Øvelse 38 – Introduktion til Continuous Integration med GitHub Actions

Formål

Formålet med øvelsen er at opnå praktisk erfaring med Continuous Integration (CI) ved at konfigurere og anvende en automatiseret buildproces med GitHub Actions.

I skal arbejde med et eksisterende ASP.NET Core-projekt. Først opretter I en fork på jeres egen GitHub-konto og kloner projektet til jeres computer. Derefter opretter I et GitHub Actions-workflow lokalt og overfører ændringerne til GitHub.

I skal undersøge, hvordan GitHub automatisk bygger projektet, når ændringer pushes til repositoryet.

Efter øvelsen skal I kunne:

- Forklare sammenhængen mellem et lokalt Git-repository og et remote repository på GitHub.
- Anvende Git til at overføre lokale kodeændringer til GitHub.
- Oprette og anvende et GitHub Actions-workflow.
- Forklare de enkelte trin i en automatiseret buildproces.
- Analysere resultaterne fra et succesfuldt og et fejlet workflow-run.

Øvelsen fokuserer på **Continuous Integration**. Automatiserede tests og sikkerhedsscanninger introduceres i de efterfølgende øvelser.

Baggrund

Continuous Integration (CI) er en udviklingspraksis, hvor kodeændringer regelmæssigt integreres i et fælles repository og automatisk kontrolleres.

En typisk CI-proces kan se således ud:

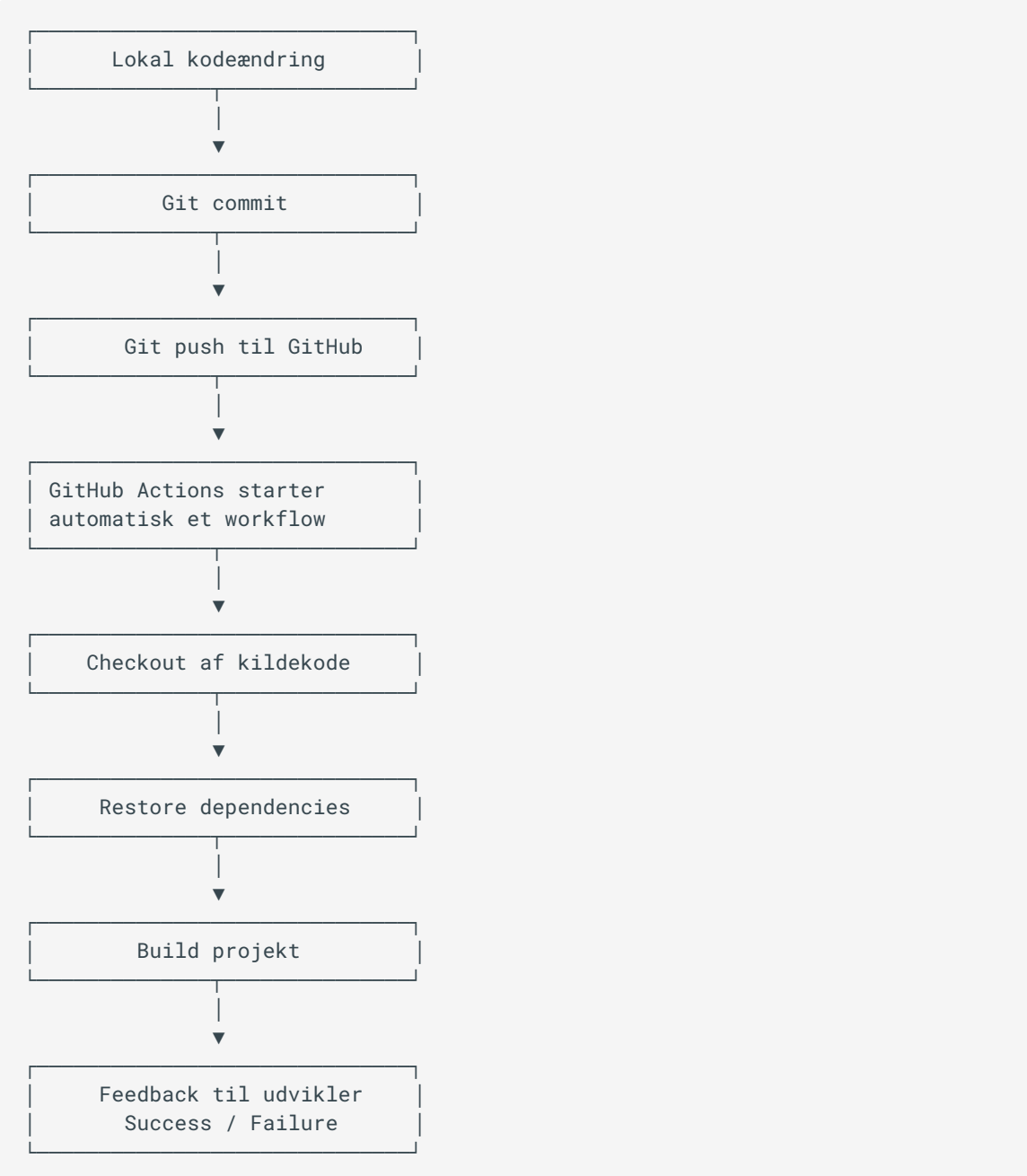
1. En udvikler ændrer kildekoden lokalt.
2. Ændringerne registreres i et Git-commit.
3. Ændringerne overføres til GitHub med `git push`.
4. GitHub registrerer ændringen og starter et workflow.

