CI/CD, GitHub Actions

Hvad er formålet med CI?

Helt kort, så er formålet med CI at vise et lille grønt hak i ens GitHub, som indikerer at ens kode stadig virker. Det er smart!

I'et i CI er det samme "integration" som i "integrationstest". I praksis er der mange andre ting end integrationstests som bliver kørt under CI. For eksempel: Compiler koden, og lykkes ens unit tests?

C'et i CI står for "continuous": At tests køres automatisk og løbende som der bliver tilføjet og ændret kode i en kodebase. Det øger kvaliteten af det man laver, når man er sikker på at ens tests bliver kørt som en påkrævet del af ens workflow.

CI sker automatisk, så folk ikke glemmer eller undgår at gøre det. CI sker på en anden computer end ens laptop gør, så testene kan fortsætte til de er færdige, så man ikke stopper dem fordi man skal bruge computeren til noget andet, eller slukker den fordi man skal hjem.

Den computer som CI bliver afviklet på kaldes en "CI runner": Det er en lille virtuel maskine, som bor i skyen, som starter hver gang CI er blevet aktiveret. Man kan ikke være sikker på at det er den samme CI runner som afvikler ens kode hver gang, men man kan definere hvilken software der skal være på en CI runner før den går i gang.

Tjek relateret til æstetik og sikkerhed:

- At koden er korrekt formatteret (format-tjek)
- At koden overholder etablerede regler (linter-tjek)

- At koden er simpel (cyklomatisk kompleksitets-tjek)
- At koden ikke har velkendte huller (sikkerheds-tjek)

Tjek relateret til dependencies:
- At koden kun har dependencies med gyldige licenser
- At kodens dependencies er indbyrdes kompatible
- At kodens dependencies er tilgængelige og virker
- At kodens dependencies er blevet audit'ed

Tjek relateret til byggeprocessen:
- At koden compiler, og at kodens
- At kodens tests lykkes: Unit tests, performance tests, load-tests, stresstests, chaostests og så videre. Man kan også tjekke at koden har en tilstrækkelig test coverage, og fejle hvis ny kode ikke har.
- At koden ikke har nogen regressioner i performance
- At kodens dokumentation bliver bygget korrekt, og at der ikke mangler dokumentation påkrævede steder (fx Javadoc)
- At kodens *build artifacts* (fx .jar-filer eller Docker images) bliver bygget korrekt
- At der ikke mangler nogen sprogfiler hvis programmet er oversat til flere sprog

Tjek relateret til versionsstyring:
- At der ikke er nogen uforløste git-konflikter i historikken
- At git commit-beskeder overholder bestemte formater
- At alle ændringer er korrekt dokumenteret i en *changelog*

Tjek relateret til kørsel
- At filer relateret til kørsel virker: application.properties, filer med miljøvariable i, compose.yaml,

- At databasen kan genereres korrekt ud fra database-laget, fx JPA-klasser. Hvis databasen benytter *migrations*, at de kan afspille korrekt.

Hvad er et CI workflow?

Måden som en programmør arbejder på kaldes et workflow. For eksempel at man skifter mellem at snakke med en kunde, beskrive sit program, programmere det, og teste det.

En fil der beskriver CI til et projekt kaldes også et workflow. Et "CI workflow" eller en "CI pipeline" er de filer som beskriver handlinger som er påkrævet for at gennemføre CI (jobs, steps).

Det lyder lidt mærkeligt, men man kan tænke på det som filer der tilføjer noget automatisering til den måde man arbejder med koden på.

Når programmører samarbejder kræver det, at deres workflows er kompatible med hinandens. Programmører kan have forskellige workflows på det samme projekt. Men man har CI workflows til fælles som en slags kontrakt for hvad der er acceptabel omgang med fælles kode: Hvis CI melder fejl, har nogen brudt en fælles aftale, fx at koden på main-branch'en altid skal compile og at tests skal lykkes.

