

ISSAM CHAREF
SALEH CHAABAN

Prof. Dr. Stephan Kurpjuweit
| Hochschule Worms

Dokumentation & Beschreibung für 4Free



RASA Chatbot



Alexa Skill

Voice Assistant

Abgabe für das Modul
„Digitale Sprachassistenten“

Inhaltsverzeichnis

Projektidee	3
Highlights	3
Voice User Interface Design (VUI-Design).....	5
Einsatz von SSML anhand ausgewählter Beispiele	9
Technische Umsetzung – Codedokumentation	12
Vorwort.....	12
App.py	12
AccountLinking.py	15
Benutzer.py	16
Database.py	17
FindeNahegelegeneStaedte.py	18
AbfrageEigeneAdresseIntent.py.....	20
AbfrageEigeneInserateIntent.py	21
AuskunftIntent.py.....	23
DetaillierteSucheIntent.py	24
InseratErzeugenIntent.py	25
SucheStartenIntent.py.....	26
Weitere Intents	27
RASA Dokumentation	28
Was ist RASA.....	28
Vorteile von RASA.....	28
Installation von RASA.....	29
Chatbot für 4Free	38
Probleme mit RASA	38
Dokumentation zur Genauigkeitserkennung (NLU Vergleich).....	39

Alexa

Projektidee

4Free ist ein Skill für den Alexa-Sprachassistenten, welchen Benutzern ermöglicht ihre nicht mehr benötigten Gegenstände an andere Menschen zu verschenken, statt diese einfach nur auf den Sperrmüll zu stellen oder wegzuschmeißen. Dies funktioniert, indem Benutzer sich über den Account-Linking-Mechanismus von Amazon mit dem Skill verbinden und eine Anzeige mit einer detaillierten Beschreibung ihres Artikels schalten. Im Anschluss hat ein weiterer Benutzer von *4Free* als Interessent die Möglichkeit durch die Suchfunktion diesen Artikel zu finden und sich mit dem Anbieter in Verbindung zu setzen. Ein Anbieter hat dabei die Wahl, ob er eine Abholung vor Ort erlaubt oder nur ein Versand möglich ist. Diese Information wird in der Artikelbeschreibung vermerkt und ist für jeden Interessenten sichtbar.

Der Skill ist im Allgemeinen für jede Zielgruppe gedacht, aber speziell für die Menschen, welche neue Gegenstände auch Mal gerne aus Second Hand beziehen. Insgesamt hat der Skill das Ziel der Umwelt zu helfen und anderen Menschen eine Freude zu bereiten.

Highlights

Folgende Funktionalitäten sind bei *4Free* besonders hervorzuheben:

1. Multi-Modale Umsetzung unter Einsatz der *Alexa Presentation Language (APL)* am Beispiel des *AuskunftIntent*

2. Einsatz verschiedener *Speech Synthesis Markup Language (SSML)* Elemente für ausgewählte Dialoge. Insbesondere anzumerken ist die Verwendung von Audio, Phonemen und eine Veränderung der Stimmlage
3. Verknüpfen des eigenen Amazonkontos und das Teilen sensibler Daten mit dem Skill (*Account-Linking*)
4. Speicherung von temporären Sitzungsattributen bzw. persistenten Attributen unter Einsatz von *MongoDB*
5. Verkettung von Intents (*Intent-Chaining*) am Beispiel des *SucheStartenIntent*. Insbesondere die Übergabe von Slotwerten an einen nächsten Intent
6. Einbeziehung eines vom Benutzer ausgewählten Suchradius für die Artikelsuche unter Verwendung von Standortdiensten
7. Manuelle Delegation von Dialogen an Alexa und manuelles Abfragen von Slots im Backend
8. Realisierung eines Onboarding-Prozesses beim erstmaligen Skillstart

Voice User Interface Design (VUI-Design)

Für das VUI-Design erforderte 4Free insbesondere Dialoge an verschiedenen Stellen des Skills, wie zum Beispiel zur Erstellung von Inseraten oder zur Suche nach Artikeln. Dabei müssen die Benutzer jeweils eine Reihe von Fragen beantworten, welche erst zum Schluss als ganzes evaluiert wird und eine Operation in der Datenbank auslöst. Für vereinzelte Slots wurden in der Alexa Developer Console eigene Slot Typen mit einer definierten Spanne an Werten eingeführt, so dass ein nicht gültiger erkannter Wert durch die definierte Slot Validierung zu einer erneuten Abfrage des Slots führt.

[Intents](#) / [SucheStartenIntent](#) / [detaillierteSuche](#)

Slot Type

Allgemeine_Bestaetigung



Auto delegation is on for this intent (inherited from skill setting).

Dialogs **Validations**

Create new Validation:

Choose a validator



A valid slot type is needed before you can add a validation

Active Validations:

Value within slot types' slot values



Here is some description about value within slot types' slot values

What might Alexa say to prompt for a valid slot value?



<speak> {detaillierteSuche} ist keine gültige Antwort. Antworte bitte nur mit: Ja oder: Nein. Möchtest du eine detaillierte Suche starten?</speak>



Slot Types / [Allgemeine_Bestaetigung](#)

Custom slot types with values define a representative list of possible values, IDs and synonyms.

Slot Values (2)

[Bulk Edit](#) [Export](#)

Search



Enter a new value for this slot type



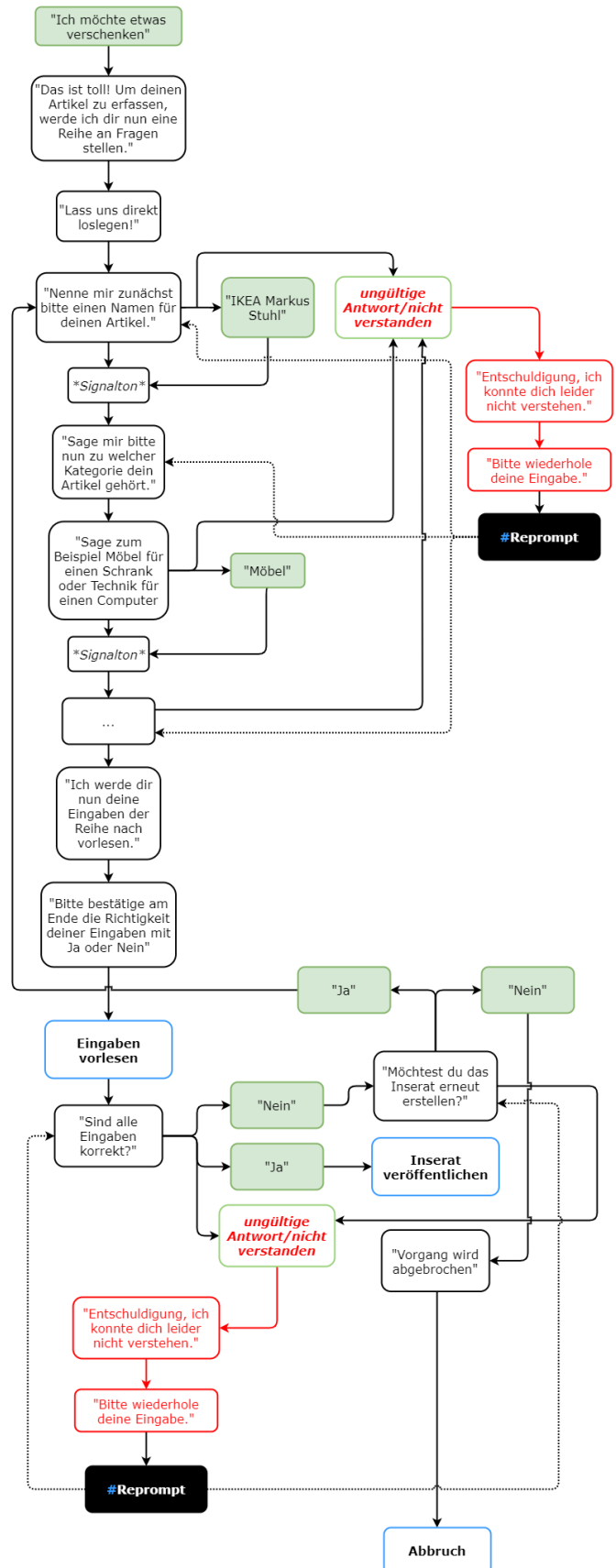
VALUE	ID (OPTIONAL)	SYNONYMS (OPTIONAL)			
Ja	Enter ID	Add synonym	+	Yes x	
Nein	Enter ID	Add synonym	+	No x	

Slotvalidierung am Beispiel des SucheStartenIntents für den Slot detaillierteSuche mit dem Slottypen Allgemeine_Bestaetigung

Abhängig vom jeweiligen Slot gibt es unterschiedliche Kategorien von Antwortmöglichkeiten. So existieren zum Beispiel für einen Slot *zustand* mit zum Beispiel „*sehr schlecht*“ bis „*sehr gut*“ eine Bewertungsspanne, also Kategorien, wohingegen für einen Slot *anmerkung* eine offene Antwort formuliert werden kann. Beschränkte Antworten mit „*ja*“ oder „*nein*“ als Antwort finden sich häufig zur Bestätigung eines Intents wieder.