5. Workflowet henter kildekoden og bygger projektet.
6. Udvikleren modtager feedback på, om buildprocessen lykkedes.

Illustration af CI-processen

Illustration af CI-processen

Figuren nedenfor viser den grundlæggende CI-proces, som I skal arbejde med i denne øvelse.



CI gør det muligt at opdage eksempelvis kompileringsfejl tidligt i udviklingsprocessen.

I denne øvelse anvender vi **GitHub Actions**, som er GitHubs platform til automatisering af udviklingsprocesser.

Et workflow beskrives i en YAML-fil og udføres på en såkaldt *runner*.

Forudsætninger

I skal have:

- En GitHub-konto.
- Git installeret lokalt.
- .NET 10 SDK installeret.
- En kodeeditor, eksempelvis Visual Studio eller Visual Studio Code.
- Adgang til undervisningsprojektet:

https://github.com/mesn1985/CI-CD-and-DevSecOps-exercises

Der kræves ikke adgang til Microsoft Azure eller andre cloudtjenester.

Del 1 – Fork projektet på GitHub

Vi starter med at oprette en kopi af undervisningsprojektet på jeres egen GitHub-konto.

Trin 1 – Åbn repositoryet

Åbn:

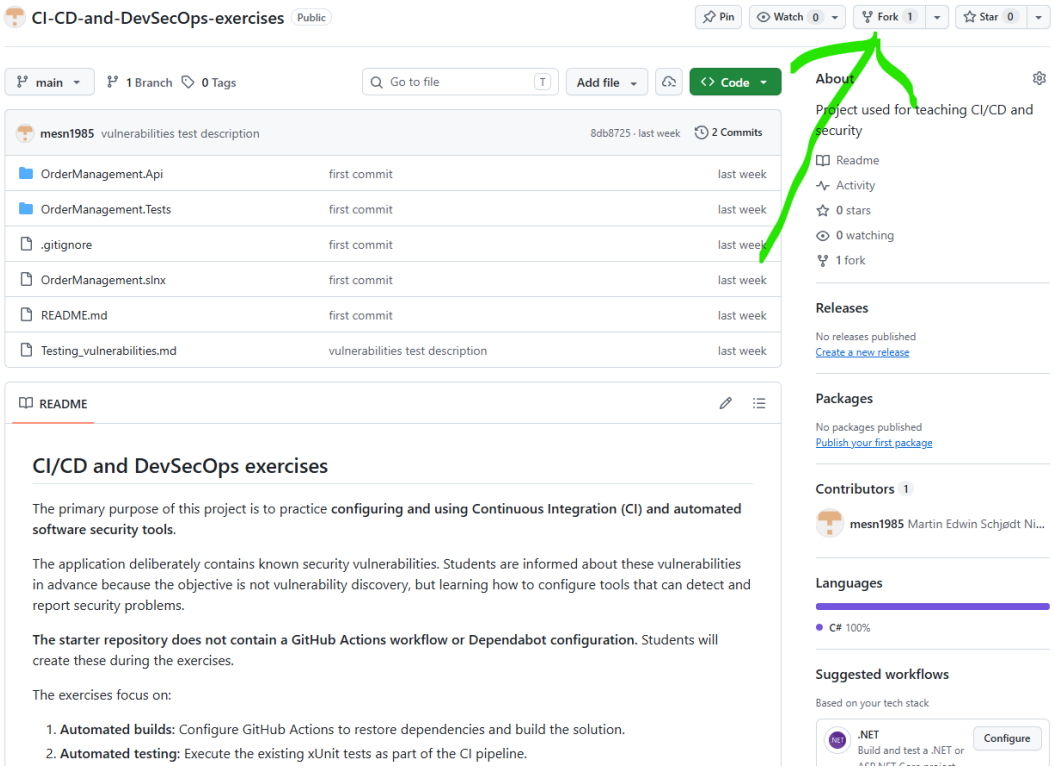
https://github.com/mesn1985/CI-CD-and-DevSecOps-exercises

Projektet indeholder blandt andet:

- `OrderManagement.Api` – ASP.NET Core REST API.
- `OrderManagement.Tests` – xUnit-testprojekt.
- `OrderManagement.slnx` – solution-fil.
- `README.md` – beskrivelse af projektet.

Trin 2 – Opret en fork

1. Klik på **Fork** øverst til højre.



2. Vælg jeres egen GitHub-konto som ejer.
3. Behold det foreslåede repository-navn.
4. Klik på **Create fork**.

GitHub opretter nu en kopi af projektet på jeres konto.

