

GITLAB CI/CD

Sonntag, 30. Dezember 2018 16:29

<https://en.wikipedia.org/wiki/YAML>

.gitlab-ci.yml

```
before_script:
- apt-get update -qq && apt-get install -y -qq sqlite3 libsqlite3-dev nodejs
- ruby -v
- which ruby
- gem install bundler --no-ri --no-rdoc
- bundle install --jobs $(nproc) "${FLAGS[@]}"
```

```
rspec:
script:
- bundle exec rspec
```

```
rubocop:
script:
- bundle exec rubocop
```

Jobs:

- Definiert eine Reihe von Jobs mit Einschränkungen, die angegeben, wann sie ausgeführt werden sollen. Unbegrenzte Anzahl von Jobs angeben, die als Elemente der obersten Ebene mit einem beliebigen Namen definiert sind und immer mindestens die Skriptklausel enthalten müssen

```
job1:
  script: "execute-script-for-job1"
-
job2:
  script: "execute-script-for-job2"
```

- Natürlich kann ein Befehl Code direkt ausführen (./configure;make;make install) oder ein Skript (test.sh) im Repository ausführen.
- Aufträge werden von Runnern abgeholt und in der Umgebung des Läufers ausgeführt. Jobs werden unabhängig voneinander ausgeführt.
- Jeder Job muss einen eindeutigen Namen haben, es gibt jedoch Schlüsselwörter, die nicht als Jobnamen verwendet werden können.

Keywords:

- **Image**
- **Services**
- **Stages**
- **Types**
- **Before_script**
- **After_script**
- **Variables**
- **cache**

Keyword	Required	Description
script	yes	Defines a shell script which is executed by Runner
extends	no	Defines a configuration entry that this job is going to inherit from
image	no	Use docker image, covered in Using Docker Images
services	no	Use docker services, covered in Using Docker Images
stage	no	Defines a job stage (default: <code>test</code>)
type	no	Alias for <code>stage</code>
variables	no	Define job variables on a job level
only	no	Defines a list of git refs for which job is created
except	no	Defines a list of git refs for which job is not created
tags	no	Defines a list of tags which are used to select Runner
allow_failure	no	Allow job to fail. Failed job doesn't contribute to commit status
when	no	Define when to run job. Can be <code>on_success</code> , <code>on_failure</code> , <code>always</code> or <code>manual</code>
dependencies	no	Define other jobs that a job depends on so that you can pass artifacts between them
artifacts	no	Define list of job artifacts
cache	no	Define list of files that should be cached between subsequent runs
before_script	no	Override a set of commands that are executed before job
after_script	no	Override a set of commands that are executed after job
environment	no	Defines a name of environment to which deployment is done by this job
coverage	no	Define code coverage settings for a given job
retry	no	Define when and how many times a job can be auto-retried in case of a failure
parallel	no	Defines how many instances of a job should be run in parallel

Extends:

Definiert einen Eintragsnamen, von dem ein Job, der erben wird. Unterstützt mehrstufige Vererbung, es wird jedoch nicht empfohlen, mehr als drei Ebenen zu verwenden. Max: 10 Verschaltungsebenen.

```
.tests:
  script: rake test
  stage: test
  only:
    refs:
      - branches

rspec:
  extends: .tests
  script: rake rspec
  only:
    variables:
      - $RSPEC
```

- Rspec-Job:

```
rspec:
  script: rake rspec
  stage: test
  only:
    refs:
      - branches
    variables:
      - $RSPEC
```

```
rspec:
  script: rake rspec
  stage: test
  only:
    refs:
      - branches
  variables:
    - $RSPEC
```

.tests ist in diesem Beispiel ein versteckter Schlüssel, es ist jedoch auch möglich, von regulären Jobs zu erben.

```
.tests:
  only:
    - pushes

.rspec:
  extends: .tests
  script: rake rspec

rspec 1:
  variables:
    RSPEC_SUITE: '1'
  extends: .rspec

rspec 2:
  variables:
    RSPEC_SUITE: '2'
  extends: .rspec

spinach:
  extends: .tests
  script: rake spinach
```

<https://docs.gitlab.com/ce/ci/yaml/README.html>

Pages:

<https://docs.gitlab.com/ce/user/project/pages/index.html>

Spezielle Aufgabe, mit der statische Inhalte in Gitlab hochgeladen werden, die für die Bereitstellung ihrer Website verwendet werden können. Es hat eine spezielle Syntax, daher müssen die beiden Voraussetzungen erfüllt sein:

- Jeder statische Inhalt muss sich in einem *public* Verzeichnis befinden.
- *Artifacts* mit einem Pfad zum Verzeichnis *public/* - Verzeichnis müssen definiert werden.

Im folgenden Beispiel werden einfach alle Dateien aus dem Stammverzeichnis des Projekts in das Verzeichnis *public/* verschoben. Die *.public*-Problemmumgebung ist so dass *cp* nicht in einer Endlosschleife auch *öffentlich/* sich selbst kopiert

Images:

Benutzerdefiniertes Docker-Image und eine Liste von Dienen angeben, die für die Zeit des Jobs verwendet werden können:

<https://docs.gitlab.com/ce/ci/docker/README.html>

Before script:

Befehl wird definiert, der vor allen Jobs ausgeführt werden soll, einschließlich Bereitstellungsjobs, aber nach der Wiederherstellung von Artefakten. Dies kann ein Array oder eine mehrzeilige Zeichenfolge sein.

Before_Script und das Haupt-*Script* wird separat ausgeführt. Abhängig vom Executor sind Änderungen außerhalb des Arbeitsbaums möglicherweise nicht sichtbar, z.B. Software, die im *before_script* installiert ist.

after script:

Wird der Befehl definiert, der nach allen Jobs ausgeführt wird, einschließlich fehlschlagender Jobs. Dies muss ein Array oder eine mehrzeilige Zeichenfolge sein.

```

before_script:
  - global before script

job:
  before_script:
    - execute this instead of global before script
  script:
    - my command
  after_script:
    - execute this after my script

```

Stages: Zur Definition von Stufen verwendet, die von Jobs verwendet werden können und sind global definiert. Spezifikation der Stufen ermöglicht flexible mehrstufige Pipelines. Die Reihenfolge der Elemente in Stufen bestimmt die Reihenfolge der Ausführung von Jobs.

1. Jobs der gleichen Stufe werden parallel ausgeführt.
2. Aufträge der nächsten Stufe werden ausgeführt, nachdem die Aufträge der vorherigen Stufe erfolgreich abgeschlossen wurden.

```

stages:
  - build
  - test
  - deploy

```

1. Alle *Build-Aufträge* parallel ausgeführt
2. Wenn *Build-Aufträge* erfolgreich sind, werden die Testjobs parallel ausgeführt.
3. Falls erfolgreich, alle *deploy-Jobs* parallel durchgeführt.
4. Wenn alle Bereitstellungsjobs erfolgreich sind, wird der Commit als bestanden markiert.
5. Wenn einer der vorherigen Jobs fehlschlägt, wird das Festschreiben als fehlschlagen markiert und es werden keine Jobs der weiteren Stufe ausgeführt.

Zwei Randfälle:

- Wenn in *.gitlab-ci.yml* keine Phasen definiert sind, dürfen *Build*, *Test* und *Deploy* standardmäßig als Jobphase verwendet werden.
- Wenn ein Job keine Phase angibt, wird dem Job die Testphase zugewiesen

Stage:

- Stufe wird pro Job definiert und basiert auf Stufen, die global definiert werden. Es ermöglicht das Gruppieren von Jobs in verschiedenen Phasen und Jobs derselben Phasen werden parallel ausgeführt.

```

stages:
  - build
  - test
  - deploy

job 1:
  stage: build
  script: make build dependencies

job 2:
  stage: build
  script: make build artifacts

job 3:
  stage: test
  script: make test

job 4:
  stage: deploy
  script: make deploy

```

Types: Sind veraltet und könnte in einer der zukünftigen Version entfernt werden. Stattdessen *Stages* verwenden.

Script: Skript ist das einzige erforderliche Schlüsselwort, das ein Job benötigt. Es ist ein Shell-Skript, das vom Runner ausgeführt wird.

```

job:
  script: "bundle exec rspec"

```

Dieser Parameter kann auch mehrere Befehle enthalten, die ein Array verwenden.

```

job:
  script:
    - uname -a
    - bundle exec rspec

```

Manchmal müssen Skriptbefehle in einfache oder doppelte Anführungszeichen gesetzt werden. Zum Beispiel müssen Befehle, die einen *Doppelpunkt* (:) enthalten, in Anführungszeichen gesetzt werden, damit der *YAML-Parser* das ganze Ding als String und nicht als *key-value-Paar* interpretieren kann.

