022-Rust-Preisdaten-Collector

Rust- und Systemaufgabe
Anforderungsniveau: Mittel

1 Ziel der Aufgabe

Sie schreiben einen Dienst in Rust, der in festen Intervallen zwei HTTP-Quellen abfragt und die gelieferten Preise historisiert in PostgreSQL ablegt. Ein solcher Sammler läuft über Monate unbeaufsichtigt, deshalb liegt der Schwerpunkt nicht auf dem Abholen, sondern auf dem, was danach kommt: Wiederholungen mit wachsendem Abstand, Begrenzung der Anfragerate je Quelle und ein Datenmodell, das auch nach dem dritten Neustart keine Dubletten enthält. Beide Quellen laufen auf Ihrem eigenen Rechner, ein fremder Anbieter wird nicht gebraucht.

Veranschlagte Bearbeitungszeit: 6–8 Stunden

- Rust mit `tokio`, `reqwest`, `serde`; PostgreSQL mit `sqlx` oder `diesel`, Schema über versionierte Migrationen
- Wiederholung mit exponentiellem Backoff und Jitter, Ratenbegrenzung je Quelle, geordnetes Herunterfahren, strukturierte Protokollierung

2 Aufgabenbeschreibung

1. **Quellen und Konfiguration**

   Welche Quellen der Dienst abfragt, steht in der Konfiguration, nicht im Code. Sie stellen beide selbst bereit – so hängt nichts an der Verfügbarkeit eines fremden Anbieters, und Fehlerfälle lösen Sie nach Belieben aus. Zwei Antwortformen, als Dateien im Ordner `fixtures/`:

```
# fixtures/fx.json -- Devisenkurse, nur Tagesdatum
{"base": "EUR", "date": "2026-02-17",
 "rates": {"USD": 1.0842, "CHF": 0.9571, "GBP": 0.8536}}
# fixtures/quotes.json -- voller Zeitstempel, Beträge als Text
{"generated_at": "2026-02-17T09:15:00Z",
 "quotes": [{"symbol": "BTC", "quote": "EUR", "last": "61234.55"},
            {"symbol": "ETH", "quote": "EUR", "last": "3288.10"}]}
```

   Ausgeliefert wird das von einem lokalen Server: `python3 -m http.server 8080` im Ordner `fixtures/` genügt. Besser ist ein kleiner eigener Dienst (`axum` oder zwanzig Zeilen Python), der bei jedem Abruf einen frischen Zeitstempel und leicht veränderte Kurse liefert und auf Wunsch `503`, `429` oder eine verstümmelte Antwort zurückgibt – sonst entstehen über mehrere Durchläufe keine neuen Zeilen. Legen Sie ihn mit ins Repository. Einen öffentlichen Endpunkt dürfen Sie zusätzlich eintragen, gefordert ist er nicht.

   Die Konfiguration liegt als TOML-Datei vor, ihr Pfad kommt über Argument oder Umgebungsvariable:

```
[database]
url = "postgres://collector:hunter2@localhost:5432/prices"
max_connections = 5
```

```
[[sources]]
name     = "fx"
url      = "http://127.0.0.1:8080/fx.json"
format   = "fx"
interval = "60s"
timeout  = "5s"
max_requests_per_minute = 3
max_retries = 4
```

Wählen Sie die Werte so, dass sich die Ratenbegrenzung beobachten lässt: eine zweite Quelle mit kurzem Intervall und niedriger Grenze weist nach, dass die Bremse wirkt. Die beiden Antwortformen sind absichtlich verschieden – jede Quelle bekommt einen eigenen Parser auf ein gemeinsames internes Format. Der Rest des Dienstes kennt nur dieses Format.

```
pub struct PriceObservation {
    pub source: String,      // "fx"
    pub instrument: String,  // "EUR/USD" oder "BTC/EUR"
    pub price: Decimal,      // rust_decimal, kein f64
    pub currency: String,    // "USD"
    pub observed_at: DateTime<Utc>,   // UTC
}
```

Für Beträge nehmen Sie `rust_decimal` oder einen ganzzahligen Typ mit festem Nachkommaanteil – Gleitkommazahlen sind für Preise die falsche Wahl.

