Informatik
Hauptcampus

H O C H
S C H U L E
T R I E R

Der Coaching Bot

Eine standardisierte und automatisierte Anmeldeplattform für Coaching Sessions

Maximilian Wellenhofer

Master-Projektstudium

Betreuer: Prof. Dr. Georg Schneider

Zürich, 28.03.2022

Kurzfassung

Anmeldung und Terminvereinbarung für Coaching-Programme sollen automatisiert werden. Gängig wird dies über ein Webformular abgewickelt. Allerdings erfreut sich diese Herangehensweise keiner großen Beliebtheit, weshalb eine Alternative gesucht wird. Diese Arbeit behandelt den Versuch, eine Chat-Bot-Technologie zu nutzen, um die Problematik zu lösen. Nach einer etwas detaillierteren Einführung in die Problematik, wird eine Reihe an bereits verfügbaren Systemen analysiert. Es wird klar, dass diese den Anforderungen aus diversen Gründen nicht genügen. Daher wird aus einer Kombination und Erweiterung mehreren existierender Systeme ein Lösungskonzept erarbeitet und präsentiert. Gefolgt von einer detaillierten Beschreibung der Realisierung und Implementierung. Dabei werden Teile der Applikation beispielhaft stärker beleuchtet als Andere. Auf Herausforderungen und komplexere Systemzusammenhänge wird insbesondere eingegangen. Es wird genau erklärt, wie der Coaching-Bot funktioniert, um ein Verständnis herzustellen, das hinreichend ist, um das System als Coach, selbst mit wenigen programmatischen Vorkenntnissen, anpassen zu können.
Nach einigen visuellen Beispielen für die Anwendung und Interaktion mit der neuen Applikation wird noch auf potenzielle Anwendungsszenarien eingangen, um dann eine kurze Zusammenfassung und ein Ausblick für eine Weiterentwicklung gegeben.

Abstract

Registration and managing appointments for coaching programs should be automated. This is usually done via a web form. However, this approach is rather unsuccessful, which is why an alternative is to be found. This work makes the attempt to use a chat bot technology to solve the problem. After a more detailed introduction to the problem, a number of already available systems are analyzed. It is quickly revealed that these do not meet the requirements for a variety of reasons. Therefore, a solution concept is developed and presented from a combination and extension of several existing systems, followed by a detailed description of the realization and implementation of that concept. Parts of the application are exemplarily highlighted more than others. Challenges and more complex system and their relationships are addressed in particular. It is explained in detail how the coaching bot works in order to establish an understanding that is sufficient for other coaches to be able to adapt the system even with little previous programming knowledge.
After some visual examples of how to use and interact with the new system, potential application scenarios are discussed, followed by a brief summary and outlook for further development.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1

Einleitung und Problemstellung

Viele junge Menschen die nach einem abgeschlossenen Studium in die Arbeitswelt einsteigen möchten, haben keine oder wenig Erfahrung damit, wie sie sich vorbereiten sollen oder welche Schritte erforderlich sind, um einen erfolgreichen Start zu schaffen. Persönliche Beratungsleistungen und speziell EinzelCoaching können dabei helfen, sich effektiv vorzubereiten und einen erheblichen Wettbewerbsvorteil bieten, sind aber für Berufseinsteiger ohne signifikante finanzielle Mittel meist weder zugänglich noch erschwinglich. Aber auch Professionals mit hinreichenden Mitteln, haben oft Hemmungen und die Einstiegshürde ich hoch.

Um diese Zielgruppen unkompliziert und intuitiv anzusprechen und so die Einstiegshürde für derlei Services herabzusetzen, gewinnen digitale, mobile Messenger-Dienste und Social-Media-Kanäle zunehmend an Relevanz. Interaktionen mit jungen Absolventen auf klassischen Websiten stellen sich erfahrungsgemäß vor Allem im persönlichen Networking als wenig erfolgreich heraus. Daneben geht man als Coach mit einem Erstgespräch immer in eine relativ riskante Vorleistung, weil die erste Kennenlern-Session meist gratis angenoten werden muss, um überhaupt Neukunden zu gewinnen. Aus dem Bedürfnis, mit geringem kontinuierlichen Planungs- und Koordinationsaufwand ein breit gefächertes Klientel im Coaching-Bereich aufzubauen, hat man sich dazu entschieden, diverse Kanäle zu prüfen und ggf. Systeme und Technologien für einen standardisierten Onboarding-Mechanismus / Workflow zu etablieren, um zumindest den Prozess bis zum Kennenlernen von manuellem Aufwand zu abstrahieren und Kosten dahingehend auf ein Minimum zu reduzieren. Der Standardisierungscharakter ist deshalb sinnvoll, weil eine erste Kontaktaufnahme und Vorbereitung auf eine erste Sitzung erfahrungsgemäß sehr kongruent zueinander oder gar einem Skript folgend verlaufen.

Die Abwendung vom Web-Browser als Kommunikationsmedium ist keine Neuerung der letzten Jahre und schreitet mit der steten Optimierung von Messenger-Diensten wie WhatsApp, Signal, Telegram und Kommunikationsoptionen via Social Media Portalen wie Instagram, Facebook, TikTok, SnapChat, etc. weiter voran. So ist bspw. die Conversion Rate auf einem Web-Formular erheblich niedriger als die auf einem auf der gleichen Website eingebundenen Chat-Bot.
Ziel des Projekts ist es, eine erste Version des Coaching Bots zu programmieren,

die es ermöglicht, persönliche Angaben zu machen und den Nutzer zu einer Terminvereinbarung hinführt. Um den Entwicklungsaufwand so gering, wie möglich zu halten, wurde hierzu eine Reihe an Systemen analysiert. Eine detaillierte Aufschlüsselung der Systeme folgt in 1 Verwandte Arbeiten und 3 Grundlagen.

2

Verwandte Arbeiten

Ziel der Recherche war es, einen kleinen, selbst wartbaren, quelloffenen Chatbot zu finden, der von angehenden Coaches mit minimalen Vorkenntnissen und einer einfach verständlichen Dokumentation ohne monetäre Mittel auf die eigenen Bedürfnisse angepasst werden kann. Gleichzeitig sollte es dem Coach überlassen sein, wo der Service gehostet wird. Der Nutzer soll eine einfache, geskriptete OnBoarding-Phase durchlaufen und schließlich einen Termin vereinbaren können. Einen Bedarf für Natural Language Processing besteht in einer ersten Version nicht. Eine Analyse der meisten Frameworks ergibt: Die Kandidaten bieten umfangreiche und komplexe Feature-Sets, die für Enterprise-Grade-Software wahrscheinlich bestens geeignet, aber für die in der Problemstellung beschriebenen Zwecke zu umfangreichend sind. Daneben sind ansonsten valable Optionen teilweise nicht direkt, aber in zweiter Instanz zu eng mit nicht kostenlosen Systemen verknüpft, als dass diese ohne Weiteres genutzt werden könnten.

Analysierte Frameworks

1. Microsoft Bot Framework
2. Botkit
3. Botpress
4. Rasa
5. Wit.ai
6. OpenDialog
7. Botonic
8. Claudia Bot Builder
9. Tock
10. BotMan
11. Bottender
12. DeepPavlov
13. Golem
14. ParlAI by Facebook AI
15. Ana
16. Bot Libre

Für eine bessere Übersicht sind die Systeme in 3 Gruppen zusammengefasst:

2.1 Systeme, die mit Kosten verbunden sind

Diese Frameworks gehören entweder einem Großunternehmen oder sind so eng mit Hyperscaler Services verknüpft, dass sie für eine kostenfreie und quelloffene Verwendung nicht geeignet sind:

2.1.1 Microsoft Bot Framework

Als Corporate ist eine Nutzung des von Microsoft bereitgestellten Frameworks sicher aufgrund vielerlei Plug-and-play-Integrationen sinnvoll. Allerdings bindet man sich damit an die mit Kosten verbundene Cloud Plattform Azure. Das Kriterium der Kostenfreiheit ist somit nicht erfüllt.[1]

2.1.2 Botkit

Botkit ist im Microsoft Bot Framework aufgegangen und unterliegt damit den gleichen soeben genannten Einschränkungen.[2]

2.1.3 Wit.ai

Wit.ai gehört Facebook (inzwischen Meta) und entspricht damit nicht unserer Vorstellung von freier Software.[3]

2.1.4 ParlAI by Facebook AI

Als Teil des Facebook- / Meta-Universums bietet ParlAI wahrscheinlich eines der besten NLPs, die aktuell verfügbar sind. Allerdings befindet sich das Framework noch in Produktion und das Feature wird nicht benötigt.[4]

2.1.5 OpenDialog

Open Dialog ist zwar open source, die Nutzung ist aber mit Lizenzgebühren verbunden.[5]

2.1.6 Tock

Tock ist eine valable Stand Alone Lösung für Chat Bots. Allerdings ist die Kompatibilität mit Plattformen, die ausschließlich kommerziellen Corporationen gehören, nicht mit den Zielen des Coaching Bots vereinbar.[6]

[1] `https://github.com/microsoft/botframework-sdk`
[2] `https://github.com/howdyai/botkit-cms`
[3] `https://github.com/wit-ai`
[4] `https://ai.facebook.com/blog/state-of-the-art-open-source-chatbot/`
[5] `https://www.opendialog.ai/`
[6] `https://github.com/theopenconversationkit/tock`

2.1.7 Botpress

Botpress ist ein sehr mächtiges Bot Framework, das alles in 1 genannten Anforderungen entspricht. Der Umfang der Dokumentation sowie die Komplexität der Anwednungsfälle lässt darauf schließen, dass man als Laie umfangreiche Einarbeitung benötigt. Im Zeitrahmen dieser Arbeit ist eine Einarbeitung leider nicht plausibel. Darüber hinaus übersteigt das Natural Language Understanding, das für fortgeschrittene Chatbos eines der Hauptfeatures ist. die hier geforderten Zwecke.[7]

2.1.8 Bottender

Auch Bottender erfüllt auf den ersten Blick alle Anforderungen, die an das Framework gestellt wurden. Allerdings scheint es, als wären die Features, die für die Zwecke des Coaching-Bots benötigt werden, nicht einfacher zu implementieren als auf einer Chatbot Boiler Plate. Durch Bottender wären aber Komplexität und Gewicht der Applikation erheblich erhöht.[8]

2.1.9 Rasa

Rasa bietet mit seinem Story-Feature genau das, was man sich als Coach wünscht. Nämlich, den potenziellen Coachee mit auf seine persönliche Reise zu nehmen. Allerdings bedarf Rasa, um gut zu funktionieren eines umfangreichen Datensatzes, anhand dessen die AI lernen kann und dieser liegt uns leider nicht vor.[9]

2.1.10 Botonic

Botonic bietet genau das, was wir gesucht haben: Eine Kombination aus Text- und grafischen Schnittstellen. Allerdings sind wie auch für die Nutzung von Botonic, wie für Botpress, umfangreiche Vorkenntnisse erforderlich. Eine Weiterverwendung und Individualisierung durch weitere Coaches ist daher unwahrscheinlich.[10]

2.1.11 Claudia Bot Builder

Claudia Bot Builder reduziert die Komplexität, einen Bot selbst zu bauen und zu konfigurieren erheblich und bietet somit genau die Features, die eine einfache Adaption ermöglichen. Leider ist die Software aber ausschließlich auf AWS Lambda ausführbar und somit mit regelmäßigen Kosten verbunden.[11]

2.1.12 BotMan

Als das populärste Bot-Framework der Welt, stellt BotMan einen soliden Kandidaten für unseren Coaching Bot dar.[12]

[7] `https://botpress.com/`
[8] `https://github.com/yoctol/bottender`
[9] `https://github.com/RasaHQ/rasa`
[10] `https://github.com/hubtype/botonic`
[11] `https://github.com/claudiajs/claudia-bot-builder`
[12] `https://github.com/botman/botman`

2.1.13 DeepPavlov

Das auf mächtige und qualitativ hochwertiges NLP ausgelegte Framework DeepPavlov ist weitaus zu mächtig und entspricht nicht dem geskripteten OnBoarding-Prozess, der für den CoachingBot verfolgt werden soll.[13]

2.1.14 Golem

Aus den gleichen Gründen wie bei Bottender und DeepPavlov ist uns auch Golem nicht dienlich. Weder werden für die erste Version des Bots NLU benötigt, noch bietet Golem mehr relevante Features, als die Vanilla-Version des Telegram-Bots.[14]

2.1.15 Ana

Ana bietet ein SDK, über das ein Chatbot in Applikationen integriert werden kann. Da wir aber bestehende Messenger Applikationen nutzen möchten, schließen wir Ana aus.[15]

2.1.16 Bot Libre

Bot Libre ist auf Android beschränkt. Ein dignifikanter Anteil aller Mobile-User wäre dadurch von unserer Zielgruppe ausgeschlossen.[16]

2.1.17 Telegram Bot

Der Instant Messenger Telegram ist (neben vielen anderen) ein beliebtes Kommunikations- und Interaktionsmedium, das Funktionen weit über den einfachen Nachrichtenaustausch hinaus bietet. Unter Anderem bietet Telegram mit seiner sehr intuitiven und einfach zu bedienenden Telegram Bot API ein Framework, das all unseren Anforderungen für eine erste Version des Bots entspricht und es uns erlaubt, eine schlanke, geskriptete OnBoarding-Applikation zu erstellen.

[13] `https://github.com/deepmipt/deeppavlov`
[14] `https://github.com/prihoda/golem`
[15] `https://www.ana.chat/`
[16] `https://www.botlibre.com/`

3

Grundlagen

Die folgenden Sprachen und Systeme und deren grundsätzliches Verständnis dienen als Grundlage für die im Rahmen dieses Projekts entwickelte Applikation. Eine Liste aller eingebundenen Bibliotheken kann dem Pipfile des Projets entnommen werden.

3.1 Python

Python ist aufgrund ihrer Simplizität beim Erlernen und im Syntax sowie ihrer Interpretationsfähigkeit eine der populärsten Programmiersprachen des 21. Jahrhunderts, was dem Anwender im Hinblick auf die Anforderung, den Code leicht und ohne umfangreiche Vorkenntnisse adaptieren zu können, entgegen kommt. Um den CoachingBot (fortan auch „Bot“) zu programmieren wird eine Sprache bevorzugt, die für Server-seitiges Development geeignet ist, geskriptete Abläufe gut abbilden kann, einfache Bindungs-Mechaniken an Datenbanken bereithält und dem Entwickler die Freiheit lässt, seine Applikation methodisch, objektorientiert oder prozedural zu entwickeln.[Ref21] Python erfüllt nicht nur all diese Anforderungen - auch die Telegram-API ist für Python-Implementierungen geschrieben.[1] Der größte Teil der Applikation ist daher in Python 3.8.6 [Pyt21d] geschrieben.[2] Für einen Einstieg in die Programmiersprache Python und als Vorbereitung auf Kapitel 5.7 Implementierung wird das benutzerfreundliche Python Tutorial von W3C-Schools empfohlen.[3]

3.2 Telegram Chat Bots

Telegram Chat Bots sind Applikationen, die auf einer quelloffenen API [Tol21b] basieren, auf allen Komplexitätsstufen adaptierbar sind und es jedermann ermöglichen, einen Chatbot zu bauen. Die einzige suboptimale Einschränkung besteht im Vendor Lock-In der Telegram-App. (Der Bot ist nur in Verbindung mit der Telegram-App [Tel21a] nutzbar.) Aufgrund der technischen Vorteile, die dieses Framework

[1] `https://python-telegram-bot.org/` Andere Sprachen sind verfügbar, aber nicht einfacher und bauen ebenfalls auf der Python-Version der Telegram-API auf.
[2] Das war zum Stand des Entwicklungsbeginns 2021 die aktuell stabile Python-Version.
[3] `https://www.w3schools.com/python/default.asp`

bietet, ist dieser Nachteil jedoch in einer ersten Version in Kauf zu nehmen. Sollte das Prinzip Erfolg versprechen, so kann die Logik mittels Frameworks wie 2.1.12 Botman auf andere Umgebungen erweitert werden. Die meisten Menschen in Deutschland und der DACH-Region verwenden immer noch WhatsApp [Meh22] und doch bietet uns der populärste Messenger nicht die Freiheiten und den Funktionsumfang, den wir uns für unseren CoachingBot wünschen. Telegram jedoch hält genau diese Offenheit für uns bereit. [Kre21] So bietet der Dienst, die Möglichkeit, via einer API direkt in die Entwicklung einzusteigen und hält sogar basale State-Machines für uns bereit, die uns die Komplexität für den Kern des Bots nicht komplett abnehmen, aber als Gerüst für den CoachingBot dienen können.

3.3 Telegram API Framework

Große Teile des CoachingBots basieren auf der API des Instant Messaging Dienstes Telegram[Tol21b] sowie deren Extension [Tol21c].

3.3.1 Telegram API

„Die Telegram Bot-API ist eine HTTP-basierte Schnittstelle für Entwickler, die Bots für Telegram erstellen möchten.“[Tel21b]

3.3.2 Telegram API Extension

Die `telegram.ext` baut auf der reinen API-Implementierung aus 3.3.1 auf. Sie besteht aus mehreren Klassen. Die beiden Wichtigsten für den CoachingBot sind `telegram.ext.Updater` und `telegram.ext.Dispatcher`. Die Updater-Klasse holt kontinuierlich neue Aktualisierungen von Telegram ab und gibt sie an die Dispatcher-Klasse weiter. Ein Updater-Objekt erstellt einen Dispatcher und verknüpft diesen mit einer Warteschlange. Im Dispatcher-Handler können dann verschiedene Typen registriert werden, die die vom Updater abgeholten Aktualisierungen entsprechend den registrierten Handlern sortieren und an eine vordefinierte Callback-Funktion übergeben. Für die Nutzung ist ein Access Token erforderlich.[BJ22] Mehr Informationen dazu, wie ein solches Token erstellt wird in Abschnitt Realisierung 5.1.1.