Sobald ein Benutzer keine oder eine falsche Eingabe tätigt, wird er darauf aufmerksam gemacht und erhält ein oder mehrere konkrete Beispiel-Utterances, welche für den jeweiligen Slot gültig sind, damit er den Dialog fortsetzen kann.

Für die einzelnen Fragen des Fragenkatalogs bei der Suche und Anzeigenerstellung wurde explizit auf eine jeweilige Bestätigung der Benutzereingabe verzichtet, da dies ansonsten den Dialogfluss negativ beeinflusst. Hingegen wird zum Schluss der Inseraterstellung eine Zusammenfassung der bisherigen Angaben erstellt und eine explizite



Bestätigung gefordert, um versehentliche oder inkorrekte Anzeigen nach Möglichkeit zu vermeiden.

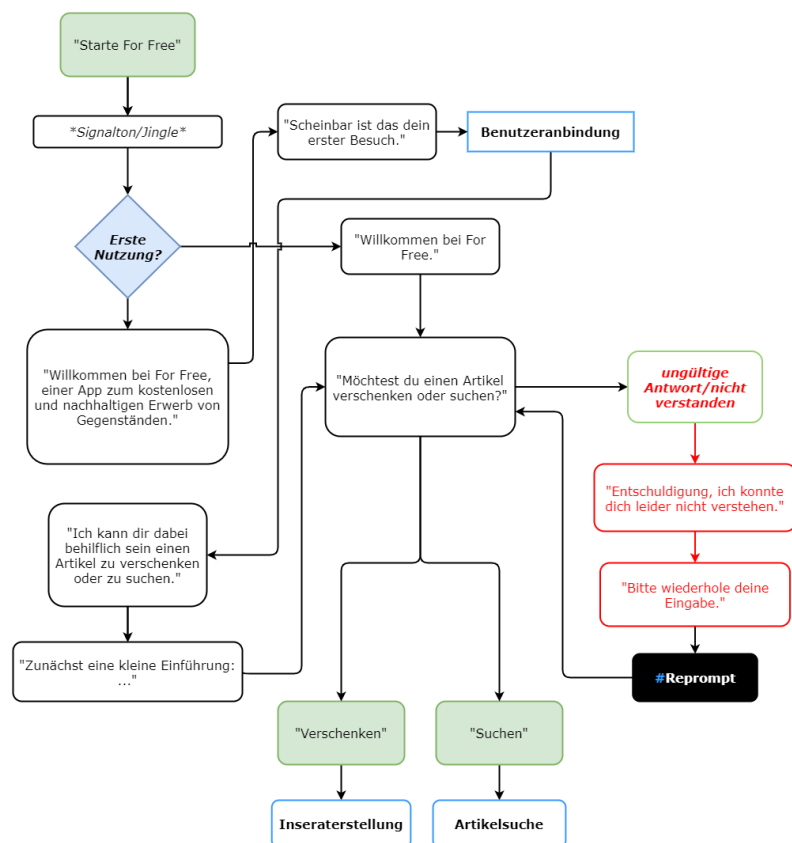
Damit der Benutzer beim Beantworten der Fragen abschätzen kann an welchem Punkt er sich in einem Dialog ungefähr befindet, wurden *Conversational Markers* eingesetzt. Diese finden immer dann Anwendung, wenn ein Benutzer bei der Suche oder bei der Inseraterstellung eine Frage erfolgreich beantwortet. Daraufhin ertönt ein Erfolgston, welcher dem Benutzer signalisiert, dass es für ihn zur nächsten Frage übergeht. Zusätzlich wird an manchen Stellen des Dialogs ausgegeben wie viele Fragen noch verbleiben, insbesondere immer dann, wenn ein Benutzer bei der letzten Frage angekommen ist. Sobald ein Benutzer seine Anzeige final bestätigt und diese erfolgreich erzeugt wurde, bekommt dieser einen weiteren Erfolgston zu hören, welcher sich jedoch von den vorherigen unterscheidet, um zu signalisieren, dass die Anzeige erfolgreich erstellt wurde und der Dialog somit abgeschlossen ist. Für den Fall, dass bei der Speicherung der Anzeige ein Fehler auftritt, ertönt ein unterschiedlicher Signalton, welcher einen Fehlschlag andeutet. Sowohl Erfolgston als auch Fehlerton werden zusätzlich um sprachliche Ausgaben wie zum Beispiel „*Inserat erfolgreich erstellt*“ bzw. „*Dein Inserat konnte aufgrund eines Fehlers nicht erzeugt werden*“ ergänzt. Auch die Aufzählungen der Funktionalitäten im *AuskunftIntent* mit „*Erstens*“, „*Zweitens*“ usw. sind Teil der *Conversational Markers*.

Bei den Intents zur Suche von Artikeln kommt nach Ertönen des letzten Erfolgstons zusätzlich die Ausgabe „*Artikel wird gesucht*“, um ein gewisses Erwartungs-Management gegenüber dem Benutzer zu pflegen, damit er abschätzen kann was als nächstes für eine Aktion folgt.

Anschließend werden die für den Benutzer passenden Suchergebnisse anhand seiner Suchparameter ausgegeben. Mit einer Hand voll definierter Follow-Up-Fragen bzw. Anweisungen kann der Benutzer im direkten Anschluss bestimmte Aktionen tätigen, wie zum Beispiel zwischen den einzelnen Suchergebnissen zu navigieren, etwa mit den Sprachbefehlen „*vorherige*“ und „*nächste*“. Auch andere Aktionen wie das Favorisieren einer Anzeige oder das Springen an den Anfang

bzw. an das Ende der Suchergebnisse sind möglich. Zusätzlich bekommt der Benutzer zu Beginn nur ein paar wenige (weniger als sieben) und nur die wichtigsten Informationen über den derzeitigen Artikel, da diese an erster Stelle genannt werden sollten und der Benutzer nicht mit zu vielen Daten auf einmal konfrontiert werden sollte. Weitere bzw. weniger wichtige Details lassen sich dann zum Beispiel über „*weitere Details*“ abfragen. Über die beispielhafte Anweisung „*kehre zurück*“ hat der Benutzer auch die Möglichkeit zur Unterbrechung der Suche und wird aus dem Intent für die Suchergebnisse hinausgeleitet.

Wird 4Free das erste Mal ausgeführt, wird mit dem neuen Benutzer zunächst ein kurzes Onboarding durchgeführt, bei dem er über den Zweck des Skills erfährt und wie er an weitere Informationen bekommt. Demnach gibt es eine Unterscheidung zwischen einem erstmaligen Benutzer und einem Benutzer, welcher den Skill bereits öfter verwendet hat und diesbezüglich keine einführenden Informationen mehr erhält, sondern nur regulär vom Skill begrüßt wird.



Insgesamt wurden die Antworten des Alexa-Skills so formuliert, dass eine explizite Frage erst zum Ende einer jeweiligen Ausgabe platziert wurde, damit die Benutzer bis zum Ende der Ansage aufmerksam zuhören, bevor sie eine Antwort geben und keine wichtigen Informationen verpassen. Vor der jeweiligen Frage ist meist ein

Beispiel für die Antwortmöglichkeiten des jeweiligen Slots aufgeführt, um den Benutzer über seine Möglichkeiten aufzuklären.

Die Alexa-Ausgaben variieren je nachdem, ob ein Benutzer bereits sein Konto verknüpft hat. So wird ein Benutzer mit verknüpftem Konto beim Starten des Skills beispielsweise mit seinem Namen begrüßt, wohingegen jemand ohne Kontoverknüpfung nur regulär ohne Namen begrüßt wird. Zudem sind einige Funktionalitäten wie das Erstellen von Anzeigen nur dann möglich, wenn eben ein solches Benutzerkonto mit dem Skill verknüpft ist. So werden Benutzer ohne Benutzerkonto beim Versuch eine neue Anzeige zu schalten mit einer alternativen Antwort abgewiesen und erhalten stattdessen einen Hinweis zum Kontoverknüpfungsprozess. Weiterhin sind viele der Sprachausgaben so

```
"ANZAHL_VERBLEIBENDE_FRAGEN": [  
  "Nur noch {} Fragen!",  
  "Es sind noch {} Fragen übrig!",  
  "Es verbleiben noch {} Fragen!",  
  "Nur noch {} Fragen verbleibend!",  
  "Du hast es fast geschafft!",  
  "Fast geschafft!"  
],
```

gestaltet, dass es für ein und dieselbe Aussage mehrere definierte Alternativen gibt, welche durch den Zufalls-Algorithmus der Pythonklasse *random* ausgewählt werden.

de-DE.json: Definierter JSON-Eintrag mit mehreren alternativen Ausgabemöglichkeiten

Benutzer, welche bereits Erfahrung mit dem Skill gesammelt haben könnten eventuell in Erfahrung gebracht haben, dass sie mit bestimmten Utterances eine einfache bzw. detaillierte Suche direkt betätigen können, ohne vorher explizit gefragt worden zu sein, ob sie eine einfache oder detaillierte Suche wünschen.

