



# CI/CD

*Niklas Meyer*

---

**Was ist CI/CD?**

# Was ist CI/CD?

- **Continuous Integration/Continuous Delivery**
- **Ziel:** Häufiger und sicherer Code bereitzustellen (fließende Einbindung in Entwicklung und Betrieb)
- **Mittel:** Automatisierung
  - Vorher: alles manuell:
    - Bauen – Testen – Zusammenführen – Bauen – Testen – Analysen – Quality Gates – Deployen
  - Nun: Automatisieren in einer Pipeline/einem Arbeitsablauf
    - Build-, Test- und Deploymentprozesse

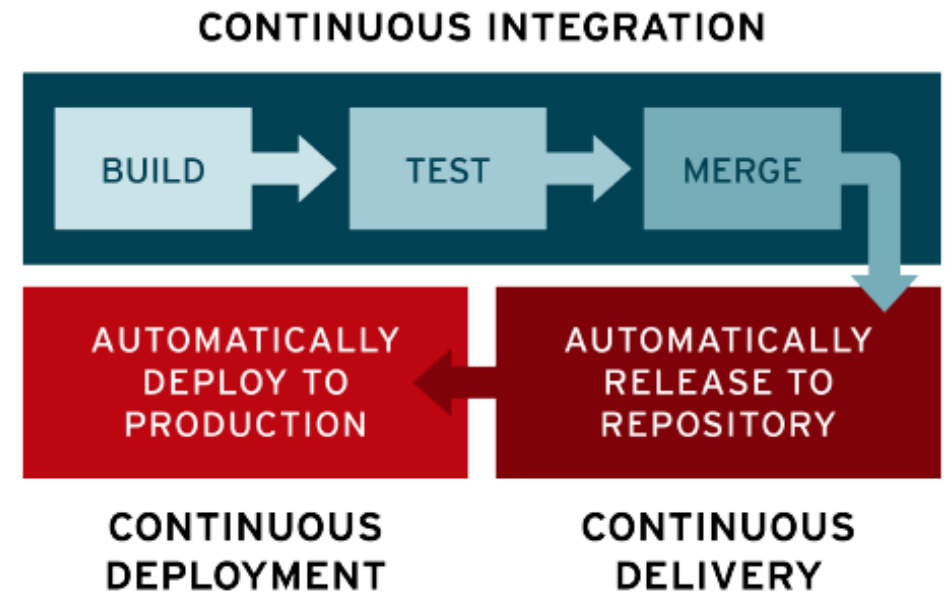
---

# Lernziele

- Den Zweck und Ablauf von CI/CD erklären können
- Wissen wie und mit welchen Tools dies realisiert werden kann
- Vor- und Nachteile von CI/CD kennen
- Gute Einsteigerkenntnisse in GitHub Actions gewinnen
- Einfache GitHub-Actions-Workflows schreiben und verstehen können
- Automatisierte Tests in Workflows integrieren können
- Mit Fehlern in Workflows umgehen können (Debugging, Logs lesen)
- Spezifischere Features einbauen können (Artefakte archivieren, Variablen, Parallelisierung, ...)

# Was unterscheidet CI von CD?

- **Continuous Integration:**
  - stetig Code zusammenführen (und Testen) (jede Code-Änderung)
- **Continuous Delivery:**
  - stetige Bereitstellung des Codes, sodass man ihn jederzeit deployen könnte (in unterschiedlichen Umgebungen z.B. Prod)
- **Continuous Deployment:**
  - stetiges Veröffentlichen dieser neuen Versionen



<https://www.plusserver.com/blog/was-bedeutet-ci-cd-in-der-entwicklung/>

---

# Wie realisiert man das?

- Benutzung von CI/CD Automatisierungstools (z.B. GitHub Actions, GitLab CI/CD, Jenkins, ...)
- Beschreibung, welche Schritte alles ausgeführt werden müssen mit einem tool-spezifischen Schema und Befehlen je nachdem welche Build-, Test- u.ä. Tools man verwendet
- Bei Ausführung werden diese Schritte dann der Reihe nach ausgeführt
- Mehr Magie ist es nicht

