HAFAS-RS

Entwicklung eines in Rust geschriebenen Programms zur Abfrage von "Deutsche Bahn"-Daten

Hintergrund

Die Deutsche Bahn AG (kurz DB oder auch die Bahn) ist das staatliche Eisenbahnunternehmen Deutschlands. Die DB ist eines der größten Transportunternehmen der Welt und der größte Eisenbahnbetreiber in Europa (laut dieser Quelle), wobei ein komplexes Netzwerk aus Fahrplänen, Zügen und Strecken in Deutschland und andere Europäische Länder müssen organisiert werden müssen. Zur Erleichterung der Online-Fahrplanauskunft wird die Software HAFAS (HaCon Fahrplan-Auskunfts-System) der Firma HaCon (Hannover Consulting, gehörend zu Siemens) verwendet. Rust ist eine System-Programmiersprache mit hervorragender Leistung und Sicherheit. Es ist nicht erforderlich, einen Müllsammler[1] oder Referenzzählung[2] zu verwenden, um die Speichersicherheit[3] sicherzustellen. Sicherheit. Rust kann für die Systemprogrammierung verwendet werden und bietet gleichzeitig High-Level-Funktionen. Beispielsweise können mit Rust Programme erstellt werden, um Daten aus unterschiedlichsten Programmierschnittstellen[4] zu ziehen, z.B. HAFAS.

Dokumentation

Der empfohlene Zugriff auf die Dokumentation erfolgt über die generierte HTML-Version im Ordner `docs`. Es gibt sowohl eine `rustdoc` - als auch eine `mdbook` -Dokumentation zu diesem Projekt, die sich im Ordner `docs` befindet.

- Rustdoc
- [en] Buch
- [de] Buch

Über das Projekt

Das Zeil dieses Projekts mit dem Namen HAFAS-RS, für eine Highschool-Präsentation (GFS) von Adrian Struwe (Eskaan) erstellt, war ein Programm zu entwickeln, um gezielt

Daten zu Fahrten, Zugtypen, Standorten und Betreibern der Deutschen Bahn zu sammeln und aufzulisten.

Es ist derzeit unter der GPL3-Lizenz lizenziert, die eine offene Verteilung und Änderung des Codes erlaubt. Das Projekt ist vollständig dokumentiert, sowohl im Quellcode (per rustdoc) als auch in diesem mdbook.

[1] Englisch Garbage Collector [2]: Englisch Reference Counting [3]: Englisch Memory Safety [4]: Englisch API, Application Programming Interface

Ziele

Das primäre Ziel des Projekts war die Extrahierung von Daten, z.B. die meistgenutzten Stationen und sie in einer Datenbank anzuzeigen.

Ein weiteres Ziel war die Abfrage und der Vergleich von Zugfahrplänen zu Echtzeitdaten. Leider wurde mir nach vielen Experimenten klar, dass diese API entwickelt wurde, um solchen Datenerfassungsversuchen zu widerstehen.

In Zukunft möchte ich versuchen, einen praktikablen Weg zu finden (der die API nicht überlastet), um solche Daten zu sammeln.

Projektstruktur

Allgemeine Struktur

Das Projekt gliedert sich in zwei Hauptteile:

- hafas-wrap, die Bibliothek[1] für den Zugriff auf HAFAS[2].

  Es bietet einfache Möglichkeiten, über mehrere Methoden auf das HAFAS zuzugreifen und ein generisches Typsystem, das auch benutzerdefinierte Anforderungs- und Antworttypen außerhalb der allgemeinen Verwendung unterstützt.

- database-cli, das über die Bibliothek abgefragte Daten verwendet und bestimmte Teile zur weiteren Verarbeitung in eine Postgresql-Datenbank einfügt.

  Es enthält derzeit auch die Unterbefehle zum ausgeben eines Diagramms mit den am häufigsten verwendeten Stationen in der Datenbank. **Wenn Sie nach der Nutzung dieses Projekts suchen, müssen Sie hier nachsehen.**

Der größte Teil des Projekts, wenn auch nicht alles, ist in einem prozedualen Stil geschrieben, da dieses Programm auf einem CLI und einer entsprechenden Bibliothek basiert. Es gibt mehr objektorientierte Teile in der Bibliothek.