Einsatz von SSML anhand ausgewählter Beispiele

Ein Teil des VUI-Designs stellt auch der Einsatz von SSML dar. Die Beschreibungssprache kann dazu verwendet werden, um die Sprachausgaben von Alexa auf unterschiedliche Art und Weise zu variieren. Das ermöglicht Abwechslung und eine spannende und gleichzeitig einzigartige Konversation mit

dem Benutzer. Zu beachten ist jedoch, dass der vermehrte Einsatz von SSML oder der Einsatz an unpassenden Stellen wiederum auch für Verwirrung bei einem Benutzer sorgen kann und die Benutzererfahrung ggf. verschlechtert. Aus dem Grund ist es wichtig SSML nur entschieden und an den passenden Stellen in einem Dialog einzusetzen.

Das erste Beispiel zeigt den Einsatz von Phonemen im *EntwicklerInfoIntent*. Phoneme sind Laute einer Sprache und können dazu verwendet werden die Aussprache bestimmter Buchstaben anzupassen. Dies ist zum Beispiel immer dann nützlich, wenn ein Wort oder ein Name ursprünglich aus einer anderen Sprache mit anderen Betonungen und Lauten stammt und auch so ausgesprochen werden soll, ohne dass das gesamte Sprachpaket geändert werden muss. Der Name der Entwickler von *4Free* stammt beispielsweise aus dem arabischen Raum und wird daher unterschiedlich betont und ausgesprochen als in der deutschen Sprache.

```
For Free wurde von <phoneme alphabet='x-sampa' ph='/\ "i_?\\ssa:m SA_?\\:re:f/'>{</phoneme>
```

de-De.json: Einsatz von Phonemen in SSML

Zusätzlich lassen sich über SSML auch Audiodateien einbinden, so dass innerhalb einer Konversation auch bestimmte Sounds und Lieder ausgegeben werden können. Der Einsatz geschieht über den *audio-Tag* mit Angabe einer Audio-Quelldatei. Es gibt u.a. bestimmte Voraussetzungen, welche die Audiodatei erfüllen muss. Diese sind in der SSML-Dokumentation von Amazon zu finden. In diesem Beispiel wurde eine Audiodatei aus der kostenfreien Amazon Soundbibliothek verwendet, welche für den Einsatz mit SSML in Alexa optimiert sind. Das folgende Audio wird innerhalb der Intents zur Inseraterzeugung und Artikelsuche verwendet und immer dann abgespielt, wenn ein Benutzer eine Frage aus dem Fragenkatalog mit einer gültigen Spracheingabe beantwortet.

```
<audio src='soundbank://soundlibrary/office/amzn_sfx_elevator_open_bell_01' />
```

de-De.json: Einbinden von Audiodateien in SSML

Um bestimmte Wörter, Satzteile oder Sätze hervorzuheben, kann mit Hilfe des *prosody-Tags* die Lautstärke, Geschwindigkeit und Stimmlage der Sprachausgabe in SSML angepasst werden. In diesem Fall wurde für den *InseratErzeugenIntent* lediglich die Lautstärke bestimmter wichtiger Satzteile erhöht, um diese für den Benutzer hervorzuheben.

```
Dein Artikel ist aus {} und hat die Maße: <prosody volume='x-loud'>{} mal {} mal {}</prosody>.
```

de-De.json: Erhöhung der Lautstärke für bestimmte Satzteile in SSML

Technische Umsetzung – Codedokumentation

Vorwort

Im Folgenden werden die wesentlichen Highlights und wiederkehrende Strukturen des Alexa-Skills anhand von Codebeispielen gezeigt. Dabei werden lediglich einzelne Codeausschnitte aus ausgewählten relevanten Komponenten behandelt und näher erläutert. Zu Beginn jedes Abschnittes folgt eine kurze Beschreibung über die Allgemeine Funktionalität, Verantwortlichkeit und Bedeutung der Komponente im Zusammenhang.

App.py

Die Komponente *app.py* dient im Allgemeinen dazu alle Komponenten miteinander zu verbinden und den Flask Server zu konfigurieren und zu starten. Somit ist sie der Dreh- und Angelpunkt des Alexa-Skills und bindet im Wesentlichen alle Intent-Handler über die Python-Decorator-Notation in den Code ein.

```
@sb.request_handler(can_handle_func=is_intent_name('EntwicklerInfoIntent'))
def entwickler_info_handler_wrapper(handler_input) -> Response:
    return entwickler_info_handler(handler_input)
```

app.py: Python-Decorator-Notation als Alternative zur Klassendefinition

Darüber hinaus implementiert die Komponente die üblichen und notwendigen Standard-Intents und Request-Handler, zum Beispiel für den *AMAZON.HelpIntent* oder den *SessionEndedRequest*.

Auch das Benutzer-Onboarding ist innerhalb der Komponente in der Funktion *launch_request_handler* vorzufinden.

```

geraete_id = handler_input.request_envelope.context.system.device.device_id
if db.skill_erststart(geraete_id):
    # Onboarding einleiten
    logging.info('Onboarding wird gestartet...')

    sprach Ausgabe += sprach_prompts['BENUTZER_ONBOARDING_BEGRUESSUNG']

    if db.registriere_benutzer_geraet(geraete_id):
        logging.info('Gerät wurde erfolgreich registriert')

    else:
        logging.warning('Fehler bei der Registrierung des Geräts')

    # Beim Erststart dem Benutzer erklären, wie er das Account-Linking durchführt
    if not benutzer_linked:
        sprach Ausgabe += ' ' + sprach_prompts['BENUTZER_ONBOARDING_LINKING_ANWEISUNGEN']

        sprach Ausgabe += ' ' + sprach_prompts['BENUTZER_ONBOARDING_TIPPS']
    # Falls Gerät bereits registriert
    else:
        logging.info(f'{db.skill_erststart(geraete_id)=}')

    if benutzer_linked:
        benutzer_name = Benutzer.AmazonBenutzer.get_benutzer_namen()
        sprach Ausgabe = random.choice(sprach_prompts['BENUTZER_LINKED_BEGRUESSUNG']).format(benutzer_name)
    else:
        sprach Ausgabe = random.choice(sprach_prompts['BENUTZER_BEGRUESSUNG'])

```

app.py: Onboarding im launch_request_handler

Dabei wird zunächst mit Hilfe der Methode *skill_erststart* aus der Klasse *Database* bei jedem Skillstart überprüft, ob der Benutzer den Skill das erste Mal ausführt oder bereits zuvor gestartet hat. Dafür wird die Geräte-ID des verwendeten Alexa-Geräts der Methode als Parameter übergeben, und überprüft, ob diese bereits zuvor in der Datenbank registriert wurde. Ist dies nicht der Fall, wird das Onboarding gestartet und die Geräte-ID wird registriert, sodass ein nächstmöglicher Aufruf kein Onboarding mehr auslöst. Anzumerken ist jedoch, dass die Geräte-ID keine wirkliche Geräte-ID ist, sondern sich immer dann ändert, wenn ein Benutzer sein Konto mit dem jeweiligen Skill verknüpft. Ein Deaktivieren und erneutes Aktivieren des Skills durch den Benutzer löst bei einer wiederholten Kontoverknüpfung wiederum das Onboarding aus. In der Praxis ist es jedoch sehr unwahrscheinlich, dass ein Benutzer den obigen Fall erlebt, da er sich den Skill in der Regel nur einmal herunterlädt und sein Konto einmalig verknüpft.