CI workflows er oftest kodet i en kombination af **YAML** og **shell script**. YAML er også det som man skriver Docker Compose-filer med, og shell script er de samme koder som man også kalder for "Linux-kommandoer", og er dem man skriver ind i Mac-terminalen eller "Git Bash": Det er faktisk et helt programmeringssprog!

En GitHub Action til Java CI

GitHub Actions bor i ens repository og er GitHub's indbyggede CI-system.

Når man først har lavet en god GitHub Action, kan man genbruge den i fremtidige projekter. Hvis man fx vil lave en simpel GitHub Action der tester om ens Java-kode compiler og tester, kan man bruge:

Et simpelt Java CI workflow

```yaml
# .github/workflows/java.yaml
name: Java CI

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4

      - name: Set up JDK
        uses: actions/setup-java@v4
        with:
          java-version: "21"
          distribution: "temurin"

      - name: Run tests
        run: mvn test
```

Hvad gør denne fil?

- Den starter CI hvis man push'er til main-branch'en, eller man laver en pull request der er rettet mod main-branch'en
- Den downloader koden fra git til CI runner'en
- Den installerer den nødvendige toolchain, fx Java SDK
- Den kører Maven tests

Hvad betyder de forskellige dele af en GitHub Action?

- Sektionen **`jobs:`** indikerer hvilket arbejde CI skal udføre. Hvert medlem her har en separat opgave. Hvert job har et eller flere **`steps:`**. Det er altså muligt at have flere jobs med flere steps. Formålet med at have flere jobs er, at man kan afvikle dem i parallel med forskellige miljøer, hvis fx man compiler flere programmer på flere programmeringssprog.
- Der er ét job, **`test:`**. Det er bare et navn og ikke et keyword.
- **`runs-on:`** indikerer at CI runner'en skal køre Ubuntu.
- Sektionen **`steps:`** indikerer de enkelte trin der skal lykkes for at test-jobbet bliver gennemført. Som udgangspunkt vil CI stoppe helt, hvis et step fejler. (Man kan godt slå det fra.)
- **`uses:`** og **`run:`**-linjerne er de faktiske steps. Et `uses:`-step kalder en GitHub Action, som bor i et andet repository, mens et `run:`-step kører en shell-kommando. For eksempel:
  - `uses: actions/checkout@v4`
  - `uses: actions/setup-java@v4`
  - `run: mvn test`

Vi vender tilbage til `uses:`-linjerne når vi snakker sikkerhed.

Starte GitHub Actions manuelt

Det er muligt at starte GitHub Actions manuelt, hvis man tilføjer `workflow_dispatch:` i listen af *triggers* i starten. Det ser sådan her ud:

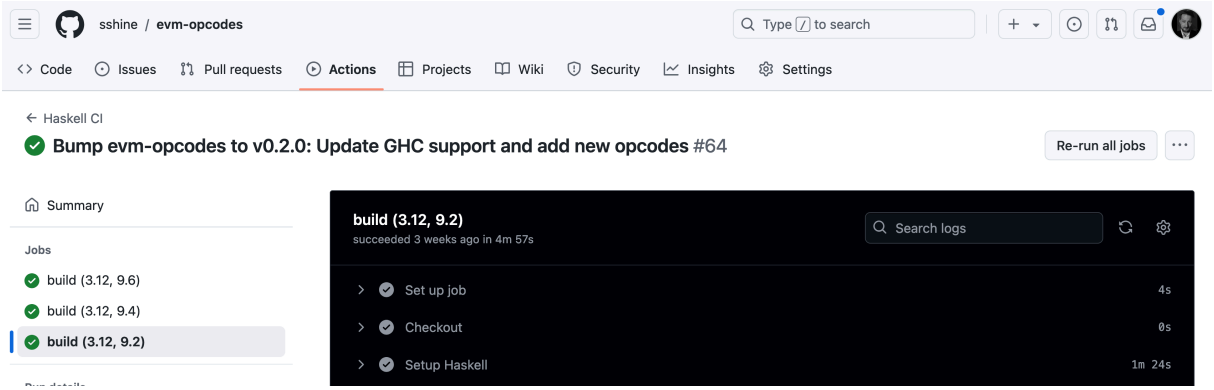
```yaml
# .github/workflows/java.yaml
name: Java CI

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main
  workflow_dispatch:

# ...
```

Det ligner at man har glemt noget fordi der bare står kolon, men det er sådan set gyldig YAML-kode. Det er muligt at putte flere argumenter på `workflow_dispatch:`, men ikke et krav.