---

# Was kann dadurch automatisiert werden?

- Eigentlich fast alles
  - es werden Umgebungen eingerichtet, bei denen man die üblichen Befehle, die zuvor manuell eingegeben wurden nun dort ausführen lassen kann
  - **Bauen**: z.B. mvn clean install (Java + Maven)
  - **Testen** (Unit-Tests, Integration, E2E): z.B. mvn test (Java + Maven)
  - **Statische Analyse** (+ deren Ergebnisse abwarten)
  - **Quality Gates**
  - **Ausliefern der Anwendung**

---

# **Vor- und Nachteile von CI/CD**



# Vor- und Nachteile von CI/CD

Vorteile	Nachteile
schnellere Release-Zyklen, da Code nicht so weit auseinander laufen kann und man Sicherheit gewinnt	automatisches Deployment birgt Risiken => gut über Pipelineschritte (z.B. Test) abzusichern, aber trotzdem meist sicherer als manuell
Automatisierung spart Zeit	Initialer Einrichtungsaufwand
Frühzeitige Fehlererkennung	Knowhow muss aufgebaut und in stand gehalten werden
Verbesserte Qualität durch automatische Testausführung	ohne automatisierte Tests, fallen die korrespondierenden Vorteile weg z.B. verbesserte Qualität oder Sicherheit
Kostensenkung, durch eingesparte Zeit und Ressourcen	Kosten für Infrastruktur

---

# **Vor- und Nachteile von CI/CD**

CI/CD ist kein Selbstzweck, sondern eine Methode, Risiken zu reduzieren und Entwicklung effizienter zu gestalten – aber nur, wenn sie richtig implementiert wird.

---

# CI/CD-Tools

---

# CI/CD-Tools (Auswahl)



# Jenkins



GitHub Actions



# CI/CD-Tools - Unterschiede

GitHub Actions	GitLab CI/CD	Jenkins
Cloud Hosting (GitHub) oder selbst	Cloud Hosting (GitLab) oder selbst	Muss selbst gehostet werden
Keine eigene Infrastruktur nötig	Keine eigene Infrastruktur nötig	Eigene Infrastruktur nötig
Kostenlos bis Runner-Zeit-Limit (2000 min pro Monat   unbegrenzt in public Repo -> Open Source Support)	Kostenlos bis Runner-Zeit-Limit (400 min pro Monat)	Kosten für Infrastruktur und Wartung

Einfacher Einstieg

Komplexer, aber flexibler und erweiterbarer

# CI/CD-Tools - Syntax

- Trotz genannter Unterschiede, meist ähnlicher als man denkt:

```
1  name: CI
2
3  on:
4    workflow_dispatch:
5
6  jobs:
7    build:
8      runs-on: ubuntu-latest
9
10     steps:
11       - name: Building
12         run: echo Building...
13
14    test:
15      runs-on: ubuntu-latest
16
17     steps:
18       - name: Testing
19         run: echo Testing...
20
21    deploy:
22      runs-on: ubuntu-latest
23
24     steps:
25       - name: Deploying
26         run: echo Deploying...
```

```
1  pipeline {
2    agent any
3
4    stages {
5      stage('Build') {
6        steps {
7          echo 'Building...'
8        }
9      }
10     stage('Test') {
11       steps {
12         echo 'Testing...'
13       }
14     }
15     stage('Deploy') {
16       steps {
17         echo 'Deploying...'
18       }
19     }
20   }
21 }
```

# CI/CD-Tools – GitHub Actions

- Gute Einsteigerfreundlichkeit
- Hohe Relevanz im CI/CD-Bereich
- Ähnlich zu anderen weit verbreiteten Tools
- Einfache Kontrolle von Übungsaufgaben

```
1  name: CI
2
3  on:
4    workflow_dispatch:
5
6  jobs:
7    build:
8      runs-on: ubuntu-latest
9
10     steps:
11       - name: Building
12         run: echo Building...
13
14     test:
15       runs-on: ubuntu-latest
16
17       steps:
18         - name: Testing
19           run: echo Testing...
20
21     deploy:
22       runs-on: ubuntu-latest
23
24       steps:
25         - name: Deploying
26           run: echo Deploying...
```