Vorsichtig bei Sonderzeichen: { }, [], ,, & , * , # , ? , | , - , < , > , = , ! , % , @ , ` , .

Only // except:

Only und *except* sind zwei Parameter, die zum Erstellen von Jobs eingeschränkt werden sollen:

1. **Only:** Definiert nur die Namen von *branches* und *tags*, für die der Job ausgeführt wird.
2. **Except:** Definiert die Namen von *branches* und *tags*, für die der Job nicht ausgeführt wird.

Jobrichtlinien gelten folgende Regeln:

- *Only* und *except* sind inklusive. Wenn in einer Jobspezifikation beide definiert sind, wird *ref* nur *only* und *except* gefiltert.
- Verwendung regulärer Ausdrücke (<https://ruby-doc.org/core-2.6/Regexp.html>)
- *Only* und *except* erlaubt es einen Repository-Pfad zum Filtern von Jobs für *forks* anzugeben.

Only und *except* erlaubt es zum Benutzen folgender *Schlüsselwörter*:

Value	Description
branches	When a git reference of a pipeline is a branch.
tags	When a git reference of a pipeline is a tag.
api	When pipeline has been triggered by a second pipelines API (not triggers API).
external	When using CI services other than GitLab.
pipelines	For multi-project triggers, created using the API with <code>CI_JOB_TOKEN</code> .
pushes	Pipeline is triggered by a <code>git push</code> by the user.
schedules	For scheduled pipelines .
triggers	For pipelines created using a trigger token.
web	For pipelines created using Run pipeline button in GitLab UI (under your project's Pipelines).
merge_requests	When a merge request is created or updated (See pipelines for merge requests).

In diesem Beispiel wird der Job nur für Verweise ausgeführt, die mit *issue-*, wobei alle Zweige übersprungen werden:

```

job:
  # use regexp
  only:
    - /^issue-.*$/
  # use special keyword
  except:
    - branches

```

In diesem Beispiel wird der Job nur für *refs* versehen sind, oder wenn ein *Build* explizit über einen *API-Trigger* oder einen Pipeline-Zeitplan angefordert wird:

```

job:
  # use special keywords
  only:
    - tags
    - triggers
    - schedules

```

Der Repository-Pfad kann verwendet werden, um Aufträge nur für das übergeordnete Repository und nicht für *forks* auszuführen.

```
job:
  only:
    - branches@gitlab-org/gitlab-ce
  except:
    - master@gitlab-org/gitlab-ce
```

Das obrige Beispiel wird für alle *branches* auf *gitlab-org/gitlab-ce* ausgeführt, mit Ausnahme von *master*. Wenn ein Job weder nur noch eine Regel hat, wird standardmäßig nur Folgendes festgelegt: *Verzweigung, Tags*
For example,

```
job:
  script: echo 'test'
```

is translated to:

```
job:
  script: echo 'test'
  only: ['branches', 'tags']
```

GitLab unterstützt sowohl einfache als auch komplexe Strategien, sodass Sie ein Array und ein Hash-Konfigurationsschema verwenden können: Vier Schlüsselwörter sind verfügbar:

- Refs
- Variables
- Changes
- Kubernetes

If you use multiple keys under `only` or `except`, they act as an AND. The logic is:

(any of refs) AND (any of variables) AND (any of changes) AND (if kubernetes is active)

only:refs and except:refs

<https://docs.gitlab.com/ce/user/project/pipelines/schedules.html>

Die *refs*-Strategie kann die gleichen Werte wie die vereinfachte / außer Konfiguration annehmen. Im folgenden Beispiel wird der *deploy*-job nur erstellt, wenn die Pipeline für den *Master-branch* geplant ist oder ausgeführt wird.

```
deploy:
  only:
    refs:
      - master
      - schedules
```

only:kubernetes and except:kubernetes: Die *kubernetes*-Strategie akzeptiert nur das aktive Schlüsselwort. Im folgenden Beispiel wird der *deploy-Job* nur erstellt, wenn der *kubernetes service* im Projekt aktiv ist.

```
deploy:
  only:
    kubernetes: active
```

only:variables and except:variables: Schlüsselwort *variables* wird zur Definition von Variablenausdrücken verwendet. Mit anderen Worten, Sie können vordefinierten *Variablen/ Project/ group* oder *Umgebungsvariablen* verwenden, um einen Ausdruck zu definieren, den GitLab auswertet, um zu entscheiden, ob ein Job erstellt werden soll oder nicht.

```
deploy:
  script: cap staging deploy
  only:
    refs:
      - branches
    variables:
      - $RELEASE == "staging"
      - $STAGING
```

Ein anderer Anwendungsfall schließt *Jobs* abhängig von einer *commit Message* aus.

```
end-to-end:
  script: rake test:end-to-end
  except:
    variables:
      - $CI_COMMIT_MESSAGE =~ /skip-end-to-end-tests/
```

<https://docs.gitlab.com/ce/ci/variables/README.html#variables-expressions>

only:changes and except:changes: Verwendung des Schlüsselwort *Changes* mit oder nur mit Ausnahmen verwenden, können sie festlegen, ob ein Job basierend auf Dateien erstellen werden soll, die durch ein git push-Ereignis geändert wurden.

```
docker build:
  script: docker build -t my-image:$CI_COMMIT_REF_SLUG .
  only:
    changes:
      - Dockerfile
      - docker/scripts/*
      - dockerfiles/**/*
      - more_scripts/*.{rb,py,sh}
```

Wenn mehrere *Commits* in Gitlab in einen vorhandenen *Branches* verschieben, erstellt und löst GitLab den Docker *Build-Job* aus, vorausgesetzt, dass eines der Commits folgenden Änderungen enthält:

- Dockerfile
- Dateien innerhalb *docker/scripts directory*.
- Dateien und Unterverzeichnisse innerhalb des *Dockerfile-Verzeichnis*
- Dateien mit den Erweiterungen *rb, py, sh* im Verzeichnis *more_scripts*.

Wenn ein neuer Branche oder ein neuer Tag an Gitlab gesendet wird, wird die Richtlinie immer als *true* ausgewertet, und Gitlab erstellt einen *Job*. Diese Funktion ist noch nicht mit *Merge-Request* verbunden. Da GitLab Piples erstellt, bevo ein Benutzer eine *Merge-Request* erstellen kann, kennen wir zu diesem Zeitpunkt noch keinen Ziel *branche*.

Ohne einen Ziel-*branche* ist es nicht möglich zu wissen, was der gemeinsame Vorfahr ist. Daher erstellen wir in diesem Fall immer einen *Job*. Diese Funktion eignet sich am besten für stabile *Branches* wie *Master*, da in diesem Fall Gitlab das vorherige Commit verwendet, das in einem Zweig vorhanden ist, um es mit dem neusten SHA zu vergleichen, das gepusht wurde

Tags: Werden verwendet, um bestimmte Runner aus der Liste aller Runner auszuwählen, die dieses Projekt ausführen dürfen. Während der Registrierung eines *Runners* können sie die *Runner-Tags* angeben, z.B. *Ruby, Postgres, development*. Mit diesen können *Jobs mit Runnern* ausführen, denen die angegebenen *Tags* zugewiesen sind:

```
job:
  tags:
    - ruby
    - postgres
```

Diese stellt sicher, dass der Job vom einem *Runner* erstellt wird, für den *Ruby- und Postgres-Tags* definiert sind. Sie sind auch eine gute Möglichkeit, verschiedene Jobs auf verschiedenen Plattformen auszuführen. Wenn sie beispielsweise einen *OS X Runner* mit *Tag osx* und *Windows-Runner* mit *Tag-Fenster* verwenden, werden die folgenden Jobs auf den jeweiligen Plattformen ausgeführt.

```
windows job:
  stage:
    - build
  tags:
    - windows
  script:
    - echo Hello, %USERNAME%!

osx job:
  stage:
    - build
  tags:
    - osx
  script:
    - echo "Hello, $USER!"
```

allow failure: Lässt einen *Job* fehlschlagen, ohne den Rest der CI-Suite zu beeinträchtigen. Der Standardwert ist *false*, außer für manuelle *Jobs*. Ist die Option aktiviert und der Job fehlschlägt, wird auf der Benutzeroberfläche eine orangefarbene Warnung angezeigt. Der logische Fluss der *Pipeline* betrachtet den *Job* jedoch als erfolgreich/ bestanden und wird nicht blockiert.