Weitere Dokumentation

Jeder dieser Projektteile enthält seine eigene `README.md` -Datei, die seine Verwendung weiter erklärt.

Ich empfehle auch, die englische rust-doc Dokumentation für die Dokumentation auf Projektebene zu verwenden, die im Ordner `docs/rustdoc` `./docs/rustdoc/database_cli/index.html` ist.

[1] Englisch Library, in Rust auch Crate genannt. [2]: Hacon Fahrplan Auskunfts System

Die hafas_wrap-Bibliothek

Die Hauptfunktionalität kommt mit dem `Client`-Objekt, da alle Anfragen damit erledigt werden.

Diese Bibliothek stellt auch einige Profile (`hafas_profiles`) für gängige Endpunkte und Anfrage- und Antwortstrukturen bereit, um die Arbeit mit HAFAS-Endpunkten zu vereinfachen. Das Modul `util` bietet Funktionen zum De- und Codieren von Base64-AES-Strings unter Verwendung eines statisch festgelegten Schlüssels und einer md5-Hashing-Funktion, die für Anfragen benötigt werden.

Die Anfragefunktion

Es bietet ein einfaches, anpassbares System für den Zugriff auf HAFAS[1]. Es gibt drei allgemeine Anforderungsparameter:

```
pub async fn request<I: Serialize + Sized, O: DeserializeOwned>(
    &self,
    profile: &HafasProfile,
    requests: Vec<RawHafasRequest<I>>,
) -> Result<Vec<O>, RequestError> {
```

- Ein Verweis auf sich selbst, um auf den Anfrageclient reqwest zuzugreifen, der zum Stellen von Webanfragen verwendet wird
- Ein Verweis auf eine Instanz des Typs `HafasProfile` .

  Ein `HafasProfile` spezifiziert den Endpunkt, eine mögliche Verschlüsselung und zusätzliche Konfigurationsdaten, die mit der Anfrage übergeben werden müssen
- Ein Array, das Instanzen des Typs `RawHafasRequest` enthält

  Der `RawHafasRequest` -Typ (und seine generische Anfrage) müssen `serde::Serialize + Send` implementieren. Diese Implementierungen werden benötigt, damit der Typ in eine JSON-Zeichenfolge serialisiert werden kann, die in der POST-Anfrage an HAFAS[1] verwendet wird.

Die `request` -Funktion versucht auch, die resultierenden Request-Bodys in den spezifizierten generischen Typ `O` zu parsen. Dieser Typ muss `DeserializeOwned` implementieren, damit er von einem JSON-Objekt deserialisiert werden kann.

Anfrageverfahren

Es gibt zwei Methoden zum Anfordern von Daten: Entweder `request` oder `request_raw`, wobei `request` hauptsächlich ein Wrapper um die `request_raw`-Methode ist, um das Parsen der Ergebnisse zu erleichtern.

Hier ist eine Zusammenfassung dessen, was die Funktionen tun:

Beide:

1. Kopieren des `HafasProfile` in ein JSON-Objekt,

   Serialisieren der Anfragen in ein JSON-Objekt und setzen des `svcReqL`-Wert der Anfrage darauf.

   Serialisieren des JSON-Objekt in einen String

   ```
   let mut req_values = profile.config.clone();
   req_values["svcReqL"] = serde_json::to_value(&requests).unwrap();
   let req_string = serde_json::to_string(&req_values)?;
   ```

2. Optional das erstellen einer Prüfsumme für die GET-Parameter der Anfrage.

   ```
   let checksum = match &profile.secret {
       Some(secret) => util::hash_md5(&(req_string.clone() + secret)),
       None => String::new(),
   };
   ```