3.4 ConversationBot

Das hier verwendete Kernelement des CoachingBots - die Finite State Machine (z.dt. Endlicher Automat, im Folgenden nur „State Machine“) - basiert auf dem ConversationBot von Leandro Toledo et. al. [Tol21a].[4] In Abb. 38 ist eine vereinfachte Version zu sehen. Nach der Einbindung der `telegram`- und `telegram.ext`-Bibliotheken, werden die Zustände der State Machine definiert. In diesem Setup gibt es drei angedeutete Arten von Methoden (später Handler-Functions): die

[4] Das Repository enthält eine Vielzahl basaler Bot-Implementierungen, die als Startpunkt für viele Bot-Implementierung einen guten Einblick in mögliche Grundgerüste und Funktionsweisen geben können.

`start`-, die `state[n]`- und die `cancel`-Funktion. Erstere und Letztere werden verwendet, um die Konversation mit dem Bot manuell zu starten und zu beenden. Über die `state[n]`-Methoden können Nachrichten an den Nutzer, Konditionen für den Übergang zum nächsten Zustand sowie weitere Zusatzfunktionen ausgelöst werden, die aus dem entsprechenden Zustand resultieren sollen. In der `main`-Funktion werden der Updater sowie der Dispatcher inkl. aller Conversation- und Command-Handler angemeldet und konfiguriert. Außerdem wird der Abfrage-Loop an Telegram (Polling) gestartet. Auf die genaue Implementierung wird in Abschnitt 6.1 eingegangen.

```
1               from telegram import ReplyKeyboardMarkup, ReplyKeyboardRemove, Update
2               from telegram.ext import (Updater, CommandHandler, MessageHandler, Filters,
3                   ConversationHandler, CallbackContext)
4
5               STATE01, STATE02 = range(2)
6
7               def start(update: Update, context: CallbackContext) -> int:
8                   update.message.reply_text('Welcome␣Message')
9                   return STATE01
10
11              def state01(update: Update, context: CallbackContext) -> int:
12                  user = update.message.from_user
13                  update.message.reply_text('state␣dependant␣message
14  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣and␣transition␣to␣next␣state')
15                  return ConversationHandler.END
16
17              def cancel(update: Update, context: CallbackContext) -> int:
18                  user = update.message.from_user
19                  update.message.reply_text('Bye!␣I␣hope
20  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣we␣can␣talk␣again␣some␣day.',
21                  reply_markup=ReplyKeyboardRemove())
22                  return ConversationHandler.END
23
24              def main() -> None:
25                  updater = Updater("TOKEN")
26                  dispatcher = updater.dispatcher
27                  conv_handler = ConversationHandler(
28                      entry_points=[CommandHandler('start', start)],
29                      states={STATE01: [MessageHandler(Filters.text &
30                      ~Filters.command, state01)],},
31                      fallbacks=[CommandHandler('cancel', cancel)],)
32                  dispatcher.add_handler(conv_handler)
33                  updater.start_polling()
34                  updater.idle()
35
36              if __name__ == '__main__':
37                  main()
```

Listing 3.1: ConversationBot Boiler Plate

3.5 SQLite

Zur persistenten Speicherung von Nutzerdaten wird eine SQLite Datenbank [The21] genutzt. SQLite ist das weltweit am weitesten verbreitete Datenbank-System und bietet mitunter die einfachste und dennoch hinreichend verlässliche und robuste Möglichkeit, Daten über die Lebensdauer des Bots hinaus in einem Datenbankformat zu speichern. SQLite ist v.A. geeignet für lokal betriebene Applikationen, die

realtiv wenige, gleichzeitige Datenbankoperationen erwarten und nicht verteilt sind oder große Enterprise-Grade-Applikationen bedienen müssen. Das System ist kostenlos, wartungsfrei, quelloffen und leicht über die Bibliothek sqlite3 [Pyt] mit dem Python-basierten CoachingBot zu verknüpfen. Der Database-Connector bietet basale CRUD-Operationen, die durch die im Kapitel 5.7 Implementierung erläuterte Umsetzung ohne Weiteres genutzt werden können. In Abschnitt 5.5 Datenbank wird näher auf die Struktur des Database-Connectors eingegangen.

3.6 Mailing-Service und Mail-Server

Die Applikation versendet als Bestätigung der Anmeldung eine E-Mail von einem Mail-Server. Dazu wird ein privat gehosteter Mail-Server genutzt. Der Mailing-Service funktioniert mit jedem beliebigen Mail-Server. Zugangsdaten dazu können in den `_constants` angepasst werden. Vertieftes Wissen über die Funktionsweise sind nicht erforderlich.

3.7 Web-GUI

Um dem Coach eine Übersicht über Anmeldungen und Nutzerinformationen anzuzeigen, stellt die Applikation eine Web-GUI via HTML, CSS und Flask bereit. (Die GUI kann angepasst werden. Dies ist aber weder für die Verwendung des Bots noch der Übersicht erforderlich.) Sollte eine Anpassung an diesem Modul gewünscht sein, sind Grundkenntnisse der drei folgenden Elemente empfohlen:

- HTML[5] - Mark-Up der im Webbrowser auszugebenden Inhalte
- CSS[6] - optische Aufbereitung der Web-GUI via CSS-Stylesheet
- Flask[7] - Audbau eines lokalen Web-Servers, um HTML und CSS an den Browser zu übergeben

3.8 Google Calendar API

Die Applikation bindet die Google Calendar API [Goo22b] an, um es dem Nutzer zu ermöglichen, einen Termin mit dem Anbieter zu vereinbaren und diesen später auch wieder in der eigegen Kalender-Applikation abzulehnen. Um die API nutzen zu können, bedarf es der Installation der API (via pipenv) und der Einrichtung der quickstart.py. Sie stellt den Rahmen für die Authentifizierung gegenüber dem Google OpenAuthorization (oauth2) Protokoll und bindet erste Bibliotheken ein.[8] Für den CoachingBot ist die quickstart.py bereits konfiguriert und wurde durch

[5] `https://www.w3schools.com/html/default.asp`
[6] `https://www.w3schools.com/css/default.asp`
[7] `https://flask.palletsprojects.com/en/2.0.x/#user-s-guide`
[8] Eine genaue Dokumentation zu Aufbau und Nutzung der quickstart.py findet sich hier: `https://developers.google.com/calendar/api/quickstart/python`

einige Erweiterungen zum Calendar Manager weiterentwickelt (siehe 6.5 calendar_manager.py). Zur Nutzung durch Dritte bedarf es dabei individueller Schlüssel sowie Zugangsberechtigungen, durch deren Setup nun geführt wird. Vorgängig ist die Dokumentation zur Google Cloud Console zu sichten.[9]

3.8.1 Scope

Die API kann auf verschiedene sog. ‚Scope'(z.dt. Umfang oder Reichweite) eingestellt werden. So wird festgelegt, welche Rechte dem Kalender Manager gegenüber der API zur Verfügung stehen und welche Methoden, die die API bietet, genutzt werden können. So wäre bspw. der Scope „.../readonly"verfügbar, über den ein Kalender nur abgefragt, aber keine Termine erstellt werden können. Der Bot nutzt den umfangreichsten Scope.[10]. Über ihn stehen alle Operationen der API zur Verfügung.

3.8.2 Zugangsdaten

Um sich via OAuth zu authentifizieren, bedarf es folgender Schritte in der Google Cloud Console. Obwohl diese Schritte bereits durchgeführt wurden, so sind diese bei einer Fremdimplementierung dennoch erneut erforderlich - damit die erforderlichen Schlüssel erstellt werden.

1. Erstellung eines Google Accounts
2. Registrierung dieses Accounts als Google Developer Account
3. Anlegen eines Projekts in diesem Google Developer Account
4. Deklaration des Projekts als Testprojekt
5. Eintragen eines Testers (das Gleiche oder ein anderes Google-Konto kann verwendet werden.)
6. Generierung eines Schlüsselpaares zur Authentifizierung
7. Verifizierung der eigenen Website[11]
8. Freigabe der Redirect-URI für dieses Schlüsselpaar
9. Generierung und Herunterladen der Zugangsdaten (credentials.json)
10. Installation der quickstart.py im eigenen Repository
11. Anpassung der quickstart.py (Angabe des Pfads zum credentials.json)
12. Ausführen der quickstart.py zur Generierung des lokalen PartnerTokens für die Authentifizierung
13. Anpassung der quickstart.py (Angabe des Pfads zum SicherheitsToken)
14. Erneutes Ausführen der quickstart.py, um zu testen, ob die ersten 10 Events des angegebenen Kalenders abgefragt werden konnten.

Bei Erfolg kann die quickstart.py als Testskript bestehen bleiben, wird aber für den Bot nicht mehr benötigt.

[9] `https://console.cloud.google.com/`
[10] `https://www.googleapis.com/auth/calendar`
[11] `https://www.google.com/webmasters/verification/home?hl=en`

3.9 TheCoachingBot

Der gesamte Quellcode inklusive aller Abhängigkeiten findet sich in einem öffentlichen GitHub-Repository.[12]

[12] `https://github.com/mwel/coaching_bot`

4

Konzept

Um sich auf die in 1 Einleitung und Problemstellung gennanten Zielgruppen (Personas „Nutzer“und „Coach“) einzulassen, hat man sich nach einiger Analyse entschieden, in einem ersten Schritt einen Chat-Bot zu programmieren, der basale Informationen vom Nutzer abfragt und den Bewerber einen Termin vereinbaren lässt. Weitere Iterationen sind nach erfolgreicher Beta-Phase möglich und ein Ausblick wird in 8.1.2 Zusammenfassung und Ausblick gegeben. Die Recherche ergibt ein Zweistufenmodell nach dem in einem ersten Schritt die Telegram Bot API genutzt wird, um einen Proof of Concept zu erstellen und eine geskriptete Variante des Bots zu schreiben, mit der der Approach getestet werden kann. In einem zweiten Schritt kann das BotMan Framework genutzt werden, um die Logik des Bots inkl. der bis dahin gesammelten Erfahrungswerte in eine Version 2 einfließen zu lassen, die mit weiteren Plattformen kompatibel ist und durch ihre bessere Skalierbarkeit auch kommerzialisiert werden könnte. Im Rahmen dieser Arbeit wird Schritt 1 des Zweistufenmodells umgesetzt.

4.1 Grundkonzept, User Journey und Features

Die Persona „Nutzer“soll einen vordefinierten Workflow durchlaufen, der im Folgenden skizziert und im Abschnitt 4.3 näher beschrieben wird:

1. Ein Nutzer kommt entweder via Link oder durch die Telegram-Suchfunktion zum CoachingBot.[1]
2. In Telegram angekommen öffnet sich ein neuer Chat mit dem CoachingBot.[2]
3. Der Bot stellt sich vor und beginnt eine Reihe an Fragen zu stellen. Antworten oder deren Format sind z.T. vordefiniert und werden vorgeschlagen.
4. Der Nutzer teilt Texte, Bilder und andere Informationen.
5. Sowohl Coach als auch Coachee sollen spätestens nach Abschluss des Konversationsflusses die Möglichkeit haben, eingegebene Daten einzusehen.
6. Der Nutzer erhält seine Zusammenfassung automatisch nach der Stufe `SUMMARY` via Chat sowie E-Mail und kann sie zusätzlich manuell vom Bot abfragen.

[1] `https://t.me/thecoachingbot?start=start`

[2] Abhängig davon, ob der start-Zusatz schon in der URL inkludiert war oder nicht, muss jetzt der Befehl start eingegeben werden.

7. Sobald alle Informationen eingereicht sind, erhält der Nutzer die Möglichkeit, einen Termin zu vereinbaren. Dem Nutzer werden dazu drei Terminvorschläge über die nächsten zehn Tage angeboten, aus denen er frei wählen kann. Die Dauer pro Termin beträgt 50 Minuten. Es können nur Termine ausgewählt werden, die noch nicht belegt sind und innerhalb der Geschäftszeiten liegen. (Auf die Auswahl der Zeitzone wird hier verzichtet, da das Coaching-Angebot aktuell nur in mitteleuropäischer Zeit angeboten wird.) Im Hintergrund wird ein Google Calendar gemanaged.
8. Der Nutzer bekommt eine schriftliche Bestätigung in Form einer E-Mail mit dem vereinbarten Termin.[3] Darüber hinaus kann der Termin jederzeit vom Bot abgefragt werden.
9. Der Bot verabschiedet sich. (Ende der Konversation)

Der Prozess kann natürlich zu jeder Zeit unter- oder abgebrochen werden. Außerdem besteht die Möglichkeit für den Nutzer, seine Daten jederzeit einzusehen, zu löschen oder den Prozess neu zu starten.

Für die Persona „Coach“sollen eingereichte Informationen über alle Bewerber inkl. Termin in einer Übersicht im Web-Browser eingesehen werden können. Außerdem sind alle vereinbarten Termine auch im Calender des Coaches einsehbar.

4.2 Technischer Aufbau

Wie in Abbildung 4.1 rechts mittig skizziert, besteht der Kern des Bots aus einem endlichen Automaten (State Machine), der Zustände vordefiniert und festlegt, wann sich welcher Nutzer in welchem Zustand befindet und von welchem in welchen Zustand er sich wann bewegen darf. An diesen Kern sind als zentrales Steuerungselement des Bots alle anderen Systeme angebunden. Dazu gehören:

1. Die SQLite Datenbank zur Speicherung der Nutzerdaten
2. Die Telegram API, über die die Kommunikation mit dem Telegram Client abgewickelt wird
3. Die Google Calendar API, über die Events erstellt und versendet werden können
4. Der Mail Server, über den E-Mails an den Nutzer versendet werden können.

Der Nutzer interagiert mit der Applikation durch drei Kanäle - in Abb. 4.1 links ersichtlich:

1. Telegram Client: Kommunikation mit dem Bot
2. Calendar Client: Erhalt, Annahme sowie Ablehnung der vereinbarten Termine
3. Mail Client: Erhalten der Zusammenfassung und Bestätigung

Der Bot wird von Nutzern (in Abb. 4.1 links ersichtlich) via einem der verfügbaren Telegram Clients (Mobile oder Desktop) angesprochen und reagiert auf die Eingabe entsprechend. So können verschieden Funktionen ausgelöst werden. Bspw. werden Antworten zurückgegeben, Informationen gespeichert oder es

[3] Sofern der Nutzer seinen Kalender so konfiguriert hat.

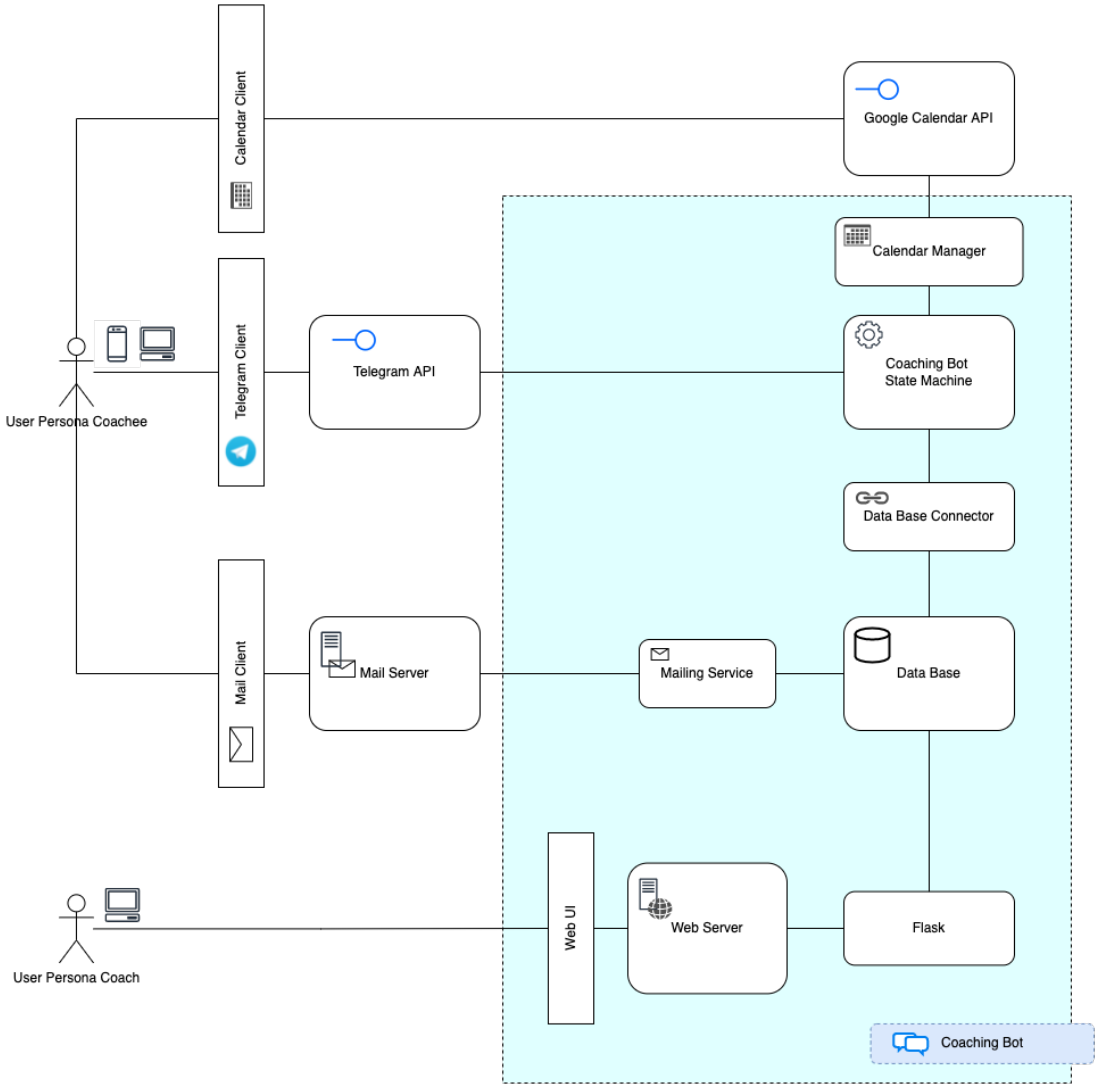

Abbildung 4.1: Konzeptionelle Architektur für das Projekt *Der Coaching Bot*

wird ein Vorschlag gemacht und an den Nutzer zurückgegeben. Der Bot soll mit mehreren Benutzern gleichzeitig sprechen können. Das wird ermöglicht, weil alle Reaktionen des Bots mit der Kennung des jeweiligen Nutzers verknüpft sind. So spricht der Bot den Nutzer mit Namen an oder kann sich daran erinnern, welche Fragen schon beantwortet wurden und welche nicht.