Konsekvensen er at man nu har en knap oppe i højre hjørne, hvis man trykker "Actions > Java CI" (eller hvad ens action nu hedder), og så trykker **Re-run all jobs** eller lignende.



Optimere CI for miljøets skyld

Man kan spare meget tid i CI hvis man indfører cache på de rigtige niveauer. Det løber op i tusindvis af sekunder, minutter og timer. Man kan derfor påvirke den samlede CO_2-udledning, hvis man kan forkorte køretiden. Her er en udvidelse til ovenstående Java CI, hvor man har tilføjet caching af dependencies:

Et simpelt Java CI workflow med caching

```yaml
# .github/workflows/java.yaml
name: Java CI

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4

      - name: Set up JDK
        uses: actions/setup-java@v4
        with:
          java-version: "21"
          distribution: "temurin"

      - name: Cache Maven dependencies
        uses: actions/cache@v4
        with:
          path: ~/.m2/repository
          key: maven-${{ hashFiles('**/pom.xml') }}
          restore-keys: maven-${{ runner.os }}-

      - name: Run tests
        run: mvn test
```

Hvad gør "Cache Maven dependencies"-trinnet?

- `path:` er de stier som skal gemmes. `~/.m2` er Maven's cache-mappe (dependencies), så formålet med denne cache er at spare tid på at downloade Spring Boot hver gang CI kører.
- `key` er den nøgle som bestemmer om cachen skal fornys. Hvis nøglen skifter, bliver den gamle cache smidt ud. I eksemplet er der kun én ting som kan forårsage at cachen bliver fornyet:

  `hashFiles('**/pom.xml')`.

Hvorfor skal cachen fornys når pom.xml ændres?

Hvis man laver en eneste ændring i filen pom.xml, downloader den alle dependencies forfra. Det er måske lidt oftere end vi behøver: Der kan være mange andre grunde til at opdatere pom.xml end at tilføje, fjerne eller ændre dependencies. Selvom vi tilføjer en enkelt dependency, har vi ikke nødvendigvis brug for at downloade alle de andre dependencies påny.

Det er det man kalder en *heuristik*: Vi smider cachen ud oftere end vi behøver, sådan at vi er sikre på at smide cachen ud hver gang vi behøver. Samlet set cacher vi filerne oftere, end hvis vi ikke har nogen cache, og vi cacher dem aldrig når en dependency er blevet ændret.

Hvad ville der ske, hvis vi ikke fornyede cachen, når vi ændrer på pom.xml?

Så ville vi risikere at compile vores kode med gamle versioner af vores dependencies. Det kan lede til fejl. Hvis vi aldrig rydder cachen, vil cachen blive ved med at gro og aldrig krympe, og så vil CI runner'en løbe tør for diskplads.

Afprøve Dockerfile i GitHub Actions

Indtil nu har vi kun testet vores Java-applikation med `mvn test`, men vi har ikke testet om vores Dockerfile faktisk virker. Hvis vi deployer vores applikation ved hjælp af Docker, bør vi også teste at vores Docker-image kan bygges og køres korrekt.

At teste sin Dockerfile er en integrationstest – I'et i CI. Ved at teste vores Dockerfile i CI opdager vi potentielle problemer før de rammer produktion.

