



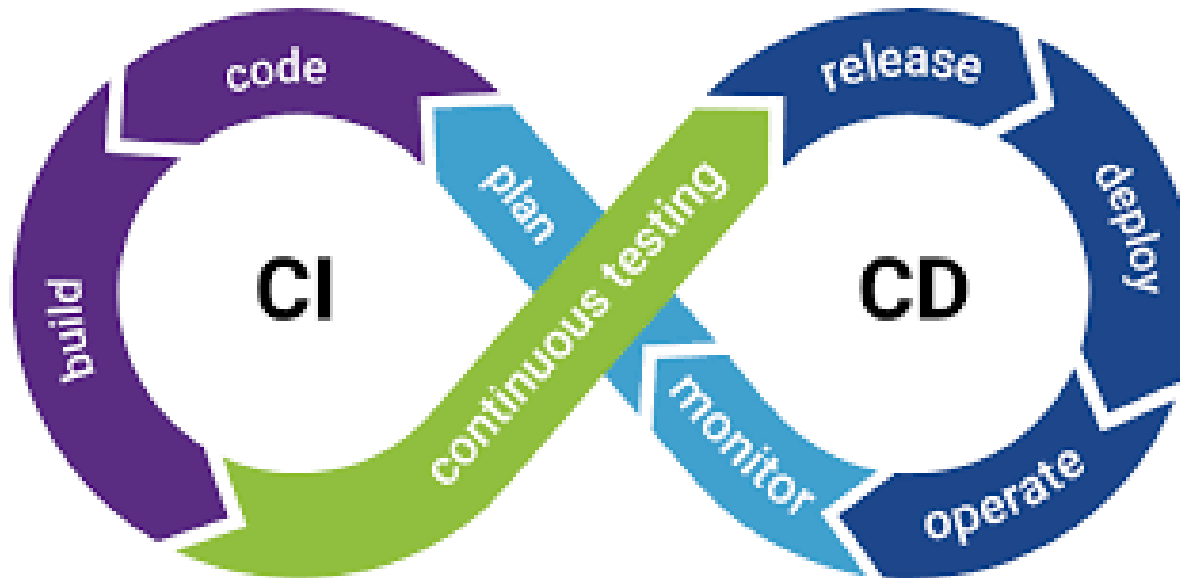
# Tag 2: Vertiefung Git-Workflow, CI/CD & GitLab CI

18.06.2024, Daniel Krämer & Malte Fischer

© Copyright 2024 anderScore GmbH

- **Tag 1 – Einführung in Git und GitLab, Git-Workflow im Team**
  - Einführung & Kursüberblick
  - Grundlagen von Git
  - Git Rebase und Merge-Strategien
  - Git Remote
  - Grundlagen von GitLab
  - Git-Workflow im Team
- **Tag 2 – Vertiefung Git-Workflow, CI/CD & GitLab CI**
  - Gitflow-Workflow
  - Tags, Releases & deren Verwaltung
  - GitLab-Runner
  - Einführung in GitLab CI/CD & gitlab.yml
- **Tag 3 – GitOps, Docker in der Entwicklung und Deployment-Strategien**
  - GitOps Grundlagen
  - Lokale Entwicklung mit Docker
  - Container/Docker-Registry
  - Erstellen von Release- und Tagged-Images
  - Möglichkeiten des Deployments & Verwaltung von Konfiguration
  - Abschlussübung & Diskussion

# Grundlagen von **CI/CD**



<https://encrypted-tbn0.gstatic.com/images?q=tbn:AND9GcRAABUoTg0hRIRysVXsNZg21ojLCOSsljUEIA&s>

## Continuous Integration (CI)

- Definition:
  - Regelmäßiges Zusammenführen von Codeänderungen in das Haupt-Repository
- Schlüsselprinzipien:
  - **Häufige Commits:** Regelmäßiges integrieren kleiner Änderungen
  - **Automatisierte Builds:** Jeder Commit triggert einen Build
  - **Automatisierte Tests:** Jeder Build wird getestet
  - **Feedback:** Schnelles Feedback für Entwickler bei Fehlern
- Vorteile:
  - Früherkennung von Fehlern
  - Verbesserte Zusammenarbeit und Codequalität

