Øvelse 39 – Automatiserede unit tests med GitHub Actions

Formål

Formålet med øvelsen er at udvide en eksisterende Continuous Integration-pipeline med automatiserede unit tests.

I øvelse 38 konfigurerede I GitHub Actions til automatisk at hente afhængigheder og bygge Order Management API'et.

Nu skal I udvide samme workflow, så det også udfører projektets eksisterende unit tests.

I skal desuden undersøge, hvordan CI-pipelinen reagerer, når en test fejler, selvom applikationen stadig kan kompileres.

Efter øvelsen skal I kunne:

- Forklare forskellen mellem automatiseret build og automatiseret test.
- Konfigurere GitHub Actions til at udføre unit tests.
- Fortolke resultaterne fra `dotnet test`.
- Forklare, hvordan fejlede tests påvirker CI-pipelinens status.
- Beskrive, hvordan automatiserede tests bidrager til kvalitetssikring i softwareudvikling.

Baggrund

Et succesfuldt build viser, at kildekoden kan kompileres, og at de nødvendige afhængigheder kan anvendes.

Det betyder imidlertid ikke, at applikationens funktionalitet er korrekt.

En applikation kan eksempelvis kompilere uden fejl, men stadig indeholde fejl i dens forretningslogik.

Unit tests anvendes til automatisk at kontrollere afgrænsede dele af en applikations funktionalitet.

Når unit tests integreres i en CI-pipeline, kan de udføres automatisk ved kodeændringer.

En CI-pipeline kan dermed kontrollere både:

1. Om applikationen kan bygges.
2. Om applikationens eksisterende tests gennemføres korrekt.

Hvis en test fejler, kan pipelinen markeres som fejlet og dermed give udviklerne feedback om problemet.

Denne mekanisme er en vigtig del af automatiseret kvalitetssikring.

Forudsætninger

I skal have gennemført øvelse 38.

Det betyder, at I har:

- Oprettet en fork af undervisningsprojektet.
- Klonet projektet til jeres lokale computer.
- Oprettet `.github/workflows/ci.yml`.
- Konfigureret automatisk restore og build med GitHub Actions.
- Verificeret, at CI-workflowet fungerer.

I skal fortsætte med det samme lokale repository og den samme GitHub Actions-konfiguration.

I skal ikke oprette et nyt repository eller et nyt workflow.

Del 1 – Undersøg projektets eksisterende unit tests

Order Management API'et indeholder allerede en række unit tests.

Åbn testprojektet:

`OrderManagement.Tests`

Undersøg, hvilke tests projektet indeholder.

Testene kontrollerer blandt andet:

- At en gyldig ordre kan oprettes.
- At en ordre med ugyldigt antal produkter afvises.
- At en oprettet ordre kan annulleres.
- At en oprettet ordre kan markeres som leveret.
- At en leveret ordre ikke efterfølgende kan annulleres.

Testene er implementeret med **xUnit**.

Trin 1 – Udfør testene lokalt

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

Kør:

```
dotnet test
```

Kommandoen finder og udfører testene i løsningen.

Trin 2 – Undersøg resultatet

Kontroller:

- Hvor mange tests blev udført?
- Hvor mange tests blev gennemført?
- Var der tests, som fejlede?
- Hvilken samlet status rapporterede `dotnet test`?

Forventet resultat

Alle fem eksisterende tests skal gennemføres korrekt.

Refleksion

1. Hvilke dele af applikationens funktionalitet kontrolleres af testene?
2. Hvad er forskellen mellem at udføre `dotnet build` og `dotnet test`?
3. Kan alle fem tests gennemføres, selvom applikationen stadig indeholder sikkerhedssårbarheder?

Del 2 – Udvid CI-workflowet med automatiserede tests

Vi skal nu udvide workflowet fra øvelse 38.

Åbn filen:

`.github/workflows/ci.yml`

Trin 1 – Tilføj et test-step

Tilføj følgende step efter `Build solution`:

```
      - name: Run unit tests
        run: dotnet test --no-build
```

Trin 2 – Kontroller det samlede workflow

Jeres workflow skal nu se således ud:

```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

      - name: Run unit tests
        run: dotnet test --no-build
```

Gem filen.

