

324-08B - JEST Tests Einführung

Inhalt

Ziel der Umsetzung.....	1
1. Anpassung der Pipeline-Struktur.....	1
2. Backend-Tests: test:java-api	2
Erläuterungen zur Konfiguration	2
3. Frontend-Tests: test:react-app.....	3
Erläuterungen zur Konfiguration	3
4. Korrektur eines fehlerhaften Frontend-Tests.....	4
5. Ergebnis: Erfolgreiche Pipeline.....	4

Ziel der Umsetzung

Damit die Qualität unserer Software langfristig gesichert ist, sollen alle bestehenden und zukünftigen Unit-Tests automatisch in der CI/CD-Pipeline ausgeführt werden. So erkennen wir Fehler frühzeitig – noch bevor ein fehlerhaftes Image gebaut oder ausgeliefert wird.

1. Anpassung der Pipeline-Struktur

Zunächst wurde die bestehende Pipeline um eine zusätzliche **Test-Stage** erweitert in dem folgendes ins `.gitlab-ci.yml` reingeschrieben wurde:

```
stages:
- build
- test      # ← NEU: Tests nach dem Build
- docker-build
- docker-push
```

Warum ist das sinnvoll?

- Tests werden nur ausgeführt, wenn der Build erfolgreich war.
- Fehlerhafte Images durch fehlschlagende Tests werden verhindert.
- Fehler im Code stoppen die Pipeline frühzeitig – Ressourcen werden geschont.

2. Backend-Tests: test:java-api

Für das Backend wurde ein eigener Job eingerichtet, der die Unit-Tests via Maven ausführt. Diese Änderung wurde im .gitlab-ci.yml vorgenommen.

```
test:java-api:
  stage: test
  image: maven:3.8.3-openjdk-17
  script:
    - cd backend && mvn test
  artifacts:
    reports:
      junit:
        - backend/target/surefire-reports/TEST-*.xml
    expire_in: "1 week"
  dependencies:
    - build:java-api
  only:
    - merge_requests
    - main
```

Erläuterungen zur Konfiguration

Einstellung	Zweck	Warum?
stage: test	Weist den Job der neuen Test-Stage zu	Klare Trennung der Prozessschritte
image: maven:3.8.3-openjdk-17	Verwendet ein offizielles Maven-Image	Gleiche Umgebung wie beim Build
cd backend	Wechselt ins Backend-Verzeichnis	Tests befinden sich dort
mvn test	Führt die Unit-Tests aus	Standardbefehl für JUnit
artifacts: junit	Liefert Testergebnisse an GitLab	Ermöglicht Auswertung direkt in der CI
dependencies	Wartet auf das Backend-Build	So sind alle nötigen Dateien vorhanden
only	Einschränkung auf relevante Branches	Nur main und MRs werden getestet

3. Frontend-Tests: test:react-app

Auch für das React-Frontend wurde ein Test-Job eingerichtet. Er verwendet Jest mit Code-Coverage. Diese Änderung wurde im .gitlab-ci.yml vorgenommen.

Job-Konfiguration

```
test:react-app:
  stage: test
  image: node:18.12.1
  script:
    - cd frontend && npm install && CI=true npm test -- --coverage --watchAll=false
  artifacts:
    paths:
      - frontend/coverage/
    expire_in: "1 week"
  dependencies:
    - build:react-app
  only:
    - merge_requests
    - main
```

Erläuterungen zur Konfiguration

Einstellung	Zweck	Warum?
stage: test	Zuweisung zur Test-Stage	Einheitlich zum Backend
image: node:18.12.1	Node.js-Umgebung	Gleiche Version wie im Build-Job
npm install	Installiert Abhängigkeiten	Neuinstallation nötig im CI-Job
CI=true	Verhindert interaktiven Modus	Jest läuft automatisch durch
npm test -- --coverage --watchAll=false	Führt Tests & Coverage aus	Coverage ist wichtig für Codequalität
artifacts: coverage	Speichert Coverage-Report	Kann später eingesehen oder veröffentlicht werden