## Continuous Delivery (CD)

- Definition:
  - Sicherstellen, dass der Code jederzeit bereit für den Release ist
- Schlüsselprinzipien:
  - **Automatisierte Tests:** Umfassende Tests zur Sicherstellung der Codequalität
  - **Release Management:** Vorbereitung auf häufige Releases
  - **Deployments:** Manuelle oder automatisierte Bereitstellung in Staging-Umgebungen
- Vorteile:
  - Schnelle Bereitstellung neuer Features
  - Reduzierung von Risiken und Fehlern bei Releases

## Continuous Deployment

- Definition:
  - Vollständig automatisierte Bereitstellung in die Produktion
- Schlüsselprinzipien:
  - **Automatisierte Deployment-Pipeline:** Kein manueller Eingriff notwendig
  - **Monitoring:** Kontinuierliche Überwachung und schnelle Reaktion auf Probleme
  - **Rollback-Strategie:** Mechanismen zur schnellen Rücknahme fehlerhafter Deployments
- Vorteile:
  - Extrem schnelle Veröffentlichung von Änderungen
  - Sofortige Reaktion auf Marktanforderungen und Benutzerfeedback

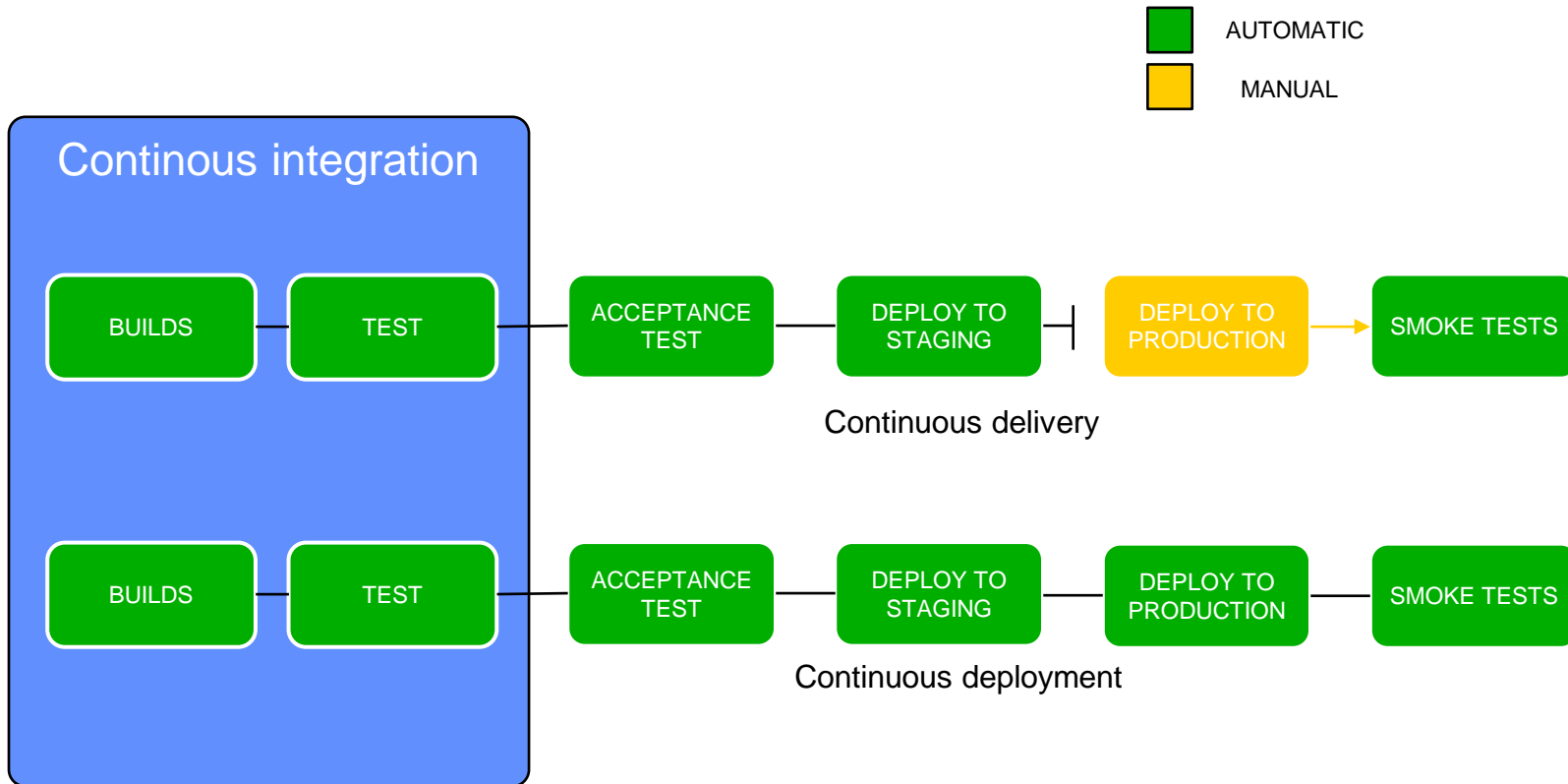
## Vorteile von CI/CD

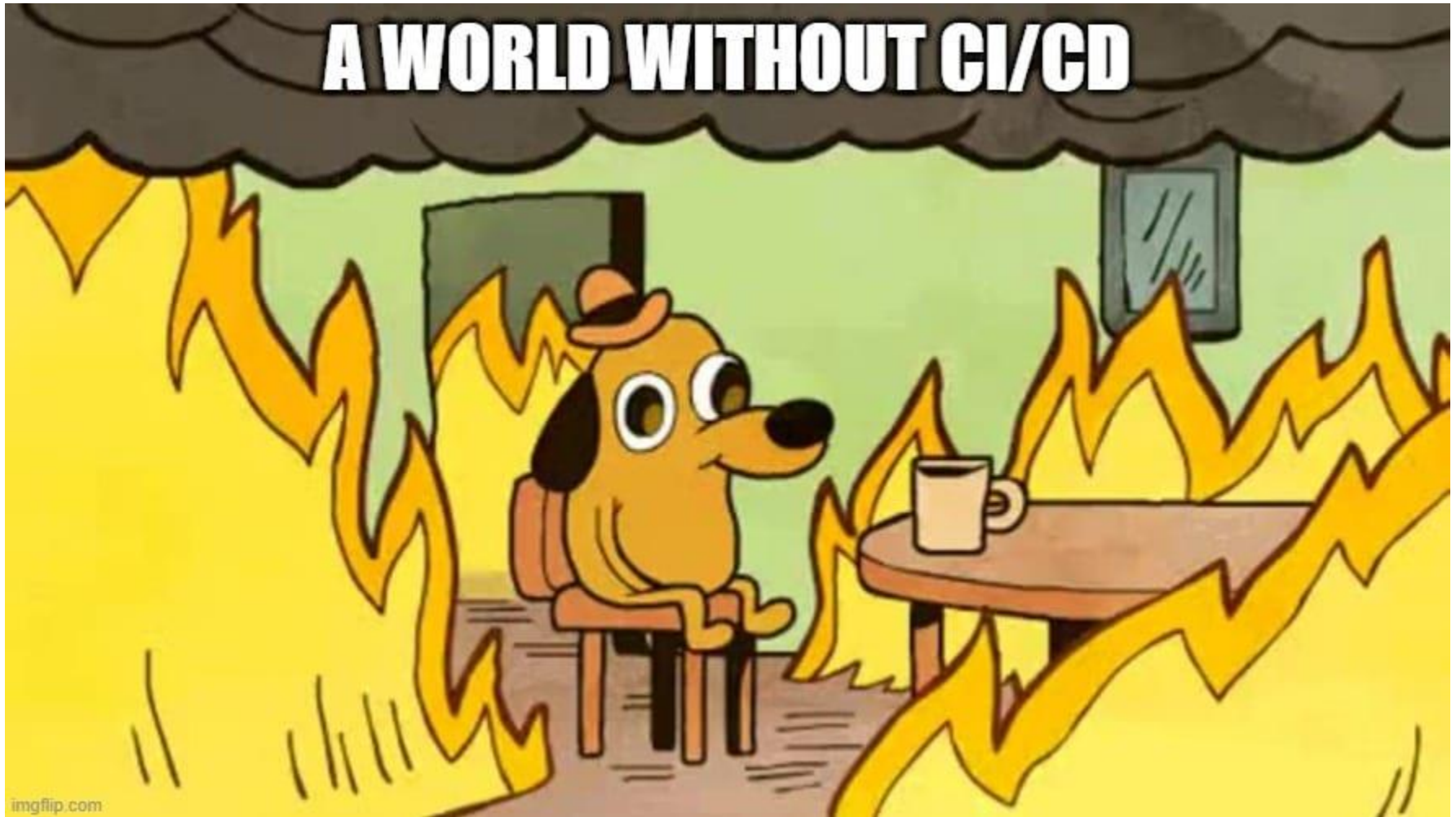
- Schnellere Lieferung:
  - Schnellere Bereitstellung von Updates und Features
- Höhere Qualität:
  - Regelmäßige Tests und Builds verbessern die Codequalität
- Bessere Zusammenarbeit:
  - Förderung von Teamarbeit und kontinuierlichem Feedback
- Reduzierte Risiken:
  - Früherkennung und Behebung von Fehlern minimiert Produktionsrisiken