# CI/CD-Tools – GitHub Actions

- **YAML-Syntax:**
  - Einrückungen sind wichtig und konsequent einzuhalten
  - Listenelemente beginnen mit "-"
  - Kommentare mit "#"
  - ...

```
1  name: CI
2
3  on:
4    workflow_dispatch:
5
6  jobs:
7    build:
8      runs-on: ubuntu-latest
9
10     steps:
11       - name: Building
12         run: echo Building...
13
14     test:
15       runs-on: ubuntu-latest
16
17     steps:
18       - name: Testing
19         run: echo Testing...
20
21     deploy:
22       runs-on: ubuntu-latest
23
24     steps:
25       - name: Deploying
26         run: echo Deploying...
```



# CI/CD-Tools – GitHub Actions

- **Workflow:**

- Größte Einheit in Actions
- Ablauf von mehreren Jobs wie Bauen und Testen einer Anwendung
- Jede YAML-Datei in ".github/workflows/" wird versucht als Workflow interpretiert zu werden

- **Jobs:**

- Teile eines Workflows (meist für je ein Thema wie Bauen, Testen, Linter, Deployen, ...)
- Ablauf von mehreren Schritten
- Definiert über "jobs:"

- **Steps/Schritte:**

- Kleinste Einheit in Actions
- Ablauf von mehreren Befehlen wie "echo Hello World!"
- Definiert über "steps:"

```
1  name: CI
2
3  on:
4    workflow_dispatch:
5
6  jobs:
7    build:
8      runs-on: ubuntu-latest
9
10     steps:
11       - name: Building
12         run: echo Building...
13
14     test:
15       runs-on: ubuntu-latest
16
17       steps:
18         - name: Testing
19           run: echo Testing...
20
21     deploy:
22       runs-on: ubuntu-latest
23
24       steps:
25         - name: Deploying
26           run: echo Deploying...
```

# CI/CD-Tools – GitHub Actions

- **"name:"** - Name eines Workflows, Jobs oder Steps festlegen
- **"on:"** - Trigger, also wann soll der Workflow ausgeführt werden
  - **"workflow\_dispatch"** - Manueller Start
  - **"push"** - Bei einem Push (man kann auch bestimmte Branches angeben über "branches:")
  - **"pull\_request"** - Bei Erstellung oder Aktualisierung eines Pull-Requests
  - **"schedule"** - Zeitlich gesteuerter Start
- **"runs-on:"** - Auf welcher Umgebung ein Job laufen soll
  - Bei uns einfachheitshalber immer **"ubuntu-latest"**
- **"run:"** - Führt einzelne Befehle aus

```
1  name: CI
2
3  on:
4    workflow_dispatch:
5
6  jobs:
7    build:
8      runs-on: ubuntu-latest
9
10     steps:
11       - name: Building
12         run: echo Building...
13
14     test:
15       runs-on: ubuntu-latest
16
17     steps:
18       - name: Testing
19         run: echo Testing...
19
20     deploy:
21       runs-on: ubuntu-latest
22
23     steps:
24       - name: Deploying
25         run: echo Deploying...
```

---

**Findet die Fehler**



```
1  name=Fehlersuche
2
3  on:
4    manually:
5
6  workflow:
7    jobs:
8      build:
9        run: ubuntu-latest
10       steps:
11         echo Building
12         echo Done
```



```
1  name: Fehlersuche
2
3  on:
4    workflow_dispatch:
5
6  jobs:
7    build:
8      runs-on: ubuntu-latest
9      steps:
10       - run: echo Building
11       - run: echo Done
```

