DuSchuldestMirBot

Prototyp einer Chatbot-basierten Erinnerung zum Einlösen von Schulden aller Art

Inhaltsverzeichnis

1. Einführung

Dieses Dokument dient der technischen Dokumentation des „DuSchuldestMirBot“. Ein Telegram-Bot, entwickelt vom Kurs TIM19 der DHBW RV.

Der „DuSchuldestMirBot“ soll ein Prototyp darstellen, der zur Erinnerung an Schulden jeglicher Art entwickelt wurde. Anwender können durch Chatinteraktionen mit einem Bot (also einem Programm) Schulden von anderen Anwendern anfordern oder Schulden begleichen (Getränke, Fahrten, Essen, etc.). Dabei wird der Schuldner vom Bot informiert, welche Schuld er zu begleichen hat (inklusive Deadline). Der Schuldner kann die Schuld annehmen oder ablehnen.

Im Fall der Schuldannahme wird der Schuldner regelmäßig an das Einlösen der Schuld erinnert. Um die Schuld zu begleichen, kann er diese im Chat als beglichen eintragen.

Die Kriterien der Entwicklung des Chatbots waren nach Absprache wie folgt: Der Chatbot soll ein Telegram-Bot sein, der mit den Anwendern interagiert und chattet. Die Entwicklung bzw. das Skript soll mit Python3 unter der Einbindung der python-telegram-bot-Bibliothek erfolgen. Die Schulden/User werden als JSON abgespeichert.

2. Technische Voraussetzungen/Umsetzung

Um den Telegram-Bot in Betrieb zu nehmen, wird zu allererst ein Telegram-Account benötigt. Wenn die Accounterstellung abgeschlossen ist, sucht man mit der Suchfunktion von Telegram nach dem „Botfather“. Diesen kann man mit /start starten. Mit dem Befehl /newbot erstellt man nun seinen eigenen Bot und folgt den Anweisungen des „Botfather“. Bei erfolgreicher Erstellung eines Bots erhält man einen Token, mit dem man die http API ansprechen kann.

Dieser Token wird nun in Zeile 40 der „main.py“ als Variable in „BOT_HTTP_TOKEN“ gespeichert. Hierfür wird die Installation von Python 3.8.2 inklusive pip und der Bibliothek „python-telegram-bot“ vorausgesetzt. Um die Bibliothek zu installieren kann man folgenden Befehl benutzen: `$ pip install python-telegram-bot --upgrade`

Nun ist der Bot startbereit. Nach Starten des Bots kann man anfangen, sich mit dem Bot über den Telegramchat zu unterhalten. Jeder User kann den Bot mit dem vom User erstellten Nicknamen mit der Telegram-Suchfunktion finden. Durch Öffnen des Chats kann man sich registrieren.

Nach erfolgreicher Registrierung beim Bot kann man anderen Usern, die sich bei dem Bot registriert haben, Schulden zuweisen bzw. kann selbst Schulden zugewiesen bekommen.

3. Programmablauf/Benutzung

Wie schon in den technischen Vorrausetzungen erklärt, kann man den Bot mit der Telegram-Suchfunktion über den vom Ersteller festgelegten Nicknamen suchen.

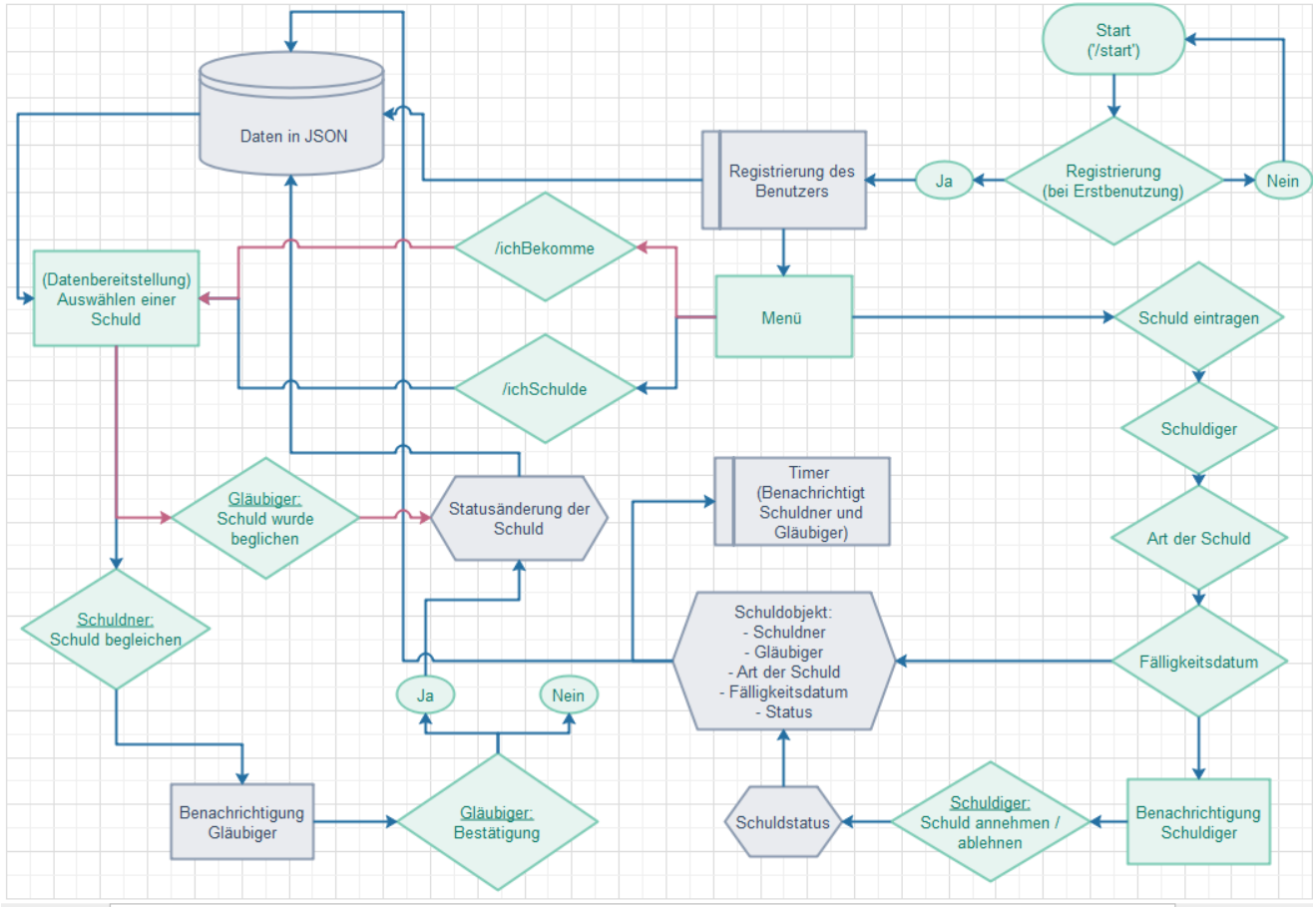
Beim Öffnen des Chats muss sich der User bei Erstbenutzung beim Bot registrieren. Der User benötigt jedoch zuerst einen Telegram-Username. Falls noch kein Username festgelegt wurde, wird der User vom Bot darauf aufmerksam gemacht. Die Ausführung weiterer Befehle kann erst nach Abschluss der Registrierung vorgenommen werden.

