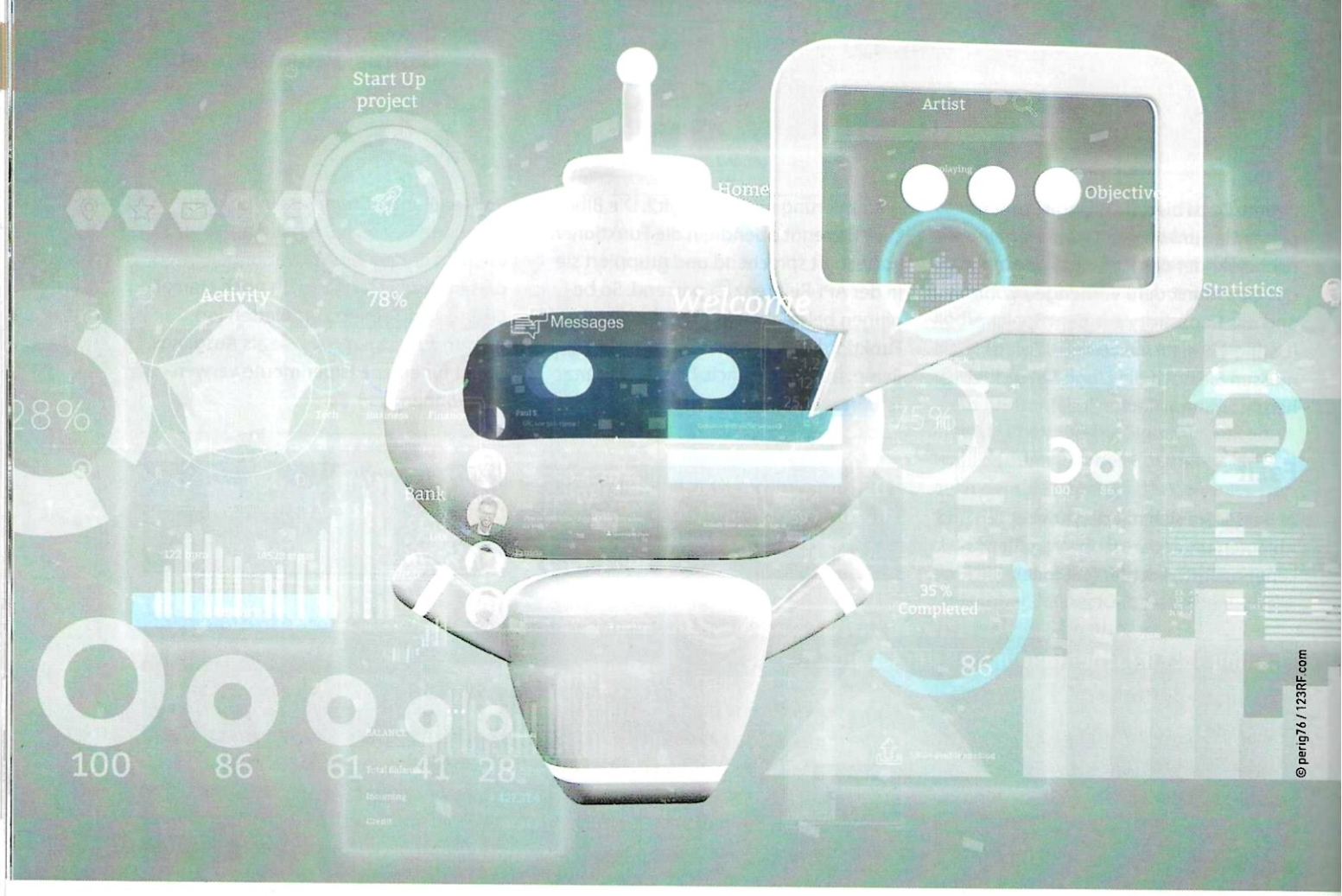

Planet Rust, Folge 9: Einen Chatbot in Rust bauen

Kleines Pläuschchen

ChatGPT mit Rust zu nutzen, erfordert eine HTTP-Kommunikation kombiniert mit asynchroner Verarbeitung. Es gibt dafür spezielle Schnittstellenbibliotheken, alternativ bietet Rust Standardpakete wie Reqwest und Tokio. Damit erstellen Sie sogar einen Chatbot. Gerhard Völkl

Vor mehr als einem Jahr besuchte mich Tobias, ein guter Bekannter und freier Programmierer. Er zeigte sich restlos begeistert von ChatGPT: „Das Ding erstellt jetzt alle meine Shell-Skripte – einfach so!" Da ich seit 20 Jahren im Bereich des maschinellen Lernens tätig bin, aber bis dahin mit ChatGPT direkt nicht in Kontakt gekommen war, dachte ich mir: „Manchen kann man schon mit wenig eine Freude machen." Tobias holte sein Notebook heraus und gab eine Anfrage nach der nächsten ein. Mit jeder Antwort von ChatGPT wuchs meine Überraschung – einen solchen Quantensprung in diesem Bereich der KI hatte ich nicht erwartet.