Für den Fall, dass ein Benutzer sein Konto nicht verknüpft hat, wird zusätzlich ein Session-Attribut für den Suchradius gesetzt, da dieser ohne eine Kontoverknüpfung nicht persistent gespeichert werden kann. Der Suchradius beträgt dabei standardmäßig fünfzehn Kilometer und kann jederzeit sowohl von Benutzern mit als auch ohne Kontoverknüpfung über den *RadiusEinstellenIntent* angepasst werden.

```
# Suchradius in Session-Attribut speichern <=> Kein Benutzerkonto
if not benutzer_linked:
    # Standard-Suchradius von 15km
    session_attr['suchradius'] = 15
    logging.info(f'{session_attr=}')

```

app.py: Speichern des Suchradius für Benutzer ohne Kontoverknüpfung im launch_request_handler

Noch bevor das eigentliche Onboarding durchgeführt wird, werden Land und die Postleitzahl des Benutzers bei eingeschalteten Ortungsdiensten in Sitzungsattributen zwischengespeichert, um diese hinterher für die Suche verwenden zu können.

```
session_attr['plz'] = get_geraete_standortdaten(handler_input)[1]
session_attr['land'] = 'DE' \
    if get_geraete_standortdaten(handler_input)[3] == 'Deutschland' \
    else get_geraete_standortdaten(handler_input)[3]
logging.info(f'{session_attr["plz"]=}, {session_attr["land"]=}')

```

app.py: Speichern von Land und Postleitzahl des Benutzers in Sitzungsattributen im launch_request_handler

Die Strings für sämtliche Sprachausgaben befinden sich in der Datei *de-DE.json* und werden über einen sogenannten *Global Interceptor* an jeden Handler geliefert. Eine Global Interceptor schaltet sich **vor jeden Handler** und wird häufig dazu verwendet notwendige Setups für diese einzurichten.

```

@sb.global_request_interceptor()
def localization_request_interceptor(handler_input):
    logger.info('Localization Interceptor wird ausgeführt...')

    try:
        with open('languages/de-DE.json') as sprach_prompt_daten:
            sprach_prompts = json.load(sprach_prompt_daten)
            handler_input.attributes_manager.request_attributes['_'] = sprach_prompts
    except FileNotFoundError as e:
        logger.exception(f'Alexa-Sprachbefehle konnten nicht geladen werden: {e}')
        exit()

```

app.py: Global Interceptor zur Lieferung der Sprachdaten an die Handler

AccountLinking.py

Die Komponente *AccountLinking.py* wird von der Komponente *app.py* dazu verwendet, um zu Überprüfen, ob ein Benutzer sein Konto mit dem Skill verknüpft hat. Das geschieht über einen *Account Linking Access Token* seitens der Amazon-Plattform. Sofern er sein Konto verknüpft hat, wird ein temporäres *AmazonBenutzer-Objekt* erzeugt, welches die wichtigsten Daten über den Benutzer, wie zum Beispiels seine *uid* für alle Komponenten einfach zugänglich macht. Andernfalls wird dem Benutzer eine *Account-Linking-Card* präsentiert mit dem Hinweis zur Verknüpfung des Kontos.

```

# Session-Token abrufen
benutzer_linking_token = get_account_linking_access_token(handler_input)
benutzer_linked = False
if benutzer_linking_token is None:
    # Nach Account-Linking Neustart erforderlich
    logging.info('Account-Linking wird gestartet...')

    response_builder.set_card(LinkAccountCard())
else:
    logging.info('Benutzer bereits verlinkt')

    benutzer_linked = True
    # Statisches Benutzerobjekt erzeugen
    benutzer = AmazonBenutzer(benutzer_linking_token)

```

AccountLinking.py: Überprüfen einer bestehenden Kontoverknüpfung

Benutzer.py

Die Komponente *Benutzer.py* dient wie bereits beschrieben dazu die Daten eines Benutzers mit verknüpftem Konto anderen Komponenten einfacher zugänglich zu machen. Im Grunde genommen handelt es sich dabei um eine Klasse *AmazonBenutzer*, welche lediglich aus statischen Methoden besteht, da pro Gerät immer nur ein Benutzer gleichzeitig aktiv ist und deshalb keine Unterscheidung zwischen mehreren Benutzern notwendig ist. Bei der Initialisierung des Benutzer-Objektes werden alle wichtigen persönlichen Daten des Benutzers in Klassenattributen gespeichert. Diese durch den als Parameter übergebenen Benutzertoken über eine API-Anfrage zugänglich. Zusätzlich wird, falls nicht bereits vorhanden jeweils ein Eintrag für die Statistiken und Einstellungen des Benutzers in der Datenbank erzeugt, um sich zum Beispiel Werte wie den Suchradius persistent zu merken.

```
class AmazonBenutzer:
    uid = None
    name = None
    email = None
    plz = None

    def __init__(self, benutzer_token):
        url = f'https://api.amazon.com/user/profile?access_token={benutzer_token}'
        benutzer_daten = requests.get(url).json()
        logging.info(f'{benutzer_daten=}')

        # Instanzvariablen setzen
        AmazonBenutzer.uid = benutzer_daten['user_id']
        AmazonBenutzer.name = benutzer_daten['name'].split()[0]
        AmazonBenutzer.email = benutzer_daten['email']
        AmazonBenutzer.plz = int(benutzer_daten['postal_code'])
        # Prüfen, ob Datum bereits eingetragen wurde
        if AmazonBenutzer.benutzer_get_statistik('datum') is None:
            AmazonBenutzer.anmeldedatum_speichern()
        # Benutzereinstellungen erzeugen, falls nicht vorhanden
        if not Database.MongoDB.benutzer_hat_einstellungen_eintrag():
            Database.MongoDB.benutzer_einstellungen_erzeugen()
```

Benutzer.py: Initialisierung des Benutzer-Objektes

Database.py

Die Komponente *Database.py* ist die Schnittstelle zur MongoDB Datenbank. Das Herstellen einer Verbindung zur Datenbank funktioniert standardmäßig wie in der Vorlesung gezeigt und wird daher nicht näher erläutert. An dieser Stelle werden nur die wichtigsten und interessantesten Methoden der Klasse veranschaulicht.

Über die Methode *get_eigene_inserate* werden die vom Benutzer erzeugten Anzeigen aus der Datenbank geladen. Dafür ist einerseits die Benutzer-UID notwendig, um den passenden Eintrag zu filtern. Zusätzlich gibt ein Benutzer beim Starten des *InseratErzeugenIntent* eine Sortierung vor, welche ebenfalls als Parameter übergeben wird und dafür sorgt, dass die Ergebnisse in absteigender bzw. aufsteigender Reihenfolge nach ihrem Erstellungsdatum geordnet werden. Sofern kein Ergebnis vorliegt oder ein Fehler auftritt, wird der Wert *None* zurückgegeben.

```
@staticmethod
def get_eigene_inserate(sortierung: str) -> list | None:
    try:
        benutzer_uid = Benutzer.AmazonBenutzer.get_benutzer_uid()
        eigene_inserate = MongoDB.get_db_instance()['benutzer_inserate'].find(
            {'meta_beschreibung.anbieter_uid': benutzer_uid},
            {'meta_beschreibung.anbieter_uid': 0}
        )

        eigene_inserate_sortiert = None
        match sortierung:
            case 'neuste zuerst' | 'neueste zuerst' | 'älteste zuletzt':
                eigene_inserate_sortiert = list(eigene_inserate.sort(
                    'meta_beschreibung.erstellungs_datum',
                    pymongo.DESCENDING
                ))
            case 'neuste zuletzt' | 'neueste zuletzt' | 'älteste zuerst':
                eigene_inserate_sortiert = list(eigene_inserate.sort(
                    'meta_beschreibung.erstellungs_datum',
                    pymongo.ASCENDING
                ))
        except (pymongo.errors.NetworkTimeout, pymongo.errors.InvalidDocument, pymongo.errors.ConnectionFailure) as e:
            logging.exception(f'{{__name__}}: Dokumente konnten nicht gelesen werden: {e}')
        return None
```

Database.py: Abrufen eigener Inserate mit Sortierung

Eine weitere wichtige Methode ist *finde_passende_artikel*. Dieses lädt die zu den Suchparametern passenden Ergebnisse für den gesuchten Artikel aus der Datenbank und bezieht dabei den festgelegten Suchradius mit ein. Die Artikelbezeichnung aus den Suchparametern wird vor der Suche in einen Regex-

Ausdruck überführt, um auch Artikel zu finden, welche den Namen lediglich enthalten. An der Stelle sei gesagt, dass in der Realität gänzlich andere Mechanismen für die Artikelsuche zum Einsatz kommen, wie beispielsweise die Vergabe von Tags und/oder eine Namensdatenbank mit bekannten Artikeln. Aus Komplexitätsgründen wurde eine einfache Art der Implementierung bevorzugt. Anschließend werden die Suchparameter samt Meta- und Artikeldaten zusammengefügt. Das resultierende Dictionary wird dann in einem letzten Schritt über die Methode *flatten* aus dem externen Paket *flatten_dict* um eine Ebene abgeflacht, da sich ein Zugriff auf die Datenbank mit zu tief verschachtelten Parametern als problematisch erwiesen hat.

```
benutzer = Benutzer.AmazonBenutzer
benutzer_land = session_attr['land']
benutzer_plz = benutzer.get_benutzer_plz() if benutzer.benutzer_existiert() else session_attr['plz']
radius = MongoDB.benutzer_get_umkreis() if benutzer.benutzer_existiert() else session_attr['suchradius']
staedte_im_umkreis = get_staedte_im_umkreis(land=benutzer_land, plz=benutzer_plz, radius=radius)
plz_im_umkreis = [plz[1] for plz in staedte_im_umkreis]
# Bezeichnungen, welche das Wort enthalten zulassen
slot_wertepaare['bezeichnung'] = {'$regex': f'.*{slot_wertepaare["bezeichnung"]}.*'}
# Suchanfrage muss wegen Verschachtelung in Punktnotation konvertiert werden
such_params = flatten(
    {'meta_beschreibung': {'anbieter_plz': {'$in': plz_im_umkreis}}, 'artikeldata': slot_wertepaare},
    reducer='dot',
    max_flatten_depth=2
)
```