Vorausgesetzt, dass alle anderen *Jobs* erfolgreich sind, wird auf der Stufe und in der *Pipeline des Jobs* dieselbe orangefarbene Warnung angezeigt. Das zugehörige Commit wird jedoch ohne Warnungen als *bestanden* markiert.

Im folgenden Beispiel werden zwei *Jobs* parallel ausgeführt. Falls *Job1* fehlschlägt, wird die Ausführung der nächsten *Stage* nicht angehalten, da sie mit *allow_failue: True* markiert ist.

```

job1:
  stage: test
  script:
    - execute_script_that_will_fail
  allow_failure: true

job2:
  stage: test
  script:
    - execute_script_that_will_succeed

job3:
  stage: deploy
  script:
    - deploy_to_staging

```

When: Wird verwendet, um *Jobs* zu implementieren, die im Fehlerhaft oder trotz des Fehler ausgeführt werden sollen. Kann auf einen der folgenden Werte gesetzt werden:

1. *On_success*-Job nur ausführen, wenn alle *Jobs* aus der vorherigen *Stage* erfolgreich sind (oder als erfolgreich angesehen werden, da sie als *allow_failue* markiert sind).
2. *On_failure*-Job nur ausführen, wenn mindestens ein Job aus der vorherigen *Stage* fehlschlägt.
3. *Allways*: Job unabhängig vom Status der *Jobs* aus früheren *Stages* ausführen.
4. *Manuell*: Job manuell ausführen (<https://docs.gitlab.com/ce/ci/yaml/README.html#whenmanual>)

```

stages:
  - build
  - cleanup_build
  - test
  - deploy
  - cleanup

build_job:
  stage: build
  script:
    - make build

cleanup_build_job:
  stage: cleanup_build
  script:
    - cleanup build when failed
  when: on_failure

test_job:
  stage: test
  script:
    - make test

deploy_job:
  stage: deploy
  script:
    - make deploy
  when: manual

cleanup_job:
  stage: cleanup
  script:
    - cleanup after jobs
  when: always

```

Das Skript macht folgendes:

1. Führen sie *cleanup_build_job* nur aus, wenn *build_job* fehlschlägt.
2. Führen sie *cleanup_job* immer als letzten Schritt in der Pipeline aus, unabhängig von Erfolg oder Misserfolg.
3. Gestattet es Ihnen, *deploy_job* manuell von der GitLab Benutzeroberfläche auszuführen.

when:manual: Manuelle Aktionen sind spezielle *Jobs*, die nicht automatisch ausgeführt werden. Sie müssen explizit von einem Benutzer gestartet werden. Ein Beispiel für die Verwendung *manueller Aktionen* wäre die Bereitstellung in einer Produktionsumgebung. *Manuelle Aktionen* können aus den Ansichten *Pipeline*, *Job*, *Umgebung* und *Deployments* gestartet werden. Diese können optional oder blockierend sein. Durch das Blockierend wird die Ausführung der Pipeline in dem Stadium blockiert, in dem diese Aktion definiert ist. Sie können die Ausführung der Pipeline forsetzen, wenn jemand eine manuelle Blockierungskation ausführt, indem Sie auf eine Wiedergabetaste klicken. Wenn eine Pipeline blockiert ist, wird sie nicht zusammengeführt, wenn

<https://docs.gitlab.com/ce/ci/environmental.html#manually-deploying-to-environments>

Zusammenführen bei erfolgreicher Pipeline eingestellt ist. Gesperrte Pipelines haben auch einen besonderen Status, der als manuell bezeichnet wird. Manuelle Aktionen sind standardmäßig nicht blockierend. Wenn sie eine manuelle Aktion blockieren möchten, müssen sie `allow_failure:false` zur Jobdefinition in `.gitlab-ci.yml` hinzufügen. Für optionale *manuelle Aktionen* ist `allow_failure:true` voreingestellt und ihre Status tragen nicht zum Gesamtstatus der Pipeline bei. Wenn als eine manuelle Aktion fehlschlägt, wird die Pipeline letztendlich erfolgreich sein. Manuelle Aktionen werden als *write actions* betrachtet. Daher werden Berechtigungen für *protected branches* verwendet, wenn der Benutzer eine Aktion auslösen möchte. Mit anderen Worten, um eine manuelle Aktion auszulösen, die einem *Branche* zugewiesen ist, muss der Benutzer die Möglichkeit haben, sich mit diesem *branche mergen*.

when:delayed (https://gitlab.com/gitlab-org/gitlab-ce/merge_requests/21767):

Verzögerter *Job* ist für die Ausführung von *Scripts* nach einer bestimmten Zeit. Dies ist nützlich, wenn Sie verhindern möchten, dass Aufträge sofort in den Status *Anstehend* wechseln. Sie können die Periode mit `start_in` key ist eine verstrichene Zeit in Sekunden, sofern keine Einheit angegeben wird. `Start_in` key muss kleiner oder gleich einer Stunde sein.

Wenn sich in einer Phase ein *Job* mit einer Verzögerung befindet, wird die Pipeline erst ausgeführt, wenn der *Job* mit Verzögerung abgeschlossen ist. Das bedeutet, dass dieses Schlüsselwort auch zum Einfügen von Verzögerung zwischen verschiedenen *Stages* verwendet werden kann. Der Timer eines verzögerten *Jobs* beginnt unmittelbar nach Abschluss der vorherigen Phase. Ähnlich wie bei anderen Arten von *Jobs* wird der Timer eines verzögerten *Jobs* erst gestartet, wenn die vorherige *Stage* bestanden ist. Im folgenden Beispiel wird ein *Job* mit dem Namen *timed rollout 10%* erstellt, der 30 Minuten nach Abschluss der vorherigen Phase ausgeführt wird.

```
timed rollout 10%:
  stage: deploy
  script: echo 'Rolling out 10% ...'
  when: delayed
  start_in: 30 minutes
```

Sie können den aktiven Zeitgeber eines verzögerten *Jobs* anhalten, indem Sie auf die Schaltfläche *Unschedule* klicken. Dieser Job wird in der Zukunft niemals ausgeführt, es sei denn, sie führen ihn manuell aus.

environment (<https://docs.gitlab.com/ce/ci/environments.html>)

Mit der *environment* wird definiert, dass ein Job in einer bestimmten Umgebung bereitgestellt wird. Wenn eine *environment* angegeben wird und keine Umgebung unter diesem Namen vorhanden ist, wird automatisch eine neue Umgebung erstellt.

```
deploy to production:
  stage: deploy
  script: git push production HEAD:master
  environment:
    name: production
```

Im obigen Beispiel wird der Job zur Bereitstellung in der Produktion als Bereitstellung in der Produktionsumgebung gekennzeichnet.

environment:name

Environment kann folgenden Namen beinhalten:

- Letters
- Digits
- Spaces
- -
- _
- /
- \$
- {
- }

Gängige Namen sind *qa*, *staging*, *production*. Anstatt den Namen der Umgebung direkt nach dem *environment* keyword zu definieren, können Sie ihn auch als separaten Wert definieren.

```
deploy to production:
  stage: deploy
  script: git push production HEAD:master
  environment:
    name: production
```

environment:url

Parameter *url* kann alle definierten CI-Variablen verwenden, einschließlich vordefinierter, sicherer Variablen und *.gitlab-ci.yml*-Variablen. Sie können jedoch keine unter Skript definierten Variablen verwenden. Dies ist ein optionaler Wert, der, wenn festgelegt, Schaltflächen an verschiedenen Stellen in Gitlab verfügbar macht. Wenn sie darauf klicken, gelangen zur definierten *URL*.

```

deploy to production:
  stage: deploy
  script: git push production HEAD:master
  environment:
    name: production
    url: https://prod.example.com

```

environment: on_stop:

Wenn Umgebung über eine *Stop-Aktion* definiert ist, löst GitLab automatisch eine *Stop-Aktion* aus, wenn der zugehörige *Branche* gelöscht wird. Das Schließen (Anhalten) von Umgebungen kann mit dem unter Umgebung definierten Schlüsselwort *on_stop* erreicht werden. Es deklariert eine andere *Job*, der ausgeführt wird, um die Umgebung zu schließen.

environment:action:

Wird verwendet in Verbindung mit *on_stop* verwendet und in dem Job definiert, der zum Schließen der Umgebung aufgerufen wird.

```

review_app:
  stage: deploy
  script: make deploy-app
  environment:
    name: review
    on_stop: stop_review_app

stop_review_app:
  stage: deploy
  script: make delete-app
  when: manual
  environment:
    name: review
    action: stop

```

Im obigen Beispiel haben wir den Job *review_app* für die Bereitstellung in der *deploy environment* eingerichtet und unter *on_stop* einen neuen Job *stop_review_app* definiert. Wenn der *review_app*-Job erfolgreich abgeschlossen ist, wird der *stop_review_app*-Job basierend auf dem unter *when* definierten Job ausgelöst. In diesem Fall haben wir die Einstellung auf *manual* festgelegt, sodass zur Ausführung eine manuelle Aktion über die Web-Oberfläche von GitLab erforderlich ist.