Forklaring af det nye step

```yaml
- name: Run unit tests
  run: dotnet test --no-build
```

`name: Run unit tests`

Angiver navnet på steppet, som vises i GitHub Actions.

`run: dotnet test --no-build`

Udfører projektets unit tests.

Parameteren `--no-build` betyder, at testkommandoen genbruger de filer, som allerede er bygget i det foregående step.

Dette er muligt, fordi alle steps i vores job udføres på den samme runner.

Det samlede workflow består nu af følgende aktiviteter:

Step	Formål
Checkout repository	Henter kildekoden
Setup .NET	Klargør .NET 10 SDK
Restore dependencies	Gendanner NuGet-afhængigheder
Build solution	Kompilerer løsningen
Run unit tests	Udfører automatiserede tests

Refleksion

1. Hvorfor placerer vi test-steppet efter build-steppet?
2. Hvad betyder parameteren `--no-build`?
3. Hvad forventer I, at der sker med workflowet, hvis en unit test fejler?

Del 3 – Commit og push ændringerne

Workflowet er nu opdateret lokalt.

Ændringen skal overføres til GitHub, før den kan anvendes af GitHub Actions.

Udfør følgende kommandoer:

```
git add .github/workflows/ci.yml
git commit -m "Add unit tests to CI workflow"
git push origin main
```

Åbn derefter jeres repository på GitHub.

Vælg fanen **Actions**, og find det nye workflow-run.

Forventet resultat

Workflowet skal nu indeholde det nye step:

`Run unit tests`

Dette step skal udføre de fem eksisterende tests.

Når alle tests gennemføres, skal workflowet få status `Success`.

Del 4 – Analyser testresultaterne i GitHub Actions

Åbn det seneste workflow-run i GitHub Actions.

Klik på jobbet `build`, og udvid steppet:

`Run unit tests`

Undersøg loggen fra testeksekveringen.

Trin 1 – Identificer testresultatet

Find oplysninger om:

- Antallet af udførte tests.
- Antallet af gennemførte tests.
- Antallet af fejlede tests.
- Den samlede status for testeksekveringen.

Trin 2 – Sammenlign med den lokale test

Sammenlign resultatet fra GitHub Actions med resultatet fra `dotnet test` på jeres lokale computer.

Forventet resultat

Begge miljøer skal kunne udføre de fem tests med succes.

Refleksion

1. Hvilke fordele giver det at udføre de samme tests både lokalt og automatisk i CI?
2. Hvorfor er det relevant, at CI-pipelinen udfører tests på en runner frem for udelukkende på udviklerens computer?

Del 5 – Undersøg hvordan CI reagerer på en fejlet test

I øvelse 38 introducerede I en kompileringsfejl og observerede, hvordan buildprocessen fejlede.

Nu skal I undersøge en anden situation:

Applikationen kan bygges, men en unit test fejler.

Det demonstrerer forskellen mellem buildkontrol og funktionel test.

Trin 1 – Opret en midlertidig test

Opret følgende fil i testprojektet:

`OrderManagement.Tests/CiPipelineTests.cs`

Indsæt:

```
using Xunit;

namespace OrderManagement.Tests;

public class CiPipelineTests
{
    [Fact]
    public void DemonstrateFailingTest()
    {
        Assert.True(false, "Intentional failure for CI exercise");
    }
}
```

Testen er bevidst konstrueret til at fejle.

Formålet er ikke at teste applikationens forretningslogik, men at demonstrere CI-pipelinens reaktion på en fejlet test.

Trin 2 – Kontroller resultatet lokalt

Kør:

```
dotnet build
```

Buildprocessen skal gennemføres korrekt.

Kør derefter:

```
dotnet test
```

Nu skal testeksekveringen fejle.

Undersøg fejlmeddelelsen.

Forventet resultat

Projektet kan stadig bygges, men mindst én test fejler.

Det viser, at et succesfuldt build ikke er en garanti for, at testene gennemføres.

Trin 3 – Commit og push ændringen

Udfør:

```
git add OrderManagement.Tests/CiPipelineTests.cs
git commit -m "Add intentional failing test"
git push origin main
```

Åbn derefter GitHub Actions.

