030-CI-Pipeline-Rust

DevOps- und Administrationsaufgabe
Anforderungsniveau: Mittel

1 Ziel der Aufgabe

Sie bauen für ein kleines Rust-Backend eine durchgehende CI-Pipeline. Bei jedem Push prüft sie Formatierung, Lint, Tests und einen Release-Build und legt das gebaute Programm als abholbares Artefakt ab. Zusätzlich überwacht die Pipeline die Versionspflege in der `Cargo.toml`: Wer den Code funktional ändert, ohne die Version anzuheben, bekommt das von der Maschine gesagt und nicht erst im Review. Geprüft wird weniger das Backend als das, was darum herum läuft.

Abgrenzung zu Aufgabe 029. Dort ging es um einen abgeschotteten Bau, der alle Bestandteile über einen internen Spiegel bezieht. Hier ist das ausdrücklich nicht gefragt: Die Pipeline darf crates.io, die Registry des CI-Anbieters und den Marktplatz der Aktionen direkt benutzen. Ein eigener Paketspiegel, `cargo vendor` oder ein Bau ohne Weg nach außen bringen in dieser Aufgabe keinen Punkt und kosten nur Zeit. Gefragt ist allein, was die Pipeline selbst prüft und belegt.

Veranschlagte Bearbeitungszeit: 6–8 Stunden

Verwendete Techniken:

- Rust mit `cargo`, ein schlankes HTTP-Backend mit `axum` oder `actix-web`
- `cargo fmt`, `cargo clippy`, `cargo test`, `cargo build --release`
- GitHub Actions oder GitLab CI: Jobs, Abhängigkeiten, Zwischenspeicher, Artefakte
- SemVer und ein eigenes Prüfskript gegen die `Cargo.toml` des Zielzweigs, mit Exit-Codes und dokumentiertem Gegenbeweis je Prüfung

2 Aufgabenbeschreibung

1. **Das Backend**

   Das Backend ist Mittel zum Zweck und darf klein bleiben. Es hält Aufgaben im Speicher und bietet mindestens:

   ```
   GET    /gesundheit        -> 200, {"status":"ok"}
   GET    /version           -> 200, {"version":"0.1.0"}
   GET    /aufgaben          -> 200, Liste aller Aufgaben
   POST   /aufgaben          -> 201 bei gueltiger, 400 bei ungueltiger Eingabe
   GET    /aufgaben/:id      -> 200 oder 404
   ```

   Die Version wird nicht von Hand gepflegt, sondern zur Bauzeit aus dem Manifest gezogen (Makro `env!` mit `CARGO_PKG_VERSION`). Schreiben Sie Tests, die mehr als den guten Fall abdecken: mindestens ein Modultest auf die Validierung und ein Integrationstest für den 400er. `Cargo.lock` wird eingecheckt – es handelt sich um ein Programm, nicht um eine Bibliothek.

2. **Werkzeugkonfiguration ins Repository**

   Formatierer und Lint laufen in der Pipeline mit denselben Regeln wie auf dem Rechner des Entwicklers. Legen Sie `rustfmt.toml` (mindestens `edition`, `max_width`), `clippy.toml` (dort

gehören Schwellen und das `msrv` hinein, passend zum festgelegten Kanal; Lint-Stufen dagegen nach `[lints.clippy]` in der `Cargo.toml`) und `rust-toolchain.toml` mit fest gewähltem `channel` und den Komponenten `rustfmt` und `clippy` ins Repository. Ein Formatierer ohne Projektkonfiguration formatiert bei der nächsten Version des Werkzeugs die halbe Datei um – genau das soll hier nicht passieren.

3. **Pipeline-Gerüst**

   Die Pipeline läuft bei jedem Push und bei jedem Pull- bzw. Merge-Request. Ein laufender Durchlauf desselben Zweigs wird abgebrochen, wenn ein neuer Push kommt. Jeder Job bekommt ein Zeitlimit. Die Beispiele in dieser Aufgabe zeigen GitHub Actions; mit GitLab CI übertragen Sie sie sinngemäß (`rules` statt `on`, `interruptible` statt `concurrency`, `GIT_DEPTH: 0` statt `fetch-depth: 0`, `artifacts:expire_in` statt `retention-days`). Ein öffentliches Repository bei einem der beiden Anbieter genügt; die Ausführung ist dort kostenfrei. Beispiel für GitHub Actions:

```
on:
  push:
    branches: ["**"]
  pull_request:

concurrency:
  group: ci-${{ github.ref }}
  cancel-in-progress: true

env:
  CARGO_TERM_COLOR: always
```

   Aktionen, Abbilder und Toolchain werden mit fester Version angegeben.