## Nachteile von CI/CD

- Komplexität der Einrichtung:
  - Hoher Aufwand für CI/CD-Implementierung.
- Kulturelle Anpassungen:
  - Erfordert Veränderung der Team-Arbeitsweise.
- Abhängigkeit von Automatisierung:
  - Starkes Vertrauen auf Automatisierung kann problematisch sein, wenn die automatisierten Prozesse fehlschlagen oder Fehler enthalten
- Kosten:
  - Zusätzliche Kosten für Tools und Schulungen.

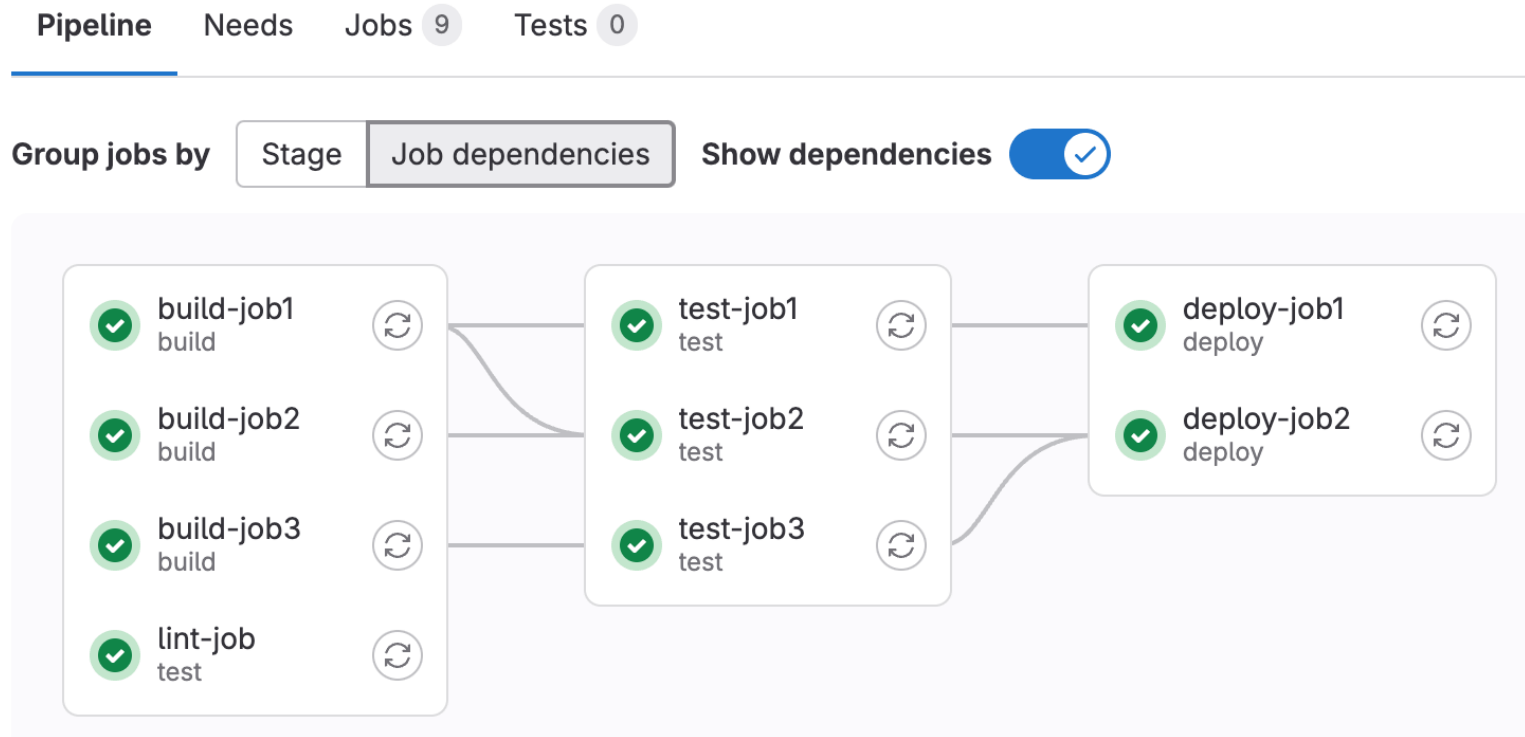




<https://b1286009.smushcdn.com/1286009/wp-content/uploads/2020/04/a-world-without-ci.cd-meme.jpg?lossy=1&strip=1&webp=1>

## Gitlab Stages und Jobs

- Die Stages werden oben in der .gitlab-ci.yml definiert
- Eine Stage kann mehrere Jobs haben
- Die Jobs einer Stage können Parallel laufen



## Gitlab Stages und Jobs

```
stages:
```

- build

```
frontend:
```

```
  stage: build
```

```
  script:
```

- echo "Building the frontend..."

```
backend:
```

```
  stage: build
```

```
  script:
```

- echo "Building the backend..."

## Aufgabe 1: Einführung in GitLab CI/CD

**1. Ziel:** Verstehe die Grundlagen von GitLab CI/CD.

**2. Schritte:**

- Erstelle ein neues GitLab-Repository.
- Füge eine `.gitlab-ci.yml`-Datei im Stammverzeichnis des Projekts hinzu.
- Schreibe eine einfache Konfiguration, die einen Job namens `hello_world` definiert, der "Hello, World!" ausgibt.

## Aufgabe 1: Einführung in GitLab CI/CD

- Mögliche Lösung

```
stages:  
  - build
```

```
hello_world:  
  stage: build  
  script:  
    - echo "Hello, World!"
```

## Aufgabe 2: Verwendung von Stages

**1. Ziel:** Verstehe, wie Stages in GitLab CI funktionieren und wie sie zur Strukturierung von Jobs verwendet werden.

**2. Schritte:**

- Erweitere die `.gitlab-ci.yml`, um zwei Stages (build und test) zu definieren.
- Füge einen Job in der build-Stage hinzu, der eine Dummy-Datei erstellt.
- Füge einen Job in der test-Stage hinzu, der diese Datei überprüft.