Nach erfolgreicher Registrierung erscheint in dem Chat zwischen User und Bot eine Auswahl an Befehlen, die von hier aus aufgeführt werden können. Diese können jederzeit zur Steuerung des Bots verwendet werden.

Mit dem Befehl /schuld kann man einem registrierten User Schulden zuweisen. Hierfür wählt man die Art, Menge und das Datum, zu der die Schuld spätestens beglichen sein soll (Deadline). Der Schuldner bekommt daraufhin eine Benachrichtigung, bei der er die Schuld bestätigen/ablehnen kann. Wenn die Schuld bestätigt wurde, bekommen Schuldner und Gläubiger in regelmäßigen Abständen eine Benachrichtigung, die einen an die Schuld erinnern soll.

Mit dem Befehl /ichBekomme kann sich der User alle seine ausstehenden Forderungen anzeigen lassen. Durch Auswahl einer Schuld kann der Gläubiger diese jederzeit als beglichen markieren.

Mit dem Befehl /ichSchulde kann sich der User alle seine ausstehenden Schulden anzeigen lassen. Durch Auswahl einer Schuld kann die Begleichung angefragt werden. Der Gläubiger bekommt in diesem Fall eine Benachrichtigung, um zu bestätigen, ob die Schuld wirklich beglichen wurde. Erst nachdem es bestätigt wurde, gilt die Schuld als beglichen und wird auch so gespeichert.



4. Funktionen/Technische Umsetzung

Hier werden die einzelnen Funktionen des Programms technisch und logisch erläutert.

4. 1. Registrierung und Start-GUI

Beim Start des Bots wird geprüft, ob der User bereits registriert ist. Falls dies nicht der Fall ist, wird der User gefragt, ob eine Registrierung gewünscht ist. Dadurch wird eine ungewollte Registrierung verhindert und es bleibt dem User selbst überlassen, ob er sich registrieren möchte. Dies kann über Buttons entschieden werden. Wenn er sich registrieren möchte, erfolgt eine Weiterleitung ins Startmenü und der Nutzer wird in der Datenbank gespeichert.

```
51  def start(update, context):
52      """handles /start command,
53      user registration
54      """
55      username = update.message.from_user.username
56      first_name = update.message.from_user.first_name
57      chat_id = str(update.effective_message.chat_id)
58
59      is_registrated = DB.user_exists(chat_id)
60
61      # sends keyboard to existing users
62      if is_registrated:
63          update.message.reply_text(
64              "Hey " + first_name + "!" + "\nWillkommen zurück!"+"\U0001F609",
65              reply_markup=get_start_keyboard())
```

Falls der registrierte User den Chatbot neu startet, erhält er keine neue Abfrage zur Registrierung, sondern wird mit Namen begrüßt und direkt ins Startmenü weitergeleitet.

Die Abfragen werden mittels einem *InlineKeyboard* durchgeführt und die Reaktion in der Methode *button* weiterverarbeitet.

```
89              # yes / no keyboard
90              keyboard_yn = [[InlineKeyboardButton("\U0001F44D", callback_data=data_yes),
91                              InlineKeyboardButton("\U0001F44E", callback_data=data_no)]]
92              reply_markup = InlineKeyboardMarkup(keyboard_yn)
93
94              update.message.reply_text(
95                  "Möchtest Du dich registrieren?", reply_markup=reply_markup)
96
97              UPDATER.dispatcher.add_handler(
98                  CallbackQueryHandler(handle_registration_response))
```

Diese Methode wird aufgerufen, wenn bei der Registrierung der Button „Ja“ oder „Nein“ gedrückt wird. Der jeweilige Button speichert beim Auslösen einen für ihn spezifischen Wert in der Variable *callback_query.data,* welche wiederum eine Reaktion auslöst mit entsprechender Methode.