4. **Job: Prüfung**

   Ein Job führt die drei statischen Prüfungen als getrennte Schritte aus, damit im Protokoll sichtbar ist, welche gefallen ist. Der Job bricht nicht beim ersten Fehler ab, sondern meldet alle drei Ergebnisse – sonst verteilt sich das Aufräumen über mehrere Durchläufe.

```
cargo fmt --all -- --check
cargo clippy --all-targets --all-features -- -D warnings
cargo test --all-features --verbose
```

   Das `target`-Verzeichnis wird zwischengespeichert (`Swatinem/rust-cache` oder `actions/cache` mit dem Hash der `Cargo.lock` als Schlüssel); die README nennt die Laufzeit kalt und warm. Ein erster Schritt gibt `rustc --version` aus, damit im Protokoll steht, welche Werkzeugkette der Lauf tatsächlich benutzt hat – die Datei aus Punkt 2 soll nachweislich greifen. Eine Matrix über mehrere Kanäle ist hier nicht verlangt.

5. **Job: Release-Build und Artefakt**

   Nach erfolgreicher Prüfung (`needs`) baut ein zweiter Job im Release-Profil und lädt das Ergebnis als Artefakt hoch. Der Name trägt Version und Commit, damit das Archiv später zuzuordnen ist:

```
aufgaben-api-0.2.1-a1b2c3d-linux-x86_64.tar.gz
```

   Dem Archiv liegt eine im selben Job erzeugte `SHA256SUMS` bei; die README zeigt, wie ein Empfänger sie prüft. Das Release-Profil (etwa `lto`, `strip`) ist gesetzt und begründet, die Aufbewahrungsdauer nicht dem Standard überlassen.

6. **Job: Versionspflege**

   Das Kernstück. `skripte/version-pruefen.sh` vergleicht die Version im Arbeitsstand mit der im Zielzweig und entscheidet, ob die Änderung eine Anhebung verlangt.

```
#!/usr/bin/env bash
set -euo pipefail

BASIS="${1:-origin/main}"

neu=$(cargo metadata --no-deps --format-version 1 | jq -r '.packages[0].version')
alt=$(git show "$BASIS:Cargo.toml" | sed -n 's/^version *= *"\(.*\)"/\1/p' | head
    -1)

# 1. Liegen funktionale Aenderungen vor? Geaenderte Dateien unter src/ oder
#    tests/ und Aenderungen an [dependencies] zaehlen als funktional, reine
#    Aenderungen an README.md oder Bildern nicht.
# 2. Ist "neu" gueltiges SemVer (MAJOR.MINOR.PATCH)?
# 3. Ist "neu" echt groesser als "alt"?
```

   Der Vergleich erfolgt nach SemVer-Regeln, nicht als Zeichenkettenvergleich – `0.10.0` ist größer als `0.9.0`, eine Textsortierung sieht das anders.

```
0   Version passt: angehoben, oder keine funktionale Aenderung
1   Aufrufproblem: Basiszweig fehlt, Cargo.toml nicht lesbar
2   funktionale Aenderung vorhanden, Version aber unveraendert
3   Version ist kein gueltiges SemVer
4   Version wurde gesenkt
```

   Jede Fehlermeldung nennt beide Versionen und die Dateien, die den Befund ausgelöst haben – ein blankes „Versionsprüfung fehlgeschlagen“ hilft niemandem. Bei Erfolg genügt `VERSION 0.1.3 -> 0.2.0 ok (12 Dateien unter src/)`. Damit das Skript den Zielzweig sieht, braucht der Checkout die Historie (`fetch-depth: 0`).

7. **Gegenbeweis je Prüfung**

   Eine grüne Pipeline beweist nur, dass sie durchgelaufen ist. Zeigen Sie für jede der vier Prüfungen, dass sie im Fehlerfall rot wird: je ein Zweig mit genau einem eingebauten Fehler, Pipeline laufen lassen, Ergebnis festhalten.

