021-Rust-CLI-Logauswertung

Rust- und Systemaufgabe
Anforderungsniveau: Einfach

1 Ziel der Aufgabe

Sie schreiben ein kleines Kommandozeilenwerkzeug in Rust, das eine Zugriffs-Logdatei zeilenweise einliest und daraus ein paar Kennzahlen berechnet. Geprüft wird weniger der Umfang als die Sorgfalt: ein Parser, der auch die unangenehmen Zeilen aushält, und ein eigener Fehlertyp statt `unwrap`.

Veranschlagte Bearbeitungszeit: 2–3 Stunden.

Verwendete Techniken:

- Rust in der Edition, die `cargo new` anlegt (2021 oder 2024), Aufbau als Binärprojekt mit Bibliotheksteil
- Argumentbehandlung mit `clap` (Derive-API)
- Eigener Fehlertyp mit `thiserror`, Rückgabe über `Result`
- Unit-Tests mit `#[cfg(test)]`

2 Aufgabenbeschreibung

1. **Projekt aufsetzen**

   Legen Sie ein neues Cargo-Projekt an:

   ```
   cargo new logstat && cd logstat
   cargo add clap --features derive
   cargo add thiserror
   ```

   Teilen Sie den Code in `src/lib.rs` (Parser und Auswertung) und `src/main.rs` (Argumente, Ein- und Ausgabe) auf – nur so lässt sich die Auswertung testen, ohne einen Prozess zu starten.

2. **Logformat verstehen**

   Eingabe ist das Combined Log Format, wie es Apache und nginx schreiben. Eine Zeile sieht so aus:

   ```
   192.0.2.10 - - [12/Mar/2024:08:15:42 +0100] "GET /api/kunden?seite=2 HTTP/1.1" 200
       5312 "https://example.invalid/start" "Mozilla/5.0"
   ```

   Die Felder in dieser Reihenfolge: Client-Adresse, Identd (ungenutzt), Benutzername, Zeitstempel in eckigen Klammern, Anfragezeile in Anführungszeichen, Statuscode, Antwortgröße in Bytes (`-` bei leerer Antwort), Referrer, User-Agent. Interessant sind nur vier davon:

   ```
   #[derive(Debug, Clone, PartialEq)]
   pub struct Eintrag {
       pub adresse: String,
   ```

```
    pub pfad: String,
    pub status: u16,
    pub bytes: u64,
}

pub fn zeile_parsen(nr: usize, zeile: &str) -> Result<Eintrag, LogFehler>;
```

Ein Abfrageteil im Pfad (alles ab `?`) wird abgeschnitten, damit `/api/kunden?seite=2` und `?seite=3` als derselbe Pfad zählen; ein `-` im Byte-Feld gilt als 0. Eine fertige Parserbibliothek für Logzeilen ist nicht erlaubt – ob Sie `split`, einen Zustandsautomaten oder `regex` nehmen, bleibt Ihnen überlassen.

3. **Fehlertyp definieren**

   Er unterscheidet zwischen einem Ein-/Ausgabefehler und einer nicht lesbaren Zeile und nennt bei einer kaputten Zeile deren Nummer.

```
#[derive(Debug, thiserror::Error)]
pub enum LogFehler {
    #[error("Datei nicht lesbar: {0}")]
    Datei(#[from] std::io::Error),
    #[error("Zeile {zeile}: unerwartetes Format ({grund})")]
    Format { zeile: usize, grund: String },
}
```

   Im Bibliotheksteil gibt es kein `unwrap`, `expect` oder `panic!` auf Eingabedaten – fehlerhafte Eingaben sind ein erwarteter Fall, kein Programmierfehler.