---

# **Live-Demo zur GitHub-UI**



```
1  name: Test
2
3  on:
4    push:
5      branches:
6        - main
7    workflow_dispatch:
8
9  jobs:
10   example:
11     name: Example Job
12     runs-on: ubuntu-latest
13
14     steps:
15       - name: Hallo
16         run: echo "Hello World"
```

---

# Übung 1: "Hello World!"-Workflow

## Ziel:

- Ersten einfachen Workflow erstellen und ausführen

## Aufgabe:

- Legt im Ordner ".github/workflows/" die Datei "hello\_world.yml" an
- Schreibt dort einen ersten Workflow mit dem Namen "Hello World"
  - Der Workflow soll von uns manuell gestartet werden können
  - Als Job soll dieser "say-hello" haben, welcher auf "ubuntu-latest" laufen soll
    - Der Job soll nur einen Step mit Namen "Say Hello" haben und den Befehl "echo Hello World!" ausführen






```
1  name: Hello World
2
3  on:
4    workflow_dispatch:
5
6  jobs:
7    say-hello:
8      runs-on: ubuntu-latest
9      steps:
10         - name: Say Hello
11           run: echo Hello World!
```

# GitHub Actions – Bauen und Testen

- Es existiert ein Marketplace mit vielen vorgefertigten Workflows, die man als Step ausführen lassen kann
  - Hier holt sich die Umgebung, auf der der Workflow läuft den Code vom Repository
- Manche Workflows kann oder muss man dann noch mit Parametern bestücken
  - Hier richten wir die in den "with:"-Parametern angegebene JDK in der Umgebung ein



```
1  steps:
2      - name: Checkout repository
3        uses: actions/checkout@v4
```



```
1  steps:
2      - name: Set up JDK 21
3        uses: actions/setup-java@v4
4        with:
5          distribution: 'temurin'
6          java-version: '21'
```

---

# **Live-Demo zum Actions-Marketplace**

---


# Übung 2: JUnit-Workflow

## Ziel:

- Erstellen eines Workflows, welcher die im Projekt befindlichen JUnit Tests ausführt (+ eventuell fehlerhafte Tests berichtigen – Hinweise in Logs)

## Aufgabe:


- Legt einen neuen Workflow "Java Workflow" an
- Der Workflow soll manuell und durch einen Push auf "main" gestartet werden können
  - Erstellt dann den Job "build-test" mit folgenden Steps
    - "Checkout repository"
    - "Set up JDK 21"
    - "Build and Test with Maven"



```
1  name: JUnit Workflow
2
3  on:
4    push:
5      branches:
6        - main
7    workflow_dispatch:
8
9  jobs:
10   build-test:
11     name: Build and Test with Maven
12     runs-on: ubuntu-latest
13
14     steps:
15       - name: Checkout repository
16         uses: actions/checkout@v4
17
18       - name: Set up JDK 21
19         uses: actions/setup-java@v4
20         with:
21           distribution: 'temurin'
22           java-version: '21'
23
24       - name: Build and Test with Maven
25         run: mvn -B clean verify
```

# GitHub Actions – Autonomie von Jobs und Artefakte

- Jobs sind autonom (neuer Job = neue Umgebung)
  - Eventuell müssen Steps zum Einrichten der Umgebung wiederholt werden z.B. "Checkout Repository"
- Man kann allerdings Artefakte hochladen und diese in anderen Jobs wieder herunterladen (für Zwischenergebnis-Sicherung oder auch um sowas wie Test-Reports zum Herunterladen anzubieten)



```
1  steps:
2    - name: Archive test results
3      if: always()
4      uses: actions/upload-artifact@v4
5      with:
6        name: junit-results
7        path: target/surefire-reports/
```