Adressen vil ligne:

`https://github.com/<brugernavn>/CI-CD-and-DevSecOps-exercises`

Forklaring

En *fork* er en kopi af et repository under en anden GitHub-konto.

I kan nu ændre projektet uden at påvirke det oprindelige undervisningsrepository.

Kontrol: I skal kunne åbne projektet på jeres egen GitHub-konto.

Del 2 – Klon repositoryet lokalt

I har nu oprettet en fork af undervisningsprojektet på jeres egen GitHub-konto.

Klon jeres fork til jeres lokale computer, og åbn projektet i jeres foretrukne udviklingsmiljø.

Kontroller, at I arbejder med jeres egen fork, og at I kan foretage ændringer lokalt og efterfølgende overføre dem til GitHub.

Kontrol: Projektet skal nu være tilgængeligt lokalt, og jeres lokale Git-repository skal være forbundet med jeres fork på GitHub.

Del 3 – Kontroller projektet lokalt

Inden vi konfigurerer Continuous Integration, skal vi sikre, at projektet kan bygges lokalt.

Åbn en terminal i projektets rodmappe, hvor `OrderManagement.slnx` ligger.

Trin 1 – Gendan afhængigheder

Kør:

```
dotnet restore
```

Denne kommando henter de NuGet-pakker, som projektet afhænger af.

Trin 2 – Byg projektet

Kør:

```
dotnet build --no-restore
```

Denne kommando kompilerer kildekoden.

Parameteren `--no-restore` betyder, at .NET anvender de afhængigheder, som allerede er gendannet i det foregående trin.

Forventet resultat

Projektet skal kunne bygges uden kompileringsfejl.

Det kan forekomme, at .NET rapporterer advarsler om sårbare NuGet-afhængigheder.

Disse er bevidst introduceret i undervisningsprojektet og undersøges i en senere øvelse.