Der Coach interagiert mit der Applikation nur durch einen Kanal (in Abb. 4.1 links unten ersichtlich) - den Web Browser. Hier steht eine einfache Übersicht über Anmeldungen, gesammelte Informationen und der aktuelle, monatliche Terminkalender zur Verfügung.[4]

[4] Der Kalender könnte auch über einen Calender-Client synchronisiert werden können.

4.3 Zustände & Konversationsfluss

Im folgenden Zustandsdiagramm ist der Konversationsfluss des Bots auf hohem Abstraktionsniveau - als endlicher Automat (State Machine) - abgebildet. Es wird deutlich, dass der Bot einem Skript folgt und Loops soweit als möglich vermieden werden sollen. Der Haupt-Pfad ist fett eingezeichnet. Daneben besteht für die meisten Schritte die Möglichkeit für den Nutzer, eine Stufe zu überspringen, wenn er diese Angabe nicht machen möchte. Was aber, wenn der Nutzer den Vorgang unterbrechen möchte oder der Bot aufgrund eines technischen Grundes neu gestartet werden muss und der Nutzer erst danach wieder zur Konversation zurückkehrt? Es muss dem Nutzer möglich sein, dass nach einiger Zeit zum Bot zurückzukehren und an der Stelle weiterzumachen, an der er aufgehört hat. Zu diesem Zweck, dient der Zustand `S0` (START) als zentraler Einstiegspunkt. Hier wird analysiert, ob der Nutzer schon bekannt ist und falls ja, bis wohin er den Prozess bereits durchlaufen hat. Dann wird er dorthin weitergeleitet. Daher ist es möglich von START aus zu allen anderen Zuständen zu gelangen, auch wenn das nicht die Regel ist. Hat der Nutzer den Prozess bereits abgeschlossen, so kann er sogar von S0 direkt in S10 (ENDE) landen und wird entsprechend benachrichtigt. Da dem Nutzer die Möglichkeit gegeben werden soll, den Prozess jederzeit zu beenden, ist es auch möglich, von jedem Zustand in `S10` (ENDE) zu gelangen.[5] Übergänge sind nach dem im Abschnitt 38 ConversationBot skizzierten Format der `state[n]`-Methoden aufgebaut. Eine detaillierte Beschreibung des Aufbaus dieser Methoden findet sich beispielhaft im Abschnitt **??** Zustands-Funktionen: Handler-Functions im Kapiel 5.7 Implementierung. Der Nutzer löst die Übergänge durch seine Eingabe aus und wird dann durch die State Machine automatisch zum entsprechenden nächsten Zustand geleitet. Sobald der Bot gestartet wird, befindet er sich im Zustand S0.

[5] Der Konversationsfluss ist in einer sehr detaillierteren Ansicht verfügbar, in der der Unterschied zwischen dem hohen Abstraktionsniveau des Automaten und der Realität sichtbar wird. So lässt sich leicht erkennen, wo die Konversation beginnt und welche Zustände und Übergänge möglich sind: `https://github.com/mwel/coaching_bot/blob/main/thesis/images/220320_PA28464_Conversation_Flow.svg`

Zustände	**Beschreibung**	
S0	START	Eingabe Biographie oder nächster Schritt oder Abbruch
S1	BIO	Eingabe Geschlecht oder nächster Schritt oder Abbruch
S2	GENDER	Eingabe Geburtsdatum oder nächster Schritt oder Abbruch
S3	BIRTHDATE	Eingabe E-Mail Adresse oder Abbruch oder Abbruch
S4	EMAIL	Eingabe Telefonnummer oder nächster Schritt oder Abbruch
S5	TELEPHONE	Eingabe Ort oder nächster Schritt oder Abbruch
S6	LOCATION	Eingabe Bild oder nächster Schritt oder Abbruch
S7	PHOTO	Alle Daten angegeben oder übersprungen oder Abbruch
S8	SUMMARY	Ausgabe Zusammenfassung
S9	APPOINTMENT	Terminvereinbarung oder Abbruch
S10	ENDE	Ende: Applikation beendet (Neustart möglich)

Tabelle 4.1: Zustände des Konversationsflusses

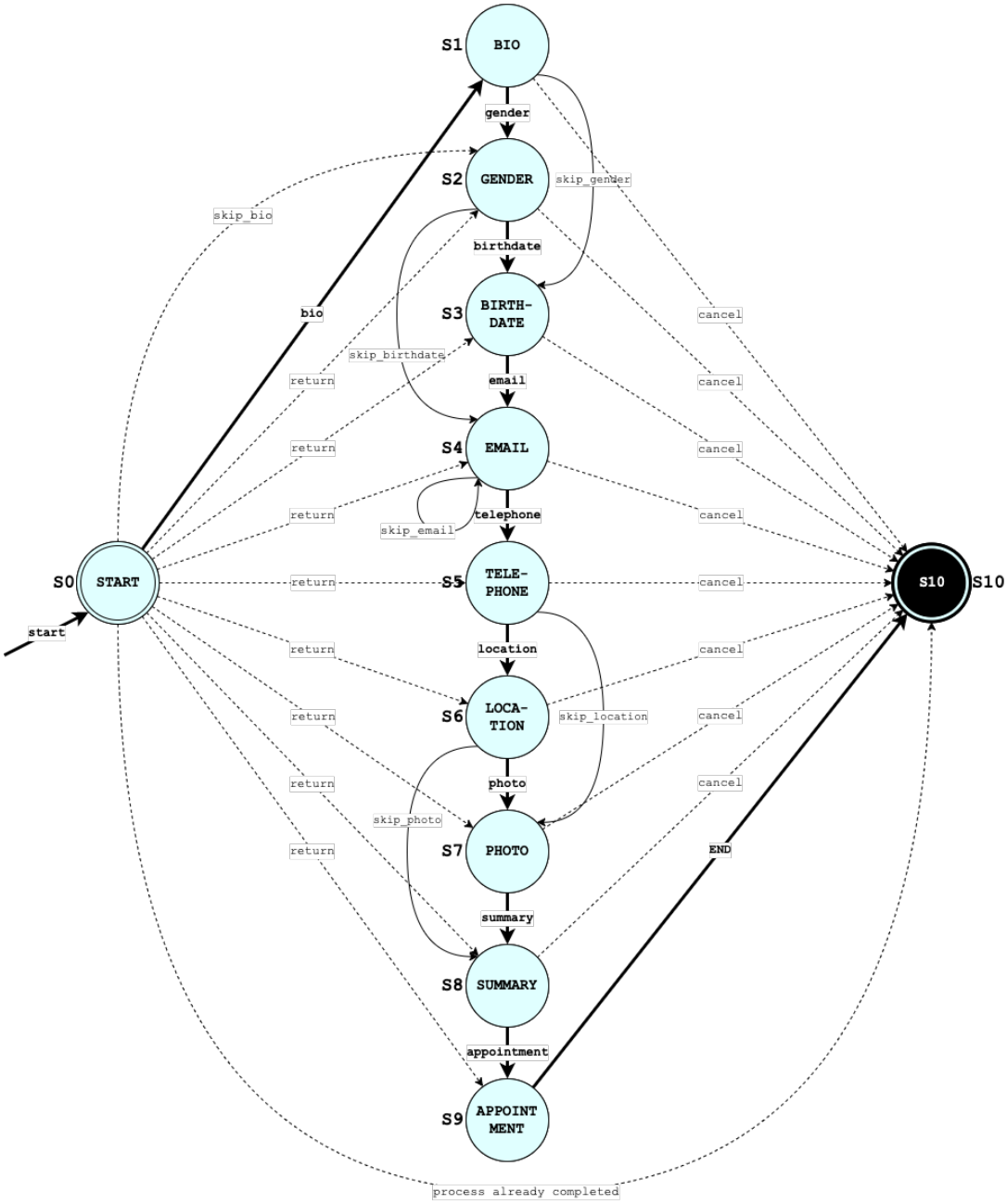

Abbildung 4.2: Endlicher Automat des Konversationsflusses des Bots

5

Realisierung

Das Konzept aus Kapitel 3.9 wird nun weiterentwickelt und Überlegungen dazu angestellt, wie das Konzept realisiert werden kann. Es wird auf das Telegram Framework und dessen Setup allgemein und dann auf die Funktionen im Zusammenhang mit den einzelnen Zuständen des Konversationsflusses eingegangen. In darauffolgenden Abschnitt wird die Lösungsansätze für die Datenbank und das Kalender-Feature erklärt.

5.1 Telegram Bot Framwork

Bevor mit der eigentlichen Implementierung begonnen werden kann, muss zunächst eine Art Bot-Rohling registriert werden. Daraufhin kann ein Template verwendet werden oder ein ganz neuer Bot implementiert werden. Im Folgenden werden diese beiden Schritte erläutert:

5.1.1 Generierung Telegram Bot

Als ersten Schritt zur Erschaffung eines Telegram-Bots wird der Bot-Father[1] konsultiert. Er erstellt das Framework, registriert den Bot und gibt ein API-Token zurück, das verwendet werden kann, um sich gegenüber der Telegram-Bot-API als Entwickler zu identifizieren. [Tol21b] Das Token dient als einzige Identifikationsmethode. Jeder, der in Besitz des Schlüssel ist, kann den zugehörigen Bot theoretisch nutzen und bearbeiten. Das Token ist also an einem sicheren Ort aufzubewahren und nicht in einer öffentlichen Versionierung freizugeben.

5.1.2 Vanilla Bot Implementierung

Als Basis (boiler plate) für den Bot nutzen wir die breit in der Community abgestütze Implementierung **Conversation Bot** [Tol21a]. Sie stellt uns eine basale Implementierung einer State Machine zur Verfügung, die in der Implementierung erheblich ausgebaut wird, uns aber als guter Einstieg dient. Im Gegensatz dazu ist der „Nested Converation Bot“schon bei Weitem zu umfangreich und zu mächtig für unsere Zwecke.

[1] `https://core.telegram.org/bots#6-botfather`

5.2 Rahmen- und Meta-Funktionen

Als Rahmen für die Interaktion zwischen Nutzer und Bot soll es neben den Zustands-Funktionen in Abschnitt 5.3 (Funktionen, die zum Gesprächsfluss gehören), eine Reihe an Meta-Funktionen geben, die dem Nutzer zur Verfügung stehen, um eine Konversation zu starten, zu beenden, Daten zu löschen oder eine Hilfe aufzurufen.

5.2.1 Start: Eine Konversation beginnen

Der Nutzer startet den Bot über den Aufruf eines Links oder mit dem Befehl `/start` (Aktion) und der Bot gibt eine Begrüßungsnachricht zurück (Reaktion). Gleichzeitig erfasst er grundsätzliche Informationen des Nutzers und schreibt diese in eine Datenbank. Ab diesem Zeitpunkt kennt die Applikation den Nutzer und kann weitere Informationen über ihn speichern oder individuell auf Eingaben reagieren. Der Bot soll einen Nutzer wiedererkennen und ihn am richtigen Punkt zurück in den Konversationsfluss platzieren. Diese Erfahrung soll für den Nutzer nicht angestrengt wirken, sondern so, als würde der Bot ihn schon kennen und einfach da weitermachen, wo er aufgehört hat. Die technische Komplexität besteht darin, dass das Feature besonders dann funktionieren soll, wenn der Bot neu gestartet wurde. Am Ende jeder Zustands-Funktion muss die nächste Stufe also aus den Nutzerdaten bekannt sein, damit der Bot in S0 weiß, auf welche Stufe der Nutzer weitergeleitet werden muss. Der Zustand in dem der Nutzer sich zuletzt befunden hat, wird also ab der ersten Kontaktaufnahme persistent in der Datenbank gespeichert. Die Memory-Funktion findet zu Beginn der Konversation statt.

5.2.2 Ende: Konversation manuell beenden

Hat der Nutzer eine Konversation gestartet, so kann er diese auch zu jedem Zeitpunkt wieder beenden. Die Konversation muss nicht zu Ende geführt worden sein. Über einen kurzen Befehl `/cancel` wird der Bot beendet und personenbezogene Daten werden aus der Datenbank gelöscht. Dabei kann der Nutzer nur seine eigenen Daten löschen. Hat der Nutzer seine Konversation bereits beendet, so ist `/cancel` nicht mehr verfügbar, da der Conversation-Handler bereits beendet ist und keine Aktionen mehr entgegennimmt. Möchte der Nutzer seine Daten dennoch löschen, so steht ihm stattdessen folgende Funktion zur Verfügung:

5.2.3 Persönliche Daten löschen

Zu jeder Zeit hat der Nutzer die Möglichkeit, die eigenen Daten via dem Befehl `/delete` zu löschen. Die Funktion ist mit einem „Reset-Knopf“zu vergleichen. Das Resultat ist nämlich, dass der Bot den Nutzer nicht mehr kennt. Er weiß nicht, dass er schon einmal da war und auch nicht, welche Angaben er gemacht hat oder nicht. So kann man den Bot nach fehlerhafter Eingabe oder, falls man neu anfangen möchte, einfach zurücksetzen.

5.2.4 Hilfe-Funktion aufrufen

Manchmal wünscht man sich als Nutzer eine Übersicht über verfügbare Befehle. Die Hilfe-Funktion gibt eine Beschreibung der Aktions-Optionen aus, die es gegenüber dem Bot gibt. So werden alle Befehle einfach erklärt und können auch direkt aus der Hilfe heraus aufgerufen werden.

5.3 Zustands-Funktionen

Die Kommunikationslogik des Bots basiert auf der State Machine aus Abb. 4.2. Die Zustände, in denen der Bot sich befinden kann, sind vordefiniert und immer mit einer Aktion und einer Reaktion verbunden. Aktionen werden von Seiten des Benutzers durch eine Eingabe oder einen Befehl ausgelöst oder vom Bot ausgelöst. Reaktionen sind in Funktionen prädefiniert. Deren Umfang wird im Folgenden funktional und in Abschnitt 6.3 in Kapitel 5.7 Implementierung technisch beschrieben.

Die Übergänge zwischen den Zuständen des Konversationsflusses sind in Funktionen definiert. Auf deren Realisierung wird hier eingegangen:

5.3.1 Hintergrund des Nutzers

Für eine Coaching-Session ist es besonders wichtig, den Coachee besser kennenzulernen. Zu diesem Zweck hat der Nutzer die Möglichkeit, etwas über sich zu erzählen. Erwartet wird hier kein komplettes Motivationsschreiben, sondern einfache, kurz formulierte Beweggründe dafür, dass man gerne mit dem Personal Coaching beginnen möchte.

5.3.2 Abfragen des Geburtsdatums

Um zu erfahren, wie alt der Bewerber ist, möchten wir das Geburtsdatum abfragen. Dabei ist wichtig, dass das Datum in einem sinnvollen Format eingegeben wird. (Siehe Eingabe-Validierung.)

5.3.3 Abfragen des Geschlechts des Nutzers

Um den Nutzer in der Folegkommunikation korrekt anzusprechen, wird nach dem Geschlecht des Nutzers gefragt. Neben der Option, die Frage überspringen zu können, präsentiert der Bot den Nutzer mit mehr als 2 Optionen, um diversen Geschlechtern gerecht zu werden.

5.3.4 Abfragen der E-Mail Adresse des Nutzers

Um dem Nutzer eine E-Mail mit allen erfassten Daten zusenden zu können und dem eigentlichen Zweck des Bots nachzukommen - einen Termin vereinbaren zu können - benötigt der Bot eine valide E-Mail-Adresse des Nutzers. Um die Wahrscheinlichkeit zu erhöhen, dass bei dieser Eigabe keine Fehler passieren, ist auch hier eine Eingabe-Validierung hinterlegt.

5.3.5 Abfragen der Telefonnummer des Nutzers

Am Ende des Konversationsflusses hat der Nutzer die Möglichkeit, einen ersten Termin zu vereinbaren. Dabei handelt es sich um einen unverbindlichen Telefontermin. Um den Nutzer zu einer festgelegten Zeit erreichen zu können, wird hier die Telefonnummer des Nutzers erfasst. Da der Service aktuell nur in der DACH-Region angeboten wird, können hier nur Telefonnummern mit der Länderkennung Deutschland, Österreich und der Schweiz angegeben werden.

5.3.6 Abfragen des Standorts des Nutzers

Der Coaching-Service soll primär und vorerst nur in der DACH-Region angeboten werden. Daher soll der Standort des Nutzers abgefragt werden. Eine Geo-Fencing-Funktion würde für unseren Zweck hier zu weit gehen, weil wir auch Personen die Chance geben wollen, sich für den Dienst anzumelden, die aktuell im Ausland sind. So bietet die Telegram-App dem Nutzer die Möglichkeit, den Ort, den er teilen möchte, spotan selbst auszuwählen.

5.3.7 Abfragen des Bilds des Nutzers

Informationen aller Nutzer werden als Resultat der Teilnahme am On-Boarding in einer Web-GUI ausgegeben. Hier wird neben den Informationen zum Bewerber auch ein Bild angezeigt. So kann der Coach sich besser auf ein erstes Treffen einstellen.