## Aufgabe 2: Verwendung von Stages

- Mögliche Lösung

```
stages:  
  - build  
  - test  
  
build_job:  
  stage: build  
  script:  
    - echo "Building the project..."  
    - touch dummy_file.txt  
  
test_job:  
  stage: test  
  script:  
    - echo "Testing the project..."  
    - ls -l dummy_file.txt
```

## Artifacts

- Dateien aus CI/CD-Pipelines (Build-Ergebnisse, Testberichte, Logs)
- Dateien bleiben für eine gewisse Zeit verfügbar
- Zugriff
  - Download über Gitlab-Webinterface
  - Weitergabe zwischen Jobs
- Arten von Artifacts:
  - Standard-Artifacts: Allgemeine Dateien
  - Reports: Test-, Sicherheits-, Qualitätsberichte
  - Cache: Temporäre Dateien

## Artifacts

```
stages:
```

- build

```
pdf:
```

```
  stage: build
```

```
  script: xelatex mycv.tex
```

```
  artifacts:
```

```
    paths:
```

- mycv.pdf

## Aufgabe 3: Verwendung von Artifacts

1. **Ziel:** Verstehe, wie man Artifacts verwendet, um Dateien zwischen Jobs und Stages zu teilen.
2. **Schritte:**
  - Modifiziere den build\_job, um die dummy\_file.txt als Artifact zu speichern.
  - Ändere den test\_job, um dieses Artifact herunterzuladen und zu verwenden.

## Aufgabe 3: Verwendung von Artifacts

- Mögliche Lösung

```
stages:
  - build
  - test

build_job:
  stage: build
  script:
    - echo "Building the project..."
    - touch dummy_file.txt
  artifacts:
    paths:
      - dummy_file.txt

test_job:
  stage: test
  script:
    - echo "Testing the project..."
    - ls -l dummy_file.txt
```

## Aufgabe 4: Erweiterung mit einem Deploy-Job

1. **Ziel:** Lerne, wie man einen Deployment-Job hinzufügt und Artifacts verwendet, um Build-Artefakte zu deployen.
2. **Schritte:**
  - Füge eine deploy-Stage hinzu.
  - Erstelle einen `deploy_job`, der das Artifact herunterlädt und einen simulierten Deployment-Prozess ausführt.

## Aufgabe 4: Erweiterung mit einem Deploy-Job

- Mögliche Lösung

```
stages:
  - build
  - test
  - deploy

build_job:
  stage: build
  script:
    - echo "Building the project..."
    - touch dummy_file.txt
  artifacts:
    paths:
      - dummy_file.txt

test_job:
  stage: test
  script:
    - echo "Testing the project..."
    - ls -l dummy_file.txt

deploy_job:
  stage: deploy
  script:
    - echo "Deploying the project..."
    - ls -l dummy_file.txt
```

## Variables

- Umgebungsvariablen für CI/CD-Pipelines
- Arten von Variablen:
  - CI/CD-Variablen: In `.gitlab-ci.yml` definiert
  - Projekt-Variablen: Im Projekt unter Einstellungen -> CI/CD
  - Gruppen-Variablen: Auf Gruppenebene definiert
  - Benutzerdefinierte Variablen: Vom Benutzer erstellt
  - Vordefinierte Variablen: Von GitLab bereitgestellt (z.B. `CI_COMMIT_SHA`)
- Sicherheitsaspekte
  - Geschützte Variablen: Nur für geschützte Branches/Tags
  - Vertrauliche Variablen: Verstecken den Wert im Job-Log



## Variables

```
variables:
  GLOBAL_VAR: "A global variable"

job1:
  variables:
    JOB_VAR: "A job variable"
  script:
    - echo "Variables are '$GLOBAL_VAR' and '$JOB_VAR'"

job2:
  script:
    - echo "Variables are '$GLOBAL_VAR' and '$JOB_VAR'"
```