Ein Beispiel hierfür wäre: „True“ wird für „Ja“ in der *callback_query.data* abgespeichert und leitet entsprechende Reaktion und Methode weiter ein, wenn der Button „Ja“ gedrückt wird.

```
130  def handle_registration_response(update, context):
131      """checks wether user wants to be registered or not
132      """
133
134      query = update.callback_query
135      data = json.loads(update.callback_query.data)
136      user_response = data['data']
137
138      # user clicks yes and will be registered
139      if user_response:
140          start_menu(update, context)
141          chat_id = str(query.message.chat_id)
142          user_name = str(query.from_user.username)
143          DB.add_user(chat_id, user_name)
```

Falls der User keine Registrierung wünscht, wird die Methode *cancel* aufgerufen. Hier wird der *ConversationHandler* beendet und der User bekommt den Hinweis durch /start den Prozess neu zu starten. Somit besteht die Möglichkeit, sich nachträglich noch zu registrieren.

```
150  def cancel(update, context):
151      """close the chat
152      """
153      query = update.callback_query
154      bot = context.bot
155      # Closing text
156      bot.edit_message_text(
157          chat_id=query.message.chat_id,
158          message_id=query.message.message_id,
159          text="Schade!" +
160          "\nFalls Du es dir anders überlegst, kannst du mit /start den Prozess neu starten." +
161          "\U0001F47C"
162      )
163      return ConversationHandler.END
```

Das Hauptmenü kann durch die Befehle: /schuld, /ichBekomme und /ichSchulde gesteuert werden. Diese Befehle können jederzeit ausgewählt werden. Die Methode erzeugt eine Auswahl an Befehlen, die der Benutzer auswählen kann. Jeder Befehl führt die dazugehörige Methode aus.

```python
def start_menu(update, context):
    """Start menu to select what you want to do (enter debts, - settle)
    """

    query = update.callback_query
    bot = context.bot
    # get ChatId
    chat_id = str(query.from_user.id)

    bot.edit_message_text(
        # generate startmenu
        chat_id=chat_id,
        message_id=query.message.message_id,
        text="Bitte wähle dein Anliegen aus:\n")

    context.bot.send_message(
        chat_id, text="Klicke auf \u27A1 /schuld um Schulden einzutragen...\n"
        "Klicke auf \u27A1 /ichSchulde um einzusehen, wem du was schuldest..."
        "\nKlicke auf \u27A1 /ichBekomme um einzusehen, was dir wer schuldet...",
        reply_markup=get_start_keyboard())
```

4. 2. Speicherung der Schulden

Um die Speicherung der Daten zu ermöglichen wurden drei Klassen erstellt:

User:

Die User Klasse stellt einen Verbund der Attribute eines Benutzers dar.

Ein User Objekt besitzt folgende Attribute:

chat_id: Chat ID des Chats zwischen Benutzer und Bot

name: Username des Benutzers

Die Klasse besitzt folgende Funktionen:

to_dict: Gibt die Attribute des User Objekts in einem Dictionary zurück
from_dict: Wandelt ein gegebenes Dictionary in ein User Objekt um und gibt das User Objekt anschließend zurück

Debt:

Die Debt Klasse stellt ein Verbund der einzelnen Attribute eines Schuldenprozesses dar.

Ein Debt Objekt besitzt folgende Attribute:

debt_id: ID des Schuldenprozesses, als UUID Zahl

creditor: Chat ID des Gläubigers

category: Kategorie der Schuld, z.B. Mittagessen

deadline: Einlösefrist der Schuld

debtor: Chat ID des Schuldners

is_accepted: Zeigt, ob der Schuldner die Schuld akzeptiert hat

is_paid: Zeigt, ob der Schuldner die Schuld beglichen hat

Die Klasse besitzt folgende Funktionen:

to_dict: Gibt die Attribute des Debt Objekts in einem Dictionary zurück

from_dict: Wandelt ein gegebenes Dictionary in ein Debt Objekt um und gibt das Debt Objekt anschließend zurück

Database:

Die Database Klasse interagiert mit der JSON Datei. Also werden mithilfe dieser Klasse die einzelnen User und Debt Objekte gespeichert beziehungsweise ausgelesen.

Die Klasse besitzt folgende Attribute:

path_to_json: Dateipfad der JSON Datei

users: Liste der User Objekte

debts: Liste der Schuld Objekte

Die Klasse besitzt folgende Funktionen:

init_json: Lädt die Daten der JSON Datei und initialisiert die User- und Debtliste

update_json: Speichert die aktuellen Listen in der JSON Datei

user_exist: Überprüft anhand der Chat ID, ob ein bestimmter User bereits in der Datenbank existiert

add_user: Fügt einen neuen User zur Datenbank hinzu

add_debt: Fügt einen neuen Schuldenprozess zur Datenbank hinzu

get_open_debts: Gibt alle offenen Schulden eines Users zurück

get_open_claim: Gibt alle offenen Forderungen eines Users zurück

get_user_by_chat_id: Gibt das passende User Objekt zu einer Chat ID zurück

get_debt_by_debt_id: Gibt das passende Schuld Objekt zu einer Debt ID zurück

set_accepted: Setzt das is_accepted Attribut eines Debt Objektes

set_paid: Setzt das is_paid Attribut eines Debt Objektes

Im Folgenden wird auf das verwendete Dateiformat JSON eingegangen:

JSON steht für „JavaScript Object Notation“ und ist ein für den Mensch und die Maschinen leicht zu verstehendes Textformat zum Austausch von Daten. Es basiert auf dem JavaScript-Standard ECMA-262 3rd Edition, welcher im Dezember 1999 verabschiedet wurde. Das Format besteht aus Key / Value Paaren, welche ineinander verschachtelt werden können. Beispielsweise kann der Value eines Keys aus beliebig vielen weiteren Key / Value Paaren bestehen. Dies ist aus einigen Programmiersprachen als Dictionary, Listen bzw. Arrays bekannt. Daher ist JSON leicht und ohne großen Aufwand zu implementieren. Die JSON Daten werden in der Datenbank Klasse mithilfe der JSON Python Library verarbeitet.

Verwendung der JSON Library:

Zur Verarbeitung der User- und Schuldobjektdaten, wird auf Funktionen des importierten JSON Moduls zurückgegriffen. Folgende Funktionen wurden verwendet:

Json.loads:

Deserialisiert eine String-Instanz, welche ein JSON-Dokument enthält, in ein Python-Objekt.