5.3.8 Überspringen

Die meisten Zustände des Bots erlauben es dem Nutzer, die aktuelle Frage zu überspringen. Vor allem, wenn es um personenbezogene oder private Informationen geht, die der Nutzer preisgeben soll, ist der Befehl `/skip` verfügbar. Für jeden Zustand, in dem `/skip` verfügbar ist, ist eine individuelle Reaktion auf das Überspringen vorgesehen, die den Nutzer trotzdem abholt um in den nächsten Zustand leitet.

1. `/skip_bio`: Stufe `BIO` kann ohne Konditionen übersprungen werden. Dann liegen keine Biographie bzw. Freitext-Information über den Nutzer vor.
2. `/skip_gender`: Stufe `GENDER` kann ohne Konditionen übersprungen werden. Eine Angabe über das Geschlecht des Nutzers ist nicht zwingend notwendig.
3. `/skip_birthdate`: Stufe `BIRTHDATE` kann ohne Konditionen übersprungen werden. Das Alter des Nutzers ist in einem nicht-kommerziellen Setup zweitrangig.[2]
4. `/skip_email`: Stufe `EMAIL` kann nicht übersprungen werden. Ohne E-Mail-Adresse können weder eine Zusammenfassungs-E-Mail, noch eine Termineinladung gesendet werden.

[2] Sollte der Coaching-Service kommerzialisiert werden, dann können Services nur für volljährige Nutzer erbracht werden und hier sollte eine Input-Validierung eingebaut werden.

5. `/skip_telephone`: Stufe `TELEPHONE` kann ohne Konditionen übersprungen werden. Es wird aber nicht empfohlen, da der Sinn des Bots die Vereinbarung eines Telefontermins ist.
6. `/skip_location`: Stufe `LOCATION` kann ohne Konditionen übersprungen werden. Die Information über den Standort des Nutzers ist in einem nichtkommerziellen Setup zweitrangig.
7. `/skip_photo`: Stufe `PHOTO` kann ohne Konditionen übersprungen werden. Ohne Bild kann im Überblick für den Coach kein Avatar angezeigt werden, aber erforderlich für die Vereinbarung des Termins ist ein Bild nicht.

Auf die Umsetzung der einzelnen Übersprungsfunktionen wird im Kapitel 5.7 Implementierung genauer eingegangen.

5.3.9 Zusammenfassungs-Funktion

Ziel des Bots ist ein hohes Maß an Transparenz auf allen Seiten. Der Nutzer weiß nicht nur, dass seine Daten erfasst wurden, sondern am Ende des Konversationsflusses werden diese auch automatisch zurückkommuniziert. Dies passiert auf zweierlei Wegen. Neben einer Telegram-Nachricht wird dem Nutzer auch eine Zusammenfassung in Form einer E-Mail an die angegebene Adresse gesendet. Darüber hinaus hat der Nutzer die Möglichkeit, die Zusammenfassung manuell via des Befehls `/summary` direkt vom Bot abzufragen.

5.4 Support-Funktionen

5.4.1 Eingabe-Validierung

Bei einigen Angaben ist es besonders wichtig, dass Eingaben auf korrekte Formate geprüft werden. So müssen bspw. E-Mail-Adresse oder Telefonnummer des Nutzers stimmen, um weitere Features des Bots zu nutzen. Um die Wahrscheinlichkeit dafür zu erhöhen, dass diese Eingaben korrekt sind, werden Nutzereingaben für Geburtsdatum, E-Mail-Adresse und Telefonnummer via regulärem Ausdruck oder dafür vorgesehenen nativen Python-Bibliotheken auf Formatfehler geprüft und der Nutzer bei falscher Eingabe um eine erneute Eingabe gebeten. Liefert der Nutzer eine korrekte Eingabe, landet er im nächsten Zustand. (siehe auch 6.3.2) Input-Validierung: validation.py) Die Übersprungsfunktion besteht, wie in Abschnitt 5.3.8 Überspringen beschrieben, weiterhin.

5.4.2 Konstruktion E-Mail

Die E-Mail, die am Ende des Konversationsflusses ausgegeben wird, wird separat aus verschiedenen Bausteinen zusammengesetzt. Dafür kommen Informationsabfragen gegen die Datenbank mit der Ansprache eines Mail-Servers zusammen.

5.5 Datenbank

Sobald der Nutzer durch den Beginn des Konversationsflusses eine Verbindung zum Bot herstellt hat, wird seine Telegram-ID abgefragt und in eine bestehende oder neue Datenbank geschrieben. Informationen, die der Nutzer angibt, reichern den Datensatz des Nutzers in jeder Stufe sukzessive an. Fast alle Informationen über den Nutzer werden so gespeichert. Ausgenommen ist das Bild, das der Nutzer hochlädt.[3] So können einzelne Werte jederzeit verwendet werden, um Nutzerspezifische Reaktionen zu gestalten. Dem Nutzer stehen die meisten Datenbank-Operationen implizit und wenige explizit zur Verfügung. Daten werden implizit gespeichert und abgerufen. Explizit können Daten gelöscht werden. Zur Realisierung wird, wie in den Grundlagen erwähnt, eine SLQlite Datenbank genutzt, die für den Zweck des Coaching-Bots vollkommen ausreichend ist. Weder ist mit immensen Nutzerzahlen, noch mit vielen gleichzeitigen Operationen oder einer riesigen Datemenge zu rechen, was für mächtigere Lösungen sprechen würde. Da keine komplexen Berechnungen auf den Daten durchgeführt werden, sondern nur basale CRUD-Operaionen geplant sind, kommt der Bot mit einer Tabelle zurecht (siehe Abb. 5.1), in der alle Nutzerdaten gespeichert sind.

5.5.1 Anbindung Datenbank an Python

Der Connector soll prüfen können, ob die Datenbank sie existiert, sich selbst aufbauen können, klassische CRUD-Operationen durchführen können und schnell Antworten liefern, um die Wartezeit bei Zustands-Funktionen, die auf Antwort von der Datenbank warten, so kurz wie möglich bleibt. Folgendes Schema ist für die benötigten Funktionen angedacht:

1. Verbindung zur DB aufbauen. Falls keine DB existiert, eine Neue erstellen.
2. Einen Cursor erstellen.
3. Query erstellen, die abfragt, ob ein Element (hier die Tabelle selbst) schon existiert.
4. Falls die Tabelle nicht existiert, Neue erstellen.
5. Daten lesen, schreiben oder löschen
6. Neuen Stand bestätigen / committen
7. Verbindung schließen.

Eine genauere Beschreibung des für den CoachingBot gebauten Database-Connectors findet sich in Abschnitt 6.4 Datenbank in Kapitel 5.7 Implementierung.

5.6 Kalender

Um am Ende des Konversationsflusses einen ersten Termin mit einem Coach vereinbaren zu können, muss der Nutzer einen Termin auswählen können und für

[3] Es wird als Datei gespeichert - genau wie andere Medien, die man mit einem anderen Teilnehmer einer Konversation über einen Messenger wie Telegram teilt.

users	
user_id (PK)	INTEGER
time_stamp	TEXT
first_name	TEXT
last_name	TEXT
gender	TEXT
photo	BLOB
birthdate	INTEGER
email	TEXT UNIQUE
telephone	TEXT UNIQUE
longitude	INTEGER
latitude	INTEGER
bio	TEXT
state	INTEGER
mail_sent	BOOLEAN
appointment	TEXT
event_id	TEXT

Abbildung 5.1: coachingBot_DB - Datenbankmodell

diesen eine Einladung beantragen. Zu diesem Zweck soll die Google Calendar API angebunden werden. Der Nutzer wird zunächst gefragt, ob er überhaupt einen Termin vereinbaren möchte. Daraufhin wird die API abgefragt und dem Nutzer werden einige Terminvorschläge gemacht. Vorschläge sollen über eine Spanne von 10 Tagen verteilt sein und nur an Wochentagen und zu Geschäftszeiten möglich sein. Da sich Geschäftszeiten ändern können und dazu keine Anpassung am Programmcode notwendig sein soll, werden Geschäftszeiten direkt im entsprechenden Kalender festgelegt. Mit einem Klick kann der gewünschte Termin dann ausgewählt werden. Kurz darauf erhält der Nutzer eine Termineinladung an die zuvor angegebene E-Mail-Adresse und kann diese im persönlichen Kalender-Client annehmen

oder ablehnen.

5.7 Web-GUI

Um gesammelte Daten und vereinbarte Termine am Ende anzeigen zu können, wird eine einfach Web-GUI mittels HTML und CSS erstellt und auf einem lokalen Flask Web-Server deployed. Die Anbindung an die Datenbank ist bereits über den Database Connector implementiert, der auch für die Web-GUI die Daten liefert. Für eine einfache, direkte Einsicht in vereinbarte Termine, wird eine Google-Calendar View eingebunden.

6

Implementierung

Die im Kapitel 4.3 Realisierung erarbeiteten Ansätze wurden umgesetzt. Die Systeme und Ergebnisse werden im Folgenden beschrieben.

6.1 main.py - Anmeldung Bot, Updater, Dispatcher und Handler-Konfiguration

Auf die Mechanismen der `main.py`-Funktion wird in den folgenden drei Code-Abschnitten eingegangen:

```
1  # Hand over API_TOKEN to the bot
2  bot = telegram.Bot(token=API_KEY)

4  def main() -> None:
5  # Creates the updater and passes the API_TOKEN to it.
6  updater = Updater(API_KEY)
```

Listing 6.1: bot/main.py(1) - Authentifizierung und Schlüssel-Übergabe an den Updater

Die `main.py`-Funktion basiert auf dem im Kapiel 3 Grundlagen vorgestellten Conversation Bot. Sie importiert alle Handler, authentifiziert sich gegenüber der Telegram API (wie in Listing 7 zu sehen), instanziiert den Bot und seinen Dispatcher und bindet dann in einem genesteten Aufbau Message- und Command-Handler and Conversation-Handler und diese wiederum an den Dispatcher, um den Bot in einen Zustand zu versetzen, in dem er Befehle entgegennehmen und entsprechend reagieren kann.

Dispatcher liefern Nachrichten an den Nutzer aus. Pro Bot gibt es grundsätzlich mind. einen Dispatcher. Der Coaching Bot hat aber mehrere für mehrere Konversationsstränge. An den Dispatcher werden Conversation- sowie Command-Handler gebunden und konfiguriert (wie in Listing 7 ersichtlich).
Der Conversation-Handler bildet die in Abb. 4.2 konzipierte State Machine ab. Als solche, kontrolliert er den Konversationsfluss zwischen dem Nutzer und dem Bot. Pro Bot kann es mehrere Conversation-Handler geben. Der CoachingBot hat aber nur Einen: den `conv_handler`. Der Conversation-Handler koordiniert alle im

Haupt-Konversationsfluss enthaltenen Command-Handler.
Pro Zustand aus der State Machine gibt es einen Command-Handler. Für jeden Command-Handler gibt es ein Set an Kriterien und / oder einen Befehl, der die zugeordnete Zustands-Funktion auslöst. i.e. den Befehl `/start` für die Funktion `start` aus `start.py` im gleichnamigen Command-Handler, der hier als Einstiegspunkt definiert ist. Funktionen werden nur mittelbar vom Nutzer und unmittelbar von der State-Machine ausgelöst.
Befehle werden vom Nutzer eingegeben und vom Message-Handler entgegengenommen. Ein Message-Handler prüft die Eingabe eines Nutzers in einem bestimmten Zustand auf vordefinierte Kriterien und meldet das Ergebnis an den Conversation-Handler zurück, der auf dieser Basis dann entscheidet, ob er die zugehörige Zustands-Funktion auslöst oder nicht. Diese Kriterien werden direkt in der main.py in einem der Message-Handler definiert.
Handler-Funktionen sind Zustands-Funktionen (siehe 5.3). Sie werden ausgelöst, wenn Eingaben vom Message-Handler als valide interpretiert werden.

```
1       # Gets the dispatcher to register handlers
2       dispatcher = updater.dispatcher

4       # bot state machine and main conversation handler
5       conv_handler = ConversationHandler(
6           entry_points=[CommandHandler('start', start)],
7           states={
8               states.BIO:         [MessageHandler
9                   (Filters.text& ~Filters.command, bio),
10                  CommandHandler('skip', skip_bio)],
11              states.GENDER:      [MessageHandler
12                  (Filters.regex('^(Gentleman|Lady|Unicorn)$'),gender),
13                  CommandHandler('skip', skip_gender)],
14              states.BIRTHDATE:   [MessageHandler
15                  (Filters.text& ~Filters.command, birthdate),
16                  CommandHandler('skip', skip_birthdate)],
17              states.EMAIL:       [MessageHandler
18                  (Filters.text& ~Filters.command, email),
19                  CommandHandler('skip', skip_email)],
20              states.TELEPHONE:   [MessageHandler
21                  (Filters.text& ~Filters.command, telephone),
22                  CommandHandler('skip', skip_telephone)],
23              states.LOCATION:    [MessageHandler
24                  (Filters.location& ~Filters.command, location),
25                  CommandHandler('skip', skip_location)],
26              states.PHOTO:       [MessageHandler
27                  (Filters.photo   & ~Filters.command, photo),
28                  CommandHandler('skip', skip_photo)],
29              states.SUMMARY:     [MessageHandler
30                  (Filters.regex('^(COMPLETE␣SIGN␣UP)$'),  summary)],
31              states.APPOINTMENT: [MessageHandler
32                  (Filters.text& ~Filters.command, appointment),
33                  CommandHandler('skip', skip_appointment)],
34              # more states here...
35              },
36              fallbacks = [CommandHandler('cancel', cancel)],
37      )

39      dispatcher.add_handler(conv_handler)
40      # more conversation handlers for secondary commands
41      dispatcher.add_handler(CommandHandler('summary', summary))
42      dispatcher.add_handler(CommandHandler('delete', delete))
43      dispatcher.add_handler(CommandHandler('status', status))
```

Listing 6.2: bot/main.py(2) - Dispatcher, Conversation- & Command-Handler

Über die folgenden Befehle, werden die entsprechenden, gleichnamigen Command-Handler und Funktionen ausgelöst:

- `/start` - startet den Konversationsfluss
- `/cancel` - beendet den Konversationsfluss, löscht Nutzerdaten aus der Datenbank und informiert entsprechend
- `/delete` - löscht Nutzerdaten des Nutzers und informiert ihn
- `/help` - gibt die Hilfe aus
- `/summary` - gibt die Zusammenfassung für den Nutzer aus
- `/status` - gibt gibt den aktuellen Fortschritt für den Nutzer aus

Der in dieser Applikation umfangreichste Conversation-Handler umfasst zwei Befehle: `/start` und `/cancel`. Solange der Bot ausgeführt wird, lässt sich ein Konversationsfluss mit ihm über den Befehl `/start` starten und via `/cancel` beenden. (Zur Funktionsweise der den Befehlen zugeordneten Zustands-Funktionen siehe den entsprechden Abschnitt unter 6.3.)

Schließlich wird die State Machine des Coaching Bots gestartet. Über das sog. Polling werden Aktualisierungen konstant von Telegram nachgeladen. Der Bot ist aktiv (idle) und wartet darauf, Befehle entgegenzunehmen.

```
1       # Start the Bot
2       updater.start_polling()

4       updater.idle()
```

Listing 6.3: bot/main.py(3) - Start Polling Idle

6.1.1 states.py - Zustände zentral verwalten

Die State Machine (der Conversation-Handler) muss zu jeder Zeit wissen, welche Zustände es gibt und in welcher Reihenfolge diese existieren. Dazu nutzt der Bot ein Array aus Konstanten (`STATES`). So lässt sich die Reihenfolge der Zustände auch ganz leicht ändern. Soll der Bot bspw. E-Mail und Telefonnummer zu Anfang abfragen oder sollen einige Schritte aus dem Konversationsfluss genommen werden, so sind diese hier einfach zu entfernen und die Nachrichten in den eizelnen Zustands-Funktionen leicht anzupassen.

Um Nachrichten an den Nutzer zentralisiert zu verwalten, verweisen Handler-Funktionen wo immer möglich auf eine Konstante aus dem `MESSAGES`-Dictionary. So wird vermieden, dass Strings bei Anpassungen der Zustände oder deren Reihenfolge in mehreren Dateien angepasst werden müssen.

Gleiches gilt für individuelle Tastaturen aus dem `KEYBOARDS`-Dictionary.