Für den Job *stop_review_app* müssen die folgenden Schlüsselwörter definiert sein:

- When
- environment:name
- environment:action
- Stage sollte mit *review_app* identisch sein, damit die Umgebung automatisch stoppt, wenn der *Branche* gelöscht wird.

Dynamic environments:

```

deploy as review app:
  stage: deploy
  script: make deploy
  environment:
    name: review/$CI_COMMIT_REF_NAME
    url: https://$CI_ENVIRONMENT_SLUG.example.com/

```

Bereitstellung als Überprüfungs-App-Job wird als Bereitstellung markiert, um die Umgebung *review / \$CI_COMMIT_REF_NAME* dynamisch zu erstellen, wobei *\$CI_COMMIT_REF_NAME* eine von Runner festgelegte Umgebungsvariable ist. Die Variable *\$CI_ENVIRONMENT_SLUG* basiert auf dem Umgebungsnamen, ist jedoch für die Aufnahme in *URLs* geeignet. In diesem Fall kann auf diese Umgebung mit einer *URL* wie <https://review-pow.example.com/> zugegriffen werden, wenn der Job zum Bereitstellen als *Review-App* in einem *Branche* namen *pow* Kausgeführt wurde. Dies bedeutet, dass der zugrunde liegende Server, auf dem sich die Anwendung befindet, ordnungsgemäß konfiguriert ist. Der übliche Anwendungsfall besteht darin, dynamische Umgebungen für *Branches* zu erstellen und diese als *Review-Apps* zu verwenden.

<https://gitlab.com/gitlab-examples/review-apps-nginx/>

Cache: Hiermit wird eine Liste von Dateien und Verzeichnissen angegeben, die zwischen Jobs zwischengespeichert werden sollen. Sie können nur Pfade verwenden, die sich im Projektarbeitsbereich befinden. Wenn der *Cache* außerhalb des Auftragsumfangs definiert ist, bedeutet dies, dass der global festgelegt ist und alle Aufträge diese Definitionen verwendet.

cache:paths:

Verwenden Sie die Direktive *path*, um auszuwählen, welche Dateien oder Verzeichnisse zwischengespeichert werden sollen. Platzhalter können ebenfalls verwendet werden. Alle Dateien in Binärdateien zwischenspeichern, die auf *.apk* und die *.config*-Datei enden:

```
rspec:
  script: test
  cache:
    paths:
      - binaries/*.apk
      - .config
```

Locally defined cache overrides globally defined options. The following `rspec` job will cache only `binaries/`:

```
cache:
  paths:
    - my/files

rspec:
  script: test
  cache:
    key: rspec
    paths:
      - binaries/
```

Da der Cache von Jobs gemeinsam genutzt wird, sollten Sie, wenn Sie unterschiedliche Pfade für verschiedene Jobs verwenden, auch einen anderen Cache festlegen. Andernfalls kann der Cache-Inhalt überschrieben werden

cache:key:

Mit der `key`-Direktive können Sie die Affinität der Zwischenspeicherung zwischen Jobs definieren. So haben Sie einen einzigen Cache für alle Jobs, Cache pro Job, Cache pro Zweig oder jede andere Art und Weise, die zu Ihrem Workflow passt. Auf diese Weise können Sie die Zwischenspeicherung optimieren und Daten zwischen verschiedenen Jobs oder sogar verschiedenen Zweigen zwischenspeichern.

Die Variable `cache: key` kann jede der vordefinierten Variablen verwenden. Wenn der Standardschlüssel nicht festgelegt ist, handelt es sich lediglich um einen literalen Standardwert. Dies bedeutet, dass standardmäßig alle Pipelines und Jobs ab GitLab 9.0 gemeinsam genutzt werden.

For example, to enable per-branch caching:

```
cache:
  key: "$CI_COMMIT_REF_SLUG"
  paths:
    - binaries/
```

If you use **Windows Batch** to run your shell scripts you need to replace `$` with `%`:

```
cache:
  key: "%CI_COMMIT_REF_SLUG%"
  paths:
    - binaries/
```

cache:untracked:

Setze `untracked: true`, um alle Dateien zwischenspeichern, die nicht in Ihrem Git-Repository gespeichert sind:

```
rspec:
  script: test
  cache:
    untracked: true
```

Cache all Git untracked files and files in `binaries`:

```
rspec:
  script: test
  cache:
    untracked: true
    paths:
      - binaries/
```

cache:policy:

Das Standardverhalten eines Zwischenspeicherungsauftrags besteht darin, die Dateien zu Beginn der Ausführung herunterzuladen und am Ende erneut hochzuladen. Dadurch können alle vom Job vorgenommenen Änderungen für zukünftige Läufe beibehalten werden. Dies wird als Pull-Push-

Cache-Richtlinie bezeichnet.

Wenn Sie wissen, dass der Job die zwischengespeicherten Dateien nicht ändert, können Sie den Upload-Schritt überspringen, indem Sie die Richtlinie festlegen: Ziehen Sie die Jobspezifikation ab. Normalerweise wird dies zu einem früheren Zeitpunkt mit einem gewöhnlichen Cache-Job verbunden, um sicherzustellen, dass der Cache von Zeit zu Zeit aktualisiert wird:

```
stages:
  - setup
  - test

prepare:
  stage: setup
  cache:
    key: gems
    paths:
      - vendor/bundle
  script:
    - bundle install --deployment

rspec:
  stage: test
  cache:
    key: gems
    paths:
      - vendor/bundle
    policy: pull
  script:
    - bundle exec rspec ...
```

Dies hilft, die Jobausführung zu beschleunigen und die Belastung des Cache-Servers zu reduzieren, insbesondere wenn eine große Anzahl von Cache-Jobs parallel ausgeführt wird. Wenn Sie über einen Job verfügen, der den Cache ohne Verweis auf den vorherigen Inhalt unbedingt neu erstellt, können Sie außerdem `policy: Push` in diesem Job verwenden, um den Download-Schritt zu überspringen

Artifacts: https://docs.gitlab.com/ce/user/project/pipelines/job_artifacts.html

artifacts wird verwendet, um eine Liste von Dateien und Verzeichnissen anzugeben, die nach erfolgreichem Abschluss an den Job angehängt werden sollen. Die Artefakte werden nach erfolgreichem Abschluss des Jobs an GitLab gesendet und stehen in der GitLab-Benutzeroberfläche zum Download zur Verfügung.

artifacts:paths (<https://docs.gitlab.com/ce/ci/yaml/README.html#dependencies>)

Sie können nur Pfade verwenden, die sich im Projektarbeitsbereich befinden. Informationen zum Übergeben von Artefakten zwischen verschiedenen Jobs finden Sie unter Abhängigkeiten.

Senden Sie alle Dateien in Binärdateien und `.config`:

```
artifacts:
  paths:
    - binaries/
    - .config
```

To disable artifact passing, define the job with empty `dependencies`:

```
job:
  stage: build
  script: make build
  dependencies: []
```

Möglicherweise möchten Sie Artefakte nur für gekennzeichnete Releases erstellen, um zu vermeiden, dass der Build-Server-Speicher mit temporären Build-Artefakten gefüllt wird.

Artefakte nur für Tags erstellen (Standard-Job erstellt keine Artefakte):

```

default-job:
  script:
    - mvn test -U
  except:
    - tags

release-job:
  script:
    - mvn package -U
  artifacts:
    paths:
      - target/*.war
  only:
    - tags

```

artifacts:name:

Mit der Direktive *name* können Sie den Namen des erstellten Artefaktarchivs definieren. Auf diese Weise können Sie jedem Archiv einen eindeutigen Namen geben, der nützlich sein kann, wenn Sie das Archiv von GitLab herunterladen möchten. Die Variable *artefakte: name* kann jede der vordefinierten Variablen verwenden. Der Standardname ist Artefakte, der beim Herunterladen zu Artefakten wird.

```

job:
  artifacts:
    name: "$CI_COMMIT_REF_NAME"
    paths:
      - binaries/

```

To create an archive with a name of the current job and the current branch or tag including only the binaries directory:

```

job:
  artifacts:
    name: "$CI_JOB_NAME-$CI_COMMIT_REF_NAME"
    paths:
      - binaries/

```

To create an archive with a name of the current *stage* and branch name:

```

job:
  artifacts:
    name: "$CI_JOB_STAGE-$CI_COMMIT_REF_NAME"
    paths:
      - binaries/

```

If you use **Windows Batch** to run your shell scripts you need to replace `$` with `%`:

```

job:
  artifacts:
    name: "%CI_JOB_STAGE%-CI_COMMIT_REF_NAME%"
    paths:
      - binaries/

```

If you use **Windows PowerShell** to run your shell scripts you need to replace `$` with `$env:`:

```

job:
  artifacts:
    name: "$env:CI_JOB_STAGE-$env:CI_COMMIT_REF_NAME"
    paths:
      - binaries/

```

artifacts:untracked:

Artefakte: untracked wird verwendet, um alle nicht protokollierten Git-Dateien als Artefakte hinzuzufügen (entlang der in *Artefakte: path* definierten Pfade).

