_GOAL"
```

in Projects -> Build -> Pipeline Schedules

- New Schedule -> APP1_snapshot_builder
  - Zeit festlegen
  - Wichtig: Variable setzen:
    - CRON_BUILD_TYPE : snapshot
- New Schedule -> APP1_release_builder
  - Zeit festlegen
  - Wichtig: Variable setzen:
    - CRON_BUILD_TYPE : release



training.tn1 › jochentest1 › Schedules › **#412297**

Description

APP1_snapshot_builder

Interval Pattern

○ Every day (at 1:00am)
○ Every week (Friday at 1:00am)
○ Every month (Day 5 at 1:00am)
● Custom (Learn more.) ?

58 * * * *

Cron Timezone

[UTC+2] Berlin

Target branch or tag

main

Variables

Variable	CRON_BUILD_	snapshot
Variable	Input variable	Input variable

Exercises

build with maven and using artifacts

- https://github.com/jmetzger/training-gitlab-ci-cd/blob/main/gitlab/11-build-war-with-maven.md

gitlab ci/cd - docker

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"
```

Selbst gebauten Container manuell ausführen

```
1. docker auf tomcat server installieren (schnellste Weg) - Ubuntu

sudo su -
snap install docker


2. registry - image runterziehen testen (gitlab)



## image wird runtergezogen
## 1. Es wird eine interaktive Shell gestartet -it
## 2. und es wird das Programm bash (Shell) gestartet
docker run -it registry.gitlab.com/training.tn11/jochentest1 bash

-> Mhm, geht nicht, keine Berechtigung

3. Einloggen in Docker (Versuch 1)

docker login registry.gitlab.com/training.tn11 -utraining.tn11 -p<Dein Passwort>

-> Mhm, geht nicht, brauchen wir vielleicht ein Token

4. Access Token anlegen

-> unter Profil -> bearbeiten -> Linkes Menü -> Access Token
nur registry lesen


5. Einloggen mit Token an der registry (Token dient als Password)

docker login registry.gitlab.com/training.tn11 -utraining.tn11 -p<Dein Access Token>

6. Image starten

docker run -it registry.gitlab.com/training.tn11/jochentest1 bash

7. ist ssh drin ?
## hinter dem Prompt eingeben
```

```
ssh
cat /etc/os-release
```

Neues Image in gitlab ci/cd aus gitlab registry verwenden

gitlab-ci.yml

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


build-new:
  stage: build
  image: registry.gitlab.com/training.tn11/jochentest1
  script:
    - which ssh
    - echo $PATH
```

Ausführen und glücklich sein !

Einloggen mit Docker Credentials

```
echo -n "username:access-token" | base64

DOCKER_AUTH_CONFIG

"auths": {
  "registry.gitlab.com": {
    "auth": "LSBuIHRyYWluaW5nMTE6Z2xwYXQtTlpILXNTNXhtNEZBeFdTekpBZnkK"
  }
}
```

```
Eintragen von DOCKER_AUTH_CONFIG -> in Settings -> CI/CD -> Variables
```

Refs:

- https://mherman.org/blog/gitlab-ci-private-docker-registry/

Tipps&Tricks

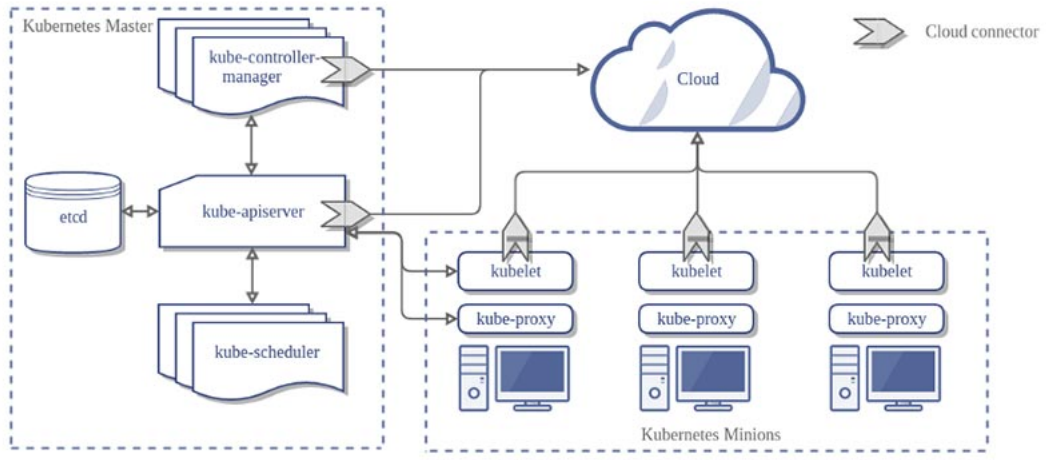
Image/Container debuggen in mit gitlab ci/cd

```
## in .gitlab-ci.yml

## standardmäßig wird in ruby image verwendet (wenn nichts anderes genannt wird)
## image: maven

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

build-job:       # This job runs in the build stage, which runs first.
  stage: build
  script:
    - echo "Compiling the code..."
    - echo "Compile complete."
    - cat /etc/os-release # Distribution Welche ? anzeigen
    - pwd # aktuelle Verzeichnis, in dem ich bin ! # in welchem Verzeichnis bin ich
    - ls -la # aktuelles verzeichnis in schöner "Admin" - Form
```

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/