6.2 Rahmen- und Meta-Funktionen

Die in Kapitel 4.3 Realisierung beschriebenen Rahmen- und Meta-Funktionen wurden folgendermaßen umgesetzt:

6.2.1 start.py - Beginn und Weiterführung des Konversationsflusses

In sechs Abschnitten wird nun die wichtige Funktion `start` beschrieben. Sie fungiert als Eingangstor für jeden Nutzer. Wann immer der Befehl `/start` an den Bot schickt wird, löst der Command-Handler die Funktion `start` aus.
Zunächst wird geprüft, ob es eine Datenbank gibt. Ist dies der Fall, wird geprüft, ob der Nutzer, der die Funktion ausgelöst hat, bereits in der Datenbank existiert. Ist dies wiederum der Fall, gibt die Funktion eine Willkommen-zurück-Nachricht aus.

```
1  def start(update: Update, context: CallbackContext) -> int:
2      # CREATE DB, IF NOT EXISTS
3      create_db()
4
5      user_id = update.message.from_user.id
6
7      user_exists = select_db.user_search(user_id)
8      if user_exists:
9          # get user's state from db
10         state = int(select_db.get_value(user_id, 'state'))
11
12         update.message.reply_text(
13             f'Welcome␣back␣{update.message.from_user.first_name},\n'
14             'Let\'s␣continue␣where␣we␣left␣off...',
15             reply_markup=ReplyKeyboardRemove(),
16             )
```

Listing 6.4: start.py - Aufbau DB und Prüfung ob Nutzer bekannt

Nun wird abhängig vom für den Nutzer gespeicherten Zustand zwischen unterschiedlichen Reaktionen differenziert. Befindet der Nutzer sich im Zustand `SUMMARY`, hat also bereits alle Fragen beantwortet, aber noch keinen Termin vereinbart, so werden in diesem Zustand sinnvolle Optionen empfohlen. Der Nutzer kann sich den Status seiner Bewerbung ausgeben lassen, die Zusammenfassung erneut beantragen oder alle seine Daten löschen.

```
1          if state == states.SUMMARY:
2          # sign up was apparently already completed for this user
3              reply_keyboard = [
4                  ['/status'],
5                  ['/summary'],
6                  ['/delete']]
7              update.message.reply_text(
8              'Ah!␣I␣see,␣you␣have␣already␣completed
9  ␣␣␣␣␣␣␣␣␣␣␣␣␣␣the␣sign␣up.\nYou␣now␣have␣multiple␣options␣below:\n'
10             'If␣you␣have␣not␣made␣an␣appointment␣yet
11 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣and␣would␣like␣to␣do␣so,␣enter␣/summary.\n\n'
12             'If␣you␣want␣to␣/delete␣your␣record␣entirely,
13 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣press␣/delete.',
14             reply_markup=ReplyKeyboardMarkup(
15             reply_keyboard, one_time_keyboard=True,
16             input_field_placeholder='SIGN␣UP␣COMPLETE'
```

```
17                  )
18              )
```

Listing 6.5: start.py - Zustand Zusammenfassung

Befindet der Nutzer sich im Zustand `APPOINTMENT`, hat aber noch keinen Termin vereinbart, die Zusammenfassung aber bereits erhalten, erhält er zusätzlich zur Option, sich die Zusammenfassung erneut ausgeben zu lassen und so die Terminfindung zu starten, nur die Status- und Lösch-Optionen. Natürlich kann der Nutzer auch manuell alle Befehle jederzeit eingeben, aber die Tastatur ist so für Optionen vordefiniert, dass der Nutzer in eine bestimmte Richtung gelenkt wird. Nach der Ausgabe dieser Nachricht, beendet der Conversation-Handler die Kommunikation.

```
1           elif state == states.APPOINTMENT and select_db.get_value(user_id, 'appointment') == 'N
2               reply_keyboard = [
3                   ['/status'],
4                   ['/delete']]
5               update.message.reply_text(
6               'If␣you␣have␣not␣made␣an␣appointment␣yet
7   ␣␣␣␣␣␣␣␣␣␣␣␣and␣would␣like␣to␣do␣so,␣enter␣/summary.\n\n',
8               reply_markup=ReplyKeyboardMarkup(
9               reply_keyboard, one_time_keyboard=True,
10              input_field_placeholder='SIGN␣UP␣COMPLETE'
11              )
12          )
13              return ConversationHandler.END
```

Listing 6.6: start.py - Zustand Terminvereinbarung(noch nicht vereinbart)

Befindet der Nutzer sich im Zustand `APPOINTMENT` und hat bereits einen Termin vereinbart, so werden Informationen zu dem Termin aus der Datenbank abgerufen und direkt ausgegeben. Auch in diesem Fall wird die Konversation nun beendet, da keine weiteren Interaktionen mit dem Nutzer vorgesehen sind.

```
1           elif state == states.APPOINTMENT:
2               appointment_made = select_db.get_value(user_id,
3               'appointment')
4               reply_keyboard = [
5                   ['/status'],
6                   ['/delete']]
7               update.message.reply_text(
8               f'Cool.␣You␣already␣have␣an␣appointment␣on
9   ␣␣␣␣␣␣␣␣␣␣␣␣{appointment_made}␣\n\n'
10              'In␣case␣you␣would␣like␣to␣cancel,
11  ␣␣␣␣␣␣␣␣␣␣␣␣you␣can␣do␣that␣via␣your␣calendar␣app.\n\n'
12              'Otherwise,␣we␣are␣looking␣forward␣to␣our␣call.',
13              reply_markup=ReplyKeyboardMarkup(
14              reply_keyboard, one_time_keyboard=True,
15              input_field_placeholder='SIGN␣UP␣COMPLETE'
16              )
17          )
18              return ConversationHandler.END
```

Listing 6.7: start.py - Zustand Termin (bereits vereinbart)

Egal für welchen Fall die Funktion sich entscheidet, der Nutzer wird immer korrekt in den Konversationsfluss zurückgeführt und zwar genau vor der Frage, die zuletzt

nicht beantwortet wurde.[1] Dazu wird die Datenbank abgefragt und der Wert aus `state` für die entsprechende `user_id` an den Conversation-Handler weitergegeben. Dieser präsentiert als Antwort darauf die nächste Frage im Konversationsfluss.

```
2               # call next function for user
3               update.message.reply_text(states.MESSAGES[state], reply_markup=states.KEYBOARD_MAR
4               return state
```

Listing 6.8: start.py - Weiterleitung in Zustand

Treffen all diese Konditionen nicht zu, wurde der Nutzer also nicht in der Datenbank gefunden, so startet der Bot ganz normal mit einer Begrüßung, nachdem initiale Daten von der Telegram-Instanz des Nutzers abgefragt und in die Datenbank geschrieben wurden.
Ist die Nachricht an den Nutzer ausgeliefert, aktualisiert der Bot den Zustand für den Nutzer in der Datenbank, damit der Bot weiß, welche Fragen der Nutzer schon beantwortet hat und er den Nutzer bei einer Rückkehr wieder am richtigen Punkt in den Konversationsfluss einfügen kann.
Bevor der Bot den Nutzer zur nächsten Stufe weiterleitet, speichert er noch einen Zeitstempel, damit man nachvollziehen kann, wann der Nutzer seinen Prozess begonnen hat.

```
2           logger.info(f'+++++␣NEW␣USER:␣{update.message.from_user.first_name}
3   ␣␣␣␣␣␣␣␣{update.message.from_user.last_name}␣+++++')

5           # write user info to db
6           insert_update(user_id, 'first_name', update.message.from_user.first_name)
7           insert_update(user_id, 'last_name', update.message.from_user.last_name)
8           insert_update(user_id, 'appointment', 'None')
9           insert_update(user_id, 'event_id', '0')

11          update.message.reply_text(
12              f'Hi␣{update.message.from_user.first_name},\n'
13              'I␣am␣a␣coaching␣bot␣by␣wavehoover.␣You␣have
14  ␣␣␣␣␣␣␣␣␣␣␣␣taken␣the␣first␣step␣on␣your␣journey␣to␣success
15  ␣␣␣␣␣␣␣␣␣␣␣␣by␣contacting␣me.␣I␣will␣guide␣you␣through␣the
16  ␣␣␣␣␣␣␣␣␣␣␣␣application␣process␣for␣your␣first␣coaching␣session.␣'
17              'It\'s␣super␣easy.␣Just␣follow␣the␣questions,␣answer
18  ␣␣␣␣␣␣␣␣␣␣␣␣or␣skip␣them␣-␣that\'s␣it.\n\n'
19              '[You␣can␣send␣/cancel␣at␣any␣time,␣if␣you␣are
20  ␣␣␣␣␣␣␣␣␣␣␣␣no␣longer␣interested␣in␣a␣conversation.]\n\n'
21              f'Now,␣{update.message.from_user.first_name}␣-
22  ␣␣␣␣␣␣␣␣␣␣␣␣{states.MESSAGES[states.BIO]}',
23              reply_markup=ReplyKeyboardRemove(),
24              )

26          # save state to DB
27          insert_update(user_id, 'time_stamp', datetime.now())
28          insert_update(user_id, 'state', states.BIO)
29          return states.BIO
```

Listing 6.9: start.py - Standard Einstieg Konversationsfluss

[1] Eine Frage zu überspringen gilt dabei auch als Beantwortung.

6.2.2 cancel.py - Konversationen beenden und Nutzerdaten löschen

Wurde eine Konversation mit dem Bot gestartet und noch nicht beendet, so kann diese auch manuell beendet werden. Über den Befehl `/cancel`, der im Conversation-Handler als Fallback definiert ist, wird die Funktion `cancel` aufgerufen, alle Daten des Nutzers aus der Datenbank gelöscht und eine Bestätigung an den Nutzer ausgegeben. Sollte es bei diesem Vorgang zu einem Fehler kommen, so wird der Nutzer auch darüber benachrichtigt.[2] Schließlich beendet der Bot die Konversation.

Wurde eine Konversation bereits beendet, kann der Conversation-Handler nicht mehr auf den Befehl `/cancel` reagieren. Für diesen Fall gibt es einen extra Dispatcher, der auf den Befehl `/delete` hört und die Daten des Nutzers ebenfalls restlos löscht. So ist sichergestellt, dass der Nutzer seine Daten auch löschen kann, wenn die Konversation mit dem Bot aus irgendeinem Grund unterbrochen oder bereits beendet wurde.

6.2.3 help.py - Hilfe ausgeben

Die Funktion `help` ist eine Meta-Funktion und gibt durch Aufruf des Befehls `/help` die Hilfe aus. Sie setzt ein sog. Dictionary aus einer Liste an Befehlen zusammen, das flexibel befüllt und dann ausgegeben werden kann. Ein Dictionary bietet die Möglichkeit, die Hilfe jederzeit einfach anzupassen, um Elemente zu erweitern oder zu reduzieren ohne die Logik, über die die Hilfe ausgegeben wird, zu beeinflussen. Dazu wird die `collections`-Bibliothek eingebunden, die es erlaubt, ein geordnetes Dictionary zu erstellen. Nachdem der String für die Hilfe zusammengesetzt ist, wird dieser einfach via der Funktion `send\_message` ausgegeben.

6.3 Aufbau und Beispiele Zustands-Funktionen(Handler-Funktionen)

Im Folgenden gehen wir detailliert auf die einzelnen Handler-Funktionen ein, beschreiben deren Umfang und Aufbau und erklären ihre Funktionsweise.
Die Funktion `bio` in der `bio.py` speichert die Text-Eingabe eines Nutzers als erste Nutzereingabe nach dem `/start`-Befehl. Sie repräsentiert besonders gut den Aufbau der Handler-Funktionen, weil sie über das Speichern und weiterleiten keine weiteren Features besitzt. Daher erläutern wir den Aufbau der Handler-Funktionen beispielhaft anhand der Funktion `bio` für alle anderen Handler-Funktionen. Der Aufbau aller weiteren Handler-Funktionen ähnelt der Funktion 22. Auf signifikante Erweiterungen und Anpassungen wird in den entsprechenden Abschnitten eingegangen.

[2] Eine Ausnahme tritt auf, wenn der Nutzer seine Daten bereits gelöscht und den Bot noch nicht neu gestartet hat - es ihn also in der Datenbank gar nicht gibt oder (sehr selten), falls die SQL-Operation nicht erfolgreich war.

```
# Stores the information received and continues on to the next state
def bio(update: Update, context: CallbackContext) -> int:

    user_id = update.message.from_user.id
    bio_message = update.message.text

    # write info to DB
    insert_update(user_id, 'bio', bio_message)

    # reply keyboard for next state
    update.message.reply_text(
        'What␣a␣story!␣We␣will␣definately␣pick␣that␣up
␣␣␣␣␣␣␣␣␣␣␣␣␣in␣our␣first␣session!\n\n' + \
        'Ok␣-␣now␣let\'s␣get␣some␣basics␣down:␣\n' + \
        states.MESSAGES[states.GENDER],
        reply_markup=states.KEYBOARD_MARKUPS[states.GENDER],
        )

    # save state to DB
    insert_update(user_id, 'state', states.GENDER)
    return states.GENDER
```

Listing 6.10: bio.py - Beispiel einer Handler-Funktion

Zunächst werden ein Update- und ein CallbackContext-Objekt an die Handler-Funktion übergeben. Zurückgegeben wird der Datentyp `int`, da die State-Machine am Ende der Funktion wissen muss, in welchen Zustand der Nutzer als nächstes geschickt werden soll und die Zustände nummeriert sind.
Innerhalb der Funktion werden `user_id` und `bio_message` aus dem Update-Objekt gespeichert, da diese beiden informationen als nächstes in die Datenbank gespeichert werden sollen. Die `bio_message` ist in diesem Fall die Text-Eingabe, die an den Bot übermittelt wurde. Die `user_id` ist die Telegram-ID des jeweiligen Nutzers, an die alle Nutzerinformationen geknüpft werden. (zur Funktionalität der `insert_update`-Funktion siehe Abschnitt **??** insert_update.py) Die Hintergrundprozesse sind nun abgeschlossen und die für den Nutzer sichtbare Reaktion auf seine Nachricht kann erfolgen. Die Funktion `update.message.reply_text` erlaubt es uns, dem Nutzer einen beliebigen String sowie eine für diese Nachricht individuelle Antwort-Tastatur auszugeben. (Die zu übergebenden Parameter für die Zustandsbergänge sind für die `reply_text`-Instanzen in der `states.py` zentral gespeichert, um sich innerhalb der einzelnen Handler-Funktion soweit als möglich von Inhalten zu abstrahieren.) Ist die Ausgabe an den Nutzer erfolgt, bleibt noch die Aktualisirung des neuen Zustands des Nutzers in der Datenbank, gefolgt von der Übergabe des nächsten Zustands an den ConversationHandler. Damit ist die Zustands-Funktion beendet und der Übergang von einem in den nächsten Zustand abgeschlossen. Diese Funktion leitet in den Zustand `GENDER`.

Im Zustand `START` ist, wie aus der State Machine ersichtlich, noch eine zweite Funktion verfügbar. Die `skip_bio` kann vom Nutzer durch den Befehl `/skip` ausgelöst werden. Dieser Befehl ist in jedem Zustand spezifisch für den Command-Handler dieses Zustands definiert und hat in jedem Zustand einen angepassten

Effekt. Anhand der Funktion `skip_bio` wird wieder beispielhaft aufgezeigt, wie `skip`-Funktionen aufgebaut sind:

```
1  # Skips this information and continues on to the next state
2  def skip_bio(update: Update, context: CallbackContext) -> int:

4      user_id = update.message.from_user.id

6      # alternative message
7      update.message.reply_text(
8          'Alright. No problem...', # individual skip message
9          reply_markup=ReplyKeyboardRemove(),
10         )
11     # reply keyboard for next state
12     update.message.reply_text(
13         states.MESSAGES[states.GENDER],
14         reply_markup=states.KEYBOARD_MARKUPS[states.GENDER],
15         )

17     # save state to DB
18     insert_update(user_id, 'state', states.GENDER)
19     return states.GENDER
```

Listing 6.11: skip_bio.py - Zustände überspringen

Der Aufbau ähnelt zwar der `bio`-Funktion, allerdings liegt hier ein reduzierter Umfang und eine andere Nachricht an den Nutzer vor. Ein Update der Datenbank fällt zudem weg, da der Nutzer keine neuen Informationen angegeben hat. Hier werden zwei `reply_text`-Funktionen hintereinander verwendet. Die Erste dient dazu, eine auf diese `skip`-Funktion individuelle Nachricht zu übermitteln. Die Zweite übermittelt die Aufforderung zur Eingabe der Information für die nächste Stufe und zeigt die entsprechende Tastatur an. Alle weiteren `skip`-Funktionen sind ähnlich aufgebaut.
Der Zustand `EMAIL` ist der einzige Zustand, der nicht übersprungen werden kann. Ohne eine gültige E-Mail-Adresse des Nutzers können wichtige Folgefunktionen des Bots nicht genutzt werden und der Sinn und Zweck (eine Terminvereinbarung) kann nicht erfüllt werden. Daher ist die Funktion `skip_email` so gestaltet, dass sie keinen Zustand zurückgibt, sondern den Nutzer immer im aktuellen Zustand belässt. Es bleiben die Optionen, eine gültige Adresse anzugeben oder die Konversation zu beenden.

Nach den beiden exemplarisch vorgestellten Funktionen wird im weiteren Verlauf des Kapitels nur noch auf Besonderheiten der restlichen Handler-Funktionen eingegangen.

6.3.1 Handler-Funktionen mit Input Validation

Der Bot arbeitet in den Funktionen `birthdate`, `email` und `telephone` mit Input-Validierung. (zur Funktionsweise siehe 6.3.2 validation.py) Dazu wird die Nutzereingabe zunächst an die Funktion `validate_birthdate` übergeben und auf die Bewertung des Inputs gewartet.

```
1  def birthdate(update: Update, context: CallbackContext) -> int:

3      if validate_birthdate(update.message.text): # if entry is valid, continue.
4          insert_update(update.message.from_user.id, 'birthdate', update.message.text)
5          update.message.reply_text(...
6          insert_update(update.message.from_user.id, 'state', states.EMAIL)
7          return states.EMAIL

9      else: # else, tell user and stay in current state until correct entry is provided.
10         update.message.reply_text(
11         f'Sorry, that\'s not a valid entry. Please try again.',
12         reply_markup=ReplyKeyboardRemove(),
13         )
```

Listing 6.12: birthdate.py - Input Validation

Entspricht der Input dem prädefinierten Format, fährt der Bot wie gewöhnlich fort und übergibt den nächsten Zustand zurück an den Conversation-Handler. Ist dies jedoch nicht der Fall, so wird, wie in Listing **??** eine entsprechende Nachricht an den Nutzer ausgegeben. Da der Conversation-Handler erst dann zur nächsten Stufe geht, wenn er von der Funktion `birthdate` den entsprechenden Zustand zurückerhalten hat, entsteht hier ein Loop, der entweder durch eine gültige Eingabe oder i.e. den Befehl `/skip` gebrochen werden kann.