Send all Git untracked files:

```
artifacts:
  untracked: true
```

Send all Git untracked files and files in `binaries`:

```
artifacts:
  untracked: true
  paths:
    - binaries/
```

artifacts:when:

Artifacts:when Wird verwendet, um Artefakte bei einem Jobfehler oder trotz des Fehlers hochzuladen.

Artifacts:when Wann kann auf einen der folgenden Werte gesetzt werden:

1. *on_success* - Laden Sie Artefakte nur hoch, wenn der Job erfolgreich abgeschlossen wurde. Dies ist die Standardeinstellung.
2. *on_failure* - Hochladen von Artefakten nur, wenn der Job fehlschlägt.
3. *always* - Artefakte unabhängig vom Jobstatus hochladen.

To upload artifacts only when job fails:

```
job:
  artifacts:
    when: on_failure
```

artifacts:expire in:

Mit `expire_in` können Sie angeben, wie lange Artefakte leben sollen, bevor sie ablaufen und daher gelöscht werden. Dies gilt ab dem Zeitpunkt, zu dem sie hochgeladen und in GitLab gespeichert werden. Wenn die Ablaufzeit nicht definiert ist, wird standardmäßig die Einstellung für die gesamte Instanz verwendet (standardmäßig 30 Tage, für immer bei GitLab.com).

Sie können die Schaltfläche "Behalten" auf der Jobseite verwenden, um den Ablauf zu überschreiben und Artefakte für immer beizubehalten.

Nach ihrem Ablauf werden Artefakte standardmäßig stündlich (über einen Cron-Job) gelöscht und sind nicht mehr zugänglich.

Der Wert von `expire_in` ist eine verstrichene Zeit in Sekunden, sofern keine Einheit angegeben wird.

```
job:
  artifacts:
    expire_in: 1 week
```

artifacts:reports:

Mit dem Schlüsselwort `reports` werden Testberichte von Jobs gesammelt und in der Benutzeroberfläche von GitLab (Zusammenführungsanforderungen, Pipeline-Ansichten) angezeigt. Lesen Sie, wie Sie dies mit JUnit-Berichten verwenden können.

https://docs.gitlab.com/ce/ci/yaml/README.html#artifacts-expire_in

artifacts:reports:junit:

Der JUnit-Bericht erfasst JUnit-XML-Dateien als Artefakte. Obwohl JUnit ursprünglich in Java entwickelt wurde, gibt es viele Fremdanbieter-Ports für andere Sprachen wie JavaScript, Python, Ruby usw.

Weitere Informationen und Beispiele finden Sie unter JUnit-Testberichte. Im Folgenden finden Sie ein Beispiel für das Erfassen einer JUnit-XML-Datei mit dem RSpec-Testtool von Ruby:

```
rspec:
  stage: test
  script:
    - bundle install
    - rspec --format RspecJUnitFormatter --out rspec.xml
  artifacts:
    reports:
      junit: rspec.xml
```

- https://docs.gitlab.com/ce/ci/junit_test_reports.html
- https://www.ibm.com/support/knowledgecenter/en/SSQ2R2_14.1.0/com.ibm.rsar.analysis.codereview.cobol.doc/topics/cac_userresults_junit.html
- <https://en.wikipedia.org/wiki/JUnit#Ports>
- https://docs.gitlab.com/ce/ci/junit_test_reports.html

Die gesammelten JUnit-Berichte werden als Artefakt in GitLab hochgeladen und automatisch in Zusammenführungsanforderungen angezeigt.

Dependencies:

Diese Funktion sollte in Verbindung mit Artefakten verwendet werden und ermöglicht es Ihnen, die Artefakte zu definieren, die zwischen

verschiedenen Jobs übertragen werden sollen. Beachten Sie, dass Artefakte aus allen vorherigen Stufen standardmäßig übergeben werden. Um diese Funktion zu verwenden, definieren Sie Abhängigkeiten im Kontext des Jobs und übergeben Sie eine Liste aller vorherigen Jobs, von denen die Artefakte heruntergeladen werden sollen. Sie können nur Jobs von Stufen definieren, die vor den aktuellen ausgeführt werden. Ein Fehler wird angezeigt, wenn Sie Jobs von der aktuellen oder der nächsten Stufe definieren. Wenn Sie ein leeres Array definieren, werden keine Artefakte für diesen Job heruntergeladen. Der Status des vorherigen Jobs wird bei der Verwendung von Abhängigkeiten nicht berücksichtigt. Wenn er fehlschlägt oder es sich um einen manuellen Job handelt, der nicht ausgeführt wurde, tritt kein Fehler auf.

Im folgenden Beispiel definieren wir zwei Jobs mit Artefakten, *build: osx* und *build: linux*. Wenn der *Test: osx* ausgeführt wird, werden die Artefakte von *build: osx* heruntergeladen und im Kontext des Builds extrahiert. Dasselbe gilt für *test: Linux* und Artefakte von *Build: Linux*.

Die Jobbereitstellung lädt Artefakte aus allen vorherigen Jobs aufgrund der Stufenpriorität herunter:

```
build:osx:
  stage: build
  script: make build:osx
  artifacts:
    paths:
      - binaries/

build:linux:
  stage: build
  script: make build:linux
  artifacts:
    paths:
      - binaries/

test:osx:
  stage: test
  script: make test:osx
  dependencies:
    - build:osx

test:linux:
  stage: test
  script: make test:linux
  dependencies:
    - build:linux

deploy:
  stage: deploy
  script: make deploy
```

Wenn die Artefakte des Jobs, der als Abhängigkeit festgelegt wurde, abgelaufen oder gelöscht wurden, schlägt der abhängige Job fehl. https://docs.gitlab.com/ce/administration/job_artifacts.html#validation-for-dependencies

Coverage:

Mit der *Coverage* können Sie konfigurieren, wie die Codeabdeckung aus der Jobausgabe extrahiert wird.

Reguläre Ausdrücke sind der einzig gültige Wert, der hier erwartet wird. Daher ist die Verwendung von umgebend / obligatorisch, um eine reguläre Ausdruckszeichenfolge konsistent und explizit darzustellen. Sie müssen Sonderzeichen mit Escapezeichen versehen, wenn Sie sie buchstäblich abgleichen möchten.

```
job1:
  script: rspec
  coverage: '/Code coverage: \d+\.\d+/'
```

Retry:

Mit *retry* können Sie konfigurieren, wie oft ein Job im Fehlerfall erneut versucht wird. Wenn ein *Job* fehlschlägt und die Wiederholung konfiguriert wurde, wird er bis zu der im Wiederholungsschlüsselwort angegebenen Anzahl erneut verarbeitet. Wenn Wiederholung auf 2 gesetzt ist und ein Job in einem zweiten Durchlauf erfolgreich ist (erster Versuch), wird er nicht erneut wiederholt. Der Wiederholungswert muss eine positive ganze Zahl sein, gleich oder größer als 0, jedoch niedriger oder gleich 2 (maximal zwei Wiederholungen, insgesamt drei Durchläufe).

```
test:
  script: rspec
  retry: 2
```

By default, a job will be retried on all failure cases. To have a better control on which failures to retry, `retry` can be a hash with the following keys:

- `max`: The maximum number of retries.
- `when`: The failure cases to retry.

To retry only runner system failures at maximum two times:

```
test:
  script: rspec
  retry:
    max: 2
    when: runner_system_failure
```

If there is another failure, other than a runner system failure, the job will not be retried.

To retry on multiple failure cases, `when` can also be an array of failures:

```
test:
  script: rspec
  retry:
    max: 2
    when:
      - runner_system_failure
      - stuck_or_timeout_failure
```

Possible values for `when` are:

- `always`: Retry on any failure (default).
- `unknown_failure`: Retry when the failure reason is unknown.
- `script_failure`: Retry when the script failed.
- `api_failure`: Retry on API failure.
- `stuck_or_timeout_failure`: Retry when the job got stuck or timed out.
- `runner_system_failure`: Retry if there was a runner system failure (e.g. setting up the job failed).
- `missing_dependency_failure`: Retry if a dependency was missing.
- `runner_unsupported`: Retry if the runner was unsupported.