Json.dumps:

Serialisiert ein Python-Objekt zu einer JSON formatierten String-Instanz.

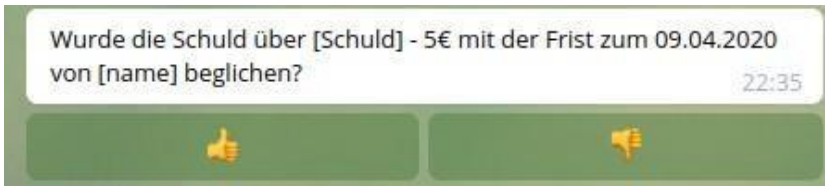
Das Datenmodell stellt die zu verarbeitenden bzw. gespeicherten Daten dar. Die Userobjektdaten werden in der Liste / dem Array „users“ gespeichert. Diese Liste beinhaltet alle Attribute eines Users in Form eines Dictionarys. Jeder User wird in einem separaten Dictionary dargestellt. Schuldobjektdaten werden ebenfalls in einer Liste gespeichert, die Liste „debts“. Für jedes Schuldobjekt existiert ein Dictionary in der Liste.

```
{
    "users": [
        {
            "chat_id": "chat_id",
            "name": "user_name"
        }
    ],
    "debts": [
        {
            "debt_id": "fbfe4e2d-6e80-11ea-9f5f-2c4d54d11061",
            "creditor": "creditor_id",
            "category": "Getränke",
            "amount": "10€",
            "deadline": "2020:03:25",
            "debtor": "debtor_id",
            "is_accepted": false,
            "is_paid": false
        }
    ]
}
```

4. 3. Bestätigungs-/Ablehnungsmechanismus

Da ein User andere User nicht beliebig mit Schulden behaften können soll, musste ein Mechanismus eingeführt werden, der den vom Gläubiger angegebenen Schuldner bestätigen lässt, dass er eine Schuld annimmt.

Außerdem muss der Bot die Funktionalität enthalten, dass ein Gläubiger die Begleichung einer Schuld bestätigen kann, nachdem ein Schuldner diese angibt.



Annehmen einer Schuld

Beim Eintragen einer Schuld durchläuft der User einen Prozess, dessen letzter Schritt das Eintragen einer Deadline ist. In derselben Funktion (*calendar_selection*), in der das Eintragen dieser Deadline behandelt wird, wird eine Nachricht mit einem Annehm- und einem Ablehnbutton an den Schuldner geschickt.

Beim Betätigen der jeweiligen Buttons wird ein Objekt der Klasse *CallbackQuery* an den Bot gesendet, dessen „data“-Attribut zwei Elemente enthält. Zum einen enthält es die Information, welcher Button betätigt wurde (in Form eines booleschen Wertes). Zum anderen die ID der behandelten Schuld. Ein Python-Dictionary, welches beide Werte enthält, wird dafür mit Hilfe von *json.loads* zu einem String konvertiert. Der erste Wert wird hierbei mit dem Schlüsselwert „1“ gespeichert, der Zweite (die Schuld-ID) mit „id“.

Eine weitere Funktion (*handle_accept_debt*) wandelt den beschriebenen String wiederum in ein Dictionary (*json.loads*) um und stellt das „accepted“-Attribut der Schuld in der JSON-Datei auf den Wert, der mit dem Schlüsselwert „1“ gespeichert wurde (*set_accepted*). Die anzupassende Schuld wird dabei über dessen eindeutige ID bestimmt (*Database.get_debt_by_debt_id*).

Außerdem bestimmt die Funktion, ob der „Ja“- oder der „Nein“-Button betätigt wurde und prüft dementsprechend den Schlüsselwert „p“ des Dictionarys auf True oder False. Wurde „Ja“ gedrückt, wird die Nachricht mit den Buttons editiert, um den Nutzer, der mit einer Schuld behaftet wurde, zu informieren, dass er die Schuld nun als angenommen markiert hat. Zusätzlich wird der Timer gestartet (*start_timer*).

Bei Betätigung des “Nein”-Buttons hingegen wird nicht nur der Schuldner, sondern auch der Gläubiger informiert, dass die Schuld nicht als angenommen markiert wurde. Ein Timer wird nicht gestartet.

Begleichung einer Schuld bestätigen

Bei diesem Teil des Codes wird die Nachricht zur Bestätigung an den Gläubiger geschickt. Dies passiert innerhalb der letzten Funktion (*handle_ask_if_debt_is_paid*), die vom *ConversationHandler* „*i_owe_handler“* aufgerufen wird. Die bei Betätigung eines Buttons mitgeschickten Daten sind hier genauso organisiert wie die in „Annehmen einer Schuld“ behandelten Daten. Lediglich der erste Schlüsselwert unterscheidet sich: anstatt „1“ wird „paid“ verwendet.

Abgefragt wird dieser String als *callback_query.data* durch erneute Umformung zu einem Dictionary (*json.loads*) in der Funktion *handle_accept_debt_is_paid*. Diese Funktion setzt in einem ersten Schritt das „paid“-Attribut der behandelten Schuld auf den Wert, der im Dictionary mit dem Key „paid“ gespeichert wurde, also True oder False.

Außerdem wird bei True die Nachricht wie beim Annehmen einer Schuld editiert und der Timer gestoppt (*stop_timer*).

Bei False wird die Nachricht ebenfalls editiert und eine Nachricht an den Schuldner gesendet, dass die Anfrage zur Begleichung abgelehnt wurde. Der Timer wird nicht gestoppt.

Handler-Organisation

Ein *ConversationHandler* (der für die Vorgänge, eine Schuld einzutragen und eine Schuld zu begleichen eingesetzt wird) ist dafür ausgelegt, die Nachrichten eines einzelnen Nutzers zu behandeln. Deswegen müssen die beschriebenen Funktionen (*handle_accept_debt, handle_accept_debt_is_paid*) zur Behandlung der *CallbackQuerys* anderer Nutzer anders aufgerufen werden als mithilfe der *ConversationHandler*. Sie brauchen ausgelagerte *CallbackQueryHandler*, um aufgerufen zu werden.