Seitdem habe ich mich intensiv mit der in ChatGPT und ähnlichen Programmen verwendeten Technologie beschäftigt: Large Language Models (LLM). Mit der Zeit reifte in mir der Plan, ChatGPT in meine eigenen Rust-Programme zu integrieren. Damit war eine neue Herausforderung geboren: einen eigenen einfachen Chatbot mit Rust zu erstellen.

ChatGPT-API

Sämtliche Dienste im Web, egal, ob es sich um einen HTML-Server, eine Cloud-SQL-Datenbank oder eben um ChatGPT handelt, funktionieren nach demselben Schema: Sie senden eine Anfrage und erhalten dann ein Ergebnis zurück.

Die KI-Hersteller offerieren ihre Dienste nicht kostenlos. Damit Sie mit einem Programm eine Anfrage schicken können, benötigen Sie einen Schlüssel (API-Key), über den der Anbieter abrechnet. Den API-Key erhalten Sie für ChatGPT unter https://platform.openai.com/apps. Jede Anfrage kostet abhängig von ihrem Umfang und der verwendeten ChatGPT-Version einige Cents. Da ältere Versionen von ChatGPT günstiger sind, ergibt es durchaus Sinn, zunächst damit zu entwickeln und erst später auf aktuelle Versionen umzusteigen.

Um keine Verwirrung zu stiften, konzentriere ich mich in diesem Artikel ganz auf OpenAI mit ChatGPT 1. Der Zugriff auf andere LLMs funktioniert ähnlich. Am Ende des Artikels zeige ich Ihnen zudem, wie Sie mit der freien Software Ollama einen eigenen Server lokal aufsetzen, zum Beispiel für Llama 3 von Meta. Solche Aufrufe kosten gar nichts, doch der verwendete Computer muss die entsprechende Rechenleistung mitbringen.

Für die Programmiersprachen Python und Javascript gibt es offizielle API-Umsetzungen für ChatGPT vom Hersteller. Auf der zentralen Webseite für Rust-Bibliotheken, Crates.io, finden Sie dafür unterschiedliche Implementierungen. Da es in diesem Artikel primär um die HTML-Kommunikation mit einem Server geht, nutze ich nicht diese inoffiziellen Rust-Bibliotheken, sondern steuere den Server direkt auf der untersten Ebene an.

HTTP-Kommunikation

Eine einfache Möglichkeit zur Datenübertragung per HTTP an einen Server bietet das Kommandozeilenprogramm Curl. In Listing 1 stellt es eine Frage an den ChatGPT-Server.

Als erster Parameter fungiert die Adresse des ChatGPT-Servers (der Endpoint) für Chat-Aufrufe. Die beiden folgenden Zeilen mit vorangestelltem `-H` enthalten die Header-Informationen des Datenpakets. Weiterhin steht der Content-Type `application/json` für den Datentyp des Paketinhalts im JSON-Format. Die zweite Zeile des Headers enthält den in der Shell-Variablen `OPENAPI_API_KEY` hinterlegten Zugriffsschlüssel.

Nach `-d` folgt das eigentliche Datenpaket. Der Slot `model` enthält die Version von ChatGPT, mit der Sie arbeiten wollen. Im Slot `messages` befindet sich die gesamte Chat-Kommunikation. Das `system` (Zeile 7), in diesem Fall ChatGPT, hat zunächst die Nachricht `You are a helpful assistant` bekommen. Der `user` stellt in Zeile 11 die Gretchenfrage nach der besten Programmiersprache.

Nach dem Absenden der Anfrage verstreicht ein wenig Zeit, bis ChatGPT den Text aus Listing 2 an Curl zurückgibt, ebenfalls im JSON-Format. Im Mittelpunkt steht dabei der Slot `content` mit der eigentlichen Antwort. Den ersten Schritt zu einem eigenen Chatbot in Rust bildet ein Programm, das ungefähr dasselbe tut wie der Curl-Aufruf: eine Anfrage an einen Server versenden und die Antwort verarbeiten.