GitHub Actions der bygger og afprøver Dockerfile

```yaml
# .github/workflows/java-docker.yaml
name: Java CI with Docker

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: docker build -t myapp:test .
      - run: |
          # Start container i baggrunden
          docker run -d --name test-container -p 8080:8080 myapp:test

          # Vent på at applikationen starter
          sleep 30

          # Test at applikationen svarer (EKSEMPEL)
          curl -f http://localhost:8080/healthz || exit 1

          # Ryd op
          docker stop test-container
          docker rm test-container
```

GitHub Actions der bygger og afprøver compose.yaml

Hvis vi bruger `compose.yaml` i vores projekt, kan vi også teste den. Det kan være smart fordi alle ens containere bliver kørt i kombination.

```
      - run: |
          # Start services
          docker compose up -d

          # Vent på at MySQL og applikationen er klar
          sleep 45

          # Test at applikationen svarer (EKSEMPEL)
          curl -f http://localhost:8080/healthz

          # Test database forbindelse (EKSEMPEL)
          curl -f http://localhost:8080/api/users

          # Ryd op
          docker compose down
```

Begge eksempler antager at den web service man kører har et endpoint som hedder `/healthz` som svarer positivt tilbage. Hvis det var en Spring Boot-applikation kunne den se sådan her ud:

```
@RestController
public class HealthController {
    @GetMapping("/healthz")
    @ResponseBody
    public Map<String, String> health() {
        return Map.of(
            "status", "UP",
            "service", "my-application"
        );
    }
}
```

Det er meget normalt for microservices at have sådan en.

Sikkerhed

Indtil nu har vi taget for givet, at GitHub Actions bare virker, og at intet kan gå galt. Men hvis vi tager hacker-brillerne på, er der et par åbne huller i vores CI pipeline:

Når man bruger et step som fx `uses: actions/checkout@v4`, så er der to sikkerhedsrelaterede emner, man kan forholde sig til: **Hvem er `actions`?** og **Hvor er `v4`?** Det lyder som nogle ret abstrakte spørgsmål, så her er forklaringen bag dem:

Hvem er `actions`? Hvert step som bruger `uses:` refererer til et GitHub Action script som bor i et GitHub repository på en anden bruger. I det her tilfælde er brugeren `actions` og repository'et er `checkout`, og svaret på "Hvem?" er "GitHub / Microsoft": Det er deres konto. Man kan besøge repository'et for denne action her: https://github.com/actions/checkout

Hvor er `v4`? Vi har etableret hvilket repository den her `checkout`-action bor i. Git har mange måder at holde styr på forskellige versioner af kode. I det her tilfælde betyder `v4` et *git tag*, dvs. noget der minder om en branch. Man kunne for eksempel også skrive `uses: actions/checkout@main`, så ville man altid få den nyeste version af action'en.

Det prøver jeg nu at argumentere for er en dårlig idé, ligesom `v4` er en dårlig idé.

Vi antager allerede en høj grad af tillid til GitHub, så vi skal ikke som sådan være bange for at GitHub / Microsoft hacker os. Og vi kan måske endda regne med, at de sørger for at deres actions er bagud-kompatible, så `v4.1` virker i alle tilfælde hvor `v4` virker. Men vi vil gerne være helt sikre.

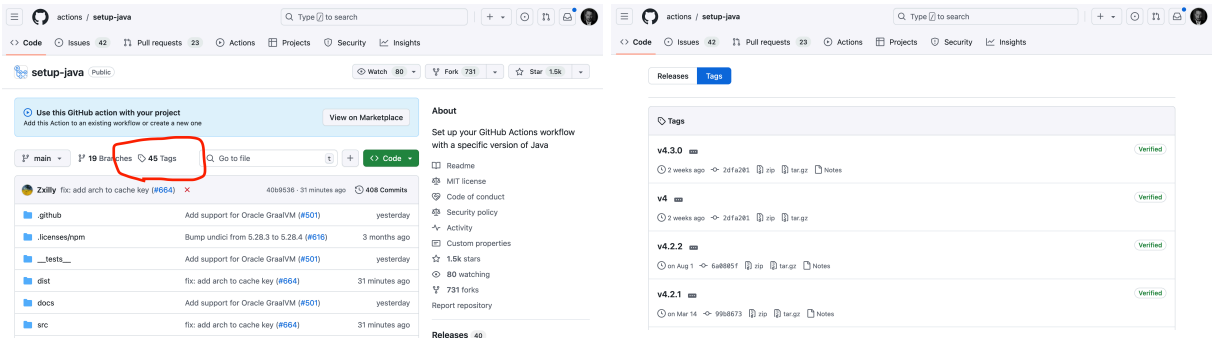
Hash-pinning: Hvad handler det om?