6.3.2 validation.py - Input-Validierung

Die drei Input-Validation-Funktionen sind einander ähnlich per `try/except` oder `if/else` Block aufgebaut:

Die Funktion `validate_birthdate` bekommt die Nutzereingabe übergeben und vergleicht diese via der Funktion `strptime` aus der `datetime`-Bibliothek [Pyt21a] mit dem in der DACH-Region gängigen Datums-Format: `TT.MM.JJJJ`
Stimmt die Eingabe mit dem definierten Format überein, gibt die Funktion `True` zurück. Ansonsten wird ein `ValueError` geloggt und die Funktion gibt `False` zurück.

Die Funktion `validate_email` bedient sich eines regulären Ausdrucks, um zu prüfen, ob die Eingabe eine E-Mail sein könnte:

`[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}`[3]

Ist der Vergleich erfolgreich, gibt die Funktion `True` zurück, ansonsten `False`.

Auch Telefonnummern werden via regulärem Ausdruck geprüft:

`^\+4[139]\d{9,12}$`

Zugelassen sind so alle Telefonnummern aus Deutschland, Österreich und der

[3] Ein vollumfänglicher regulärer Ausdruck, ein externder Dienst oder gar das versenden einer Test-E-Mail wurden aus Performance-Gründen ausgeschlossen.

Schweiz.
Ist der Vergleich erfolgreich, gibt die Funktion `validate_telephone` `True` zurück, ansonsten `False`.

6.3.3 summary.py - Zusammenfassung für den Nutzer

In der Funktion `summary` kommt alles zusammen. Der Nutzer hat nun alle Angaben gemacht oder übersprungen und befindet sich im Zustand `SUMMARY`.
Die Funktion `summary`, wie in den kommenden vier Code-Abschnitten aufgezeigt, beginnt damit, eine Reihe von Informationen von der Datenbank abzufragen und in Variablen zu speichern. Es werden nur Informationen abgefragt, die auch in der auszugebenden Nachricht genutzt werden sollen. Direkt darauf wird ein String für die zu versendende Nachricht zusammengesetzt und gespeichert. Es folgt eine einfache „Danke-Nachricht“an den Nutzer:

```
1  def summary(update: Update, context: CallbackContext) -> int:
2
3      user_id = update.message.from_user.id
4
5      first_name = get_value(user_id, 'first_name')
6      last_name = get_value(user_id, 'last_name')
7      gender = get_value(user_id, 'gender')
8      birthdate = get_value(user_id, 'birthdate')
9      email = get_value(user_id, 'email')
10     telephone = get_value(user_id, 'telephone')
11
12     summary = f"""
13         Given Name:\t\t{first_name}
14         Last Name:\t\t{last_name}
15         Gender choice:\t\t{gender}
16         Birthdate:\t\t\t{birthdate}
17         Email address:\t\t{email}
18         Phone number:\t{telephone}
19         """
20
21     # confirmation message
22     update.message.reply_text(
23         f'Thanks␣for␣signing␣up,␣{update.message.from_user.first_name}!\n\n'
24         f'SUMMARY␣for␣{update.message.from_user.first_name}
25 ␣␣␣␣␣␣␣␣{update.message.from_user.last_name}:\n\n{summary}',
26         reply_markup=ReplyKeyboardRemove(),
27     )
```

Listing 6.13: summary.py(1) - Zusammenfassung für den Nutzer direkt im Messenger

Nun prüft der Bot, ob der Nutzer bereits einen Termin vereinbart hat:

Option A

Der Nutzer ist bis zum Zustand `SUMMARY` gekommen, hat die Zusammenfassung ausgegeben bekommen, dann aber keinen Termin vereinbart und den Chat verlassen. Der Nutzer kehrt nun zum Chat zurück und gibt erneut `/start` ein, um seine Konversation wieder aufzunehmen. Der Bot findet den Nutzer in der Datenbank und leitet ihn an die Stufe `SUMMARY` weiter, um direkt vor der Terminvergabe (Zustand `APPOINTMENT`) einzusetzen. Der Nutzer kann nun einen Termin vereinbaren und durchläuft die folgenden zwei Schritte:

1. Der Bot versucht, dem Nutzer jetzt drei mögliche Terminvorschläge zu unterbreiten und startet dazu die Terminfindung (siehe 6.5 Kalender).
2. Sobald die Termine zurückkommen, präsentiert der Bot diese dem Nutzer in Form eines entsprechenden Tastatur-Layouts.[4]

```
1       # check, if the user already made an appointment.
2       appointment_made = get_value(user_id, 'appointment')
3
4       # If yes, make one. Else, inform.
5       if appointment_made == 'None':
6
7           update.message.reply_text(
8               'Ok. We will look for 3 appointment options
9   ␣␣␣␣␣␣␣␣␣␣␣␣you can choose from for your phone call. \n\n'
10              '... SEARCHING ...',
11              reply_markup=ReplyKeyboardRemove(),
12          )
13
14          free_slots = find_slots()
15
16          slot1 = str(free_slots[0])
17          slot2 = str(free_slots[1])
18          slot3 = str(free_slots[2])
19
20          # next step message
21          update.message.reply_text(
22              states.MESSAGES[states.APPOINTMENT],
23              reply_markup=ReplyKeyboardMarkup(
24                  [[slot1], [slot2], [slot3]], ['/skip'],
25                  one_time_keyboard=True,
26                  input_field_placeholder='Choose your slot...'
27                  )
28              )
```

Listing 6.14: summary.py(2) - Prüfung Termin negativ

Option B

Der Nutzer ist bis zum Zustand `SUMMARY` gekommen und hat bereits einen Termin vereinbart. In diesem Fall fragt der Bot den Termin von der Datenbank ab und gibt ihn in einer Nachricht an den Nutzer zurück. Gleichzeitig schlägt er dem Nutzer weitere mögliche Befehle vor, die an dieser Stelle Sinn machen und beendet die Konversation.

```
1       else:
2           update.message.reply_text(
3               f'Cool. You already have an appointment: {appointment_made} \n\n'
4               'In case you would like to cancel, you can do so via your calendar app.\n\n'
5               'Otherwise, we are looking forward to our call.',
6               reply_markup=ReplyKeyboardRemove(),
7           )
8
9           return ConversationHandler.END
```

Listing 6.15: summary.py(3) - Prüfung Termin positiv

[4] Das Layout ist dabei dynamisch und generiert sich bei jeder Anfrage neu.

Schließlich wird die Funktion `confirmation_mail` aufgerufen, die die gleiche Zusammenfassung nochmals per E-Mail an die Adresse des Nutzers sendet (siehe 6.3.4 confirmation_mail.py) und die Information darüber, dass an diesen Nutzer bereits eine E-Mail gesendet wurde wird neben den üblichen Abschlussbefehlen in der Datenbank gespeichert.

```
1       # trigger confirmation email
2       confirmation_mail(first_name, summary, email)
3       insert_update(user_id, 'mail_sent', '1')

5       # save state to DB
6       insert_update(user_id, 'state', states.APPOINTMENT)
7       return states.APPOINTMENT
```

Listing 6.16: summary.py(4) - Bestätigungs-Mail

6.3.4 confirmation_mail.py - Bestätigung per E-Mail

Um dem Nutzer die Zusammenfassung in Form einer E-Mail zukommen zu lassen (wie in Listing 8 aufgerufen), muss diese zunächst zusammengesetzt werden. Die Bibliotheken `mime` [Pyt21b] und `smtplib` [Pyt21e] helfen dabei, eine sichere Verbindung zu einem Mail-Server aufzubauen, über den die fertige E-Mail versenden werden kann.

Erforderliche Zugangsdaten werden außerhalb der Funktion `confirmation_mail` aus den `constants` abgefragt und für die Verwendung innerhalb der Funktion zwischen-gespeichert. So werden diese nicht bei jedem Funktionsaufruf erneut abgerufen.

Die Funktion bekommt Empfänger-Name sowie -Adresse und die Zusammenfassung aus der Funktion `summary` übergeben. Über die `smtplib` wird ein Server-Objekt erstellt. Gegenüber diesem Server authentifiziert sich der Bot nun via Benutzername und Passwort. War die Authentifizierung erfolgreich, wird die eigentliche Nachricht zusammengesetzt. Dazu benötigt werden vier Bauteile:

1. Sender-Adresse
2. Empfänger-Adresse
3. Betreff
4. Nachricht

Die Nachricht wird zuerst via der Funktion `attache` zusammengesetzt, um dann aus dem vordefinierten String ein `message`-Objekt zu konstruieren. Schließlich kann die E-Mail via `sendmail` unter Verwendung des Mail-Servers versendet werden. Die Verbindung zum Server wieder getrennt.

```
1  def confirmation_mail(recipient_name, summary, recipient_address):

3       # open connection to mail server and authenticate
4       server = smtplib.SMTP_SSL(smtp_address, smtp_port)
5       server.login(sender_address, password)
```

```
7       # create multipart object, the email consists of
8       message = MIMEMultipart()

10      # define from-address, to-address and subject of the email
11      message['From'] = sender_address
12      message['To'] = recipient_address

14      subject = 'Coaching␣Bot␣|␣Confirmation␣-␣sign␣up␣complete'
15      message['Subject'] = subject

17      # email body
18      body =   f"""Hi {recipient_name}, \t
19          thanks for signing up. This is the confirmation for your
20          sign up with the coaching program by wavehoover. \n
21          {summary}\n
22          Looking forward to meeting you!\n
23          Your wavehoover Team"""

25      # create the text object for the email
26      message.attach(MIMEText(body, 'plain'))

28      server.sendmail(message['From'], message['To'], message.as_string())
29      server.quit()
```

Listing 6.17: confirmation_mail.py - Bestätigung und Zusammenfassung für den Nutzer per E-Mail

6.3.5 appointment.py - Kalender-Event erstellen

Um einen Termin zu vereinbaren, muss ein kompatibles JSON-File an die Funktion `make_appointment`, die wiederum die Google Calendar API anspricht, formuliert werden. Ziel der Funktion `appointment` ist es also, den Zeitstempel vom Nutzer entgegenzunehmen, das Kalender-Event zu bauen und es zu übergeben.
Dazu fragt sie zunächst alle erforderlichen Informationen bei der Datenbank ab.

```
1   def appointment(update: Update, context: CallbackContext) -> int:

3       user_id = update.message.from_user.id
4       email = get_value(user_id, 'email')
5       telephone = get_value(user_id, 'telephone')

7       summary = 'wavehoover␣|␣Coaching␣Session'
```

Listing 6.18: appointment.py - DB-Abfrage der zu verbauenden Informationen

Der erhaltene Zeitstempel für den Beginn des Zeitfensters wird dann in ein Format übersetzt, das die Google Calendar API akzeptiert:
`%Y-%m-%dT%H:%M:%S+01:00`
Das Ende des Zeitfensters wird auf 50 Minuten nach dem Start gesetzt und die beiden korrekt formatierten Zeitstempel in das JSON-Event verbaut.[5]

```
1       slot_start = update.message.text
2       dt_slot_start = datetime.strptime(slot_start, '%Y-%m-%d␣%H:%M:%S')
3       iso_slot_start = str(dt_slot_start.isoformat('T') + '+01:00')
```

[5] Format und Aufbau des Events können via dem Google APIs Explorer getestet werden. [Goo22a])

```
5       slot_end = str(dt_slot_start + timedelta(minutes=50))
6       dt_slot_end = datetime.strptime(slot_end, '%Y-%m-%d␣%H:%M:%S')
7       iso_slot_end = str(dt_slot_end.isoformat('T') + '+01:00')
```

Listing 6.19: appointment.py - Formatierung des Zeitstempels in RFC3339

```
1       # build the event data into the event object
2       event = {
3           f'summary': summary,
4           'location': 'Phone␣Call',
5           'description': f'Your␣coach␣will␣call␣you
6   ␣␣␣␣␣␣␣␣under␣the␣following␣number:␣{telephone}',
7           'start': {'dateTime': iso_slot_start,
8                     'timeZone': 'Europe/Berlin',},
9           'end': {'dateTime': iso_slot_end,
10                  'timeZone': 'Europe/Berlin',},
11          'attendees': [{'email': email},],
12          },
13          }
```

Listing 6.20: appointment.py - Konstruktion des Kalender-Events

Ist der Aufruf an die Funktion `make_appointment` abgesetzt und ohne Fehler zurückgekehrt, so wird die Datenbank entsprechend um den Start-Zeitstempfel erweitert und der Nutzer wird informiert.

```
1               make_appointment(user_id, slot_start, event) # hand over user info to make appoint
2               insert_update(user_id, 'appointment', slot_start)
```

Listing 6.21: appointment.py - Konstruktion des Kalender-Events

Überspringt der Nutzer diesen letzten Schritt, gibt ihm die Funktion `skip_appointment` lediglich eine Nachricht aus, die die Option offen lässt, auf anderem Weg mit dem Coach in Kontakt zu treten.

6.3.6 status.py - Fortschritt abrufen

Die Funktion `status` ist eine Meta-Funktion und gibt dem Nutzer seinen aktuellen Fortschritt zurück. Dazu prüft die Funktion `status` zunächst, ob der Nutzer überhaupt in der Datenbank existiert. Zwei Szenarien:

1. Der Bot findet den Nutzer, gibt den aktuellen Status zurück und beendet die Konversation.
2. Der Bot findet den Nutzer nicht und zeigt dem Nutzer Optionen an, fortzufahren - namentlich die Hilfe aufzurufen oder eine neue Konversation mit dem Bot zu starten.

6.4 Datenbank

In diesem Abschnitt wird die Funktionsweise des Database-Connectors erläutert. Der Database-Connector wird unter der Zuhilfenahme der sqlite3-Bibliothek programmiert, die es ermöglicht, klassische Datenbank-Operationen direkt aus einem

Python-Skript heraus anzustoßen und hier als Basis für den Database-Connector dient. Die CRUD-Operationen selbst werden in handelsüblichem SQL formuliert und übergeben. Die Datenbankoperationen sind in separate Funktionen aufgeteilt, die immer an die Struktur aus 5.5 angelehnt sind. Zunächst wird eine Verbindung zur Datenbank geöffnet, es finden Prüfungen statt, eine CRUD-Operaion wird abgesetzt und die Antwort entweder innerhalb der Funktion analysiert und ein `Boolean` oder die übergebenen Daten aufbereitet und im entsprechenden Format zurückgegeben. Alle Funktionen sind ähnlich aufgebaut.

6.4.1 create_db.py - Datenbank und Schema aufbauen

Es wird versucht, eine Verbindung zur Datenbank `db` aufzubauen. Ist dies erfolgreich, wird der `cursor` erstellt und geprüft, ob es die Tabelle `users` schon gibt. Ist dem so, wird die Funktion mit einem Commit und dem Schließen der Verbindung zur Datenbank, beendet.
Ist eine der beiden Prüfungen nicht erfolgreich, wird das entsprechende Element neue erstellt. Dabei entspricht das `CREATE`-Statement hier dem Datenbankschema aus Abb. 5.1.

```
1  # imports
2  import sqlite3

4  db = 'coachingBotDB.db'

6  # build a new db, if none exists yet
7  def create_db ():
8      # connect to db
9      connection = sqlite3.connect(db)
10     # cursor
11     cursor = connection.cursor()

13     # table checker sql statement
14     checker = '''SELECT count(name)
15         FROM sqlite_master
16         WHERE type='table' AND name='users'
17     '''

19     # table creation sql statement
20     table_users = '''CREATE TABLE IF NOT EXISTS users (
21         user_id INTEGER PRIMARY KEY,
22         time_stamp TEXT,
23         first_name TEXT,
24         last_name TEXT,
25         gender TEXT,
26         photo BLOB,
27         birthdate INTEGER,
28         email TEXT UNIQUE,
29         telephone TEXT UNIQUE,
30         longitude INTEGER,
31         latitude INTEGER,
32         bio TEXT,
33         state INTEGER,
34         mail_sent BOOLEAN,
35         appointment TEXT,
36         event_id TEXT
37     );'''

39     cursor.execute(checker)
```

```
41      #if count is 1, table already exists - else, create it
42      if cursor.fetchone()[0]==1:
43          print('Table already exists.')
44      else:
45          cursor.execute(table_users)
46          print('Table created.')

48      connection.commit()
49      connection.close()
```

Listing 6.22: Database Connector mit sqlite3

6.4.2 select_db.py - Nutzerdaten abfragen

Hier wird kurz beispielhaft auf die beiden wichtigsten READ-Funktionen des Database-Connectors hingewiesen und wo sie verwendet werden:

Die Funktion `user-search` bekommt eine Nutzer-ID übergeben und prüft, ob ein Nutzer in der Datenbank existiert. Falls ja, wird `True` zurückgegeben - ansonsten `False`. Die Funktion wird in der Funktion `start` genutzt (siehe 17), wenn geprüft wird, ob ein Nutzer bereits bekannt ist.