---

# Übung 3: JUnit Ergebnisse hochladen

## **Ziel:**

- Erweitern des JUnit-Workflows, sodass die Test-Ergebnisse hochgeladen werden

## **Aufgabe:**

- Ihr könnt entweder den Java Workflow erweitern oder diesen duplizieren und anpassen
- Erweitert den Workflow um einen Step, der die Test-Ergebnisse, die durch das Testen über Maven unter dem Pfad "target/surefire-reports/" generiert werden, als Artefakte mit Namen "junit-results" hochlädt
- Sichten des Artefakts im Reiter Actions

---

## **Einschub: Linter**



---

# Einschub: Linter

- Analysiert Quellcode auf Stilverstöße, kann aber auch Syntaxfehler und z.B. ungenutzte Variablen erkennen
- Lint-Regeln können weiter angepasst werden
- Es gibt auch Tools, die nicht linten, sondern gleich die Stilverstöße behebt
- Kann natürlich auch in CI/CD eingebaut werden

---

# Übung 4: Linter einbauen

## **Ziel:**

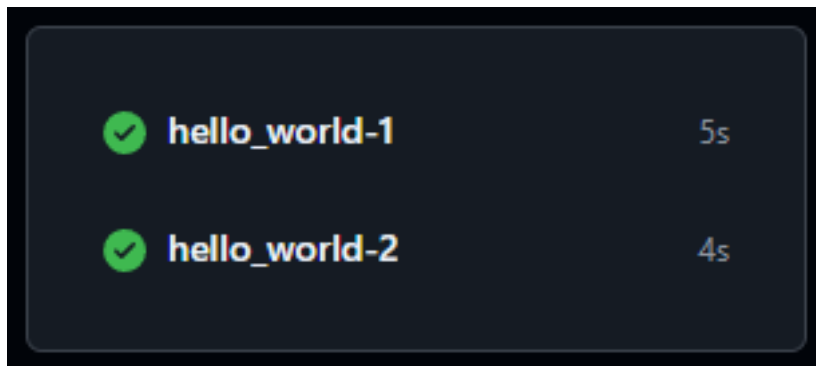
- Erweitern des Java Workflows um einen Linter, damit wir Formatierungsprobleme erkennen können

## **Aufgabe:**

- Ihr könnt wieder entweder den Java Workflow erweitern oder diesen duplizieren und anpassen
- Erweitert den Workflow um einen Job für den Linter
  - Dieser Job soll den Befehl "mvn checkstyle:check" ausführen und die Ergebnisse unter "target/checkstyle-result.xml" hochladen (egal, ob der Workflow erfolgreich war oder nicht)


# GitHub Actions – Parallelisierung

- Generell laufen Jobs parallel zueinander, also gleichzeitig
  - Kann vorteilhaft sein, aber auch Nachteile mit sich bringen
  - Wenn man einen Job zwingend erst nach einem Anderen starten will muss man mit "needs:" den vorausgehenden Job angeben wie z.B. "build-test"




# GitHub Actions - Variablen

- Es gibt Szenarien, bei denen man z.B. je nach Branch, Umgebung oder etwas anderem andere Werte benötigt
  - Hierfür kann man Variablen anlegen und deren Werte abfragen
- Strategy-Matrix - Wenn man unterschiedliche Job-Varianten ausführen lassen will
  - Job wird dann für jeden angegebenen Wert einmal ausgeführt
  - Mit "\${{ matrix.greeting }}" kann man die Werte dann benutzen
- Workflow-Variablen – Wenn man Werte in einem Workflow bereitstellen will z.B. zum Nutzen in mehreren Jobs
  - Mit "\$Greeting" kann man den Wert nutzen
- Übergreifende Variablen/Secrets - Wenn man Werte für das ganze Repo oder für Umgebungen bereitstellen will
  - Im Settings-Reiter des Repos unter Security – Secrets and Variables – Actions
  - Mit "\${{ secrets.GREETING }}" kann man die Werte dann benutzen



```
1 strategy:
2   matrix:
3     greeting: [Hallo, Servus, Hi]
```



```
1 env:
2   Greeting: Servus
```