Pruefung	Ausloeser	Job	Ergebnis
cargo fmt	Einrueckung mit 7 Leerzeichen	pruefung	rot, Diff
cargo clippy	ungenutzte Variable	pruefung	rot, Regel
cargo test	400er-Fall liefert 500	pruefung	rot, 1 failed
Versionspflege	src/ geaendert, Version gleich	version	rot, Code 2

   Diese Tabelle ist Teil der Abgabe. Ohne sie ist die Pipeline nicht nachgewiesen.

3 Anforderungen

- `cargo build`, `cargo test` und `cargo run` laufen lokal ohne zusätzliche Handgriffe.
- Die Pipeline läuft bei jedem Push und jedem Pull- bzw. Merge-Request und enthält mindestens die Jobs *Prüfung*, *Versionspflege* und *Bau*; der Bau-Job hängt vom Prüf-Job ab.
- `cargo fmt --check`, `cargo clippy -- -D warnings` und `cargo test` sind einzeln erkennbare Schritte. Ein Clippy-Hinweis lässt die Pipeline scheitern.

- **Kein Formatierer ohne Projektkonfiguration**: die Dateien `rustfmt.toml`, `clippy.toml` und `rust-toolchain.toml` liegen im Repository und werden verwendet.
- `version-pruefen.sh` beginnt mit `set -euo pipefail`, ist ausführbar eingecheckt, belegt die Exit-Codes 0 bis 4 und läuft auch lokal.
- Das Artefakt trägt Version und Commit im Namen, hat eine `SHA256SUMS` und eine gesetzte Aufbewahrungsdauer. Aktionen, Abbilder und Toolchain sind auf feste Versionen gepinnt; der Cache-Schlüssel enthält den Hash der `Cargo.lock`.
- Die Gegenbeweis-Tabelle für alle vier Prüfungen liegt in der README, mit Meldung und Exit-Code.
- Secrets kommen ausschließlich aus dem Secret-Speicher des CI-Systems; `GITHUB_TOKEN` bzw. das Job-Token bekommt nur die nötigen Rechte. Keine echten Zugangsdaten oder Kundennamen im Repository – prüfen Sie auch die Historie, nicht nur den Arbeitsbaum.

4 Abgabe

- Git-Repository mit nachvollziehbarer Historie. Die Versuchszweige aus dem Gegenbeweis bleiben erhalten oder sind in der README verlinkt.
- `README.md` mit Startanleitung (bauen, starten, ein `curl`-Beispiel je Endpunkt), Übersicht der Jobs samt Abhängigkeiten und Laufzeit (kalt und warm), Beschreibung des Versionsskripts mit Exit-Codes und der Abgrenzung funktionaler von redaktionellen Änderungen, der Gegenbeweis-Tabelle, der Prüfung des Artefakts gegen `SHA256SUMS` und einer Begründung des Release-Profils.
- `CHANGELOG.md` mit mindestens zwei Einträgen, die zu zwei tatsächlich vorgenommenen Versionsanhebungen passen. Dazu ein Screenshot oder Protokollauszug eines grünen und eines roten Durchlaufs; `.gitignore` für `target/`.

5 Bewertungskriterien

- **Vollständigkeit der Pipeline** – laufen Format, Lint, Test, Versionsprüfung und Release-Build wirklich bei jedem Push, oder nur auf dem Hauptzweig? Zwischenspeicher, Zeitlimits, Abbruch überholter Durchläufe. Sagt das Protokoll, was kaputt ist und wo?
- **Qualität des Versionsskripts** – korrekter SemVer-Vergleich, saubere Abgrenzung funktionaler von redaktionellen Änderungen, brauchbare Exit-Codes, lokal wie in der Pipeline gleich nutzbar.
- **Nachweisführung** – ist für jede Prüfung belegt, dass sie im Fehlerfall rot wird, oder wird eine grüne Pipeline als Beweis ausgegeben?
- **Artefakt und Nachvollziehbarkeit** – sprechender Name, Prüfsumme, feste Werkzeugversionen, begründetes Release-Profil.
- **Codequalität und Dokumentation** – idiomatisches Rust, Tests, die auch die Fehlerfälle abdecken; lässt sich das Projekt allein anhand der README bauen und prüfen?