Der Funktion `get_value` werden eine Nutzer-ID sowie eine Spaltenbezeichnung übergeben. So kann ein spezifischer Wert aus der Datenbank abgerufen werden. (Die meisten Handler-Funktionen bedienen sich dieser Funktion, um mit Informationen über den Nutzer zu arbeiten.)

```
1  # get a specific value of a user
2  def get_value(user_id, column):
3      ...

5      selection = f"""SELECT {column}
6          FROM users
7          WHERE user_id = {user_id}
8          """

10     # execute command to fetch all data from table users
11     cursor.execute(selection)

13     # store all data from selection in table_data
14     table_value = (str(cursor.fetchone())).lstrip("('").rstrip("',)")
15     return table_value
```

Listing 6.23: select_db.py - Datenbankabfrage einzelner Nutzerinformationen

6.4.3 insert_value_db.py - Werte schreiben

Um sicherzustellen, dass eine Datebank existiert, wird zunächst die Funktion `create_db` aufgerufen. Diese kommt schnell zurück, da die Datenbank in den meisten Fällen bereits existiert. Nun wird in einem `try/except`-Block versucht (siehe Listing **??**), einen existierenden Datenbankeintrag zu aktualisieren. Das funktioniert meistens, weil der Datensatz für einen Nutzer bereits bei der Eingabe von

`/start` durchgeführt wird. So kann ein Eintrag bereits von Beginn an immer weiter angereichert werden.

```
1       ...
3       try:
4           cursor.execute('INSERT␣INTO␣users␣(user_id)␣VALUES␣(?)', (user_id,))
5           logger.info (f'+++++␣CREATED␣record␣{user_id}:␣{cursor.lastrowid}␣+++++')
6       except sqlite3.IntegrityError:
7           logger.info("+++++␣FOUND␣record...␣UPDATING...␣+++++")
9       # sql command to UPDATE an existing record
10      update_command = f"UPDATE␣users␣SET␣{column}␣=␣?␣WHERE␣user_id␣=␣?"
11      update_args = (value, user_id)
13      cursor.execute(update_command, update_args)
15      ...
```

Listing 6.24: select_db.py - Datensatz eines Nutzers anreichern

6.4.4 delete_record.py

Die Funktion `delete_record` bekommt eine Nutzer-ID übergeben und prüft zunächst via der Funktion `user_search`, ob es den Nutzer mit der angegebenen Nutzer-ID gibt. Falls ja, wird in einem `try/except`-Block versucht, alle Informationen eines Nutzers via SQL-Befehl zu löschen.[6] Die Funktion wird in den Funktionen `cancel` und `delete` aufgerufen.

6.5 Terminvereinbarung mit dem Google Calendar

Um die Anforderungen aus der Realisierung in Abschnitt 5.6 zu erfüllen, ergo mit der Google Calendar API zu kommunizieren, sind die Schritte aus Kapitel 3 Grundlagen, Abschnitt **??** Google Calendar API durchzuführen. Der Kalender-Manager basiert auf der `quickstart.py` und erweitert diese. Daneben nutzt der Kalender-Manager Pythons native Bibliothek zum Zusammenfügen von Dateipfaden (path) und den soeben beschriebenen Database-Connector. Er erlaubt es dem aufrufenden System, abzufragen, ob eine Zeitspanne in einem bestimmten Kalender verfügbar ist und Termine zwischen einer festgelegten Veranstalter-Adresse und beliebig vielen Teilnehmer-Adressen zu erstellen und zu versenden. Im Folgenden werden Aufbau und Funktionsweise der `calendar_manager.py` erklärt:

Die `main`-Funktion des Kalender-Managers authentifiziert sich gegenüber der Google Calendar API und versucht daraufhin, die nächsten zehn Elemente des Kalenders auszugeben. Ist die Authentifizierung nicht erfolgreich, wird ein `HTTP`-Error ausgegeben.

[6] Die Funktion delete_value ist mit der Funktion delete_record fast identisch. Hier wird aber zusätzlich eine Spaltenbezeichnung übergeben, die es ermöglicht, nur einen einzelnen Wert zu löschen.

Die `authenticate` ist eine reduzierte Version der `main`. Sie gibt ein `service`-Objekt zurück, das genutzt werden kann, um die Funktionen der Google Calendar API anzusprechen und auszuführen.

6.5.1 Zeitspanne prüfen

Die Funktion `check_availability` nimmt einen Start- und einen Endzeitpunkt entgegen und prüft die Verfügbarkeit des entsprechenden Kalenders. Sie gibt einen `BOOLEAN` zurück, der aussagt, ob der Termin noch frei ist. Dafür werden die beiden Zeitstempel so formatiert, dass sie dem RFC3339-Format entsprechen. Das Format setzt sich zusammen wie folgt:

YYYY-MM-DD, ‚T', HH:MM:SS.ms, Buchstabe ‚Z'

Beispiel für 10:05 Uhr vormittags am 28.02.2022, koordinierte Weltzeit (UTC):

`2022-02-28T10:05:00.00Z`

Weitere Informationen zum RFC3339-Format finden sich im offiziellen Standard. [eaK02]

Die beiden Zeitstempel werden in eine `JSON`-Anfrage eingebaut:

```
1  def check_availability(start, end):
2
3      service = authenticate()
4      start_iso = str(start.isoformat('T')+'+01:00') # convert UTC to CET
5      end_iso = str(end.isoformat('T')+'+01:00') # same for end time
6      request = {
7          "timeMin": start_iso,
8          "timeMax": end_iso,
9          "timeZone": "Europe/Berlin",
10         "items": [
11             {
12             "id": coaching_calendar_ID
13             }]
14         }
```

Listing 6.25: check_availability - Komposition der Anfrage an die API

In einem `try/except`-Block wird das die Anfrage dann an die API-Funktion `freebusy` übergeben. Ist dies erfolgreich, gibt die API eine Antwort im `JSON`-Format zurück, das Python als Dictionary interpretieren und in mehreren genesteten Schleifen auslesen kann. Ist das Feld `busy: []` leer, so ist das Zeitfenster frei. Es wird `True` zurückgegeben. Ist das Feld nicht leer, gibt es einen Terminkonflikt. Die Funktion gibt `False` zurück.
Erzeugt dieser Vorgang einen Fehler, wird ein `HTTP`-Error auf der Konsole ausgegeben.

```
    try:
        response = service.freebusy().query(body=request).execute()
        # climb down the dict latter and read the busy response from HTTP-Response
        calendars = response.get('calendars')
        calendar_temp = calendars.get(coaching_calendar_ID)
        availability = calendar_temp.get('busy')
        if availability == []:
            return True

    except HttpError as error:
        logger.info('ERROR: %s' % error)
```

Listing 6.26: check_availability - Versant der Anfrage und Analyse der Antwort der API

6.5.2 Terminvorschläge generieren

Dem Nutzer sollen drei Terminvorschläge gemacht werden. Die Funktion `find_slots` sucht folgendermaßen nach drei verfügbaren Zeitfenstern und gibt diese in Form einer Liste zurück:

Die Suche für Termine startet um 08:00 Uhr am kommenden Arbeitstag. Es wird zwischen heute und morgen 08:00 Uhr unterschieden. Liegt der Zeitstempel zur Zeit der Ausführung der Funktion vor 08:00 Uhr, so wird die Suche heute um 08:00 Uhr begonnen. Ist es bereits nach 08:00 Uhr, so wird die Suche am nächsten Tag begonnen. Das Resultat ist der Zeitstempel von heute oder morgen 08:00 Uhr, der als Startzeit für die Suche verwendet wird.

```
def find_slots():

    # Get today's datetime
    datenow = datetime.datetime.now()
    # Create datetime variable for 8 AM
    dt8 = None
    # If today's hour is < 8 AM
    if datenow.hour < 8:

        # Create date object for today's year, month, day at 8 AM
        dt8 = datetime.datetime(datenow.year, datenow.month, datenow.day, 8, 0, 0, 0)

    # If today is past 8 AM, increment date by 1 day
    else:
        # Get 1 day duration to add
        day = datetime.timedelta(days=1)
        # Generate tomorrow's datetime
        tomorrow = datenow + day

        # Create new datetime object using tomorrow's year, month, day at 8 AM
        dt8 = datetime.datetime(tomorrow.year, tomorrow.month, tomorrow.day, 8, 0, 0, 0)
```

Listing 6.27: find_slots - Sucht drei Terminvorschläge heraus

Nun wird die Google Calendar API für das erste Zeitfenster abgefragt. Ist das Fenster frei, übernimmt die Funktion den Slot und fährt mit der Suche fort. Der nächste Termin wird drei Tage später gesucht, um mehrere unterschiedliche Termine anbieten zu können. Ist ein Fenster belegt, so wird der Slot zur nächsten

vollen Stunde geprüft und dies solange, bis drei Termine gefunden sind. Neben der Rückgabe als Liste erfolgt eine Ausgabe auf der Konsole.

```
1       # within the business hours, find 3 free time slots to suggest to the user
2       free_slots = []
3       slots = 0
4       start = dt8
5       round = 0
6       while slots < 3:
7           round += 1
8           end = start + datetime.timedelta(minutes=50)
9
10          if (check_availability(start, end)):
11              free_slots.append(str(start))
12              slots += 1;
13              start = dt8 + datetime.timedelta(days=3*slots)
14
15          else:
16              start = start + datetime.timedelta(hours=1)
17
18      return free_slots
```

Listing 6.28: find_slots - Sucht drei Terminvorschläge heraus

Wochenenden und Zeiten vor 08:00 sowie nach 18:00 Uhr werden übersprungen, da die Funktion `check_availability` für diese Zeiten immer `False` zurückgibt.[7]

6.5.3 make_appointment - Finalen Termin vereinbaren

Abschließend kann der Nutzer einen der drei Vorschläge auswählen. Der gewählte Vorschlag wird an die Funktion `make_appointment` übergeben inkl. der Kalender-ID an die API via der API-Funktion `insert` übergeben und so in den entsprechenden Kalender eingefügt.

```
1   def make_appointment(event):
2       service = authenticate()
3       try:
4           service.events().insert(calendarId=coaching_calendar_ID, body=event).execute()
5
6       except HttpError as error:
7           logger.info('ERROR: %s' % error)
```

Listing 6.29: make_appointment - Terminvereinbarung und Erstellung Kalender Event in Google Calendar

Google versendet eine Termineinladung, in der die Informationen enthalten, sind, die im Event mitgeschickt wurden und der Nutzer sowie der Coaching-Kalender bekommen eine entsprechende Einladung. Falls bei dieser Operation ein Fehler auftritt, wird ein `HTTP`-Error auf der Konsole ausgegeben.

[7] Welche Zeiten als busy zurückgegeben werden kann einfach im eigenen Kalender-Client angepasst werden. Eine Anpassung im Code ist nicht erforderlich.

6.6 Web-GUI

Die Web-GUI stellt dem Coach eine Liste der Nutzer, die den Prozess begonnen haben zur Verfügung und zeigt den aus den Anmeldungen resultierenden Terminkalender an. Die GUI wurde mit Flask erstellt und bindet Daten aus der Datenbank, Nutzerbilder, eine Google Calendar View zusammen auf eine HTML-Seite, die mithilfe von CSS gestyled wird.

Die `app.py` instanziiert via der Flask Bibliothek einen Webserver und konstruiert den darzustellenden Inhalt in Form eines Templates. Dieses wird aus Nutzerinformationen zusammengesetzt, die bei Aufrufen der entsprechenden URL aus der Datenbank abgefragt werden sowie aus den von Nutzern eingereichten Bildern. Das fertige Template wird dann an den Webserver zurückgegeben, der die Oberfläche an den Web-Browser liefert.

```
1 # Instantiate Flask App / Build Web Server to display GUI
2 app = Flask(__name__)
3 app.config['SEND_FILE_MAX_AGE_DEFAULT'] = 0

5 @app.route("/")
6 def home():
7     data = get_customers()
8     picture_path = os.path.join('static', 'user_pictures')
9     return render_template('home.html', **config.CONTEXT, customers=data, path=picture_path)
```

Die `home.html` besteht aus drei Teilen. Zunächst wird eine Tabelle erstellt und mit den für die Darstellung der Daten aus der Datenbank erforderlichen Spalten versehen. Daraufhin werden die Daten aus der Datenabank, die eine andere Spaltenreihenfolge haben, als die, in der sie hier dargestellt werden sollen, den entsprechenden Spalten zugeordnet. Schließlich wird eine Google Calendar View via iframe eingebunden.Die Ansicht aktualisiert sich auf Anfrage.
Es kommt ein minimalistisches CSS-Template zum Einsatz, das den Ansprüchen, Nutzerdaten einzusehen leicht genügt.

Über die IP-Adresse `127.0.0.1` und Port `5000` ist die GUI als Admin zu erreichen[8]: `http://127.0.0.1:5000/`

Damit ist das Kapitel 5.7 Implementierung abgeschlossen. Es folgt ein Beispieldurchlauf.

[8] Um das Kalender-Feature einsehen zu können, ist eine Anmeldung mit dem entsprechenden Google-Konto erforderlich.

7

Beispiele

Der Nutzer startet den Bot via dem Klick auf einen Link, den er auf einer Website findet oder der ihm zugesandt wird.