3. Antworttext einer Webanfrage an HAFAS[1] erstellen, senden und abwarten.

   ```
   let response = self
       .reqwest_client
       .post(profile.url)
       .header("Accept", "application/json")
       .header("Content-Type", "application/json")
       .query(&[("checksum", &checksum)])
       .body(req_string)
       .send()
       .await?;
   ```

`request_raw` gibt an dieser Stelle den Antwort-String zurück, während `request` mit dem Parsen der Antwort fortfährt:

Nur `Anfrage`:

5. Prüft, ob HAFAS[1] eine Fehlermeldung zurückgegeben hat.

```
        if res.get("err") != Some(&Value::String(String::from("OK"))) {
            return Err(RequestError::InternalError(
                res.get("err")
```

6. Deserialisiert die Anfrage in ein JSON-Objekt und versucht, das Antworttextsegment als Array abzurufen.

```
        res.get("svcResL")
            .ok_or(RequestError::DeserializeError(
                "Response not an Object or svcResL does not exist in it.",
            ))?
            .as_array()
            .ok_or(RequestError::DeserializeError("svcResL is not an
    array."))?
```

7. Deserialisiert den Array von JSON-Objekten in den angegebenen generischen Typ `O`.

```
            .iter()
            .map(|r| {
                Ok(serde_json::from_value(
                    r.get("res")
                        .ok_or(RequestError::DeserializeError(
                            "field res does not exist. This is most likely a
    request error.",
                        ))?
                        .clone(),
                )?)
            })
            .collect::<Result<Vec<O>, RequestError>>()
```

HAFAS-Profile

Im Modul `hafas_profiles` sind mehrere Konfigurationsvoreinstellungen für verschiedene HAFAS[1]-Endpunkte zu finden.

Diese wurden aus verschiedenen anderen Open-Source-Projekten gesammelt.

Methoden

Im Modul `Methoden` finden Sie einige Voreinstellungen für Request- und Response-Strukturen. Man muss sie nicht verwenden, aber sie wurden getestet und scheinen für die meisten Endpunkte zu funktionieren.

Derzeit verfügbare Methoden:

- HimSearch
- JourneyMatch
- Reisedetails

Dienstprogramme

Einige verschiedene Funktionen wie AES-Verschlüsselung mit einem festgelegten Schlüssel oder das Hashen eines Strings mit dem `md5` -Digest-Algorithmus.

[1] Hacon Fahrplan Auskunfts System

Die Datenbank-CLI-Toolchain

Dies ist derzeit die einzige Implementierung der oben genannten Bibliothek.

Die aktuelle Verwendung besteht darin, eine Datenbank mit Fahrten, Zugtypen, Standorten und Betreibern zu erstellen. Das genaue Layout findet man unter `database_layout.png` im Stammverzeichnis des Projekts. Man kann auch ein einfaches Diagramm aus Nutzungsstatistiken erstellen.

Obwohl das Programm keine Live-Datenbank zum Kompilieren benötigt, benötigt es eine Postgres-Datenbank, um zu funktionieren. Die Datenbank muss auf localhost liegen, mit dem Namen `db-statistics` und dem Benutzer `postgres` (passwortlos) auf alle Tabellen zugreifen zu können, die im Datenbanklayout angegeben sind. Um die Datenbank für die Verwendung vorzubereiten, führen Sie `database-climigrate` aus. Dadurch werden alle erforderlichen Schemas, Tabellen, Typen und Funktionen erstellt.

Die genaue Verwendung finden Sie in der Rustdoc-Dokumentation der Methode [`main`] oder im Abschnitt über die Nutzung im Buch. Diese Crate ist größtenteils in einem prozeduraleren (aber auch teilweise objektorientierten) Programmierstil geschrieben, wie es für eine CLI-Schnittstelle zu einer Bibliothek üblich ist. Daraus resultieren viele Module, die meist einzelne Funktionen anstelle ganzer Objekte enthalten.

Für die Modulebene empfehle ich wieder die Rust-Doc-Dokumentation für diese Crate, die hier verfügbar ist.

Besser strukturierte Informationen zu den CLI-Befehlen finden Sie in der Nutzungssektion dieses Buchs.