Da beide Annahmefunktionalitäten eines ausgegliederten „CallbackQueryHandlers“ bedürfen, wäre es denkbar, dass der finale Code zwei Objekte dieser Handler-Klasse beinhalten würde.

Dem ist nicht so, weil mehrere *CallbackQueryHandler* nicht parallel laufen können. Insbesondere, weil ein *CallbackQueryHandler* nicht auf ein *CallbackQuery* eines bestimmten Nutzers reagiert, sondern auf jedes CallbackQuery-Objekt, welches der Bot empfängt.

Also wird eine weitere Funktion (*callback_general*) definiert. Sie wird bei jeder *CallbackQuery* aufgerufen, welches nicht innerhalb einer bestehenden Konversation auftritt.

Jede *CallbackQuery* wird dabei durch einfache if-Abfragen ihrem Zweck zugeordnet und die entsprechende Funktion aufgerufen.

```
956  def callback_general(update, context):
957      """
958      General callback handler for debt is paid, registration and accept debt.
959      """
960
961      callback_data = json.loads(update.callback_query.data)
962      action = ""
963      if "action" in callback_data:
964          action = callback_data["action"]
965
966      # handle accept debt as paid
967      if "p" in callback_data:
968          handle_accept_debt_is_paid(update, context)
969
970      # handle registration
971      if action == "registration":
972          print(callback_data)
973          handle_registration_response(update, context)
974
975      # handle_accept_debt
976      if "1" in callback_data:
977
978          print("handle accept debt")
979          handle_accept_debt(update, context)
```

```
1118      # Handler /ichSchulde
1119      i_owe_handler = ConversationHandler(···
1127      )
1128
1129      # Handler /ichBekomme
1130      i_get_handler = ConversationHandler(···
1138      )
1139
1140      UPDATER.dispatcher.add_handler(CommandHandler("start", start))
1141
1142      UPDATER.dispatcher.add_handler(se_conv_handler)
1143
1144      UPDATER.dispatcher.add_handler(i_owe_handler)
1145      UPDATER.dispatcher.add_handler(i_get_handler)
```

Ergänzung

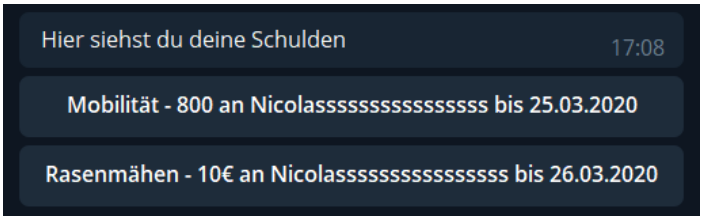
In *callback_general* wird außerdem die Funktion *handle_registration_response* aufgerufen, weil die Registrierung nicht über einen *ConversationHandler* gelöst wurde.

4. 4. Schulden begleichen

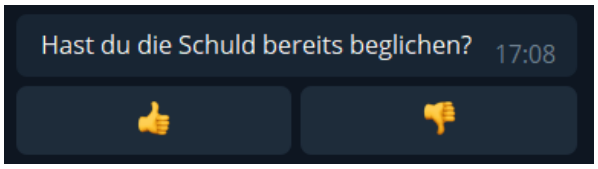
Dieser Teil des Projekts ist dafür verantwortlich, dass bereits eingestellte Schulden ordnungsgemäß beglichen werden können. Dies kann sowohl von Gläubiger- als auch von Schuldnerseite angestoßen werden. Jeder Nutzer kann sich sowohl eine Liste seiner Schulden als auch seiner Forderungen ausgeben lassen.

Implementiert wurden zwei Befehle: /ichSchulde und /ichBekomme, welche jeweils vom Hauptmenü aus aufgerufen werden können. Bei beiden Befehlen wurde darauf geachtet, dieselbe Struktur im Ablauf des Prozesses beizubehalten.

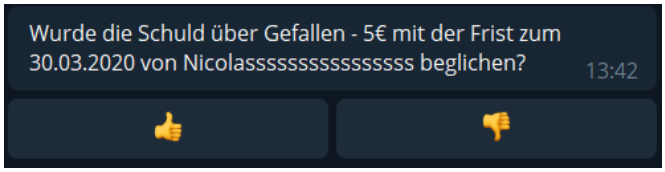
Beim Aufrufen von /ichSchulde wird dem User eine Liste seiner Schulden angezeigt, welche er an einen anderen Nutzer zurückzuzahlen hat. Diese Liste umfasst folgende Details: Beschreibung der Schuld, Gläubiger, Höhe der Schuld und Deadline.



Durch das Anklicken einer in der Liste aufgeführten Schuld wird dem User die Option einer Begleichung angeboten, welche er entweder annehmen oder ablehnen kann. Im Falle einer Ablehnung bekommt er lediglich eine kurze Rückmeldung „Ok“ und die Liste wird geschlossen. Anschließend hat er die Möglichkeit beliebig fortzufahren.

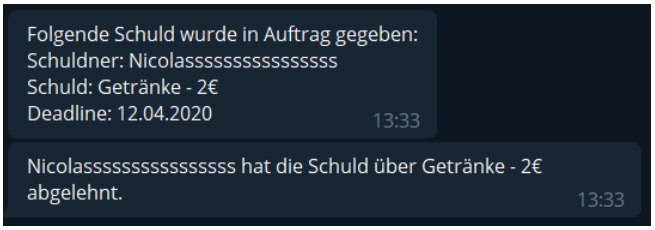


Im Falle einer Annahme wird der Gläubiger darüber benachrichtigt, dass der Schuldner seine Schuld begleichen möchte. Der Gläubiger hat nun die Möglichkeit dies zu bestätigen und damit der Begleichung zuzustimmen oder er lehnt die Begleichung der Schuld ab. Zum Beispiel dann, wenn er die Rückzahlung nicht erhalten hat.