Refleksion

Hvorfor kan det være en fordel at kontrollere, at projektet kan bygges lokalt, inden ændringer overføres til GitHub?

Del 4 – Opret et GitHub Actions-workflow

Vi skal nu konfigurere GitHub til automatisk at udføre de samme buildkommandoer, som vi netop har udført lokalt.

Trin 1 – Opret workflow-mappen

I projektets rodmappe skal I oprette følgende mappestruktur:

`.github/workflows/`

I kan oprette mappen fra jeres kodeeditor eller med PowerShell:

```
New-Item -ItemType Directory -Force -Path ".github/workflows"
```

Trin 2 – Opret workflow-filen

Opret filen:

`.github/workflows/ci.yml`

Trin 3 – Indsæt workflowet

Indsæt følgende YAML-konfiguration:

```yaml
name: CI

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

permissions:
  contents: read

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout repository
        uses: actions/checkout@v7

      - name: Setup .NET
        uses: actions/setup-dotnet@v6
        with:
          dotnet-version: '10.0.x'

      - name: Restore dependencies
```

```
        run: dotnet restore

      - name: Build solution
        run: dotnet build --no-restore
```

Gem filen.

Det er vigtigt, at indrykningen bevares, da YAML anvender indrykning til at beskrive strukturen.

Kontrol: Filen `ci.yml` skal nu findes lokalt under `.github/workflows/`.

Del 5 – Analyser workflowets opbygning

Vi skal nu gennemgå workflowets enkelte elementer og undersøge deres funktion.

1. Workflowets navn

```
name: CI
```

Dette angiver navnet på workflowet.

Navnet vises i GitHubs Actions-fane.

I vores tilfælde hedder workflowet `CI`.

2. Hændelser, som starter workflowet

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

Dette angiver, hvornår workflowet skal starte.

Der er konfigureret to hændelser:

Push

Workflowet starter, når ændringer pushes til branch `main`.

Pull request

Workflowet starter også, når der oprettes eller opdateres en pull request rettet mod `main`.

Det betyder, at ændringer kan kontrolleres automatisk, inden de flettes ind i hovedbranchen.

I denne øvelse arbejder vi primært med `push`.

3. Rettigheder

```yaml
permissions:
  contents: read
```

Denne konfiguration begrænser de rettigheder, som workflowets `GITHUB_TOKEN` tildeles.

Tokenet får læseadgang til repositoryets indhold, hvilket er tilstrækkeligt til vores buildproces.

Det er et eksempel på sikkerhedsprincippet **least privilege**.

4. Job

```yaml
jobs:
  build:
```

Et workflow består af et eller flere *jobs*.

Her opretter vi et job med navnet `build`.

Et job består af en række steps, som udføres i den angivne rækkefølge.

5. Runner

```yaml
runs-on: ubuntu-latest
```

En *runner* er det miljø, hvor jobbet udføres.

Her anvender vi en Ubuntu-baseret runner, som GitHub stiller til rådighed.

Projektet bliver dermed bygget i et standardiseret miljø, som ikke afhænger af konfigurationen på jeres lokale computer.

Dette kan hjælpe med at opdage problemer, der ellers kun viser sig i bestemte udviklingsmiljøer.

6. Checkout af kildekoden

```yaml
- name: Checkout repository
  uses: actions/checkout@v7
```

Dette step henter repositoryets kildekode til runnerens arbejdsmappe.

`actions/checkout` er en eksisterende GitHub Action.

Uden dette step vil projektets kildekode normalt ikke være tilgængelig for de efterfølgende buildkommandoer.

7. Installation af .NET SDK

```yaml
- name: Setup .NET
  uses: actions/setup-dotnet@v6
  with:
    dotnet-version: '10.0.x'
```

Dette step sikrer, at den ønskede .NET SDK-version er tilgængelig.

`dotnet-version: '10.0.x'` angiver, at workflowet skal anvende en SDK-version fra .NET 10.

Ved at angive SDK-versionen eksplicit mindskes risikoen for forskelle mellem udviklingsmiljøer.