Main

Dies ist das Wurzelmodul[1] der Crate, hier liegt die Funktion, die beim Programmstart aufgerufen wird. Dieses Modul analysiert die CLI-Argumente und ruft die entsprechenden Funktionen auf.

##request_raw_jids Dieses Modul ist für den Unterbefehl `data request_raw` verantwortlich. Es fordert Rohdaten von HAFAS an und schreibt sie in die Tabelle `raw_data` .

parse_raw_jids

Dieses Modul wird vom Unterbefehl `data parse` aufgerufen und analysiert Daten aus den `raw_data` und fügt sie in die anderen Tabellen `trips`, `locations`, `operators` und `train_types` ein.

count_location_trips

Dieses Modul ist eine Erweiterung des Unterbefehls `data parse`, der mit `dara parse_heatmap` aufgerufen wird.

vergleiche_rohdaten

Dieses Modul wird vom Unterbefehl "Datenprüfung" aufgerufen. Es verwendet das Modul `request_raw_jids`, um ein einzelnes JID anzufordern und es mit der Tabelle `raw_data` zu vergleichen.

create_heatmap_diagram

Dieses Modul enthält neben dem Herumspielen mit Datenbankdaten den einzigen praktischen Nutzen, den ich bisher gefunden habe. Es wird vom Unterbefehl `create_heatmap` aufgerufen und erstellt ein horizontales Diagramm, das die am häufigsten verwendeten Stationen in der Datenbank anzeigt.

[1] Englisch Root Module, in Java die Klasse mit main Funktion.

Installation

Abhängigkeiten

Dieses Programm ist abhängig von:

- Ein laufender Postgresql-Server auf `postgres://postgres@localhost/db-statistics`
  Getestet mit Postgresql 13
- Stabile version von Rust
- Eine funktionierende Internetverbindung

Schnellstartanleitung

Information: Der größte Teil dieser Anleitung funktioniert nur unter Linux. Weitere Informationen zu anderen Systemen finden Sie in der Postgresql-Dokumentation und auf der Rust-Website.

1. Zuerst müssen Sie postgresql installieren:
    1. Installieren Sie das Datenbankpaket auf Ihrem System:
        - Arch: `sudo pacman -S postgresql`
        - Debain & Ubuntu: `sudo apt install postgresql`
        - Sonstige: Postgresql-Installationsanleitung
    2. Melden Sie sich dann bei dem neu erstellten Benutzer an: `sudo su postgres`
    3. Erstellen Sie einen neuen Datenbank-Cluster mit `initdb -D Verzeichnis/to/store/data` .
    4. Erstellen Sie im Cluster eine neue Datenbank mit dem Namen db-statistics im Cluster: `createdb db-statistics`
    5. Starten Sie den Postgresql-Server (normalerweise `sudo systemctl start postgresql`)
2. Kompilieren Sie das Projekt:
    1. Installieren Sie das Rustup-Paket:
        - Arch: `sudo pacman install rustup`
        - Debial & Ubuntu: `sudo apt install rustup`
        - Sonstige: Rustup-Installationsanleitung
    2. Führen Sie `rustup install stable` in Ihrem Terminal aus.
    3. Führen Sie `cargo build --release` im hafas-rs-Ordner aus. Dadurch sollte ein Ordner mit dem Namen `target/release` erstellt werden, der eine ausführbare Datei mit dem Namen `database-cli` enthält. Herzlichen

Glückwunsch, Sie haben das Projekt gerade erfolgreich kompiliert!