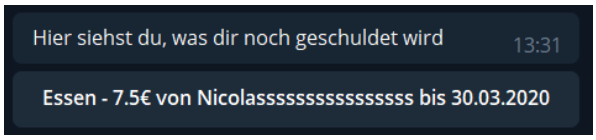


Sollte der Gläubiger die Rückzahlung bestätigen, ist der Vorgang beendet. Die Schuld wird aus der Liste des Schuldners entfernt. Wenn der Schuldner also keine Nachricht nach dem Begleichen der Schuld erhält, weiß er, dass er die Schuld erfolgreich beglichen hat. Dies kann er mit dem Befehl /ichSchulde kontrollieren, denn hier taucht diese bereits beglichene Schuld nun nicht mehr auf. Zusätzlich erhält er auch keine weiteren Zahlungserinnerungen mehr.

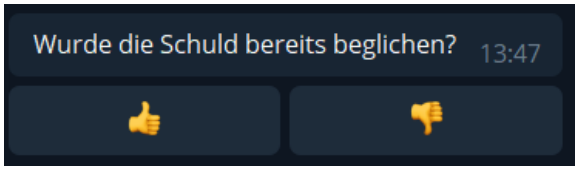
Sollte der Gläubiger die Rückzahlung ablehnen, so erhält der Schuldner eine Nachricht darüber, dass die Rückzahlung nicht akzeptiert wurde und weiß somit, dass die Schuld weiterhin besteht. Somit sind Täuschungsversuche, sowie Missverständnisse über noch offene und bereits beglichene Schulden ausgeschlossen.



Beim Aufrufen von /ichBekomme wird dem User eine Liste mit all seinen Auslagen angezeigt. Auch hier enthält die Liste folgende Details: Beschreibung der Auslage, Schuldner, Höhe der Schuld und Deadline.



Beim Auswählen einer in der Liste aufgeführten Auslage gibt es die Möglichkeit das Begleichen dieser Auslage zu bestätigen oder abzulehnen. Somit hat auch der Gläubiger die Möglichkeit eine Auslage als erledigt zu markieren.



Bei einer Ablehnung der Rückzahlung einer Auslage bekommt der Gläubiger die kurze Antwort „Ok“. Damit ist der Prozess beendet und es werden auch keine Auslagen/Schulden aus der Liste entfernt.

Bei der Bestätigung der Rückzahlung einer Auslage bekommt der Gläubiger die Nachricht darüber, dass die Auslage als beglichen markiert wurde. Der Schuldner erhält keine gesonderte Nachricht. Eine aktuelle Übersicht über die Schulden und Auslagen können sich Gläubiger sowie Schuldner jederzeit über die beiden Befehle /ichSchulde und /ichBekomme anzeigen lassen.

Umgesetzt wurde der oben beschriebene Ablauf mit insgesamt drei Handlern, die in Kombination mehrfach benutzt werden. Es werden zwei *ConversationHandler* benutzt, welche auf die „states“ reagieren, die von einer anderen Methode zurückgegeben werden.

Wie bereits erwähnt, werden in Kombination zwei weitere Handler verwendet: Der *CommandHandler* und der *CallbackQueryHandler*.

Mit dem *CommandHandler* macht man es den Usern möglich, die Befehle /ichSchulde und /ichBekomme zu nutzen und dadurch mit dem Bot zu kommunizieren. Dabei wird auf den *CallbackQueryHandler* zurückgegriffen.

Die Aufgabe des *CallbackQueryHandler* ist es, dem Bot korrekt mitzuteilen, welcher Button vom User gedrückt wurde. Beispielsweise ob der User „Ja“ oder „Nein“ als Antwort ausgewählt hat oder welche Schuld vom Nutzer ausgewählt wurde. Für die Identifikation wird eine „debt_id“ verwendet.

```
956 def callback_general(update, context):
957     """
958     General callback handler for debt is paid, registration and accept debt.
959     """
960 
961     callback_data = json.loads(update.callback_query.data)
962     action = ""
963     if "action" in callback_data:
964         action = callback_data["action"]
965 
966 
967     # handle accept debt as paid
968     if "p" in callback_data:
969         handle_accept_debt_is_paid(update, context)
```

Es gibt einen weiteren separaten *CallbackQueryHandler*, der unter anderem dazu dient, die Bestätigung des Gläubigers zu bearbeiten. Dies kann nicht im *ConversationHandler* geschehen, da der Bot mit mehreren Usern kommunizieren muss.

4. 5. Schulden eintragen

Nach erfolgreicher User-Registrierung hat dieser drei verschiedene Optionen zur Auswahl. Neben den Anzeigeoptionen für eigene Schulden und von anderen noch nicht beglichenen Schulden, lassen sich neue Schulden in Auftrag geben. Die Infos zur neuen Schuld werden nach dem Ausführen von /schuld in einer bestimmten Reihenfolge abgefragt.