Vi ønsker at løse 2 problemer:

- **Reproducérbarhed:** Vi vil gerne sikre, at vores CI ikke ændrer opførsel uden vores samtykke, dvs. med mindre vi opdaterer CI-workflowet.
- **Sikkerhed:** Vi vil gerne forhindre at vores CI pipeline bliver hijacket af tredjeparts-actions, hvis vi begynder at bruge nogen der bliver udgivet af andre end GitHub.

Problem: v4 er et git tag og kan betyde hvad som helst

Git tags kan slettes og genoprettes. Hvad der er `v3` i dag er ikke `v3` i morgen.



Her ses git tags for `actions/setup-java`. Hvis vores pipeline fejler, skal det være fordi vores kode er gået i stykker, ikke fordi vores pipeline har ændret sig uden vi har spurgt.

Hvad er en hash og hvad er en pin?

Det vi mener med hash her er selvfølgelig den her lange, tilfældige streng af hexadecimaler:

`2dfa2011c5b2a0f1489bf9e433881c92c1631f88`

Men en **pin** er rigtigt nok en metafor for at fastgøre noget: Når man har lavet et git commit, får ens commit en unik nøgle, som er outputtet af en hashfunktion. Man kalder det også for et "digest" (en værdi fordøjet af hashfunktionen).

Det eneste vi behøver at vide er:

Det er astronomisk svært at fake en hashværdi.

Hash-pinning: Hvordan ser resultatet ud?

I følgende eksempel hash-pinner vi `actions/setup-java`:

```yaml
# Før hash-pinning ser trinnet sådan her ud:
- name: Set up JDK
  uses: actions/setup-java@v3

# Efter hash-pinning ser trinnet sådan her ud:
- name: Set up JDK
  uses: actions/setup-java@2dfa2011c5b2a0f1 # v4.3.0, 2024-09-09
```

- **Kryptografisk sikker:** Den afgørende forskel er, at når GitHub Actions leder efter den rigtige version af `actions/setup-java`, så vælger den en helt bestemt version, med kryptografisk nøjagtighed. Hvor `v3` kan snydes med, er det astronomisk usandsynligt at nogen har lavet en overlappende version med hash-værdien `2dfa2011c5b2a0f1489bf9e433881c92c1631f88`.
- **Informativ:** For læsbarhedens skyld står version og dato i en kommentar. Det er ikke et krav når man hash-pinner, men det gør ens arbejde betydeligt lettere, når man vender tilbage.

Hash-pinning: Hvordan gør man?

I stedet for `v3` vil vi gerne finde hash-værdien for udgivelsen.

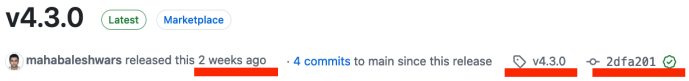
Trin:

1. Besøg action'ens GitHub-side, fx:

   https://github.com/actions/setup-java

2. Rul ned til "Releases" i højre kolonne af siden og find den nyeste version; i skrivende stund er det v4.3.0. Notér tre ting: **versionsnummeret** (`v4.3.0`), **datoen** (2024-09-24) og **git commit hash'en** som man finder øverst på siden som vist i følgende screenshot:

   Man kan finde den fulde commit hash ved at trykke på "**`2dfa201`**":

   Datoen finder man ved at holde musen over "2 weeks ago".