1 Neben der gewohnten Weboberfläche bietet ChatGPT eine API an, mit der Programme direkt Anfragen stellen.

Reqwest

Das meistverwendete Rust-Paket (Crate) für den Zugriff per HTML auf Server ist Reqwest. Listing 3 zeigt, wie Rust damit den Inhalt einer Webseite liest und sie in der Konsole ausgibt.

Die komplette Kommunikation erfolgt über das zentrale Reqwest-Objekt `Client` (dritte Zeile). In der nächsten Zeile definiert das Programm mithilfe der Methode `client.get("URL")` den Server, auf den es zugreifen will. Die Methode `send()` stößt die Übertragung an. Da sie schiefgehen kann, kommen Sie nicht um eine passende Fehlerbehandlung herum. Die einfachste Möglichkeit dazu bietet die Funktion `unwrap()`: Sie bricht bei einem Fehler schlicht das Programm ab.

Wenn alles wie gewünscht läuft, wandelt das Programm das zurückgelieferte Ergebnis in einen String um. Dieser Vorgang kann allerdings ebenfalls fehlschlagen, weswegen Sie erneut auf `unwrap()` zurückgreifen. Das Ergebnis landet schließlich in der Variablen `result`, die das Programm anschließend ausgibt (vorletzte Zeile).

Als durchaus kritisch erweist sich bei der Kommunikation mit Servern das Warten auf die Antwort. Denn hier stellt sich die Frage, was das Client-Programm währenddessen tun soll. In unserem Beispiel dreht es schlicht Däumchen – es ist blockiert (`blocking`, erste Zeile). Besäße das Programm eine Oberfläche, käme sie zum Stehen. Die Alternative zu diesem als synchron bezeichneten Vorgehen wäre eine asynchrone Verarbeitung. Sie wartet nicht etwa auf die Rückmeldung des Servers, sondern arbeitet bis zum Eintreffen der Antwort weiter und kümmert sich dann darum.

Listing 1: Abfrage via Curl

```
01 $ curl https://api.openai.com/v1/chat/completions \
02     -H "Content-Type: application/json" \
03     -H "Authorization: Bearer $OPENAI_API_KEY" \
04     -d '{ "model": "gpt-3.5-turbo", \
05           "messages": [ \
06             { \
07             "role": "system", \
08             "content": "You are a helpful assistant." \
09             }, \
10             { \
11             "role": "user", \
12             "content": "What is the best programming language?" \
13             } \
14         ]}'
```

Asynchron mit Tokio

Für die asynchrone Verarbeitung in Rust stehen die Direktive `async` und die Funktion `await()` (Listing 4) zur Verfügung. Dabei zeigt zunächst `async` an, dass Rust eine Funktion asynchron verarbeiten soll, und `await()` markiert die Stelle, an der das Programm auf etwas wartet.

Eine asynchrone Funktion ist in Rust flott definiert – schwierig wird es, wenn Sie sie ausführen möchten. Wenn Sie sie aufrufen wie jede andere Funktion, führt Rust sie nicht aus, sondern liefert stattdessen eine Datenstruktur zurück, die auf die asynchrone Funktion verweist.

Obwohl Rust standardmäßig `async` und `await()` anbietet, fehlt die Information, wie solche Funktionen auszuführen sind. Der Hintergrund: Da Rust auf den diversen Plattformen vom Mikrochip bis hin zum Multiprozessorrechner agiert, gibt es keine optimale Art, mit asynchroner Verarbeitung umzugehen. Die Details übernehmen Bibliotheken wie Tokio, Async-std oder Smol. Ich habe mich für die Library Tokio entschieden, da sie momentan auf Desktop-Rechnern am häufigsten zum Einsatz kommt.

Beim Arbeiten mit Tokio müssen Sie im ersten Schritt eine Runtime erzeugen, die sich um das Ausführen der asynchronen Funktionen kümmert (Listing 5). Mit der Methode `block_on()` erhält Tokio die asynchrone Funktion und führt sie aus. Sie sparen sich das explizite Erzeugen der Laufzeitumgebung sowie den Aufruf der Methode `block_on()`, indem Sie die Funktion `main()` genauso definieren wie in den ersten beiden Zeilen von Listing 6. Zusätzlich demonstriert das Listing, wie das Lesen einer HTML-Seite beim asynchronen Zugriff aussieht.