Das zentrale Herzstück dieses Befehls ist der *ConversationHandler*, der jeglichen User-Input verarbeitet. Verschiedene Eingaben sorgen dafür, dass der *ConversationHandler* bestimmte Methoden je nach Zustand aufruft. Die Zustände sorgen dafür, dass wenn beispielsweise die Eingabe eines Datums erwartet wird, ein Username nicht akzeptiert wird.
Somit läuft der Informationsgewinn für die Schuld nach folgendem Konzept ab:

Auswahl des Schuldners → Kategorie → Anzahl → Deadline

```
1051    se_conv_handler = ConversationHandler(
1052
1053        entry_points=[CommandHandler("schuld", new_debt)],
1054        # todo: unicode fix
1055        states={
1056 >          USER_SELECTION: [MessageHandler(Filters.regex("^Abbrechen ✕$"), ···
1071 >          CATEGORY_SELECTION: [···
1094            ],
1095 >          AMOUNT_SELECTION: [MessageHandler(Filters.regex("^Abbrechen ✕$"), ···
1114 >          CALENDAR_SELECTION: [MessageHandler(Filters.regex("^Abbrechen ✕$"), ···
1133        },
1134        fallbacks=[MessageHandler(Filters.regex("^Done$"), done)]
1135    )
```

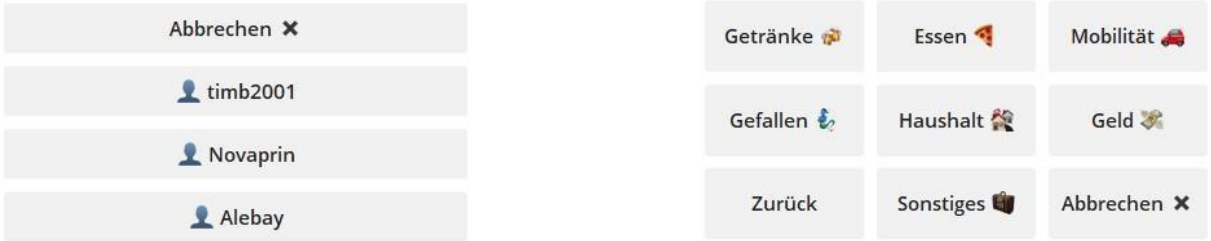
Der Input des Users wird je nach Zustand anders verarbeitet. Dafür gibt es vorgefertigte Textfilter (*regex*) oder auch nur den „rohen“ Userinput (*text*), die wiederum verschiedene Methoden (z.B. *amount_selection*) aufrufen.
Was bei allen Zuständen jedoch gleich bleibt, sind die Abbruchsfunktionen, wie z.B. das Aufrufen anderer Befehle und die „zurück“ Funktion.

```
1095            AMOUNT_SELECTION: [MessageHandler(Filters.regex("^Abbrechen ✕$"),
1096                                              cancel_define_debt),
1097                               MessageHandler(Filters.regex("^Abbrechen"),
1098                                              cancel_define_debt),
1099                               MessageHandler(Filters.regex("^Sonstiges 📝$"),
1100                                              amount_selection_manual),
1101                               MessageHandler(Filters.regex("^Zurück$"),
1102                                              amount_selection_back),
1103                               MessageHandler(Filters.regex("^/schuld"),
1104                                              new_debt),
1105                               MessageHandler(Filters.regex("^/ichSchulde"),
1106                                              i_owe),
1107                               MessageHandler(Filters.regex("^/ichBekomme"),
1108                                              i_get),
1109                               MessageHandler(Filters.regex("^/start"),
1110                                              start),
1111                               MessageHandler(Filters.text,
1112                                              amount_selection)
1113                               ],
```

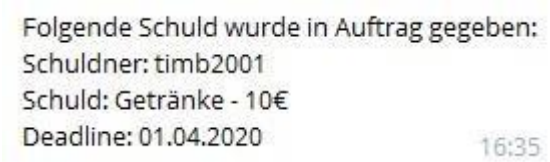
Die Methoden selbst können den Status des *ConversationHandlers* ändern und senden meist auch Informationen an den User, wie z.B. Custom Keyboards für den Fortlauf.

```
769  def amount_selection(update, context):
770      """
771      called when amount has been selected
772      --> sends calendar keyboard
773      """
774      amount = update.message.text
775      context.user_data["amount"] = amount
776      reply_keyboard = [["Heute", "Morgen"],
777                        ["Eine Woche", "Zwei Wochen"],
778                        ["Ein Monat", "3 Monate"],
779                        ["Sonstiges ℹ"],
780                        ["Zurück", "Abbrechen ✕"]]
781      markup = ReplyKeyboardMarkup(reply_keyboard, one_time_keyboard=True)
782      update.message.reply_text(
783          f"Bis wann soll {context.user_data['debtor']} die Schuld beglichen haben?",
784          reply_markup=markup)
785      return CALENDAR_SELECTION
```

Über speziell eingerichtete Tastaturen, kann der User schnell und komfortabel Eingaben in der jeweiligen Infoabfrage machen.



Über einen in Telegram integrierten Knopf, der sich über den speziell erstellten Tastaturen befindet, kann jederzeit zu einer klassischen Tastatur gewechselt werden, um bei den Eingaben alle Freiheiten zu haben. Zu Beginn wählt man eine der aufgelisteten Personen. Mit dem Wechsel auf die traditionelle Tastatur kann man Namen auch manuell eingeben, insofern die Person registriert ist und die Eingabe keine Tippfehler enthält. Mit der darauffolgenden Abfrage der Kategorie lässt sich die Art der Schuld angeben. Wie es danach weiter geht hängt von der Wahl der Kategorie ab. Je nach Art der Schuld werden verschiedene Tastaturen nach der Wahl der Kategorie geöffnet, mithilfe deren man den Wert der Schuld am besten angeben kann. Abschließend ist noch die Angabe der Deadline relevant, um alle nötigen Daten zur Erstellung eines neuen Eintrags erfasst zu haben. Neben einigen vorgefertigten Auswahlmöglichkeiten lässt sich hier ebenfalls manuell eine Deadline der Schuld eintragen. Nach dem Abschluss der Eintragungen wird eine Anfrage der Schuld an den potentiellen Schuldner geschickt. Zuletzt ist bei diesem Punkt noch zu erwähnen, dass jede funktionell angepasste Tastatur die grundlegenden Funktionen des Abbrechens und des Zurückkehrens als letzte Wahl zur Verfügung stellt.