8. Restore dependencies

```yaml
- name: Restore dependencies
  run: dotnet restore
```

Dette step gendanner projektets NuGet-afhængigheder.

Kommandoen svarer til den, vi tidligere udførte lokalt.

Hvis en nødvendig afhængighed ikke kan hentes, kan dette step fejle.

9. Build solution

```yaml
- name: Build solution
  run: dotnet build --no-restore
```

Dette step bygger projektet.

Vi anvender `--no-restore`, fordi afhængighederne allerede er gendannet i det foregående step.

Hvis kildekoden indeholder kompileringsfejl, vil buildprocessen fejle, og jobbet vil blive markeret som fejlet.

Opsamling

Element	Funktion

Element	Funktion
`name`	Navngiver workflowet
`on`	Angiver hændelser, som starter workflowet
`permissions`	Begrænser workflowets tokenrettigheder
`jobs`	Definerer de jobs, der skal udføres
`runs-on`	Vælger runner-miljø
`steps`	Definerer rækkefølgen af aktiviteter
`uses`	Anvender en eksisterende GitHub Action
`run`	Udfører en kommando på runneren

Refleksion

Besvar følgende spørgsmål:

1. Hvad er forskellen mellem et step, der anvender `uses`, og et step, der anvender `run`?
2. Hvorfor er det nødvendigt at hente kildekoden til runneren?
3. Hvorfor installerer vi en bestemt .NET SDK-version?
4. Hvorfor udfører vi `dotnet restore` før `dotnet build`?
5. Hvordan kommer princippet om *least privilege* til udtryk i workflowet?

Del 6 – Commit og push workflowet

Vi har nu oprettet workflowet lokalt.

For at GitHub kan anvende det, skal filen overføres til vores remote repository.

Trin 1 – Undersøg ændringerne

Kør:

```
git status
```

Git bør vise, at der findes en ny fil under `.github/workflows/`.

Trin 2 – Tilføj workflow-filen

Kør:

```
git add .github/workflows/ci.yml
```

Dette placerer filen i Git's staging area.

Trin 3 – Opret et commit

Kør:

```
git commit -m "Add initial CI workflow"
```

Et commit registrerer ændringen i det lokale Git-repository.

Trin 4 – Push til GitHub

Kør:

```
git push origin main
```

Denne kommando overfører de lokale commits til branch `main` på jeres remote repository.

Hvis GitHub beder om login, skal I gennemføre den relevante godkendelsesproces.

Forklaring

Kommando	Funktion
`git status`	Viser lokale ændringer
`git add`	Tilføjer ændringer til staging area
`git commit`	Registrerer ændringer i den lokale Git-historik
`git push`	Overfører lokale commits til et remote repository

Det er vigtigt at forstå, at `git commit` ikke automatisk overfører ændringer til GitHub.

Ændringerne bliver først synlige i det remote repository, når de pushes.

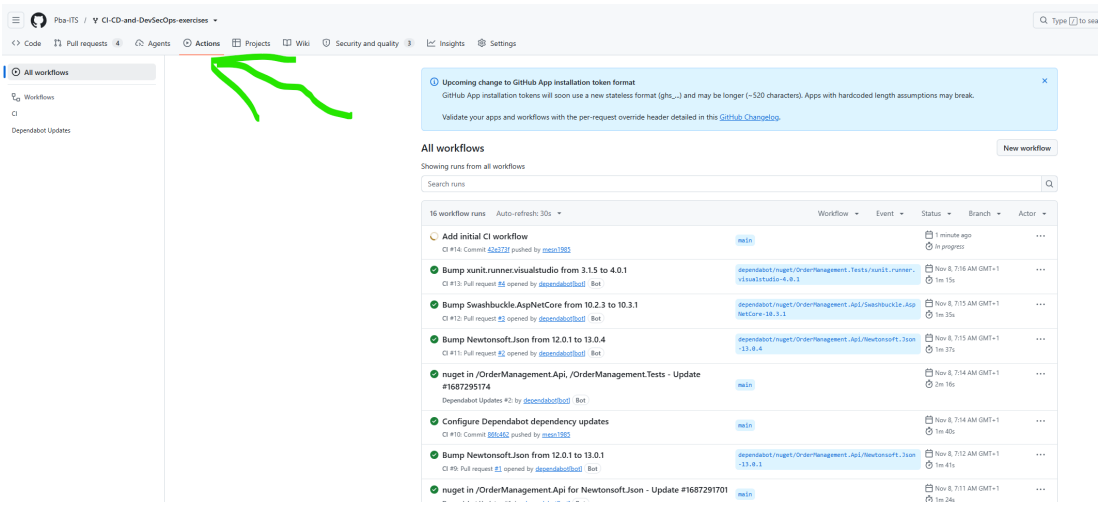
Kontrol: Åbn jeres repository på GitHub, og kontroller, at filen `.github/workflows/ci.yml` nu findes.