Database.py: Datenbankmethode finde_passende_artikel zur Suche von passenden Artikeln unter Berücksichtigung des Benutzerstandortes

FindeNahegelegeneStaedte.py

Die Komponente *FindeNahegelegeneStaedte.py* wird dazu verwendet unter Berücksichtigung des Benutzerstandortes und dem vom Benutzer festgelegten Suchradius alle Städte im Umkreis mit jeweils ihrem Namen und Postleitzahl in einer Liste aus Tupeln zu speichern. Diese Liste wird hinterher für die Suche von Artikeln verwendet, wie bereits in der vorherigen Komponente beschrieben.

```

else:
    geolocation_daten = response.json()
    logging.info(json.dumps(geolocation_daten, indent=4, ensure_ascii=False))

    staedte_namen = []
    staedte_plz = []
    for geo_entry in geolocation_daten['postalCodes']:
        staedte_namen.append(geo_entry['placeName'])
        staedte_plz.append(int(geo_entry['postalCode']))
    # Format: [(stadt, plz), ...]
    staedte_innerhalb_radius = list(zip(staedte_namen, staedte_plz))
    logging.info(f'{staedte_innerhalb_radius=}')

return staedte_innerhalb_radius

```

FindeNahegelegeneStaedte.py: Erzeugung einer Liste von Tupeln, bestehend aus dem Städtenamen und der zugehörigen Postleitzahl innerhalb der Funktion `get_staedte_im_umkreis`

Die jeweiligen Datensätze zur Ermittlung der nahegelegenen Städte stammen dabei aus der *Geonames.org-API* und sind zumindest mit gewissen Limitierungen kostenfrei nutzbar. Voraussetzung ist zunächst ein Benutzerkonto auf der Geonames.org Webseite, welches kostenlos erstellbar ist. Einem Benutzerkonto steht zu Beginn eine festgelegte Anzahl an Credits für die API-Anfragen zur Verfügung. Je nach Typ der Anfrage variiert die Höhe der einzulösenden Credits. Beispielsweise werden zur Ermittlung der Städte samt zugehöriger Postleitzahlen pro getätigte Anfrage jeweils zwei Credits benötigt. Zusätzlich ist der Suchradius für die kostenfrei Variante auf dreißig Kilometer beschränkt, so dass keine weitreichenden Suchen möglich sind. In der Praxis wäre dies nicht ausreichend und die Projektverantwortlichen oder Entwickler sollten sich mit dem Betreiber der API in Verbindung setzen oder auf die bezahlte Variante zurückgreifen.

```

url = 'http://api.geonames.org/findNearbyPostalCodesJSON'
parameter = {
    'country': land,
    'username': 'freesfor'
}
..

```

FindeNahegelegeneStaedte.py: Definition der URL und Parameter für die API-Anfrage

AbfrageEigeneAdresseIntent.py

Die Komponente für den Intent *AbfrageEigeneAdresse* ermöglicht es einem Benutzer seine derzeitigen in der Amazon-Alexa-App hinterlegten Adressdaten für das verwendete Gerät abzufragen. Ein Benutzer soll dadurch Gewissheit bekommen, ob bei der Erstellung von Anzeigen die korrekten Adressdaten einbezogen werden. Bei fehlenden oder unvollständigen Adressdaten wird der Benutzer auf das Fehlen oder die Unvollständigkeit seiner Adressdaten hingewiesen. Für fehlende Standortberechtigungen wird zusätzlich nach Berechtigungen für die Aktivierung der Standortdienste in Form einer Card (*AskForPermissionsConsentCard*) gefragt.

```
if not (request_envelope_permissions.scopes["alexa::devices:all:geolocation:read"].status.name == 'GRANTED'
        and request_envelope_permissions.consent_token):
    logging.info('Benutzer fehlt Berechtigungen für Standortdienste')

    response_builder.speak(sprach_prompts['ANFRAGE_STANDORT_BERECHTIGUNGEN'])

    standort_rechte = ['read::alexa:device:all:address']
    response_builder.set_card(AskForPermissionsConsentCard(permissions=standort_rechte))
# Standortdaten können gelesen werden
else:
    try:
        notwendige_standortdaten = get_geraete_standortdaten(handler_input)
        if None in notwendige_standortdaten:
            response_builder.speak(sprach_prompts['BENUTZER_ADRESSE_NICHT_GEFUNDEN_ODER_UNVOLLSTAENDIG_FEHLER'])
        else:
            response_builder.speak(str(sprach_prompts['ADRESSE_AUSGABE']).format(*notwendige_standortdaten))
```

AbfrageEigeneAdresseIntent.py: Prüfen der Standortberechtigungen und Ausgabe der Adressdaten

Die Adresse wird dabei über die Methode *get_full_address* mit Angabe der Geräte-ID als Parameter bezogen.

```
def get_geraete_standortdaten(handler_input) -> list:
    """Gerätebasierte Standortdaten abrufen"""
    geraete_id = handler_input.request_envelope.context.system.device.device_id
    geraete_id_client = handler_input.service_client_factory.get_device_address_service()
    standort = geraete_id_client.get_full_address(geraete_id)
    logging.info(f'{standort=}')

    benutzer_land = 'Deutschland' if standort.country_code == 'DE' else standort.country_code
```

AbfrageEigeneAdresseIntent.py: Beziehen der Adressdaten über die Geräte-ID und Service-Client-Factory

AbfrageEigenenInserateIntent.py

Der *AbfrageEigenenInserateIntent* und seine Komponente sorgen dafür, dass ein Benutzer eine Übersicht über seine eigenen inserierten Artikel erhält und diese bei Bedarf auch löschen kann. Wenn ein Benutzer keine eigenen Inserate hat, wird dieser wieder zurückgeleitet und bekommt eine entsprechende Meldung.

Für den aktuellen Artikel existiert ein Ergebniszeiger, welcher in einem Sitzungsattribut gespeichert ist, damit dieser Wert auch nach einer Benutzerinteraktion erhalten bleibt. Dieser Zeiger wird erhöht bzw. reduziert abhängig davon in welche Richtung der Benutzer navigiert.

Um zu verhindern, dass nicht bei jeder Benutzerinteraktion eine erneute Ausgabe desselben Artikels stattfindet, wird zur Regulation der Sprachausgabe ein Flag *artikel_ausgeben* eingeführt. Immer wenn der Flag gesetzt ist, wird eine jeweilige Zusammenfassung des Artikels erzeugt und zur Sprachausgabe hinzugefügt. Manche Werte werden vorab so variiert, dass ein Satzbaustein entsteht, welcher sinnvoll in einen zusammenhängenden Satz integriert werden kann.

```
# Suchergebnis zusammenbauen, falls Flag gesetzt
if artikel_ausgeben:
    aktueller_artikel = session_attr['eigene_inserate'][session_attr['ergebnis_zeiger']]
    # Bestimmte Satzbausteine anpassen, so dass es natürlicher klingt
    aktueller_artikel_stueckzahl_satzbaustein = 'ein' if aktueller_artikel['artikeldaten']['stueckzahl'] == 1 \
        else aktueller_artikel['artikeldaten']['stueckzahl']
    aktueller_artikel_zustand_satzbaustein = aktueller_artikel['artikeldaten']['zustand'] + 'en' \
        if aktueller_artikel['artikeldaten']['zustand'] not in ['ohne schäden', 'klasse', 'ohne mängel'] \
        else aktueller_artikel['artikeldaten']['zustand']

    sprach_ausgabe += sprach_prompts['EIGENE_INSERTE_ZUSAMMENFASSUNG'].format(
        session_attr['ergebnis_zeiger'] + 1,
        aktueller_artikel['meta_beschreibung']['erstellungs_datum'].split(' ')[0],
        aktueller_artikel_stueckzahl_satzbaustein,
        aktueller_artikel['artikeldaten']['bezeichnung'],
        aktueller_artikel['artikeldaten']['farbe'],
        aktueller_artikel['artikeldaten']['hersteller'],
        aktueller_artikel_zustand_satzbaustein
    )
```