/start

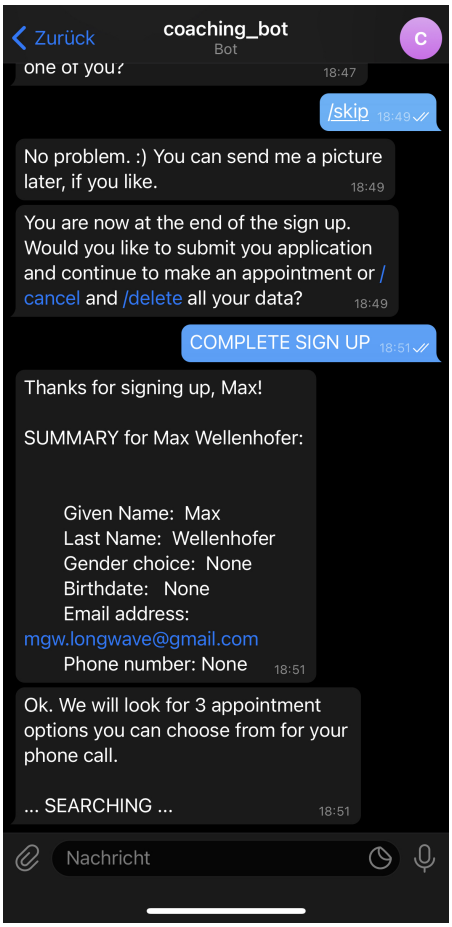

Abbildung 7.1: Bezeichnung der Abbildung

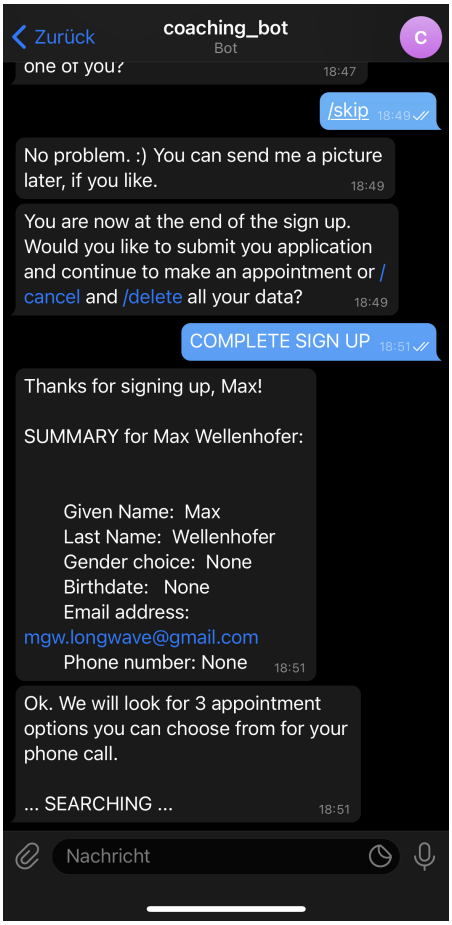

Abbildung 7.2: Bezeichnung der Abbildung

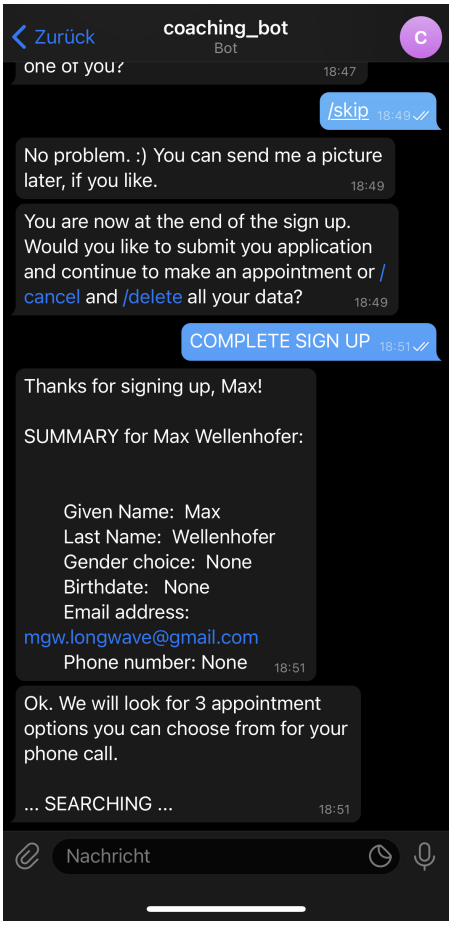

Abbildung 7.3: Bezeichnung der Abbildung

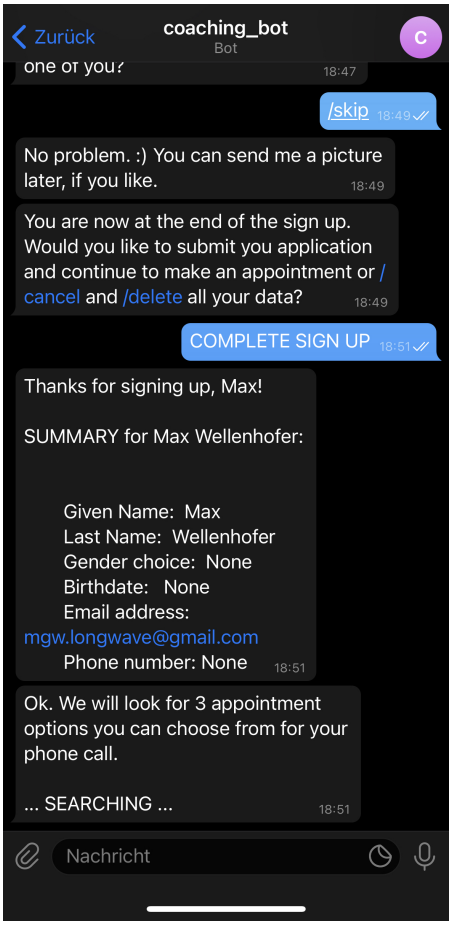

Abbildung 7.4: Bezeichnung der Abbildung

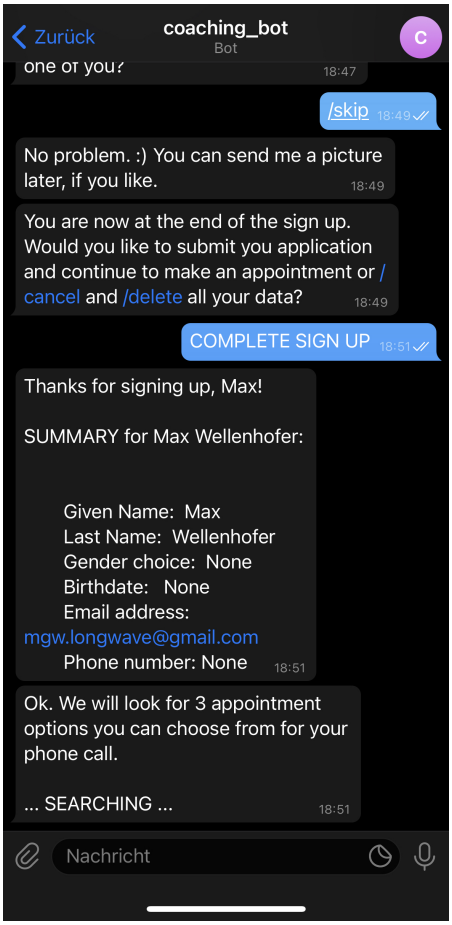

Abbildung 7.5: Bezeichnung der Abbildung

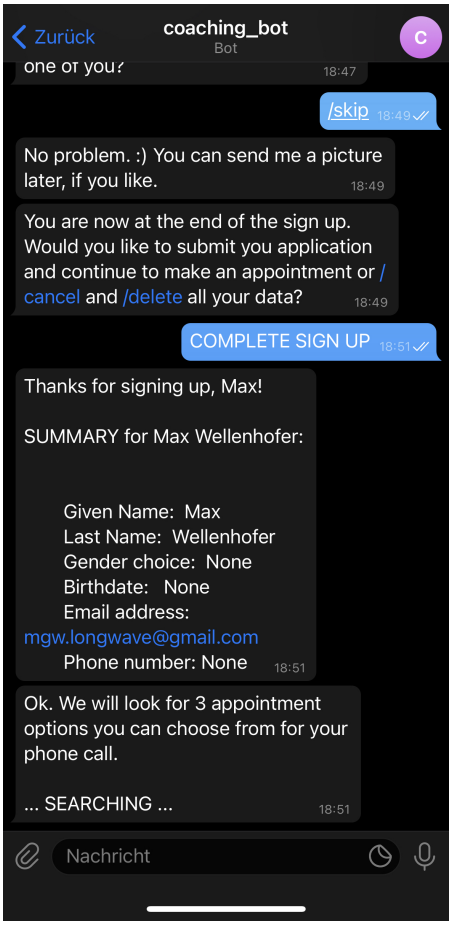

Abbildung 7.6: Bezeichnung der Abbildung

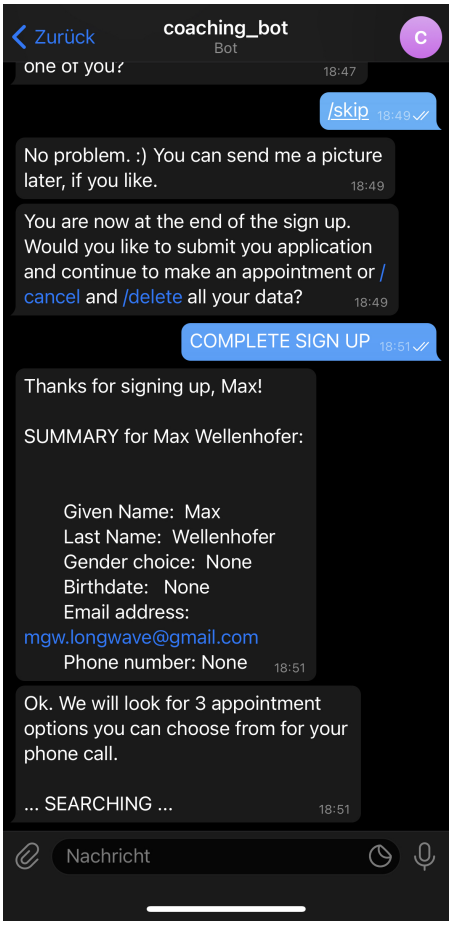

Abbildung 7.7: Bezeichnung der Abbildung

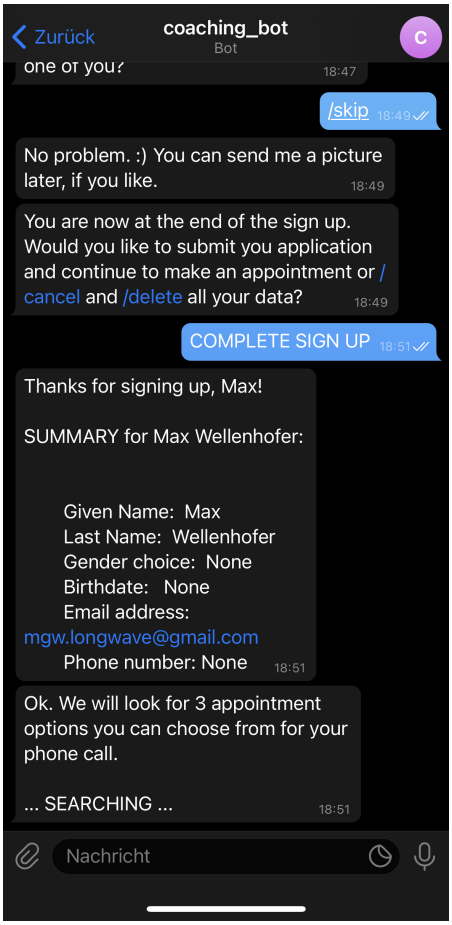

Abbildung 7.8: Bezeichnung der Abbildung

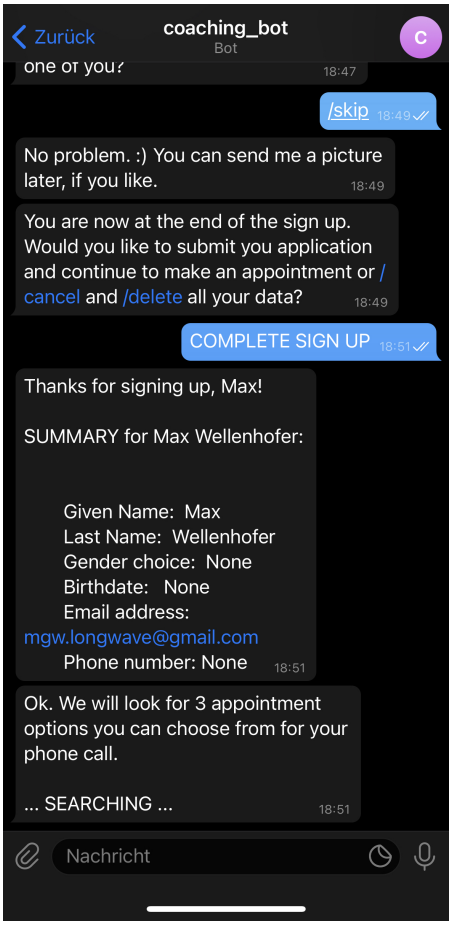

Abbildung 7.9: Bezeichnung der Abbildung

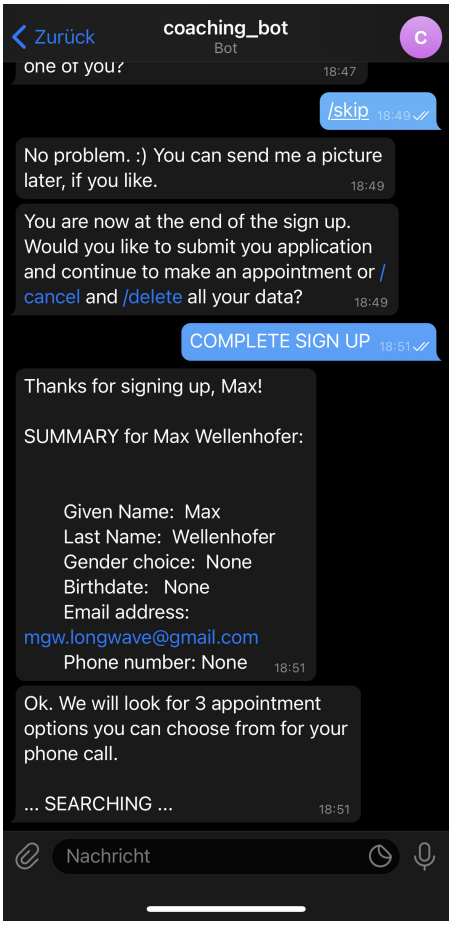

Abbildung 7.10: Bezeichnung der Abbildung

8

Anwendungsszenarien

Der Coaching Bot kann in allerlei Szenarien angewandt werden. Die in den letzten Jahren stark angewachsene Zahl an Personal Coaches kann den Bot mit basalen Programmierfähigkeiten an die eigenen Bedürfnisse anpassen und ist flexibel in der Wahl der unterliegenden Infrastruktur. Vor allem für Nebenerwerbstätige Coaches mit einem kleinen Kundenportfolio, bei dem sich Kaltaquise oft als teuer und wenig erfolgreich herausstellt, bietet der Coaching Bot eine einfache Möglichkeit, Neukunden kostenfrei und einfach onzuboarden. Der Prozess ist unkompliziert, unverbindlich, gut dokumentiert und einfach zu adaptieren.

8.1 Setup

8.1.1 pipenv - Python Package Manager

Die Applikation nutzt den Package Manager pipenv. Dieser bietet die Möglichkeit, ein projektspezifisches Dokument über alle Abhängigkeiten hinweg zu erstellen und im Projekt selbst zu speichern. So können andere Entwickler Abhängigkeiten leicht installieren und müssen dies nicht auf Systemebene tun, wo es ggf. zu Konflikten mit anderen Projekten kommen könnte.
Um alle Abhängigkeiten einzusehen, pipenv [Pyt21c] und das coaching_botPipfile entsprechend der Dokumentation nutzen, um alle automatisch zu installieren.

8.1.2 Konstanten und Schlüssel

Für die Umsysteme des Coaching Bots (Telegram-API, Google-Calendar-API und Mail-Server) sind Zugangsdaten erforderlich. Diese sind im Repository [Wel21] aus Sicherheitsgründen nicht versioniert. Anpassbare Vorlagen sind inkl. Anleitung unter `_constants`[1] zu finden.

[1] `https://github.com/mwel/coaching_bot/tree/main/bot/constants_`

9

Zusammenfassung und Ausblick

Ziel des Projekts war es, dem interessierten Nutzer eine Alternative zum klassischen Webformular anzubieten, über die er sich für einen Coaching-Service anmelden kann. Er soll die Möglichkeit erhalten, einige Informationen über sich zu vermitteln, diese sollen einem Coach in einer einfach Übersicht übermittelt werden können. Schließlich soll es zu einem Treffen zwischen Coach und Coachee kommen. Dazu muss ein Termin vereinbart werden können.
All diese Dinge bietet uns der Coaching Bot und dies unter den Randbedingungen, dass er quelloffen, kostenlos und einfach adaptierbar ist. Insofern wurden die wichtigsten Kriterien an das Projektgewerk erfüllt. Allerdings bleibt der Bot ein Proof-of-Concept und birgt natürlich Weiterentwicklungspotenzial. Coaches mit ähnlichen Bedürfnissen können den Bot selbst nutzen und hosten. Aufgrund der vielen, mächtigen, massentauglichen Systeme, die es auf dem Markt gibt und der niedrigen Priorität, die das Thema Datensicherheit gegenüber internationalen Korporationen für für Viele immer noch hat, wird die Mehrheit aber wahrscheinlich eher zu plug-and-play-Produkten i.e. WhatsApp oder Facebook Messenger von Meta greifen, die sich immer weiter in Richtung Bots entwickeln.

Für jene aber, die einen einfach adaptierbaren Bot für ihr Onboarding suchen und die sich und ihre Kunden gerne gegen die omnipräsente Datensammelfreude der Konzerne schützen möchten, stellt der Coaching Bot eine solide Startplafform dar.

So sind bspw. folgende Ausbaustufen möglich:

1. Verteilung der Systeme und Verlagerung in die Cloud für konstante und hohe Verfügbarkeit. (Containerisierung wäre i.e. eine Option.)
2. Skripten und Automatisierung weiterer Coaching-Stufen. i.e. könnte die erste Session, die oft ähnlich abläuft auch vom Bot abgehandelt werden.
3. Ton-Aufnahmen für das Biography-Modul
4. Verteilung auf weitere Messenger-Dienste, i.e. via 2.1.12 BotMan
5. NLP- oder Sprachkatalog-Integration (So würde die Sprache des Bots etwas variieren und natürlicher wirken.)

6. Ausbau der Web-GUI. Hinsichtlich der darzustellenden Informationen sowie zum Thema Daten-Sicherheit besteht hier Potenzial.

Literaturverzeichnis

BJ22. Bibo-Joshi: *Extensions – Your first Bot.* https://github.com/python-telegram-bot/python-telegram-bot/wiki/Extensions-—Your-first-Bot, 2022.

eaK02. Klyne et. al.: *Date and Time on the Internet: Timestamps.* 2002.

Goo22a. Google LLC: *Google APIs Explorer overview.* https://developers.google.com/explorer-help/, 2022.

Goo22b. Google LLC: *GoogleCalendarAPI.* https://googleapis.github.io/google-api-python-client/docs/dyn/calendar_v3.html, 2022.

IBM20. IBM: *Application Programming Interface (API).* https://www.ibm.com/cloud/learn/api, 2020.

Kre21. Kremming, Katharina: *Telegram Messenger – Alles was Du wissen musst!* https://www.messengerpeople.com/de/messaging-apps-brands-der-telegram-messenger/, 2021.

Meh22. Mehner, Matthias: *Nutzerzahlen Messenger Apps Deutschland und Weltweit.* https://www.messengerpeople.com/de/weltweite-nutzer-statistik-fuer-whatsapp-wechat-und-andere-messenger/, 2022.

Pyt. Python Software Foundation: *sqlite3 — DB-API 2.0 interface for SQLite databases.* https://docs.python.org/3/library/sqlite3.html.

Pyt21a. Python Software Foundation: *datetime — Basic date and time types.* https://docs.python.org/3.8/library/datetime.html, 2021.

Pyt21b. Python Software Foundation: *email.mime: Creating email and MIME objects from scratch.* https://docs.python.org/3.8/library/email.mime.html, 2021.

Pyt21c. Python Software Foundation: *pipenv.* https://pypi.org/project/pipenv/, 2021.

Pyt21d. Python Software Foundation: *Python 3.8.6.* https://docs.python.org/3.8/, 2021.

Pyt21e. Python Software Foundation: *smtplib — SMTP protocol client.* https://docs.python.org/3.8/library/smtplib.html#module-smtplib, 2021.

Ref21. Refsnes Data: *Python Introduction.* https://www.w3schools.com/python/python_intro.asp, 2021.

Tel21a. Telegram Messenger Inc.: *Telegram App*. https://telegram.org/, 2021.

Tel21b. Telegram Messenger Inc.: *Telegram Bot API*. https://core.telegram.org/bots/api, 2021.

The21. The SQLite Consortium: *SQLite*. https://sqlite.org, 2021.

Tol21a. Toledo, Leandro: *python-telegram-bot*. https://github.com/python-telegram-bot/python-telegram-bot/blob/master/examples/conversationbot.py, 2021.

Tol21b. Toledo, Leandro: *Telegram API Documentation*. https://python-telegram-bot.readthedocs.io/en/stable/index.html#, 2021.

Tol21c. Toledo, Leandro: *telegram.ext package*. https://python-telegram-bot.readthedocs.io/en/stable/telegram.ext.html, 2021.

Wel21. Wellenhofer, Max: *The Coaching Bot Repository*. https://github.com/mwel/coaching_bot, 2021.

appendix

A

Glossar

API	Anwendungsprogrammierschnittstellen (APIs) vereinfachen Softwareentwicklung und -innovation, indem sie Anwendungen den einfachen und sicheren Austausch von Daten und Funktionen ermöglichen. [IBM20]
Bot	hier: ChatBot - Computerprogramm, das mit einem Nutzer interagieren und vordefinierte Aufgaben selbstständig erledigen kann.
PoC	Proof of Concept: z. dt. Machbarkeitsstudie
DACH	Deutschland, Österreich, Schweiz bzw. deutschsprachige Region Europas
Vendor Lock-In	hier: Beschränkung der Anwendbarkeit eines Systems auf einen Anbieter - in diesem Fall 'Telegram'
GUI	Graphical User Interface: z.dt. Benutzeroberfläche
CSS	Cascading Style Sheets: Stylesheet-Sprache, die zur optimsch ansehnlicheren Aufbereitung von Benutzeroberflächen genutzt wird
Geo-Fencing	Virtuelle Abgrenzung eines festgelegten Gebiets oder einer Region
CRUD	CREATE, READ, UPDATE, DELETE: Standard-Operationen, die auf einer Datenbank durchgeführt werden
SQL	Search Query Language: Sprache zur Definition von Datenstrukturen in relationalen Datenbanken

B

Selbstständigkeitserklärung

☐ Diese Arbeit habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

☐ Diese Arbeit wurde als Gruppenarbeit angefertigt. Meinen Anteil habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Namen der Mitverfasser:

Meine eigene Leistung ist:

Datum

Unterschrift der Kandidatin/des Kandidaten