Del 7 – Analyser det første workflow-run

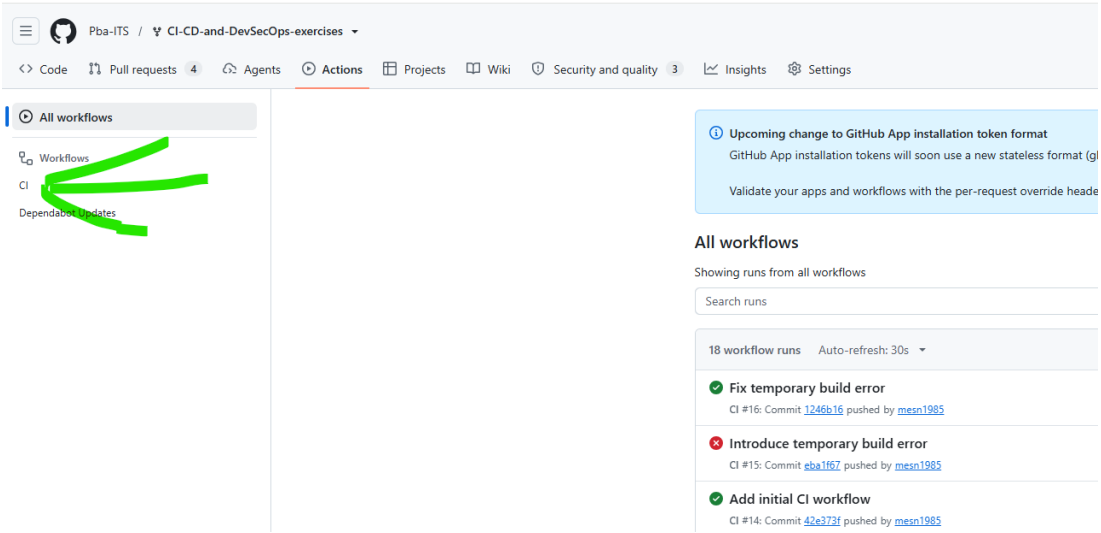
Når workflow-filen er blevet overført til GitHub, skal GitHub Actions automatisk starte buildprocessen.

Trin 1 – Åbn GitHub Actions

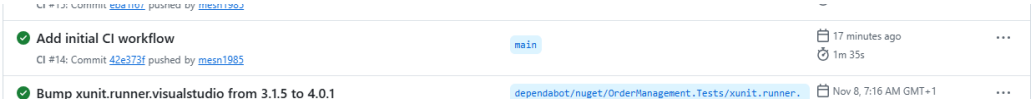
1. Åbn jeres fork på GitHub.
2. Vælg fanen **Actions**.