AbfrageEigenenInserateIntent.py: Zusammenbau der Sprachausgabe für die Artikelzusammenfassung

Welche Aktion ausgeführt wird hängt dabei immer vom Slot *aktion* ab. Dieser ist im Interaction Model innerhalb des Intents definiert. Statt wie üblich Alexa die volle Kontrolle über den Dialog und somit das Abfragen der Werte zu geben, wird der Slot manuell im Backend über die *ElicitSlotDirective* abgefragt, und zwar so lange, bis der Benutzer zum Beispiel mit dem Sprachbefehl „zurück“ wieder aus dem Intent herausnavigiert. Der große Vorteil besteht darin, dass der Entwickler wesentlich mehr Kontrolle über den Dialog erhält und insgesamt weniger Limitierungen hat. Der Nachteil ist hingegen, dass es häufig einen deutlich längeren Quellcode zur Folge hat, da jegliche Use-Cases im Backend abgedeckt werden müssen und zulässige bzw. unzulässige Slotwerte manuell im Backend zu definieren und prüfen sind.

```
return response_builder.speak(sprach_ausgabe).ask(
    sprach_prompts[ 'AKTION_REPROMPT' ]
).add_directive(
    ElicitSlotDirective(slot_to_elicit='aktion')
).response
```

AbfrageEigenenInserateIntent.py: Manuelle Abfrage des Slots „aktion“

Der Slot wird dabei über das Patternmatching (Feature ab Python 3.10) abgefragt und löst je nach enthaltenem Wert eine unterschiedliche Aktion aus. Das Patternmatching ist in etwa vergleichbar mit einem Switch-Case-Ausdruck aus anderen Programmiersprachen, jedoch erweiterten Möglichkeiten. Da beim anfänglichen Eintritt in den Intent der Slot *aktion* zunächst den Wert *None* hat (der Benutzer kann beim Starten des Intents keine Aktion sagen), wird dieser Fall dazu verwendet, um mittels der Datenbankmethode *get_eigene_inserate* alle eigenen Anzeigen abzufragen. Dieser Fall ist einmalig, da ab jeder weiteren Wiederholung der Slot immer einen Wert haben wird. Das soll verhindern, dass die eigenen Anzeigen bei jedem Durchlauf erneut abgefragt werden und dabei einen nicht notwendigen Zugriff auf die Datenbank auslösen. Zu Beginn des Intents wird der Benutzer zunächst wie bereits bei der Komponente *Database* angemerkt nach einer Sortierung gefragt, welche dann als Parameter übergeben wird. Die

Slotvalidierung findet dabei innerhalb von Alexa statt. Dafür muss der Dialog bei einer ungültigen Sortierung mit der *DelegateDirective* zurück an Alexa delegiert werden, welche dann den Reprompt durchführt. Die Sitzung beendet sich dabei automatisch nach zwei ungültigen Benutzereingaben.

```
match slots['aktion'].value:
    # Slots zu Beginn leer <=> Als erstes ausgeführt, um eigene Inserate zu laden
    case None:
        session_attr['eigene_inserate'] = Database.MongoDB.get_eigene_inserate(
            slots['reihenfolge'].value
        )
        logging.info(f'{session_attr["eigene_inserate"]=}')
        # Ungültige Sortierung angegeben
        if not session_attr['eigene_inserate']:
            # Alexa Slot Validation verwenden <=> Sitzung endet nach 2 falschen Angaben
            return response_builder.add_directive(
                DelegateDirective(updated_intent=intent)
            ).response
```

AbfrageEigeneInserateIntent.py: Auslösen eines Reprompts bei einer ungültigen Sortierung und anschließende Abfrage der eigenen Anzeigen in der Datenbank

AuskunftIntent.py

Die Komponente für den *AuskunftIntent* gibt einem Benutzer einen Gesamtüberblick über alle möglichen Funktionen, die *4Free* zu bieten hat.

Für die Komponente wurde eine beispielhafte APL-Implementierung vorgenommen, welche alle per Sprachausgabe aufgelisteten Funktionen zusätzlich in Form einer Listendarstellung auf einem Alexa-Gerät mit Bildschirm (ausgenommen Alexa-Spot und Fire-TV) abbildet, da die Informationsmenge sieben Elemente übersteigt und sich ein Benutzer diese Informationen schlecht merken kann.

Zu Beginn prüft die *get_supported_interface* Methode, ob das verwendete Gerät einen Bildschirm besitzt. Ist das der Fall, wird eine *RenderDocumentDirective* gesendet, welche das definierte APL-Dokument rendert und auf dem Alexa-Gerät

darstellt. Die Überprüfung ist an der Stelle wichtig, weil ein Alexa-Gerät ohne einen Bildschirm zu einem Fehler und Absturz führt.

```
if get_supported_interfaces(handler_input).alexa_presentation_apl:
    response_builder.add_directive(
        RenderDocumentDirective(
            document=_lade_apl_dokument('./lambda/py/apl/auskunft_apl.json'),
            datasources=get_apl_daten()
        )
    )
```

DetaillierteSucheIntent.py

Der *DetaillierteSucheIntent* implementiert mit seiner Komponente die detaillierte Suche nach Artikeln. Der Unterschied zum *EinfacheSucheIntent* besteht darin, dass der Fragenkatalog mit sieben statt vier Fragen größer ist, um akkuratere Suchergebnisse zu erzielen. Aus dem Grund gelten die folgenden Erläuterungen stellvertretend für beide genannten Komponenten.

Ein Benutzer hat beim Aufruf des Intents die Möglichkeit den Slot *bezeichnung* für den Artikelnamen zu übergeben. Sofern dieser Slot übergeben wurde, verkürzt sich der jeweilige Fragenkatalog um eine Frage bzw. die Frage wird im Dialog übersprungen. Wurde der Slot jedoch nicht übergeben, wird dieser manuell über die *ElicitSlotDirective* als zweite Frage im Dialog abgefragt.

```
# Bezeichnung abfragen <=> nicht als Input mitgeliefert
slots = intent.slots
if not slots['bezeichnung'].value:
    sprach Ausgabe += f' {random.choice(sprach_prompts["ARTIKEL_BEZEICHNUNG_ERFRAGEN"])}</speak>'
    return response_builder.speak(sprach Ausgabe).ask(
        random.choice(sprach_prompts['ARTIKEL_BEZEICHNUNG_ERRAGEN_REPROMPT'])
    ).add_directive(
        ElicitSlotDirective(slot_to_elicit='bezeichnung')
    ).response
```

DetaillierteSucheIntent.py: Manuelle Abfrage des Slots „bezeichnung“, falls dieser nicht in der Utterance übergeben wurde

Sobald der Dialog abgeschlossen ist, werden die Ergebnisse der Suche dem Sitzungsattribut *suchergebnisse* zugewiesen. Daraufhin kommt es zu einer Intent-

Weiterleitung bzw. Intent-Verkettung mit Hilfe der *DelegateDirective* unter Angabe des Namens für den Ziel-Intent (hier *SuchergebnisseIntent*). Zu beachten ist, dass eine Intent-Verkettung nur dann funktioniert, wenn der Ziel-Intent einen Dialog implementiert und die automatische Delegation deaktiviert ist. Außerdem landet der Benutzer direkt im Dialogzustand *Dialog.IN_PROGRESS* statt wie üblich in *Dialog.STARTED*, da der Benutzer indirekt in den Intent weitergeleitet wurde.

```
return response_builder.add_directive(
    DelegateDirective(
        updated_intent=Intent(
            name='SuchergebnisseIntent',
            confirmation_status=IntentConfirmationStatus.NONE
        )
    )
).response
```

DetaillierteSucheIntent.py: Intent-Verkettung bei Abschluss des Dialoges im detaillierte_suche_completed_handler

InseratErzeugenIntent.py

Die Komponente für den *InseratErzeugenIntent* dient dazu, dass ein Benutzer mit verknüpftem Konto eine Anzeige auf *4Free* schalten kann, um seinen Artikel anzubieten. Analog zu den Intents für die Suche muss der Benutzer in einem längeren Dialog per Spracheingabe alle Slots füllen. Dieser Prozess wurde bereits erläutert und wird an der Stelle nicht erneut behandelt.