# Übung 5: Parallelisierung und Variablen

## Ziel:

- Mit parallelen und nicht parallelen Jobs und Variablen arbeiten

## Aufgabe:

- Ändert den Befehl "mvn checkstyle:check" zu "mvn checkstyle:checkstyle"
- Baut ein, dass statt der JDK 21 auch jeweils einmal die Version 11 und 17 verwendet wurde (Variable)
- Sorgt außerdem dafür, dass es einen weiteren Job gibt, der etwas mit Variablen in die Konsole schreiben soll
  - Schema der Ausgabe: <Grußformel> <Begrüßter>, das Passwort für <Passwortnutzen> ist <Passwort>!
  - Es sollen auf folgenden Ebenen Variablen, für den angegebenen Teil der Konsolenausgabe angelegt und benutzt werden:  
Root-Ebene - Grußformel | Job-Ebene - Begrüßter | Step-Ebene – Passwortnutzen | Repository-Ebene – Passwort
  - Für die <Passwort>-Variable müsst ihr in eurem Fork in Settings ein Action Secret anlegen
- Außerdem sollen die bisherigen Jobs auf die erfolgreiche Ausführung des Konsolen-Jobs warten

# Übung 6: Environments

## **Ziel:**

- Mit unterschiedlichen Environments auf unterschiedliche Server und URLs deployen (simuliert)

## **Aufgabe:**

- Sorgt wieder dafür, dass nur die JDK 21 benutzt wird
- Erstellt in den Settings 3 Environments für DEV, QA und PROD
- Erstellt einen neuen Job "deploy", welcher nach allen anderen laufen soll
  - Nun soll der Job nur eine Ausgabe tätigen mit der URL, auf die er deployen würde
  - Die URL unterscheidet sich je nach Environment
  - Die Stage, auf die deployed werden sollte, soll man beim manuellen Starten des Workflows angeben müssen
  - Schaut bestenfalls, dass Jobs oder Steps geskipped werden, wenn das sinnvoll ist

---

# Übung 7: SonarCloud

## **Ziel:**

- Mit SonarCloud ein externes Tool in Workflow einbauen

## **Aufgabe:**

- Meldet euch bei SonarCloud mit eurem GitHub Account an
- Geht die Einrichtung durch, sodass euer Repository als Projekt eingerichtet ist
- In Administration und Analysis Method müsst ihr Automatic Analysis deaktivieren
- Erstellt dann in euren Account Settings unter Security einen Token
- Versucht nun einen Sonar-Job einzubauen, der eine Analyse (z.B. mit "SonarSource/sonarcloud-github-action@master") macht und danach mit "sonarsource/sonarqube-quality-gate-action@master" prüft, ob alles passt (ihr könnt euch bei Problemen die Lösung im Repo anschauen, da es teils tricky ist)

---

**Spontane Übung?**



---

# Zusammenfassung

- Mit CI/CD kann man häufig durchzuführende Aufgaben automatisieren und so u.a. Zeit sparen
- Es gibt viele Tools zur Realisierung von CI/CD, die alle aber ähnlich funktionieren (z.B. GitHub Actions, GitLab CI/CD, Jenkins, ...)
- Aufteilung in Workflows, Jobs und Steps
- Mit GitHub Actions kann man in einem Workflow andere Workflows als Step ausführen lassen
- Jobs laufen standardmäßig parallel, was man allerdings mit "needs:" steuern kann
- Man kann Jobs mit mehreren Konfigurationen ausführen lassen

# Ausblick

- Wenn man so viele Befehle in einem Step hat, dass es unübersichtlich wird oder man andere Gründe dafür hat, kann man diese in eine Script-Datei auslagern und diese ausführen lassen
- Man kann einstellen, dass bei Pull-Requests ein Workflow ausgeführt werden und dieser erfolgreich sein muss, um mergen zu können
- Es können natürlich auch viele weitere Jobs eingebaut werden, um z.B. die Code-Coverage herauszufinden, eine statische Analyse laufen zu lassen oder deployen zu können
  - Hierbei kann man auf Google, YouTube oder direkt im Marketplace suchen