2. **Datenmodell ohne Dubletten**

   Das Schema entsteht über Migrationsdateien (`sqlx migrate` oder `refinery`), nicht von Hand. Eine Tabelle reicht; den Ausgangspunkt dürfen Sie abwandeln:

```
CREATE TABLE price_observations (
    id          BIGSERIAL PRIMARY KEY,
    source      TEXT NOT NULL,
    instrument  TEXT NOT NULL,           -- "EUR/USD", "BTC/EUR"
    price       NUMERIC(20, 8) NOT NULL,
    observed_at TIMESTAMPTZ NOT NULL,
    fetched_at  TIMESTAMPTZ NOT NULL DEFAULT now(),
    UNIQUE (source, instrument, observed_at)
);
```

   Daraus ergeben sich Anforderungen an das Schreiben:

   - Der Dienst schreibt historisiert: alte Werte werden nie überschrieben oder gelöscht.
   - Zweimaliges Einspielen derselben Antwort erzeugt genau eine Zeile – über die eindeutige Einschränkung und `ON CONFLICT DO NOTHING`, nicht über ein vorgeschaltetes `SELECT`. Zwischen Prüfung und Einfügen liegt eine Lücke.
   - `observed_at` stammt aus der Antwort (Feld `date` bzw. `generated_at`); fehlt ein Zeitstempel, nehmen Sie die Abrufzeit, gerundet auf das Abfrageintervall. Regel je Quelle im README festhalten. Die Devisenquelle liefert nur ein Tagesdatum: Nach dem ersten Durchlauf entstehen dort den Rest des Tages keine neuen Zeilen mehr – kein Fehler, sondern die Dublettenregel bei der Arbeit.
   - Alle Zeitstempel in UTC, `TIMESTAMPTZ` statt `TIMESTAMP`. Die Beobachtungen einer Antwort werden in einer Transaktion geschrieben – entweder alle oder keine.

3. **Zeitplanung und Ratenbegrenzung**

   Jede Quelle läuft in ihrem Intervall; eine hängende Quelle darf die übrigen nicht blockieren.

- Je Quelle eine eigene Aufgabe (`tokio::spawn`) mit `tokio::time::interval`. Achten Sie auf `MissedTickBehavior`, damit sich Abfragen nach einer Störung nicht stauen und gebündelt losfeuern.
- Die Anfragerate ist je Quelle begrenzt (`max_requests_per_minute`) – mit `governor` oder einem `Semaphore` samt Mindestabstand. Auch Wiederholungen zählen dagegen: ein Ausfall darf nicht dazu führen, dass eine Quelle dichter befragt wird.
- Ein `reqwest::Client` wird einmal gebaut und geteilt.

4. **Wiederholung mit Backoff**

   Wiederholt werden Verbindungsfehler, Zeitüberschreitungen, `429` und `500-599`. Nicht wiederholt werden die übrigen `4xx` und Antworten, die sich nicht deserialisieren lassen – die sehen beim nächsten Mal genauso aus.

   - Der Abstand wächst exponentiell ab etwa einer Sekunde, mit Obergrenze, dazu ein zufälliger Anteil (Jitter) gegen den Gleichschritt.
   - Bei `429` respektieren Sie `Retry-After`, sofern vorhanden.
   - Nach `max_retries` vergeblichen Versuchen wird der Durchlauf mit einer Warnung aufgegeben. Der Dienst beendet sich nicht, die nächste planmäßige Abfrage findet statt.
   - Gewartet wird *nicht* mit `std::thread::sleep` – das legt in einer asynchronen Laufzeit den ganzen Arbeitsstrang lahm.

   Sie dürfen eine Bibliothek verwenden (`backon` ist gepflegt) oder selbst rechnen – dann testbar, als reine Funktion aus Versuchsnummer und Zufallswert.

5. **Betrieb**

   - Protokollierung mit `tracing`. Jede Zeile trägt Quelle, Versuchsnummer und Dauer als Felder, nicht als zusammengeklebten Text.
   - Auf `SIGTERM` und `SIGINT` fährt der Dienst geordnet herunter (`tokio::signal`): laufende Transaktionen abschließen, Verbindungspool schließen, Rückgabewert 0.
   - Ein Schalter `-once` (`clap`) führt genau einen Durchlauf je Quelle aus. Ein `docker-compose.yml` startet PostgreSQL; Zugangsdaten in `.env.example`, die echte `.env` bleibt draußen.