3. Find workflowet med navnet `CI`.



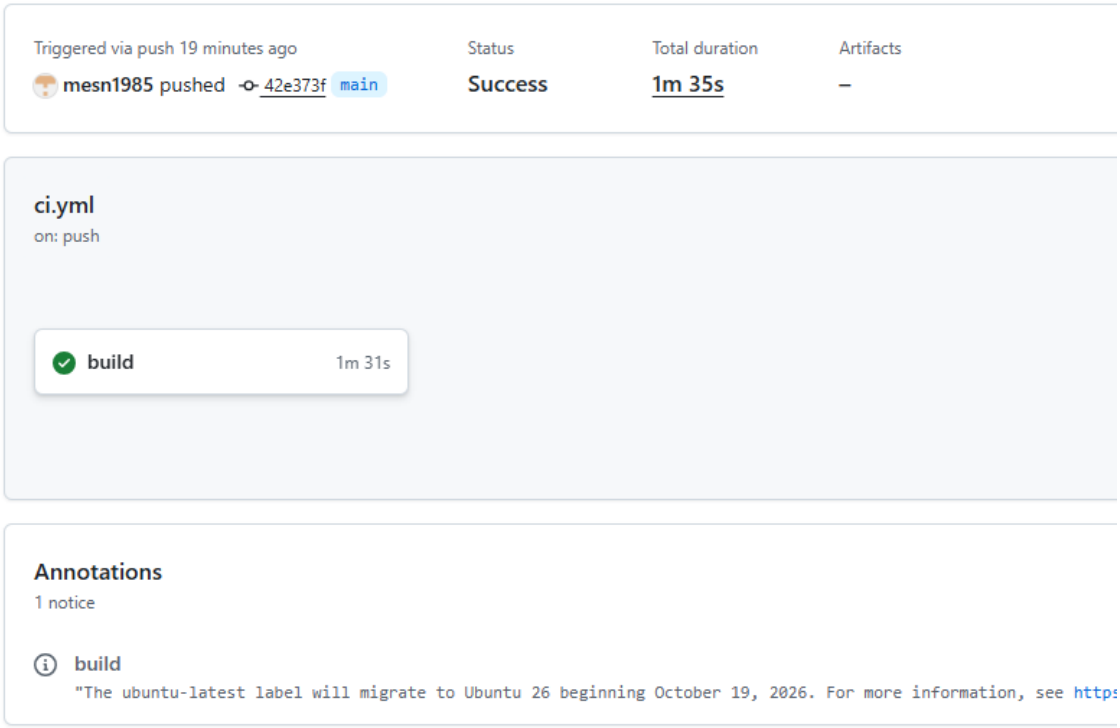
4. Åbn det seneste workflow-run.



Hvis GitHub Actions er deaktiveret for jeres fork, skal I først aktivere det via Actions-fanen.

Trin 2 – Undersøg jobbet

Klik på jobbet `build`.



I bør blandt andet kunne se følgende steps:

- Checkout repository
- Setup .NET
- Restore dependencies
- Build solution

GitHub kan desuden vise interne steps til klargøring og oprydning af runneren.

Trin 3 – Undersøg loggene

Udvid hvert step, og undersøg de kommandoer og meddelelser, der vises.

Kontroller:

- Om repositoryets kildekode blev hentet.
- Om .NET 10 SDK blev gjort tilgængelig.
- Om NuGet-afhængighederne blev gendannet.

- Om projektet blev bygget.
- Om jobbet fik status `Success`.

Forventet resultat

Workflowet skal gennemføre restore og build uden kompileringsfejl.

Der kan forekomme advarsler om sårbare afhængigheder. Det betyder ikke nødvendigvis, at workflowet fejler.

Refleksion

Besvar følgende spørgsmål:

1. Hvilken hændelse startede workflowet?
2. Hvor blev buildprocessen udført?
3. Hvilke aktiviteter udførte GitHub automatisk?
4. Hvad er forskellen mellem at udføre `dotnet build` lokalt og via GitHub Actions?
5. Hvilken information giver workflow-loggene om buildprocessen?

Del 8 – Undersøg hvordan CI reagerer på en fejl

Vi skal nu demonstrere, hvordan en CI-proces kan opdage en kompileringsfejl.