Um die Schuld nun als Sammlung aus Daten einzureichen gibt es Schnittstellen, die an andere Stellen des gesamten Programms anknüpfen. Die rohen Daten werden zunächst zur Speicherung weitergegeben und erhalten eine ProzessID. Diese wird für den nächsten Schritt benötigt, in dem die Daten zusammen mit der neu erhaltenen ID an die Accept-/Deny-Funktion weitergegeben werden, von der aus die Anfrage für die Schulden verschickt wird. Durch die ID ist die Anfrage nun zuzuordnen und kann zukünftig bei Annahme oder Ablehnung weiterverarbeitet werden.

4. 6. Erinnerungsmechanismus/Timer

Die Aufgabe des Timers ist, in regelmäßigen Abständen Schuldner und Gläubiger an ihre offenen Schulden zu erinnern. Die Grundfunktionsweise des Timers basiert auf der *JobQueue* aus der „python-telegram-bot library“.

Innerhalb der *start_timer*–Methode lässt sich die Funktionsweise der „JobQueue“ gut darstellen.

Anhand der *JobQueue* lassen sich Methoden als sogenannte „Jobs“ speichern, welche in bestimmten Zeitabständen ausgeführt werden.

Damit erhält man die *JobQueue* des aktuellen „Dispatcher-Objekts“. *tele_updater* ist hier der zugehörige *Updater*.

```
199  def start_timer(tele_updater: UPDATER, debt_id):
200      """
201      Starts the timer with a corresponding debt
202      Params: UPDATER tele_updater
203              string debt_id
204      """
205
206      cur_debt = DB.get_debt_by_debt_id(debt_id)
207
208      # Gets the current job queue of the dispatcher
209      queue = tele_updater.dispatcher.job_queue
210      time = datetime.time(hour=10, minute=0, second=0)
211
212      queue.run_daily(_callback_alarm, time, context=cur_debt)
```

Hiermit lässt sich der Job *_callback_alarm* einmal täglich zu einer festgelegten Zeit (*time)* ausführen und dient als Erinnerung an die Deadline. Als context lässt sich hier ein beliebiges Objekt übergeben.

Die *_callback_alarm* - Methode kümmert sich um das Benachrichtigen des Schuldners und des Gläubigers.

```
179  def _callback_alarm(context: CallbackContext):
180      """
181      Sends a message once the timer is triggered
182      Params: CallbackContext context
183      """
184      cur_debt = context.job.context
185
186      creditor_cid = cur_debt.creditor
187      debtor_cid = cur_debt.debtor
188      deadline_time = _parse_time_(cur_debt.deadline)
189      debt_text = str(cur_debt.amount) + " " + cur_debt.category
190
191      deadline = deadline_time.strftime("%d.%m.%Y")
192
193      context.bot.send_message(creditor_cid, text=str(DB.get_user_by_chat_id(
194          debtor_cid).name) + " schuldet dir noch " + str(debt_text) + " bis zum " + deadline)
195      context.bot.send_message(debtor_cid, text="Du schuldest " + str(DB.get_user_by_chat_id(
196          creditor_cid).name) + " noch " + str(debt_text) + " bis zum " + deadline)
```

Als Übergabe erhält die Methode lediglich den *CallbackContext*, welcher zuvor übergeben wurde.

Dieser besteht aus einer Schuld, die direkt zu Beginn in ihre benötigten Attribute aufgeteilt wird.

Zuerst wird ein String aus *Usernamen, Schuldbeschreibungen* und *Deadline* gebaut, welcher dann anhand von
context.bot.send_message() an beide User durch die jeweilige ChatID gesendet wird.

Um *deadline_time* festzulegen rufen wir die *_parse_time* – Methode auf. Diese sieht wie folgt aus:

```
166  def _parse_time_(time):
167      """
168      Conversion of the Deadline string to datetime object
169      Params: string time
170              Format: "YYYY:MM:DD"
171      """
172      year = int(time[0] + time[1] + time[2] + time[3])
173      month = int(time[5] + time[6])
174      day = int(time[8] + time[9])
175      due = datetime.datetime(year, month, day)
176      return due
```

Nach der Übergabe der Zeit in Form eines Strings mit dem Format *YYYY:MM:DD* (in Python) lassen sich Strings auch als *Character-Array* ansprechen, um den Übergabewert zu einem datetime-*Objekt* umzuwandeln.

Der Timer lässt sich anhand der *stop_timer* Methode manuell beenden.

Diese Methode benötigt als Übergabewerte den *Updater* und die *debt_id* der Schuld, welche entfernt werden soll. Die einzelnen Jobs der *JobQueue* werden in der *for*-Schleife durchlaufen und anhand einer *if*-Abfrage überprüft, ob der zu prüfende Job dieselbe *debt_id* besitzt. Ist dies der Fall, wird der Job in dem darauffolgenden Zyklus entfernt. Bei Erfolg wird *True* zurückgegeben, bei Misserfolgen wird *False* zurückgegeben.

Nun müssen wir bei einem Neustart des Bots sicherstellen, dass sämtliche Timer wieder starten, da die *JobQueue* mit dem Schließen des Bots geleert wird.

```python
def check_timers(tele_updater: UPDATER):
    '''
    Checks whether each debt has a corresponding timer running

    Params: Updater tele_updater
    '''
    debtlist = DB.debts
    queue = tele_updater.dispatcher.job_queue

    for debt in debtlist:
        job_exists = False

        for ajob in queue.jobs():
            if ajob.context.debt_id == debt.debt_id:
                job_exists = True
                break

        if not job_exists and debt.is_accepted and not debt.is_paid:

            start_timer(tele_updater, debt.debt_id)
```

Hier wird eine ähnliche Vorangehensweise verwendet, wie bei der *stop_timer*–Methode. Der größte Unterschied besteht in der Verschachtelung zweier *for*–Schleifen, welche nun alle Schulden mit allen Jobs vergleichen und die Kontrollvariable auf *True* setzen, sobald der Job als vorhanden befunden wurde.