Parallel:

Mit *parallel* können Sie konfigurieren, wie viele Instanzen eines Jobs parallel ausgeführt werden sollen. Dieser Wert muss größer oder gleich zwei (2) und kleiner oder gleich 50 sein. Dadurch werden N Instanzen desselben Jobs erstellt, die parallel ausgeführt werden. Sie werden sequentiell von Jobname 1 / N bis Jobname N / N benannt.

Für jeden Job werden die Umgebungsvariablen `CI_NODE_INDEX` und `CI_NODE_TOTAL` gesetzt.

```
test:
  script: rspec
  parallel: 5
```

Include:

Mit dem Schlüsselwort *include* können Sie das Einbinden externer YAML-Dateien zulassen.

Im folgenden Beispiel wird der Inhalt von `.before-script-template.yml` zusammen mit dem Inhalt von `.gitlab-ci.yml` automatisch abgerufen und ausgewertet:

```
# Content of https://gitlab.com/awesome-project/raw/master/.before-script-template.yml

before_script:
  - apt-get update -qq && apt-get install -y -qq sqlite3 libsqlite3-dev nodejs
  - gem install bundler --no-document
  - bundle install --jobs $(nproc) "${FLAGS[@]}"
```

```
# Content of .gitlab-ci.yml

include: 'https://gitlab.com/awesome-project/raw/master/.before-script-template.yml'

rspec:
  script:
    - bundle exec rspec
```

Sie können es entweder als einzelne Zeichenfolge definieren oder, falls Sie mehr als eine Datei einfügen möchten, ein Array mit verschiedenen Werten. Die folgenden Beispiele sind beide gültige Fälle:


```
# Single string
include: '/templates/.after-script-template.yml'
```

```
# Array
include:
  - 'https://gitlab.com/awesome-project/raw/master/.before-script-template.yml'
  - '/templates/.after-script-template.yml'
```

- *local* zu demselben Repository, auf das unter Verwendung vollständiger Pfade in demselben Repository verwiesen wird, wobei / das Stammverzeichnis ist. Zum Beispiel:

```
# Within the repository
include: '/templates/.gitlab-ci-template.yml'
```

Note: You can only use files that are currently tracked by Git on the same branch your configuration file is. In other words, when using a **local file**, make sure that both `.gitlab-ci.yml` and the local file are on the same branch.

Note: We don't support the inclusion of local files through Git submodules paths.

Remote an einem anderen Ort, auf den über HTTP / HTTPS zugegriffen wird, und der über die vollständige URL referenziert wird. Zum Beispiel:

```
include: 'https://gitlab.com/awesome-project/raw/master/.gitlab-ci-template.yml'
```

Note: The remote file must be publicly accessible through a simple GET request, as we don't support authentication schemas in the remote URL.

Note: In order to include files from another repository inside your local network, you may need to enable the **Allow requests to the local network from hooks and services** checkbox located in the **Settings > Network > Outbound requests** section within the Admin area.

Seit GitLab 10.8 mischen wir nun die in include definierten Dateien gründlich mit denen in `.gitlab-ci.yml`. Dateien, die von include definiert werden, werden immer zuerst ausgewertet und mit dem Inhalt von `.gitlab-ci.yml` zusammengeführt, unabhängig von der Position des Include-Schlüsselworts. Sie können das Zusammenführen nutzen, um Details in enthaltenen CI-Konfigurationen mit lokalen Definitionen anzupassen und zu überschreiben.

```
# Content of https://company.com/autodevops-template.yml

variables:
  POSTGRES_USER: user
  POSTGRES_PASSWORD: testing_password
  POSTGRES_DB: $CI_ENVIRONMENT_SLUG

production:
  stage: production
  script:
    - install_dependencies
    - deploy
  environment:
    name: production
    url: https://$CI_PROJECT_PATH_SLUG.$AUTO_DEVOPS_DOMAIN
  only:
    - master
```

```
# Content of .gitlab-ci.yml

include: 'https://company.com/autodevops-template.yml'

image: alpine:latest

variables:
  POSTGRES_USER: root
  POSTGRES_PASSWORD: secure_password

stages:
  - build
  - test
  - production

production:
  environment:
    url: https://domain.com
```

In diesem Fall wurden die Variablen `POSTGRES_USER` und `POSTGRES_PASSWORD` zusammen mit der in `autodevops-template.yml` definierten Umgebungs-URL des Produktionsjobs durch neue Werte überschrieben, die in `.gitlab-ci.yml` definiert sind.

Durch das Zusammenführen können Sie Wörterbuchzuordnungen erweitern und überschreiben. Sie können jedoch keine Elemente zu einem enthaltenen Array hinzufügen oder ändern. Um beispielsweise ein zusätzliches Element zum Produktionsauftragskript hinzuzufügen, müssen Sie die vorhandenen Skriptelemente wiederholen.

```
# Content of https://company.com/autodevops-template.yml

production:
  stage: production
  script:
    - install_dependencies
    - deploy
```

```
# Content of .gitlab-ci.yml

include: 'https://company.com/autodevops-template.yml'

stages:
  - production

production:
  script:
    - install_dependencies
    - deploy
    - notify_owner
```

Wenn `install_dependencies` und `deploy` in `.gitlab-ci.yml` nicht wiederholt werden, wären sie in diesem Fall nicht Bestandteil des Skripts für den Produktionsjob in der kombinierten CI-Konfiguration.

Variables: <https://docs.gitlab.com/ce/ci/variables/README.html#variables>

Mit GitLab CI / CD können Sie Variablen in `.gitlab-ci.yml` definieren, die dann in der Jobumgebung übergeben werden. Sie können global und pro Job festgelegt werden. Wenn das Schlüsselwort `variables` auf Jobebene verwendet wird, werden die globalen und vordefinierten YAML-Variablen überschrieben.

Sie werden im Git-Repository gespeichert und sollen nicht sensible Projektkonfigurationen speichern, zum Beispiel:

```
variables:
  DATABASE_URL: "postgres://postgres@postgres/my_database"
```

- <https://docs.gitlab.com/ce/ci/variables/README.html#predefined-variables-environment-variables>
- <https://docs.gitlab.com/ce/ci/variables/README.html>

Diese Variablen können später in allen ausgeführten Befehlen und Skripten verwendet werden. Die von YAML definierten Variablen werden auch auf alle erstellten Service-Container festgelegt, sodass sie fein abgestimmt werden können.

Abgesehen von den benutzerdefinierten Variablen gibt es auch diejenigen, die vom Runner selbst eingerichtet wurden. Ein Beispiel wäre `CI_COMMIT_REF_NAME`, das den Wert des Zweig- oder Variablennamens enthält, für den das Projekt erstellt wird. Neben den Variablen, die Sie in `.gitlab-ci.yml` einstellen können, gibt es auch die sogenannten Variablen, die in der Benutzeroberfläche von GitLab eingestellt werden können.

Git strategy:

Sie können `GIT_STRATEGY` festlegen, das zum Abrufen von aktuellem Anwendungscode verwendet wird, entweder global oder pro Job im Variablenabschnitt. Wenn Sie keine Angabe machen, wird der Standardwert aus den Projekteinstellungen verwendet.

Es gibt drei mögliche Werte: Klonen, Abrufen und keine.

Clone ist die langsamste Option. Es wird das Repository für jeden Job von Grund auf neu geklont, um sicherzustellen, dass der Projektarbeitsbereich immer makellos ist.

```
variables:
  GIT_STRATEGY: clone
```

Fetch ist schneller, da der Projektarbeitsbereich erneut verwendet wird (wenn auf Klonen zurückgegriffen wird, falls er nicht vorhanden ist). Mit `git clean` werden alle durch den letzten Job vorgenommenen Änderungen rückgängig gemacht. Mit `git fetch` werden Commits abgerufen, die seit dem letzten Job ausgeführt wurden.

```
variables:
  GIT_STRATEGY: fetch
```

None verwendet den Projektarbeitsbereich erneut, überspringt jedoch alle Git-Vorgänge (einschließlich des Vor-Klon-Skripts von GitLab Runner, falls vorhanden). Dies ist vor allem für Jobs nützlich, die ausschließlich mit Artefakten arbeiten (z. B. implementieren). Git-Repository-Daten sind zwar vorhanden, aber sicherlich nicht mehr aktuell. Daher sollten Sie sich nur auf Dateien verlassen, die aus dem Cache des Arbeitsbereichs oder Artefakten in den Projektarbeitsbereich geladen werden.

```
variables:
  GIT_STRATEGY: none
```

Git submodule strategy:

Die Variable `GIT_SUBMODULE_STRATEGY` wird verwendet, um zu steuern, ob / wie Git-Submodule beim Abrufen des Codes vor einem Build enthalten sind. Sie können sie im Variablenbereich global oder pro Job festlegen.