Sind alle Slots ausgefüllt, werden die Slotwerte zum Schluss in ein Dictionary-Objekt verpackt und als Parameter an die Datenbankmethode *benutzer_speichere_inserat* übergeben. Diese ergänzt das Objekt um Metadaten (u.a. die des Benutzers) und speichert dieses anschließend in seiner Gesamtheit in der Datenbank, um die Anzeige für alle Benutzer zugänglich zu machen. Wird der Artikel erfolgreich gespeichert ertönt ein Erfolgston samt Sprachausgabe. Tritt bei der Speicherung ein Fehler auf, wird dagegen ein Fehlerton mit Sprachausgabe ausgegeben.

```

inserat_dokument = {
    # Metabeschreibung wird in benutzer_speichere_inserat() hinzugefügt
    'artikeldaten': {
        'bezeichnung': slot_wert_bezeichnung,
        'material': slot_wert_material,
        'abholung': slot_wert_abholung,
        'farbe': slot_wert_farbe,
        'zustand': slot_wert_zustand,
        'hersteller': slot_wert_hersteller,
        'hoehe': float(slot_wert_hoehe),
        'breite': float(slot_wert_breite),
        'laenge': float(slot_wert_laenge),
        'stueckzahl': int(slot_wert_stueckzahl),
        'anmerkung': slot_wert_anmerkung
    }
}

if Database.MongoDB.benutzer_speichere_inserat(inserat_dokument):
    response_builder.speak(
        f'<speak>'
        f'{sprach_prompts["INSERTAT_SPEICHERN_ERFOLG_AUDIO"]}'
        f'{random.choice(sprach_prompts["INSERTAT_SPEICHERN_ERFOLG"])}'
        f'</speak>'
    )
else:
    response_builder.speak(sprach_prompts['INSERTAT_SPEICHERN_FEHLER'])

```

InseratErzeugenIntent.py: Erzeugen eines Dictionary-Objektes mit den Artikeldaten und anschließende Speicherung in der Datenbank

SucheStartenIntent.py

Die folgende Komponente hat den Zweck den Benutzer zur gewünschten Suchart zu navigieren. Dieser hat die Wahl zwischen einer einfachen Suche mit weniger Fragen und einer detaillierten Suche mit mehr Fragen. Die Weiterleitung geschieht wiederum über eine *DelegateDirective* unter Angabe des Intentnamens. Dieser variiert je nach Antwort des Benutzers und nimmt entweder den Wert *EinfacheSucheIntent* oder *DetaillierteSucheIntent* an. Sollte der Benutzer beim Aufrufen des *SucheStartenIntents* in der jeweiligen Utterance eine Bezeichnung für den Artikel mitliefern, so wird dieser Slotwert ebenfalls an den zu delegierenden

Intent weitergereicht und dort als gleichnamiger Slot *bezeichnung* mit dem übergebenen Slotwert gespeichert.

```
return response_builder.add_directive(
    DelegateDirective(
        updated_intent=Intent(
            # Weiterleitung zu passendem Intent
            name=zu_delegierender_intent_name,
            confirmation_status=IntentConfirmationStatus.NONE,
            slots={
                'bezeichnung': Slot(
                    name='bezeichnung',
                    confirmation_status=SlotConfirmationStatus.NONE,
                    # Artikelnamen übergeben, falls ex.
                    value=gesuchter_artikel_name
                )
            }
        )
    )
).response
```

SucheStartenIntent.py: Intent-Verkettung mit Übergabe des Slots „bezeichnung“

Weitere Intents

Alle weiteren Intents sind selbsterklärend oder verwenden die bereits erläuterten Mechanismen und Funktionsweisen und werden darum nicht weiter behandelt.

Rasa

RASA Dokumentation

Was ist RASA

Rasa ist eine Conversational AI Platform, welche mit dem Rasa Stack eine Open-Source Software zur Verfügung stellt, um Contextual AI Assistants und Chatbots zu erstellen. Neben dem kostenfreien RASA-Stack gibt es außerdem die RASA-Platform. Diese setzt ebenso auf den RASA-Stack, bietet allerdings für Enterprise-Kunden zusätzliche Features, wie zum Beispiel ein User-Interface mit Funktionalitäten wie zum Beispiel Training Data, Admin, Conversations und Models API. Entwickelt wird RASA vom gleichnamigen Start-up mit Hauptsitz in Berlin. Zudem gibt es über dreihundert Beteiligte, welche RASA im Open-Source Sinn voranbringen.

Vorteile von RASA

Es gibt viele Vorteile von RASA. An der Stelle werden die wichtigsten zusammengefasst:

- Durch Open Source ist man nicht auf kostenpflichtige Lizenzen angewiesen
- Durch die Einsicht des gesamten Quellcodes ist eine hohe Use-Case spezifische Konfiguration möglich
- Offlineservice: Einsatz in Offline-Systemen wie zum Beispiel IoT Geräten umsetzbar

- RASA läuft on-premises, anders formuliert man hat die vollständige Kontrolle über die Daten, die der Chatbot generiert. Allerdings muss ein leistungsfähiger Server zur Verfügung gestellt werden

Installation von RASA

Für die Installation von RASA auf Windows sind einige Werkzeuge notwendig. Die Vorgehensweise dafür wird im weiteren Verlauf erläutert. Studierende der nächsten Semester haben somit eine Anleitung für die Installation unter Windows.

Die Installation unterteilt sich in folgende Schritte:

1. Anaconda installieren oder irgendeine einfache Installation eines Linux-Betriebssystems
2. Microsoft Visual C++ Redistributable installieren
3. Verzeichnis „MyFirstBot“ erstellen und in das Verzeichnis wechseln
4. In der Kommandozeile folgende Kommandos der Reihe nach ausführen
 - *conda create --name chatbotenv python==3.7.6*
 - *conda activate chatbotenv*
 - *conda install ujson==2.1.3*
 - *conda install tensorflow=2.0 python=3.7*
 - *pip install rasa==1.10.0*
 - *rasa init*

1. Anaconda installieren

Anaconda ist ein freies Installationsprogramm (Open Source) zur einfachen Installation eines Linux-Betriebssystems und basiert auf einer Open-Source-Distribution für die Programmiersprachen Python und R.

Anaconda lässt sich über die offizielle Seite herunterladen: [Anaconda](#)

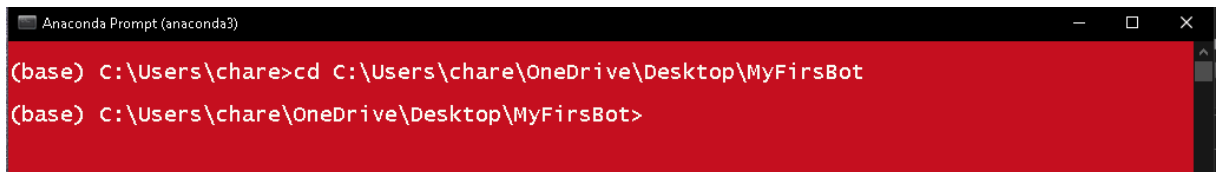
Die Installation von Anaconda verläuft wie eine übliche Installation eines Standardprogrammes.

2. Microsoft Visual C++ Redistributable installieren

Die „[Visual C++ Redistributable](#)“-Versionen installieren Laufzeitkomponenten, welche zur Ausführung von Programmen dienen, die mit der Programmiersprache C oder C++ entwickelt wurden. Um Probleme zu vermeiden, wird die Installation empfohlen.

3. Verzeichnis „MyFirstBot“ erstellen und in das Verzeichnis wechseln

Das erstellte Verzeichnis dient als Ablageort für die Dateien des Chatbots. Vom aktuellen Arbeitspfad in der Kommandozeile wird in das erstellte Verzeichnis „*MyFirstBot*“ navigiert.

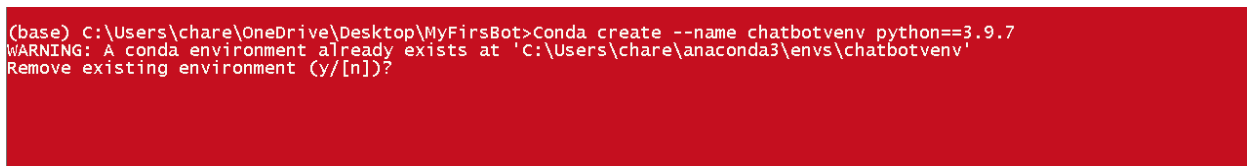
A screenshot of the Anaconda Prompt window. The title bar reads "Anaconda Prompt (anaconda3)". The command prompt shows the user navigating to a directory: (base) C:\Users\chare>cd C:\Users\chare\OneDrive\Desktop\MyFirsBot. The next line shows the user is now in that directory: (base) C:\Users\chare\OneDrive\Desktop\MyFirsBot>.

```
(base) C:\Users\chare>cd C:\Users\chare\OneDrive\Desktop\MyFirsBot
(base) C:\Users\chare\OneDrive\Desktop\MyFirsBot>
```

4. In der Kommandozeile

Nun wird eine virtuelle Umgebung (Virtual Environment) mit dem Namen „*chatbotenv*“ mit folgendem Kommando auf der Kommandozeile erzeugt:

➔ *conda create --name chatbotenv python==[Python Version]*

A screenshot of the Anaconda Prompt window showing the command to create a new environment. The command is: (base) C:\Users\chare\OneDrive\Desktop\MyFirsBot>Conda create --name chatbotenv python==3.9.7. Below the command, a warning message is displayed: WARNING: A conda environment already exists at 'C:\Users\chare\anaconda3\envs\chatbotenv'. The prompt then asks: Remove existing environment (y/[n])?.

```
(base) C:\Users\chare\OneDrive\Desktop\MyFirsBot>Conda create --name chatbotenv python==3.9.7
WARNING: A conda environment already exists at 'C:\Users\chare\anaconda3\envs\chatbotenv'
Remove existing environment (y/[n])?
```

Falls in dem Verzeichnis bereits eine virtuelle Umgebung existiert, fragt die Kommandozeile, ob diese überschrieben werden soll. Die Kommandozeile zeigt anschließend alle installierten Pakete an.