4. **Kennzahlen berechnen**

   Eine Funktion nimmt die eingelesenen Einträge entgegen und liefert eine Auswertung:

```
pub struct Auswertung {
    pub zeilen_gesamt: usize,
    pub zeilen_ungueltig: usize,
    pub bytes_gesamt: u64,
    pub nach_statusklasse: BTreeMap<String, usize>, // "2xx", "3xx", ...
    pub top_pfade: Vec<(String, usize)>,            // die fuenf haeufigsten
}
```

   Die Rangliste enthält die fünf häufigsten Pfade; bei gleicher Häufigkeit wird alphabetisch nach dem Pfad sortiert, damit zwei Läufe dieselbe Ausgabe erzeugen.

5. **Kommandozeile und Ausgabe**

   Der Aufruf ist `logstat <DATEI>`: ein Pflichtargument, dazu das von `clap` erzeugte `--help`. Ungültige Zeilen werden übersprungen, aber gezählt und am Ende ausgewiesen. Die Kennzahlen gehen nach `stdout`, Warnungen und Fehlermeldungen nach `stderr`. Der Exit-Code ist `0` bei erfolgreicher Auswertung und `1`, wenn die Eingabe nicht lesbar ist (Datei fehlt, keine Rechte).

   Die Ausgabe ist schlichter Text; ausgefeilte Spaltenausrichtung erwarten wir nicht. Ein Vorschlag:

```
Zeilen gesamt: 12841 (davon ungueltig: 7)
Bytes gesamt: 412350992
2xx: 11204   3xx: 912   4xx: 718
Top-Pfade:  /api/kunden 3120  /start 2044
```

6. **Tests schreiben**

   Vier Unit-Tests genügen, aber sie sollen etwas aussagen:

   - eine gültige Zeile wird feldgenau zerlegt
   - `/suche?q=abc` wird zu `/suche` und `-` im Byte-Feld ergibt 0
   - ein Leerzeichen innerhalb der Anführungszeichen verschiebt die folgenden Felder nicht – Status und Bytes bleiben korrekt
   - eine abgeschnittene Zeile liefert `LogFehler::Format` mit der richtigen Zeilennummer

   Schreiben Sie die Logzeilen in den Tests als Rohzeichenkette (`r#"..."#`), sonst ersticken Sie in Maskierungen. Prüfen Sie Fehlerfälle über die Variante, nicht über den Meldungstext.

3 Anforderungen

- Das Projekt übersetzt mit `cargo build` ohne Warnungen.
- Parser und Auswertung liegen in `src/lib.rs` und sind ohne Prozessstart testbar. Ein Fehlertyp mit `thiserror` unterscheidet die Fälle und nennt die Zeilennummer.
- **Kein `unwrap`, `expect` oder `panic!` auf Eingabedaten** im Bibliotheksteil. In Tests sind sie erlaubt.
- Kennzahlen nach `stdout`, Meldungen nach `stderr`. Die Exit-Codes 0 und 1 sind wie beschrieben belegt.
- Mindestens vier Unit-Tests; `cargo test` läuft grün.

4 Abgabe

- Git-Repository ohne `target/` (`.gitignore` anlegen).
- `beispiel/zugriff.log` mit etwa zehn erfundenen Zeilen, darunter eine fehlerhafte. Nur Adressen aus den Dokumentationsbereichen (`192.0.2.0/24`, `198.51.100.0/24`, `203.0.113.0/24`) und `example.invalid`, keine echten Logzeilen.
- Eine kurze `README.md` mit dem Startbefehl (`cargo run -- beispiel/zugriff.log`) und den beiden Exit-Codes. Das Projekt läuft danach ohne weitere Handgriffe.

5 Bewertungskriterien

- **Fehlerbehandlung** – durchdachter Fehlertyp, oder endet jeder Sonderfall in `unwrap`? Bleibt das Programm bei einer kaputten Zeile handlungsfähig?
- **Parserqualität** – werden Anführungszeichen, Leerzeichen im Pfad und das `-` in den Zahlenfeldern korrekt behandelt?
- **Testtiefe** – prüfen die Tests auch die unangenehmen Fälle oder nur die eine Zeile, die ohnehin funktioniert?
- **Trennung der Belange** – ist die Auswertung von Ein- und Ausgabe getrennt, oder steckt die halbe Logik in `main`?