CI/CD

- Datenbank Migration
- Code Reviews
- CI/CD Pipeline – Übersicht

Datenbank Migration

Wie läuft die Migration ab?

Die Migration erfolgt mit `pg_upgrade` – dem Standardwerkzeug für In-Place-Upgrades von PostgreSQL-Instanzen.

→ Offizielle Dokumentation zu pg_upgrade

Aufbau der Migration

Die Migration wird bei jeder Benutzeranmeldung ausgeführt. Dabei wird geprüft, ob neue `.sql`-Skripte in `sql/Pg-upgrade2` oder `sql/Pg-upgrade2-auth` vorhanden sind, die noch nicht ausgeführt wurden. Bereits angewendete Migrationen werden in der Datenbanktabelle `schema_info` protokolliert, sodass sichergestellt ist, dass jedes Skript nur einmal ausgeführt wird.

Idempotente Skripte

Der Begriff *idempotent* stammt aus dem Lateinischen:
idem („gleich“) und *potens* („wirksam“).
Ein idempotentes Skript führt bei mehrfacher Ausführung stets zum selben Ergebnis. Das bedeutet, dass sich der Zustand der Instanz nach der ersten erfolgreichen Ausführung durch weitere Ausführungen nicht mehr ändert. Wiederholte Ausführungen haben somit keinen zusätzlichen Effekt.

Beispiel

Betrachten wir folgendes einfache PostgreSQL-Skript:

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    email TEXT UNIQUE
);

CREATE INDEX idx_users_email ON users(email);
```

Wenn dieses Skript zweimal ausgeführt wird, tritt bei der zweiten Ausführung ein Fehler auf. Dies kann die Migration sowie das Update des ERP-Systems verlangsamen oder sogar vollständig verhindern.

Im Folgenden eine sichere, moderne und idempotente Alternative:

```
CREATE TABLE IF NOT EXISTS users (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    email TEXT UNIQUE
);