Trin 4 – Analyser det fejlede workflow

Find det nye workflow-run.

Undersøg følgende:

1. Blev restore gennemført?
2. Blev build gennemført?
3. Hvilket step fejlede?
4. Hvilken test var årsag til fejlen?
5. Hvilken status fik det samlede job?

Forventet resultat

- `Restore dependencies` gennemføres.
- `Build solution` gennemføres.
- `Run unit tests` fejler.
- Det samlede job markeres som fejlet.

Refleksion

Hvorfor er det vigtigt, at et fejlet test-step medfører, at CI-jobbet markeres som fejlet, selvom applikationen kan bygges?

Del 6 – Ret fejlen og verificer CI-pipelinen

Vi skal nu bringe CI-pipelinen tilbage til en succesfuld tilstand.

Trin 1 – Fjern den midlertidige test

Slet filen:

`OrderManagement.Tests/CiPipelineTests.cs`

Vi ønsker ikke at bevare den bevidst fejlede test i projektet.

Trin 2 – Udfør testene lokalt

Kør:

```
dotnet test
```

Kontroller, at alle fem oprindelige tests igen gennemføres korrekt.

Trin 3 – Commit og push rettelsen

Udfør:

```
git add -u
git commit -m "Remove intentional failing test"
git push origin main
```

Trin 4 – Verificer resultatet

Åbn GitHub Actions.

Kontroller, at det nyeste workflow-run gennemføres korrekt.

I skal nu have observeret:

- Et succesfuldt workflow med fem gennemførte tests.
- Et fejlet workflow med en bevidst fejlet test.

- Et succesfuldt workflow efter fjernelse af testen.

Refleksion

1. Hvordan kan en CI-pipeline hjælpe med at opdage regressioner?
2. Hvad sker der, hvis en fejl i forretningslogikken ikke er dækket af nogen test?
3. Hvorfor er det ikke tilstrækkeligt blot at kontrollere, at testkommandoen er blevet udført?

Samlet refleksion

Besvar følgende spørgsmål med udgangspunkt i jeres observationer:

1. Hvilken ekstra sikkerhed for funktionaliteten giver automatiserede tests sammenlignet med et build alene?
2. Hvordan påvirkes CI-pipelinen, hvis en unit test fejler?
3. Hvorfor er det relevant at udføre tests ved hver kodeændring?
4. Hvilke begrænsninger har automatiserede unit tests?
5. Kan en applikation gennemføre alle sine unit tests og stadig indeholde sikkerhedssårbarheder?
6. Hvordan kan automatiserede tests indgå som en del af en mere sikker softwareudviklingsproces?

Resultat og evaluering

Øvelsen er gennemført, når I kan demonstrere følgende:

- I har videreudviklet workflowet fra øvelse 38.
- Workflowet udfører nu automatiserede unit tests.
- De eksisterende fem tests gennemføres i GitHub Actions.
- I har observeret en fejlet test i CI.
- I kan forklare forskellen mellem et fejlet build og en fejlet test.
- I har rettet testproblemet og verificeret et succesfuldt workflow.
- I kan forklare, hvordan automatiseret test understøtter kvalitetssikring.

Det vigtigste er ikke blot, at CI-pipelinen bliver grøn, men at I kan **fortolke testresultaterne og forstå, hvad de fortæller om applikationens kvalitet**.

Videre arbejde

I øvelse 38 etablerede I en automatiseret buildproces.

I denne øvelse udvidede I processen med automatiserede unit tests.

Pipelinen kontrollerer nu både, om projektet kan bygges, og om den eksisterende funktionalitet består de implementerede tests.

Men en applikation kan stadig indeholde sikkerhedssårbarheder, selvom alle unit tests gennemføres korrekt.

I **øvelse 40** udvider vi derfor CI-pipelinen med scanning af tredjepartsafhængigheder ved hjælp af **NuGet Audit**.

Formålet bliver at identificere kendte sikkerhedssårbarheder i de biblioteker, som applikationen anvender.

Litteratur og dokumentation

- *Secure by Design*, kapitel 8 – Leveraging your delivery pipeline for security.
- Microsoft – dotnet test
- Microsoft – Unit testing with .NET
- GitHub Actions – Building and testing .NET

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