4. Korrektur eines fehlerhaften Frontend-Tests

Ein ursprünglich vorhandener React-Test schlug fehl, weil er auf veralteten Dummy-Content prüfte:

```
test('renders learn react link', () => {  
  render(<App />);  
  const linkElement = screen.getByText(/learn react/i);  
  expect(linkElement).toBeInTheDocument();  
});
```

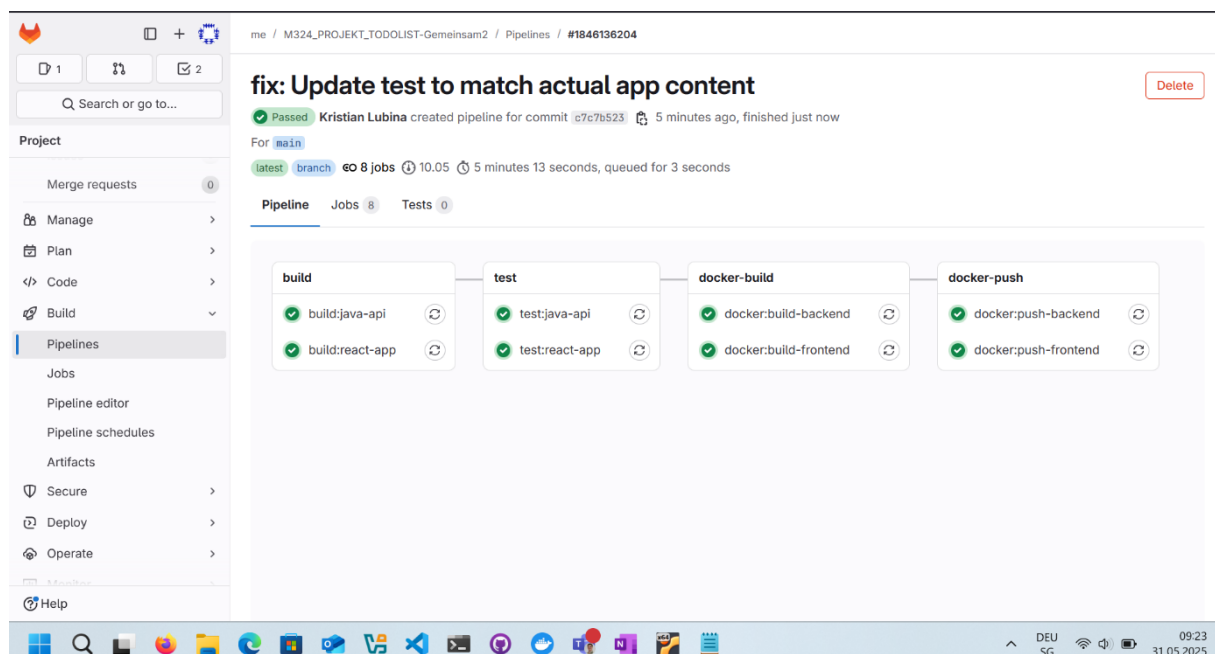
Stattdessen wurde er angepasst, um den tatsächlichen App-Text zu testen:

```
test('renders todo list title', () => {  
  render(<App />);  
  const titleElement = screen.getByText(/ToDo Liste/i);  
  expect(titleElement).toBeInTheDocument();  
});
```


Hinweis: Tests sollten sich immer am realen Output der App orientieren – nicht an Beispielen aus Templates oder Tutorials.

5. Ergebnis: Erfolgreiche Pipeline

Nach der Umsetzung liefen alle Jobs erfolgreich durch. Ein Screenshot dokumentiert den Zustand:



Was alles funktioniert, hat:

-  build:java-api – Backend kompiliert

-  build:react-app – Frontend gebaut
-  test:java-api – Backend-Tests bestanden
-  test:react-app – Frontend-Tests bestanden
-  docker:build-backend – Backend-Image erstellt
-  docker:build-frontend – Frontend-Image erstellt
-  docker:push-backend – Deployment erfolgreich
-  docker:push-frontend – Deployment erfolgreich

Damit ist die automatische Test-Integration in die Pipeline abgeschlossen und sorgt zukünftig für mehr Stabilität und Vertrauen in unseren Code.