Der große Vorteil von Tokio zeigt sich, wenn ein Programm viele asynchrone Aufrufe möglichst gleichzeitig verarbeiten soll (Listing 7). Durch die Methode `spawn()` weiß `tokio`, dass es die asyn-

Listing 2: ChatGPT-Ausgabe

```
{
  "id": "chatcmpl-9GPcrEW4R6JwpMJQZoe4z8QZFexf5",
  "object": "chat.completion",
  "created": 1713698781,
  "model": "gpt-3.5-turbo-0125",
...
  "content": "The best programming language depends on the specific
needs and requirements of a project.
  Different programming languages have different strengths and are
suited to different types of tasks.
  Some popular programming languages include Python, Java, JavaScript,
C++, and Ruby.
  It's recommended to choose a programming language based on the
particular project you are working on
  and your own familiarity with the language."
...
}
```

Listing 3: Reqwest.rs – HMTL-Seite auslesen

```
01 use reqwest::blocking::Client;
02 fn main() {
03   let client = Client::new();
04   let result = client.get("https://www.rust-lang.org")
05     .send().unwrap()
06     .text().unwrap();
07   println!("{}",result);
08 }
```

Listing 4: Asynchrone Verarbeitung

```
async fn get_result() {
  Client.get(...).await().unwrap()
}
```

Listing 5: Tokio

```
let mut rt = tokio::runtime::Runtime::new().unwrap();
rt.block_on(meine_async_fn());
```

Listing 6: Reqwest_async.rs

```
01 use reqwest::Client;
02 #[tokio::main]
03 async fn main() {
04   let client = Client::new();
05   let result = client.get
   ("https://www.rust-lang.org")
06     .send().await.unwrap()
07     .text().await.unwrap();
08   println!("{}",result);
09 }
```

Listing 7: Parallelverarbeitung

```
let _ = tokio::join!(
  tokio::spawn(get_html("http://test.com")),
  tokio::spawn(get_html("http://test1.com""))
);
```

chrone Funktion `get_html()` möglichst effizient ausführen soll. Während es beispielsweise auf den ersten Server wartet, führt es den Aufruf für den zweiten Server aus. Der Befehl `join` wartet, bis alle Tasks abgeschlossen sind.

Tokio bietet viele weitere Methoden, mit denen Sie Funktionen nahezu gleichzeitig ausführen können, ohne sich dabei um viel kümmern zu müssen.

Fehlerbehandlung

Das Programm `openai.rs` – Sie finden es im Download-Bereich zu diesem Artikel – erledigt dasselbe wie der Curl-Aufruf am Anfang dieses Artikels. Auf den ersten Blick sieht es etwas kompliziert aus, aber einige Hintergrundinformationen sorgen für mehr Verständnis.

Zu den großen Themen bei der Kommunikation via Web zählt die Fehlerbehandlung. In Rust stehen dazu grundsätzlich zwei Wege offen: Das Programm selbst kümmert sich entweder um den Fehler (Recoverable Errors) oder bricht einfach ab (`panic!`). Dass ein Programm auf den Fehler reagiert, ist der grundsätzliche Anspruch an professionelle Software, doch die Fehlerbehandlung macht alles etwas unübersichtlicher.

Übernimmt eine Funktion in einem Rust-Programm die Fehlerbehandlung, liefert sie ein Ergebnis vom Typ `Result` zurück (Listing 8). Geht alles gut, weist das Ergebnis den Wert `Ok(...)` auf, anderenfalls erhalten Sie `Err(...)` mit dem entsprechenden Fehler.

Soll das Programm einfach abbrechen, genügt die Methode `unwrap()` (Listing 9, erste Zeile). Sie ruft im Fehlerfall den Befehl `panic!` auf (Zeile 4), was in etwa dem `match`-Statement entspricht. Will sich die aufrufende Funktion des Fehlers

Listing 8: Result

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Listing 10: Main

```
async fn start() -> Result<(),
Box<dyn Error>> {
...
  Ok(())
}
```

Listing 9: Abbrechen

```
01 function_with_error().unwrap;
02   match function_with_error() {
03     Ok(value) => value,
04     Err(e) => panic!("Error:
   {:?}", e)
05   }
06   match function_with_error() {
07     Ok(value) => value,
08     Err(e) => return Err(e)
09   }
10 function_with_error()?
```

annehmen, kann sie im Zweig `Err` des `match`-Statements etwas Sinnvolles tun oder den Fehler als Rückgabewert weiterreichen (Zeilen 6 bis 9).