Es gibt drei mögliche Werte: none, normal und rekursiv:

- *none* bedeutet, dass beim Abrufen des Projektkodes keine Submodule enthalten sind. Dies ist die Standardeinstellung, die dem Verhalten vor Version 1.10 entspricht.

- *Normal* bedeutet, dass nur die Top-Level-Submodule enthalten sind. Es ist äquivalent zu:

```
git submodule sync
git submodule update --init
```

- *recursive* bedeutet, dass alle Submodule (einschließlich Submodule von Submodulen) eingeschlossen werden. Diese Funktion benötigt Git v1.8.1 und höher. Wenn Sie einen GitLab Runner mit einem nicht auf Docker basierenden Executor verwenden, stellen Sie sicher, dass die Git-Version diese Anforderung erfüllt. Es ist äquivalent zu

```
git submodule sync --recursive
git submodule update --init --recursive
```

Damit dieses Feature ordnungsgemäß funktioniert, müssen die Submodule (in `.gitmodules`) folgendermaßen konfiguriert werden:

- die HTTP (S) -URL eines öffentlich zugänglichen Repositories oder
- einen relativen Pfad zu einem anderen Repository auf demselben GitLab-Server.

https://docs.gitlab.com/ce/ci/git_submodules.html

Git checkout:

Die Variable `GIT_CHECKOUT` kann verwendet werden, wenn für `GIT_STRATEGY` entweder Klonen oder Abrufen festgelegt ist, um anzugeben, ob ein Git-Checkout ausgeführt werden soll. Wenn nicht angegeben, ist der Standardwert "*true*". Sie können sie im Variablenbereich global oder pro Job festlegen.

Falls *False*:

- *when doing fetch*: Aktualisieren Sie das Repository und belassen Sie die Arbeitskopie in der aktuellen Version.

- *when doing clone*: Sie das Repository und belassen die Arbeitskopie im Standardzweig.

Wenn diese Einstellung auf true gesetzt ist, bedeutet dies, dass der Runner die Arbeitskopie sowohl für die Klon- als auch für die Abrufstrategie auf eine mit der CI-Pipeline zusammenhängende Revision auscheckt:

```
variables:
  GIT_STRATEGY: clone
  GIT_CHECKOUT: "false"
script:
  - git checkout master
  - git merge $CI_BUILD_REF_NAME
```

Job stages attempts:

You can set the number for attempts the running job will try to execute each of the following stages:

Variable	Description
GET_SOURCES_ATTEMPTS	Number of attempts to fetch sources running a job
ARTIFACT_DOWNLOAD_ATTEMPTS	Number of attempts to download artifacts running a job
RESTORE_CACHE_ATTEMPTS	Number of attempts to restore the cache running a job

The default is one single attempt.

Example:

```
variables:
  GET_SOURCES_ATTEMPTS: 3
```

You can set them globally or per-job in the `variables` section.

Shallow cloning:

Sie können die Tiefe des Abrufs und Klonens mit `GIT_DEPTH` angeben. Dies ermöglicht ein flaches Klonen des Repositorys, was das Klonen für Repositorys mit einer großen Anzahl von *Commits* oder alten, großen Binärdateien erheblich beschleunigen kann. Der Wert wird an *git fetch* und *git clone* übergeben.

Da das Abrufen und Klonen von Git auf einem *Ref* basiert, z. B. auf einem Zweignamen, können Läufer keine bestimmten Festschreibungs-SHA klonen. Wenn sich in der Warteschlange mehrere Jobs befinden oder Sie einen alten Job wiederholen, muss die zu testende Festschreibung innerhalb des geklonten Git-Verlaufs liegen. Wenn Sie für `GIT_DEPTH` einen zu kleinen Wert festlegen, kann es unmöglich sein, diese alten *Commits* auszuführen. In den Jobprotokollen wird ein nicht aufgelöster Verweis angezeigt. Sie sollten dann erneut überdenken, `GIT_DEPTH` auf einen höheren Wert zu ändern.

Jobs, die sich auf *git compare* verlassen, funktionieren möglicherweise nicht korrekt, wenn `GIT_DEPTH` festgelegt ist, da nur ein Teil des Git-Verlaufs vorhanden ist.

Nur die letzten 3 Commits abrufen oder klonen:

```
variables:
  GIT_DEPTH: "3"
```

You can set it globally or per-job in the `variables` section.

Special YAML features:

Es ist möglich, spezielle YAML-Funktionen wie Anker (&), Aliase (*) und Map Merge (<<) zu verwenden, wodurch Sie die Komplexität von `.gitlab-ci.yml` erheblich reduzieren können.

<https://learnxinyminutes.com/docs/yaml/>

Hidden keys (jobs)

Wenn Sie einen Job vorübergehend "deaktivieren" möchten, anstatt alle Zeilen auszuwerten, in denen der Job definiert ist:

```
#hidden_job:
# script:
#   - run test
```

Sie können den Namen stattdessen mit einem Punkt (.) beginnen und wird von GitLab CI nicht verarbeitet. Im folgenden Beispiel wird `.hidden_job` ignoriert:

```
.hidden_job:
  script:
    - run test
```

Anchors:

YAML verfügt über eine praktische Funktion namens Anker, mit der Sie Inhalte ganz einfach in Ihrem Dokument kopieren können. Anker können zum Duplizieren / Erben von Eigenschaften verwendet werden. Dies ist ein perfektes Beispiel für verborgene Schlüssel, um Vorlagen für Ihre Jobs bereitzustellen.

Im folgenden Beispiel werden Anker und Kartenzusammenführung verwendet. Es werden zwei Jobs erstellt, test1 und test2, die die Parameter von `.job_template` übernehmen, für die jeweils ein eigenes benutzerdefiniertes Skript definiert ist:

```
.job_template: &job_definition # Hidden key that defines an anchor named 'job_definition'
  image: ruby:2.1
  services:
    - postgres
    - redis

test1:
  <<: *job_definition          # Merge the contents of the 'job_definition' alias
  script:
    - test1 project

test2:
  <<: *job_definition          # Merge the contents of the 'job_definition' alias
  script:
    - test2 project
```

& setzt den Namen des Ankers (job_definition), << bedeutet "den angegebenen Hash mit dem aktuellen Hash zusammenführen" und * enthält den benannten Anker (wieder job_definition). Die erweiterte Version <https://git.rwth-aachen.de/help/ci/variables/README#variablesion> sieht folgendermaßen aus:

```
.job_template:
  image: ruby:2.1
  services:
    - postgres
    - redis

test1:
  image: ruby:2.1
  services:
    - postgres
    - redis
  script:
    - test1 project

test2:
  image: ruby:2.1
  services:
    - postgres
    - redis
  script:
    - test2 project
```

Sehen wir uns ein weiteres Beispiel an. Dieses Mal verwenden wir Anker, um zwei Service-Sets zu definieren. Dadurch werden zwei Jobs erstellt, `test: postgres` und `test: mysql`, die die in `.job_template` definierte Skriptanweisung und die in `.postgres_services` bzw. `.mysql_services` definierte Services-Direktive gemeinsam nutzen:

```

.job_template: &job_definition
  script:
    - test project

.postgres_services:
  services: &postgres_definition
    - postgres
    - ruby

.mysql_services:
  services: &mysql_definition
    - mysql
    - ruby

test:postgres:
  <<: *job_definition
  services: *postgres_definition

test:mysql:
  <<: *job_definition
  services: *mysql_definition

```

```

.job_template:
  script:
    - test project

.postgres_services:
  services:
    - postgres
    - ruby

.mysql_services:
  services:
    - mysql
    - ruby

test:postgres:
  script:
    - test project
  services:
    - postgres
    - ruby

test:mysql:
  script:
    - test project
  services:
    - mysql
    - ruby

```

Triggers: <https://docs.gitlab.com/ce/ci/triggers/README.html>

Triggers können verwendet werden, um mit einem API-Aufruf die Neuerstellung eines bestimmten Zweigs, Tags oder Commits zu erzwingen.