6. **Tests**

   Erwartet werden Tests, die das Verhalten unter Störungen belegen:

   - Backoff-Berechnung: wächst der Abstand, bleibt er unter der Obergrenze, liegt der Jitter im erwarteten Bereich?
   - Parser je Quelle gegen die Dateien aus `fixtures/`, dazu eine verstümmelte Antwort: Fehler zurückgeben statt in Panik geraten.
   - Zwei Tests mit `wiremock`: zweimal `503`, dann `200` – der Abruf gelingt nach genau drei Anfragen; und `400` – es findet *keine* Wiederholung statt.
   - Ein Test gegen eine echte PostgreSQL-Instanz (`#[sqlx::test]` oder `testcontainers`), der dieselbe Antwort zweimal einspielt und danach die Zeilenzahl prüft.

```
cargo test && cargo clippy -- -D warnings && cargo fmt --check
```

Halten Sie im README fest, was der Datenbanktest voraussetzt (etwa `DATABASE_URL`), oder trennen Sie ihn so ab, dass `cargo test` ohne Datenbank nicht scheitert.

3 Anforderungen

- Rust-Ausgabe 2021 oder neuer, stabiler Compiler, `cargo clippy` ohne Warnungen. Kein `unwrap()` oder `expect()` auf Betriebspfaden; Fehler mit `thiserror` typisieren, `anyhow` nur außen.
- Preise werden nicht als `f64` gespeichert oder verrechnet. Die eindeutige Einschränkung liegt in der Datenbank, nicht nur in der Anwendungslogik.
- Ratenbegrenzung und Wiederholung sind je Quelle konfigurierbar, nicht im Code verdrahtet.
- Aufbau und Tests kommen ohne Internetzugang aus: Quellen, Beispielantworten und Datenbank laufen auf dem eigenen Rechner.
- Die Version in `Cargo.toml` wird nach SemVer gepflegt und bei jeder funktionalen Änderung mitgezogen – bei uns verbindlich.
- Setzen Sie `rustfmt` oder `clippy` mit eigenen Regeln ein, gehört die Konfigurationsdatei mit ins Repository.
- Keine echten Zugangsdaten oder Schlüssel, weder im Quelltext noch in Tests. Nur existierende Bibliotheken, `Cargo.lock` mit abgelegt.

4 Abgabe

- Ein Git-Repository mit nachvollziehbarer Historie. `README.md` mit Startanleitung (auch für den lokalen Quellserver), Aufbau der Konfiguration, Datenmodell samt Regel gegen Dubletten, Wiederholungsstrategie, Testbefehl und einer Notiz zu dem, was Sie bewusst weggelassen haben.
- Der Dienst läuft nach diesen Schritten ohne weitere Handgriffe:

```
cp .env.example .env && docker compose up -d db
(cd fixtures && python3 -m http.server 8080 &)  # oder Ihr eigener Testserver
cargo sqlx migrate run && cargo run -- --once   # ein Durchlauf je Quelle
cargo run                                       # Dauerbetrieb
```

  Die dritte Zeile steht stellvertretend für Ihr Migrationswerkzeug; was es voraussetzt, gehört in die Startanleitung.
- Eine kurze SQL-Abfrage im README, mit der sich das Ergebnis prüfen lässt. Dazu `.gitignore` für `target/` und `.env`.

5 Bewertungskriterien

- **Datenmodell** – Historisierung, Dublettenfreiheit an der richtigen Stelle, Zeitstempel und Zeitzonen.
- **Wiederholungsstrategie** – wiederholbare gegen endgültige Fehler, wachsender Abstand, Jitter, `Retry-After`.
- **Ratenbegrenzung** – je Quelle, auch für Wiederholungen.
- **Fehlerbehandlung** – typisierte Fehler, keine Panik im Betrieb, geordnetes Herunterfahren.
- **Tests und Codequalität** – Störungsfälle oder nur der glatte Ablauf? Trennung von Abruf, Umwandlung und Ablage.