Allerdings machen Sie es sich noch etwas einfacher, indem Sie den Operator `?` nutzen. Er erledigt genau dasselbe wie das `match`-Statement aus Listing 9. Die Funktion `start()` gibt jeden Fehler an die Funktion `main()` zurück (Listing 10).

Da in Rust Fehler unterschiedliche Datentypen haben können, wirkt der Rückgabewert der Funktion etwas unübersichtlich. Dem Enum `Result` werden bei der Definition zwei Datentypen mitgegeben: Der erste ist der, den das `Ok(...)` zurückliefert, der zweite ist der Fehlerdatentyp von `Err(...)`.

Im Ok-Fall liefert die Funktion `start()` nichts zurück, was in Rust so aussieht: `()`. Bei einem Fehler kann es sich um jeden beliebigen Fehlertyp handeln. Deshalb steht dort kein konkreter Typ, sondern die Schnittstelle (Trait) `std::error::Error`. Sie implementiert jeden Fehlertyp.

Anfrage senden

Das Paket Reqwest verwendet bei der Anfrage an OpenAI die Methode `post()` (Listing 11, Zeilen 28 bis 32), da es etwas absendet, bevor es etwas zurück erhält. Beim ersten Beispiel kam dagegen die Methode `get()` zum Einsatz, da Reqwest einfach einen Seiteninhalt abrief.

Die Methode `send()` schickt die Anfrage ab, und das Programm wartet gleichzeitig asynchron mit `await` auf die Antwort. Der verwendete `request_string` ist als `r#"..."#;` definiert (Listing 11, Zeilen 18 bis 26). Es handelt sich dabei um einen Raw String, der sich über mehrere Zeilen erstrecken kann und außerdem automatisch die Zeilenumbrüche mit in den Inhalt aufnimmt. Das kommt der Übersichtlichkeit zugute. Sobald das Ergebnis des Aufrufs zurückkommt, liest das Programm die Antwort mit der Methode `text()` aus (Zeile 33).

Lokales LLM mit Ollama

Wenn Sie die Beispielprogramme ohne Webservices lokal verwenden möchten, dann bietet die Software Ollama eine einfache Möglichkeit dazu. Sie installieren sie einfach entsprechend der Anleitung des Projekts auf Ihrem Linux-System. Anschließend lädt das Kommando `ollama pull llama3` beispielsweise das LLM Llama 3 von Meta auf den Rechner, und mit dem Befehl `ollama serve` starten Sie es. In den Beispielprogrammen müssen Sie lediglich die aufgerufene URL in http://localhost:11434/v1/chat/completions ändern.

Ausblick

Den vollständigen Chatbot `openai_console.rs` finden Sie im Download-Bereich zu diesem Artikel. Er verwendet einige Konsolenkomponenten, die die Bedienung vereinfachen. Wenn Sie mehr über die asynchrone Verarbeitung in Rust erfahren möchten, empfehle ich als Einstieg ein Video von Herbert *TheBracket* Wolverson auf Youtube. (*csi*)

Dateien zum Artikel herunterladen unter
www.lm-online.de/dl/50963

Weitere Infos und interessante Links
www.lm-online.de/qr/50963

Listing 11: Openai.rs

```
01 use std::error::Error;
02 use std::fs;
03 use reqwest::{Client, Url};
04 #[tokio::main]
05 async fn main() {
06   match start().await {
07     Ok(_) => println!("Bye!"),
08     Err(e) => println!("Error: {e}")
09   }
10 }
11 fn get_openai_key() -> String{
12   let key = fs::read_to_string("./secret.txt").
   unwrap();
13   key
14 }
15 async fn start() -> Result<(), Box<dyn Error>> {
16   let client = Client::new();
17   let url = Url::parse("https://api.openai.com/
   v1/chat/completions")?;
18   let request_string = r#"
19   {
20     "model": "gpt-3.5-turbo",
21     "messages": [{
22       "role": "user",
23       "content": "What is the best programming
   language?"
24     }]
25   }
26   "#;
27   let result = client
28     .post(url)
29     .header("Content-Type", "application/json")
30     .bearer_auth(get_openai_key())
31     .body(request_string)
32     .send().await?;
33   let body = result.text().await?;
34   println!("Body: {body}");
35   Ok(())
36 }
```