## Aufgabe 5: Einführung von Variablen

1. **Ziel:** Lerne, wie man Variablen in GitLab CI/CD Pipelines verwenden können, um ihren Entwicklungsprozess flexibler zu gestalten und die Wartbarkeit ihres Codes zu verbessern.
2. **Schritte:**
  - Füge eine Variable DUMMY\_FILE hinzu
  - Ersetze alle Verweise auf "dummy\_file.txt" mit der Variable

## Aufgabe 5: Einführung von Variablen

- Mögliche Lösung

```
variables:  
  DUMMY_FILE: "dummy_file.txt"  
  
stages:  
  - build  
  - test  
  - deploy  
  
build_job:  
  stage: build  
  script:  
    - echo "Building the project..."  
    - touch $DUMMY_FILE  
artifacts:  
  paths:  
    - $DUMMY_FILE  
  
test_job:  
  stage: test  
  script:  
    - echo "Testing the project..."  
    - ls -l $DUMMY_FILE  
  
deploy_job:  
  stage: deploy  
  script:  
    - echo "Deploying the project..."  
    - ls -l $DUMMY_FILE
```

## Rules

- Regeln zur Steuerung der Ausführung von Jobs in CI/CD-Pipelines
- Verwendung:
  - Konfiguration in `.gitlab-ci.yml`
  - Ersetzt **only** und **except**
- Wichtige Schlüsselwörter:
  - **if**: Bedingungen basierend auf Variablen oder Pipeline-Status
  - **changes**: Bedingungen basierend auf Dateiänderungen
  - **exists**: Bedingungen basierend auf dem Vorhandensein von Dateien
  - **when**: Bestimmt, wann ein Job ausgeführt wird (`on_success`, `on_failure`, `always`, `manual`, `delayed`)

## Rules

stages:

- deploy

deploy\_prod:

stage: deploy

script:

- echo "Deploy to production server"

when: manual

rules:

- if: \$CI\_COMMIT\_BRANCH == \$CI\_DEFAULT\_BRANCH

## Aufgabe 6: Bedingte Ausführung von Jobs

1. **Ziel:** Verstehe, wie man Jobs bedingt ausführt, basierend auf bestimmten Bedingungen wie Branches oder Tags.
2. **Schritte:**
  - Modifiziere den `deploy_job`, um ihn nur auf dem main-Branch auszuführen.

## Aufgabe 6: Bedingte Ausführung von Jobs

- Mögliche Lösung

```
variables:
  DUMMY_FILE: "dummy_file.txt"

stages:
- build
- test
- deploy

build_job:
  stage: build
  script:
    - echo "Building the project..."
    - touch $DUMMY_FILE
  artifacts:
    paths:
      - $DUMMY_FILE

test_job:
  stage: test
  script:
    - echo "Testing the project..."
    - ls -l $DUMMY_FILE

deploy_job:
  stage: deploy
  script:
    - echo "Deploying the project..."
    - ls -l $DUMMY_FILE
  only:
    - main
```

## Aufgabe 7: Parallelisierung von Jobs

1. **Ziel:** Verstehe, wie man Jobs parallelisiert, um die CI/CD-Pipeline zu beschleunigen.
2. **Schritte:**
  - Füge mehrere Jobs in der test-Stage hinzu, die verschiedene Tests parallel ausführen.



# Aufgabe 7: Parallelisierung von Jobs

- Mögliche Lösung

```
variables:
  DUMMY_FILE: "dummy_file.txt"

stages:
- build
- test
- deploy

build_job:
  stage: build
  script:
    - echo "Building the project..."
    - touch $DUMMY_FILE
  artifacts:
    paths:
      - $DUMMY_FILE

test_job_1:
  stage: test
  script:
    - echo "Running test 1..."
    - ls -l $DUMMY_FILE

test_job_2:
  stage: test
  script:
    - echo "Running test 2..."
    - ls -l $DUMMY_FILE

deploy_job:
  stage: deploy
  script:
    - echo "Deploying the project..."
    - ls -l $DUMMY_FILE
  only:
    - main
```