Trin 1 – Introducer en midlertidig fejl

Åbn filen:

`OrderManagement.Api/Program.cs`

Find følgende linje:

```
app.Run();
```

Indsæt den følgende linje **umiddelbart før** `app.Run();`:

```
ThisWillNotCompile();
```

Den relevante del af filen skal nu se således ud:

```
ThisWillNotCompile();

```

```
app.Run();
```

Metoden `ThisWillNotCompile()` findes ikke, og derfor vil koden medføre en kompileringsfejl.

Trin 2 – Commit og push ændringen

Gem filen, og udfør:

```
git add OrderManagement.Api/Program.cs
git commit -m "Introduce temporary build error"
git push origin main
```

Trin 3 – Undersøg workflowet

Åbn GitHub Actions.

GitHub bør automatisk starte et nyt workflow-run.

Undersøg resultatet.

Forventet resultat: Workflowets build-step fejler, fordi koden ikke kan kompileres.

Find fejlmeddelelsen i loggen.

Bemærk, hvordan GitHub markerer det fejlede step og jobbet.

Trin 4 – Ret fejlen

Fjern linjen:

```
ThisWillNotCompile();
```

Gem filen.

Trin 5 – Commit og push rettelsen

```
git add OrderManagement.Api/Program.cs
git commit -m "Fix temporary build error"
git push origin main
```

Trin 6 – Verificer resultatet

Åbn GitHub Actions igen.

Kontroller, at det seneste workflow-run gennemføres korrekt.

I har nu demonstreret:

- Et succesfuldt build.
- Et fejlet build.
- Et succesfuldt build efter rettelse af fejlen.

Refleksion

1. Hvilken fejlmeddelelse blev vist i GitHub Actions?
2. Hvorfor blev jobbet markeret som fejlet?
3. Hvordan kan automatiserede builds reducere risikoen for, at kode med kompileringsfejl integreres i et fælles softwareprojekt?
4. Ville samme workflow nødvendigvis opdage en logisk fejl i applikationens forretningslogik?

Samlet refleksion

Besvar følgende spørgsmål med udgangspunkt i jeres egne observationer:

1. Hvilken rolle spiller GitHub som remote repository i CI-processen?
2. Hvorfor skal ændringer pushes, før GitHub Actions kan reagere på dem?
3. Hvad er fordelen ved, at en buildproces udføres automatisk frem for udelukkende lokalt?
4. Hvordan giver GitHub Actions udviklerne feedback om problemer i kildekoden?
5. Hvorfor er et succesfuldt build ikke nødvendigvis ensbetydende med, at applikationen fungerer korrekt?
6. Hvorfor er et succesfuldt build heller ikke en garanti for, at applikationen er sikker?

Resultat og evaluering

Øvelsen er gennemført, når I kan demonstrere følgende:

- I har oprettet en fork af undervisningsprojektet.
- I har klonet projektet til jeres lokale computer.
- I har bygget projektet lokalt.
- I har oprettet et GitHub Actions-workflow.
- I har committed og pushed workflowet til GitHub.
- GitHub Actions har automatisk gennemført restore og build.

- I har observeret, hvordan workflowet reagerer på en kompileringsfejl.
- I har rettet fejlen og verificeret, at workflowet igen gennemføres korrekt.

Det vigtigste er ikke alene, at workflowet fungerer, men at I kan forklare sammenhængen mellem **lokale kodeændringer, Git, GitHub og automatiseret CI**.

Videre arbejde

I denne øvelse har vi oprettet en CI-pipeline, som kontrollerer, om projektet kan bygges.

Men selvom koden kan kompileres, kan den stadig indeholde funktionelle fejl.

I **øvelse 39** udvider vi derfor det eksisterende workflow med automatiserede unit tests.

Her skal GitHub Actions ikke blot bygge applikationen, men også kontrollere, om den eksisterende forretningslogik fungerer som forventet.

Litteratur og dokumentation

- *Secure by Design*, kapitel 8 – Leveraging your delivery pipeline for security.
- GitHub Actions – Building and testing .NET
- GitHub Actions – Workflow syntax
- GitHub Actions – setup-dotnet

Last update: 2026-11-10 20:57:39