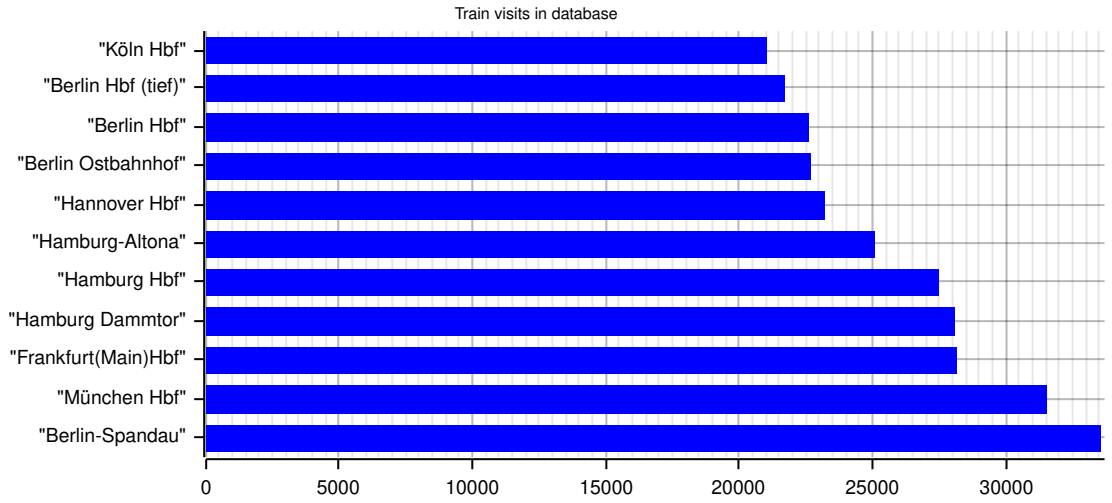
3. Verwenden Sie dann `database-cli migrate`, um die Verbindung zur lokalen Datenbank zu überprüfen und die notwendige Infrastruktur wie Schemas, Tabellen und Funktionen zu erstellen.
4. Verwenden Sie `database-cli data request_raw 2000000 --parse`, um Daten anzufordern und in die Datenbank zu parsen. Es wird eine Weile dauern, bis alle ~1,5 Millionen Einträge abgerufen sind. Ich empfehle, diesen Befehl über Nacht oder zu anderen Zeiten in denen HAFAS nicht stark ausgelastet ist auszuführen.
5. Sie sind bereit, mit den Daten in Ihrer bevorzugten SQL-Eingabeaufforderung zu spielen! Weitere Informationen zur Datenbankstruktur finden Sie im HAFAS-Abschnitt.

(Optional)

6. Verwenden Sie `database-cli data parse_heatmap`, um die Eva[1]-Zählungen der Reise in ein leichteres Format zu parsen.

7. Verwenden Sie `database-cli create_heatmap -m 10`, um ein Diagramm der 10 am häufigsten verwendeten Stationen zu erhalten.

   Nachdem Sie diese Schritte ausgeführt haben, sollte das Programm ein Bild mit dem Namen `heatmap.svg` im aktuellen Ordner erstellen. Hier ist ein Beispiel dafür:



[1] Internationale Bahnhofsnummer

Verwendung

Wie der Name der Crate schon sagt, kompiliert `database-cli` in ein Kommandozeilenprogramm[1]. Diese Schnittstelle arbeitet mit einer Postgresql-Datenbank zusammen, um HAFAS-Daten zu speichern und anzufordern.

Liste der Unterbefehle:

Alle Befehle können mit -v oder -vv übergeben werden, um sie ausführlicher zu machen. Das Argument `--help` kann zu jedem Unterbefehl hinzufügt werden, um Details zu den Argumenten anzuzeigen. Wenn Sie die CLI nicht kompilieren möchten, finden Sie Argumente im Quellcode.

▸ Verantwortlicher Code in `main.rs - zum Erweitern klicken`

- `data` Dieser Befehl kann nur zusammen mit einem seiner Unterbefehle verwendet werden, er hat keine eigenen Eigenschaften. Es sammelt im Allgemeinen Befehle, die verwendet werden, um Suchdaten für Zugfahrten in der Datenbank zu verschieben.
    - `request_raw` : Dieser Befehl fordert rohe Fahrplandaten über alle JIDs im HAFAS-Endpunkt an. Unabhängig vom `TO` -Argument gerät der Befehl in Panik, wenn er das letzte Jid erreicht. Der übliche letzte Jid liegt bei etwa 1,5 Millionen.