Skipping jobs: <https://git-scm.com/docs/git-push#git-push--oltoptiongt>

Wenn Ihre Commit-Nachricht `[ci skip]` oder `[skip ci]` enthält, wird der Commit erstellt, aber die Pipeline wird ausgelassen. Alternativ kann man die Git-Push-Option `ci.skip` übergeben, wenn Git 2.10 oder neuer verwendet wird: `$ git push -o ci.skip`

Validate the .gitlab-ci.yml

Jede Instanz von GitLab CI verfügt über ein eingebettetes Debugging-Tool namens Lint, das den Inhalt Ihrer `.gitlab-ci.yml`-Dateien überprüft. Sie finden das Lint unter der Seite `ci / lint` Ihres Projekt-Namensraums (z. B. <http://gitlab-example.com/gitlab-org/project-123/-/ci/lint>).

GitLab CI/CD Variables <https://git.rwth-aachen.de/help/ci/variables/README#variables>

Wenn Sie einen Auftrag von GitLab CI erhalten, bereitet der Runner die Build-Umgebung vor. Es beginnt mit dem Festlegen einer Liste vordefinierter Variablen (Umgebungsvariablen) und einer Liste benutzerdefinierter Variablen.

Die Variablen können überschrieben werden und haben in dieser Reihenfolge Vorrang vor einander:

1. Trigger variables or scheduled pipeline variables (take precedence over all)
2. Project-level variables or protected variables
3. Group-level variables or protected variables
4. YAML-defined job-level variables
5. YAML-defined global variables
6. Deployment variables

7. Predefined variables (are the lowest in the chain)

Wenn Sie beispielsweise `API_TOKEN = secure` als Projektvariable und `API_TOKEN = yaml` in Ihrer `.gitlab-ci.yml` definieren, nimmt `API_TOKEN` den Wert sicher an, da sich die Projektvariablen in der Kette befinden.

Es gibt Fälle, in denen einige Variablen nicht im Kontext von `a` verwendet werden können `.gitlab-ci.yml` -Definition (zum Beispiel unter `script`). Weiterlesen welche Variablen nicht unterstützt werden.

- https://git.rwth-aachen.de/help/ci/variables/where_variables_can_be_used.md

Einige der vordefinierten Umgebungsvariablen sind nur verfügbar, wenn eine Mindestversion von GitLab Runner verwendet wird. Konsultieren Sie die nachstehende Tabelle, um das zu finden Version von Runner erforderlich.

gitlab-ci.yml defined variables: https://git.rwth-aachen.de/help/ci/docker/using_docker_images.md

Mit GitLab CI können Sie Variablen `.gitlab-ci.yml` hinzufügen, die in der Build-Umgebung festgelegt sind.

Die *Variablen* werden daher im Repository gespeichert und sollen nicht sensible Projektkonfigurationen speichern, z. B. `RAILS_ENV` oder `DATABASE_URL`.

```
variables:
  DATABASE_URL: "postgres://postgres@postgres/my_database"
```

Die von YAML definierten Variablen werden auch auf alle erstellten Service-Container festgelegt, sodass sie fein abgestimmt werden können. Variablen können auf globaler Ebene, aber auch auf Jobebene definiert werden. Um global definierte Variablen in Ihrem Job zu deaktivieren, definieren Sie einen leeren Hash:

```
job_name:
  variables: {}
```

You are able to use other variables inside your variable definition (or escape them with `$$`):

```
variables:
  LS_CMD: 'ls $FLAGS $TMP_DIR'
  FLAGS: '-al'
script:
  - 'eval $LS_CMD' # will execute 'ls -al $TMP_DIR'
```

Variables: <https://git.rwth-aachen.de/help/user/group/subgroups/index.md>

Beachten Sie, dass Variablen nicht maskiert sind und ihre Werte in den Auftragsprotokollen angezeigt werden können, wenn Sie ausdrücklich dazu aufgefordert werden. Wenn Ihr Projekt öffentlich oder intern ist, können Sie die Pipelines über die Pipelines-Einstellungen Ihres Projekts als privat festlegen. Folgen Sie der Diskussion in Ausgabe # 13784, um die Variablen zu maskieren.

Mit GitLab CI können Sie pro Projekt oder pro Gruppe Variablen definieren die in der Pipeline-Umgebung festgelegt sind.

Die Variablen werden aus dem Repository (nicht in `.gitlab-ci.yml`) gespeichert und sicher an GitLab Runner übergeben, sodass sie während eines Pipeline-Laufs verfügbar sind. Dies ist die empfohlene Methode zum Speichern von Kennwörtern, SSH-Schlüsseln und Anmeldeinformationen.

Protected variables:

Variablen können geschützt werden. Wenn eine Variable geschützt ist, wird sie nur sicher an Pipelines weitergeleitet, die in den geschützten Verzweigungen oder geschützten Tags ausgeführt werden. Die anderen Pipelines erhalten keine geschützten Variablen.

Registriere Docker Runner:

```
sudo gitlab-runner register \
  --url "https://gitlab.example.com/" \
  --registration-token "PROJECT_REGISTRATION_TOKEN" \
  --description "docker-ruby-2.1" \
  --executor "docker" \
  --docker-image ruby:2.1 \
  --docker-postgres latest \
  --docker-mysql latest
```

Images nicht von Docker-Hub:

https://docs.gitlab.com/ce/ci/docker/using_docker_images.html

Never: Deaktiviert das vollständige Ziehen von Bildern. Benutzer kann nur die Images verwenden, die manuell auf dem Docker-Host gezogen wurden, auf dem der Runner ausgeführt wird.

Wird verwendet, wenn vollständige Kontrolle darüber, welche Images von dem Benutzer des Runners verwendet werden. Gute Wahl für private Runnter, die sich auf ein Projekt konzentrieren, in dem nur bestimmte Images verwendet werden. Wird nur verwendet, wenn für den ausgewählten Cloud-Anbieter vordefinierte Cloud-Instanz-Images verwendet werden. Das Images muss die installierte Docker Engine und die lokale Kopie des verwendeten Images enthalten.

If-not-present: Überprüft, ob das Image lokal vorhanden ist. Wenn dies der Fall ist, wird die lokale Version von Image verwendet. Andernfalls versucht der Runner, das Image zu ziehen. Wird verwendet, wenn Images aus entfernten Registern abgerufen werden, die Zeit für die Analyse der Bildebenenunterschiede reduziert möchte, wenn starke und selten aktualisierte Images verwendet werden. In diesem Fall müssen die Images gelegentlich manuell aus dem lokalen Docker Engine-Speicher entfernen, um die Aktualisierung des Images zu erzwingen. Wird auch verwendet, wenn Images die nur lokal erstellt werden und verfügbar sind, andererseits aber auch das Abrufen von Bildern aus entfernten Registern. Sollte nicht verwendet werden, wenn Runner von verschiedenen Benutzer verwendet werden kann, die keinen Zugriff auf private Images haben dürfen, die von anderen verwendet werden.

Always: Sorgt dafür, dass das Images immer gezogen wird. Der Runner versucht das Image abzurufen, selbst wenn eine lokale Kopie verfügbar ist. Wird verwendet, wenn der Runner öffentlich verfügbar ist und als freigegebener Runner in ihrer GitLab-Instanz konfiguriert. Dies ist die einzige Richtlinie, die sich als betrachtet werden kann, wenn der Runner mit privaten Images verwendet wird. Funktioniert nicht, wenn lokal gespeicherte Images verwendet werden müssen. In diesem Fall überspringt der Runner die lokale Kopie des Images und versucht, sie aus der Remote-Registrierung zu ziehen. Wenn das Image lokal erstellt wurde und in keiner öffentlichen Registry vorhanden ist, schlägt der Builder alarm.

Docker executor:

Service:

Definiert ein anderes Docker-Image, das während des Builds ausgeführt wird und mit dem Docker-Image verknüpft, dass das Image-Schlüsselwort definiert. Zusammenfassend lässt sich sagen, dass wenn Sie mysql als Dienst zu Ihrer Anwendung hinzufügen, dieses Image verwendet werden, um einen Container zu erstellen, der mit dem Build-Container verknüpft ist. Je nach Workflow ist dies der erste Schritt, der ausgeführt wird, bevor die eigentlichen Builds ausgeführt werden.

<https://docs.gitlab.com/runner/executors/docker.html#define-image-and-services-in-configtoml>

config.toml: <https://github.com/toml-lang/toml#user-content-example>

<https://git.rwth-aachen.de/help/ci/environments>

<https://gitlab.com/gitlab-examples/review-apps-nginx/tree/master>

Introduction to Continuous Integration with GitLab CI

<https://git.rwth-aachen.de/swc-public/teaching/introduction-to-ci-with-gitlab-ci>