CREATE INDEX IF NOT EXISTS idx_users_email ON users(email);
```

Vorteile von idempotenter Skripte

Idempotenz bietet den offensichtlichen Vorteil, dass ein Skript mehrfach ausgeführt werden kann, ohne dass sich das Ergebnis verändert. Zwar wirken idempotente Skripte oft komplexer, dennoch lohnt sich der Einsatz dieses Standards.

Weitere, weniger offensichtliche Vorteile sind:

1. Sie sind ideal für komplexe CI/CD-Umgebungen, in denen Software auf mehreren SQL-Instanzen ausgeführt wird.
2. Sie vereinfachen Automatisierung und Rollouts.
3. Sie verhindern Duplikate und Fehler.

Obwohl unser ERP-System durch eine Protokollierung bereits gewährleistet, dass jedes Skript auf einer Datenbankinstanz nur einmal ausgeführt wird, ist Idempotenz in bestimmten Fällen besonders wichtig:

1. Ein Kundenbranch enthält ein sehr ähnliches Migrationsskript.
2. Die Einträge zur Durchführung eines Skripts gehen verloren.
3. Zwei Migrationsskripte führen zum gleichen Ergebnis.

Code Reviews

> ⚠ Dieser Prozess ist nicht final und kann sich jederzeit ändern oder weiterentwickeln.

Grundprinzipien

- Code Reviews erfolgen **pro Ticket**, nicht gesammelt über mehrere Tickets hinweg
- Reviews sollen **schnell und effizient** sein (typischerweise **3–10 Minuten** inklusive eine Feedbackrunde)
  - Wenn Code deutlich länger braucht um verstanden zu werden, ist das ein Hinweis auf ein Problem
  - **Ausnahmen** gelten für komplexe oder kritische Logik (z. B. Algorithmen)

Ziele

Der Fokus liegt auf dem Gesamtbild, nicht auf Detailoptimierungen.

Im Review wird geprüft, ob:

- die Lösung zum bestehenden Code und Aufbau des Systems passt
- die Struktur des Codes nachvollziehbar und sinnvoll ist
- die Umsetzung leicht verständlich ist
- die Änderung keine negativen Auswirkungen auf andere Teile des Systems hat
- die grundlegende Qualität der Lösung stimmt

Ziel ist es, schnell zu beurteilen, ob die Lösung insgesamt sinnvoll und wartbar ist.

Ablauf

Vorbereitung

Vor dem Erstellen eines Commits:

- Sicherstellen, dass der Code funktioniert
- Relevante Tests ausführen
- Änderungen klar und nachvollziehbar halten
- Kurz beschreiben:

- Was wurde geändert?
- Warum wurde es geändert?

Review

Im Review wird vor allem geprüft:

- Ist die Struktur des Codes verständlich?
- Ist die Lösung im Gesamtsystem sinnvoll?

Diskussionen konzentrieren sich auf:

- **Sicherheit**
- **Robustheit**
- **Performance**
- **Einfachheit**

Ergebnis

Ein Review gilt als nicht bestanden, wenn:

- der Code schwer verständlich ist
- die Struktur unklar ist
- Designentscheidungen nicht nachvollziehbar sind

In diesem Fall sind Vereinfachung oder bessere Struktur erforderlich.

Präventive Code Reviews

Bei Unsicherheiten bezüglich Ansatz oder Design sollte frühzeitig Feedback eingeholt werden, bevor mit der Implementierung begonnen wird.

Best Practices

Autoren

- Code einfach und verständlich halten
- Unnötige Komplexität vermeiden
- Änderungen klar pro Ticket trennen
- Vor ein Merge Request sollte der main-Branch im Ticket-Branch gemergt werden

Reviewer

- Fokus auf Architektur und Gesamtbild
- Keine Detaildiskussionen ohne Mehrwert
- Feedback klar und begründet formulieren
- Feedback verfolgt zwei zentrale Ziele:
  1. Verbesserung der Code-Qualität
  2. Wissenstransfer und Weiterentwicklung im Team

Fazit

Code Reviews sind ein schneller, fokussierter Prozess zur Sicherstellung von verständlichem und gut integriertem Code.

CI/CD Pipeline – Übersicht

⚠ Revision 2

Ziel

Diese Pipeline dient als automatischer Merge-Gate für das CE-Repository.
Sie stellt sicher, dass Änderungen vor dem Merge:

- syntaktisch korrekt sind
- konsistent formatiert sind
- grundlegende Qualitätsanforderungen erfüllen
- funktional getestet wurden

Zusätzlich reduziert sie Merge-Konflikte durch frühzeitige, kontinuierliche Integration.

Scope

Diese Version gilt für:

- CE-Repository
- Merge Requests auf:
  - `main`
  - `release-*`

Die Pipeline läuft als Merged Results Pipeline.
Sie testet den Code so, wie er nach dem Merge aussehen würde.

Pipeline-Struktur

Die Pipeline besteht aus zwei Stages:

1. Validate

Schnelle Prüfungen:

- Syntax-Check (`perl -c`)
- Whitespace-Check (Tabs, Trailing Spaces)
- Linting (`perlcritic`)

Besonderheit:

- Diese Jobs laufen **nur für geänderte Perl-Dateien** (`.pl`, `.pm`)
- Wenn keine relevanten Dateien geändert wurden, werden die Jobs bewusst übersprungen

Ziel: frühes, automatisiertes Feedback bei grundlegenden Problemen

2. Testing

Automatisierte Tests über `t/test.pl` mit PostgreSQL-Service.

Ziel: funktionale Korrektheit automatisch sicherstellen

Technische Details

- Gemeinsames Docker-Image für alle Jobs
- Lint-Abhängigkeiten (z. B. `perlcritic`) sind im Image enthalten
- Keine Paketinstallation mehr zur Laufzeit der Jobs
- Klare und einheitliche Job-Ausgaben mit:
  - `START`
  - `FILES`
  - `RESULT`
  - `SUMMARY`

Merge-Regeln

Ein Merge Request darf nur gemerged werden, wenn:

- keine Merge-Konflikte vorliegen
- die Pipeline erfolgreich ist
- ein Code-Review erfolgt ist
- Testing-Freigabe vorliegt

Workflow

1. Branch erstellen (`feature/*`, `improvement/*`, `fix/*`, `hotfix/*`)
2. Merge Request öffnen (frühzeitig)
3. Pipeline läuft automatisch
4. Fehler beheben bis Pipeline grün ist
5. Code Review + QA
6. Merge

Nicht Bestandteil (Revision 2)

Diese Themen sind aktuell nicht enthalten:

- Review Apps
- Deployments
- Build- oder Packaging-Stages
- Push-Pipelines
- EE-Integration

Zielwerte

- Pipeline-Laufzeit: ideal < 10 Minuten
- Fokus: schnell, stabil, verständlich

Ausblick

Mögliche Erweiterungen in späteren Revisionen:

- Release-Pipelines
- Review Apps
- Staging-Deployments
- Integration mit EE-Repository