      Dies ist normalerweise der zweite Befehl, der nach `migrate` ausgeführt wird.

      Es ist zu beachten, dass dieser Befehl abhängig von Ihrem Computer und Ihrer Netzwerkverbindung normalerweise ziemlich lange dauert, bis er fertig ist. Es wird empfohlen, obwohl es nicht notwendig ist, es mit `--parse` aufzurufen.

      Es wird empfohlen, alle anderen optionalen Flags auf einem Standardwert zu belassen, um ein Timeout zu verhindern.
    - `parse` : Dieser Befehl parst die Daten aus der `raw_data` -Tabelle in ein verwendbares Format und fügt sie in die anderen Tabellen ein. Es kann auch automatisch aufgerufen werden, indem `--parse` zu den Argumenten von `request_raw` hinzugefügt wird.

      Sie können sich das Datenbankschema und hauptsächlich den [Hafas-Abschnitt] (./hafas.html) ansehen, um weitere Details darüber zu erhalten, wie Daten geparst werden.
    - `parse_heatmap` : Dieser Befehl sollte zu einem bestimmten Zeitpunkt

aufgerufen werden, bevor die Funktion `create_heatmap` verwendet wird. Es zählt alle aufgezeichneten Zugfahrten in einer eigenen Tabelle für einen schnelleren Zugriff zusammen.

Da naher und lokaler Verkehr das Endbild verschleiern kann, empfehle ich, `-o 'ICE'` als Filter zu setzen.

- `check` : Dieser Befehl prüft, ob Daten vom HAFAS-Endpunkt von den aktuellen Daten abweichen. Diese Prüfung wird nur für einen einzigen Jid durchgeführt. Ein Unterschied könnte auf eine Fahrplanänderung hindeuten.

- `create_heatmap` Erstellt ein horizontales Balkendiagramm der am häufigsten verwendeten Stationen in der Nachschlagetabelle. Es kann nach cat_code, cat_out und Suchlimit gefiltert werden. Aus derzeit unbekannten Gründen bringt alles über 11 Balken die Stationsnamen durcheinander.

- `migrate` : Erstellt die gesamte notwendige Infrastruktur auf der entfernten Datenbank. Das CLI selbst arbeitet derzeit nur mit dem lookup_data-Schema, aber dieses `sqlx` -Feature benötigt

Die meisten Befehle führen zu einem Fortschrittsbalken wie diesem:

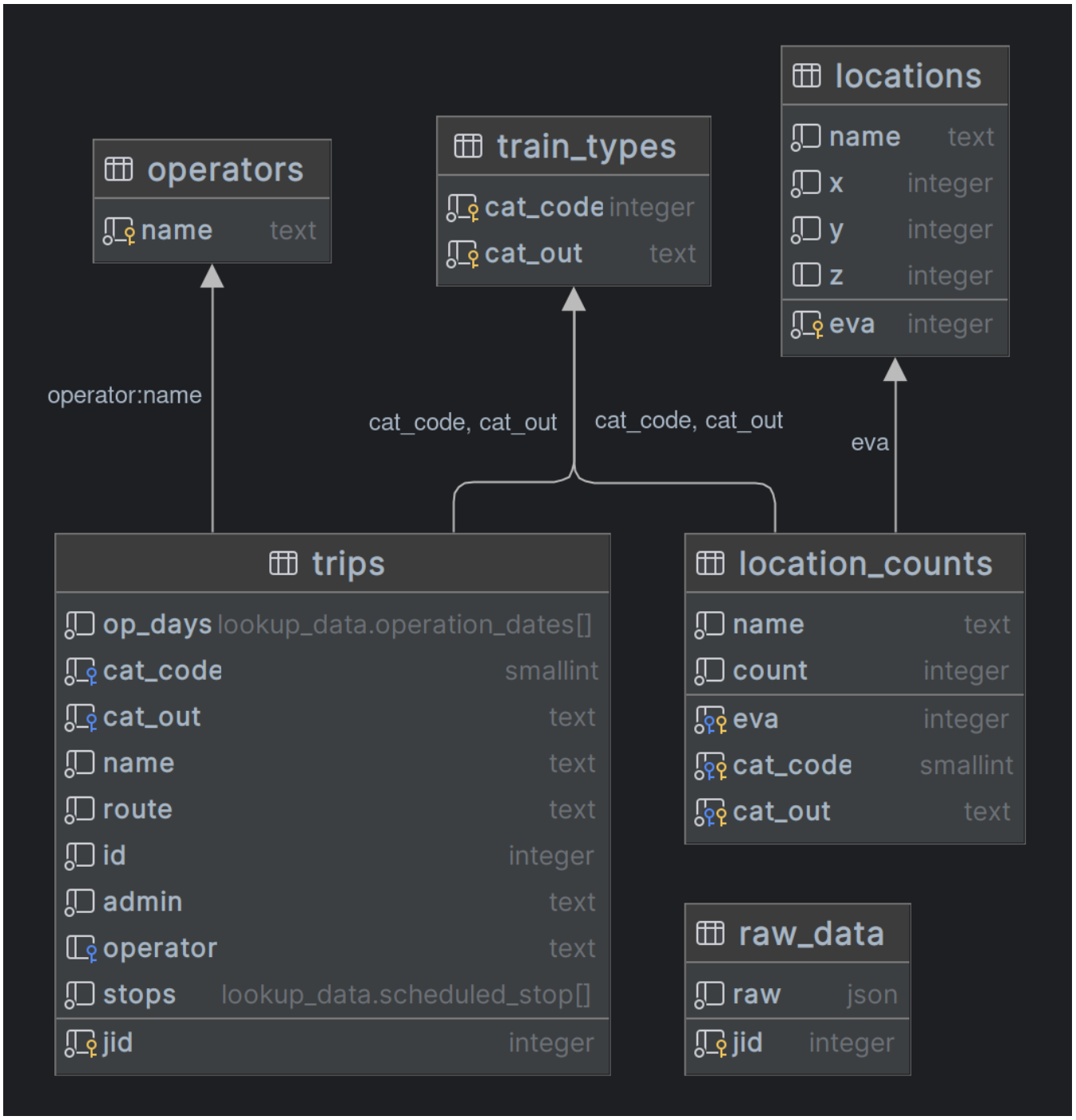
```
Parsing [=======>                                  ] 1485/9900
```

Die Zählungen am Ende sollen einen ungefähren Hinweis darauf geben, wie lange der Befehl dauern wird.

[1] Englisch CLI, Command Line Interface

HAFAS und Datenbanklayout

Datenbanklayout



▸ DDL für Datenbanklayout - zum Erweitern klicken

Die gesamte Datenbank ist derzeit im lookup_data-Schema enthalten.

Das allgemeine Schema ist oben dargestellt, mit einigen benutzerdefinierten zusammengesetzten Typen und Funktionen, die hier erklärt werden:

- `operation_dates` ist ein Typ, der ein Array von `dates`, ein `from_loc`-Eva mit entsprechendem `to_loc`-Eva und einen `Info`-Text enthält.
- `scheduled_stop` ist ein Typ, der einen Halt einer Fahrt an einer Station beschreibt. Es enthält die `eva`, die `scheduled_arrival`-Zeit und die entsprechende `scheduled_departure`-Zeit.
- `from_evas` ist eine Funktion, die ein Array von `scheduled_stop`-Elementen verwendet und nur das `eva` als int-Array daraus extrahiert.

Zukunft des Projekts

Ich plane, meine Arbeit an diesem Projekt fortzusetzen und vielleicht eines Tages eine gute HAFAS-Bibliothek für Rust zu entwickeln. Dies ist das primäre Ziel für die Zukunft, danach plane ich, eine öffentliche API zu erstellen, die besser lesbare Antworten liefert als HAFAS.

Ich freue mich immer über Issues und Pull-Requests zu meinen Projekten. Wenn sich dieses Projekt durchsetzt, werde ich es sicherlich genießen, es zu pflegen.