3. Oplysningerne indføres i stedet for `v3` i YAML-filen.

Hash-pinning: Er det virkelig nødvendigt?

GitHub anbefaler i deres dokumentation under "Security hardening for GitHub Actions" at man benytter den her teknik:

https://docs.github.com/en/actions/security-for-github-actions/security-guides/security-hardening-for-github-actions

Der er mange andre teknikker til at sikre ens GitHub Actions. De fleste har noget at gøre med hvordan man håndterer API-nøgler fortroligt i ens CI workflow.

Continuous Deployment

Indtil nu har vi fokuseret på CI (*Continuous Integration*) - at teste vores kode automatisk. Nu kigger vi på CD (*Continuous Deployment*) – at deploye kode automatisk til produktion.

Det kan gøres på rigtigt mange måder, men her er én:

1. Først bygges et Docker-image i CI (det har vi styr på)
2. Så skubbes Docker image'et til et container registry (GHCR)
3. Så sendes en besked til ens server om at der er opdateringer
4. Serveren henter de nye images ned og genstarter

Lærematerialet benytter GitHub Container Registry (GHCR) fordi det er nemt: Det koster ikke noget, og man er automatisk logget ind via GitHub Actions. Alternativt kunne man fx bruge DockerHub.

For at demonstrere at skubbe et Docker image til et container registry (GHCR) bruger vi `bashcrawl` som eksempel i stedet for en Spring Boot-applikation. Så kan vi let afprøve den resulterende container med `docker run -it`.

Der antages en kort Dockerfile, hvor der bare kopieres filer:

```
FROM alpine:latest
RUN apk add --no-cache bash
COPY entrance /bashcrawl/entrance
WORKDIR /bashcrawl/entrance
ENTRYPOINT ["/bin/bash"]
```

Se hele koden her: https://github.com/sshine/bashcrawl

CD workflow for bashcrawl

```yaml
# .github/workflows/cd.yaml
name: Continuous Deployment

on:
  push:
    branches:
      - master

env:
  REGISTRY: ghcr.io
  IMAGE_NAME: ${{ github.repository }}

jobs:
  build-and-push:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: write

    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Log in to Container Registry
        uses: docker/login-action@v3
        with:
          registry: ${{ env.REGISTRY }}
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}

      - name: Extract metadata
        id: meta
        uses: docker/metadata-action@v5
        with:
          images: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}
          tags: |
            type=ref,event=branch
            type=sha,prefix={{branch}}-
            type=raw,value=latest,enable={{is_default_branch}}

      - name: Build and push Docker image
        uses: docker/build-push-action@v6
        with:
          context: .
          push: true
          tags: ${{ steps.meta.outputs.tags }}
          labels: ${{ steps.meta.outputs.labels }}
```

Hvad sker der i CD-pipelinen?

- **`env:`** sætter nogle variable som senere trin bruger
- **`permissions:`** giver workflow'et tilladelse til at læse kode og skrive til container registry
- **`docker/login-action`** logger ind i GHCR med GitHub's indbyggede token. Hvis man vil logge ind på et andet Container Registry, skal man benytte Secrets til kodeordet.
- **`docker/metadata-action`** genererer Docker tags baseret på branch-navn og commit-hash. Metadata-action'en genererer flere tags automatisk:
  - `latest` - kun for master branch
  - `main-abc123` - branch-navn plus commit-hash
  - `main` - branch-navn
- **`docker/build-push-action`** Bygger Docker-image'et og pusher det til GHCR
- **`id: meta`** under trinnet "Extract metadata" hjælper de to trin med at kommunikere sammen: Tagget som Docker image'et gives beregnes i ét trin og benyttes i et andet trin.

Test det nye image

Efter CD-pipelinen kører, kan alle bruge det nye image direkte:

```
docker run -it ghcr.io/sshine/bashcrawl:latest
```

Så har vi et image parat til at blive deployet.