Sollte bei der Installation eine Fehlermeldung auftauchen ist mit hoher Wahrscheinlichkeit anzunehmen, dass „*Visual C++ Redistributable*“ nicht (richtig) installiert wurde.

```
Collecting package metadata (current_repodata.json): done
Solving environment: done
```

Die weitere Installation muss anschließend mit der Taste „y“ bestätigt werden. Danach wird die Umgebung fertig installiert und zeigt die Kommandos zur Aktivierung bzw. Deaktivierung der virtuellen Umgebung auf der Kommandozeile.

```
Proceed ([y]/n)? y

Downloading and Extracting Packages
sqlite-3.37.0          | 785 KB          | #####
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
#
# To activate this environment, use
#
#     $ conda activate chatbotenv
#
# To deactivate an active environment, use
#
#     $ conda deactivate
```

Zunächst muss die virtuelle Umgebung aktiviert werden. Zur Aktivierung wird folgendes Kommando verwendet:

➔ *conda activate chatbotenv*

Zur Deaktivierung wiederum:

➔ *conda deactivate*

Das die virtuelle Umgebung aktiviert wurde lässt sich daran erkennen, dass vor dem jeweiligen Verzeichnis der Name der virtuellen Umgebung in runden Klammern steht.

```
(base) C:\Users\chare\OneDrive\Desktop\MyFirsBot>conda activate chatbotenv
(chatbotenv) C:\Users\chare\OneDrive\Desktop\MyFirsBot>
```

Um RASA verwenden zu können werden noch zwei weitere Pakete benötigt:

- **Ujson:** *conda install ujson==2.1.3* (Die Version verursacht am wenigsten Probleme)

- **Tensorflow:** *conda install tensorflow=2.0 python=3.7*

Nach Abschluss der Installationen wird RASA anschließend über folgendes Kommando installiert:

```
(chatbotenv) C:\Users\chare\OneDrive\Desktop\MyFirsBot>pip install rasa==1.10.0
```

Die Version ist beliebig wählbar und kann durch ein Update auch nach der Installation auf den neusten Stand gebracht werden.

Um mit der Arbeit in RASA zu beginnen bzw. alles Notwendige dafür zu initialisieren wird folgendes Kommando verwendet:

```
(chatbotenv) C:\Users\chare\OneDrive\Desktop\MyFirsBot>rasa init
Welcome to Rasa! 🐍

To get started quickly, an initial project will be created.
If you need some help, check out the documentation at https://rasa.com/docs/rasa.
Now let's start! 📄

? Please enter a path where the project will be created [default: current directory] .
```

Die Willkommensnachricht zeigt, dass die Ausführung erfolgreich war. Zusätzlich möchte RASA ein Verzeichnis wissen, um die Projektdaten zu installieren. Ohne Angabe wird das aktuelle Verzeichnis gewählt.

In einem letzten Schritt wird gefragt, ob der Chatbot gestartet werden soll. Das kann mit der Taste „y“ bestätigt werden.

```
NLU model training completed.
Your Rasa model is trained and saved at 'C:\Users\chare\OneDrive\Desktop\MyFirsBot\models\20220110-025627.tar.gz'.
? Do you want to speak to the trained assistant on the command line? 📄 (Y/n)
```

Im Folgenden sind ein paar beispielhafte Fragen und die Reaktion des Chatbots dargestellt.

```
2022-01-10 02:59:33 INFO      root - Starting Rasa server on http://localhost:5005
Bot loaded. Type a message and press enter (use '/stop' to exit):
Your input -> hello
Hey! How are you?
Your input -> i'm sad
Here is something to cheer you up:
Image: https://i.imgur.com/nGF1k8f.jpg
Did that help you?
Your input -> Great - Thanks
Great, carry on!
Your input ->
```

Im Ordner „*MyFirstBot*“ befinden sich nun alle von RASA generierten Dateien und Verzeichnisse, wie im Folgenden zu sehen.

MyFirsBot		"MyFirsBot" durchsuchen		
	Name	Änderungsdatum	Typ	Größe
	__pycache__	10.01.2022 02:55	Dateiordner	
	data	10.01.2022 02:55	Dateiordner	
	models	10.01.2022 02:57	Dateiordner	
	tests	10.01.2022 02:55	Dateiordner	
	__init__	10.01.2022 02:48	JetBrains PyCharm	0 KB
	actions	10.01.2022 02:48	JetBrains PyCharm	1 KB
	config	10.01.2022 02:48	YML-Datei	1 KB
	credentials	10.01.2022 02:48	YML-Datei	1 KB
	domain	10.01.2022 02:48	YML-Datei	1 KB
	endpoints	10.01.2022 02:48	YML-Datei	2 KB

Das gesamte Projektverzeichnis soll nun mit einem beliebigen Editor wie zum Beispiel Visual Studio Code oder Atom geöffnet werden.

Im weiteren Verlauf werden die wichtigsten Dateien erläutert.

nlu.md

Das Ziel von NLU (Natural Language Understanding) ist es, strukturierte Informationen aus unseren Nachrichten zu extrahieren. Dies beinhaltet üblicherweise die Absicht des Benutzers und alle Entitäten, die seine Nachricht enthält.

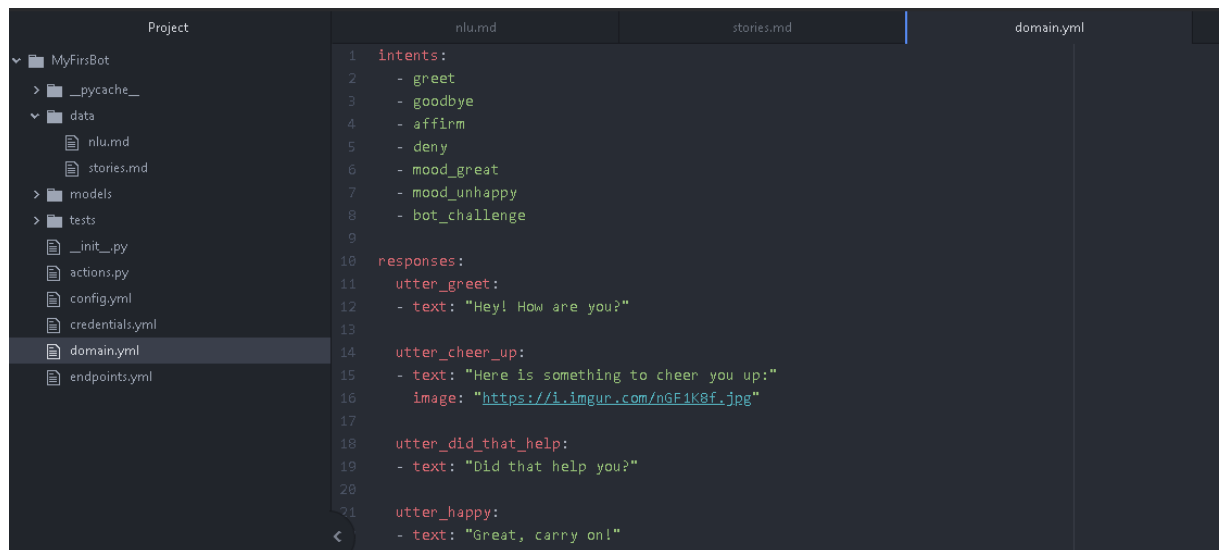
Für die Definition der Intents mit zugehörigen gültigen Utterances wird eine stichpunktartige Notation verwendet.

Beispielsweise werden für den ersten Intent „*greet*“ folgende Utterances akzeptiert:

Project	nlu.md	stories.md	domain.yml
MyFirsBot	1 ## intent:greet		
__pycache__	2 - hey		
data	3 - hello		
nlu.md	4 - hi		
stories.md	5 - good morning		
models	6 - good evening		
tests	7 - hey there		
__init__.py	8		
actions.py	9 ## intent:goodbye		
config.yml	10 - bye		
credentials.yml	11 - goodbye		
domain.yml	12 - see you around		
endpoints.yml	13 - see you later		
	14		
	15 ## intent:affirm		

domain.yml

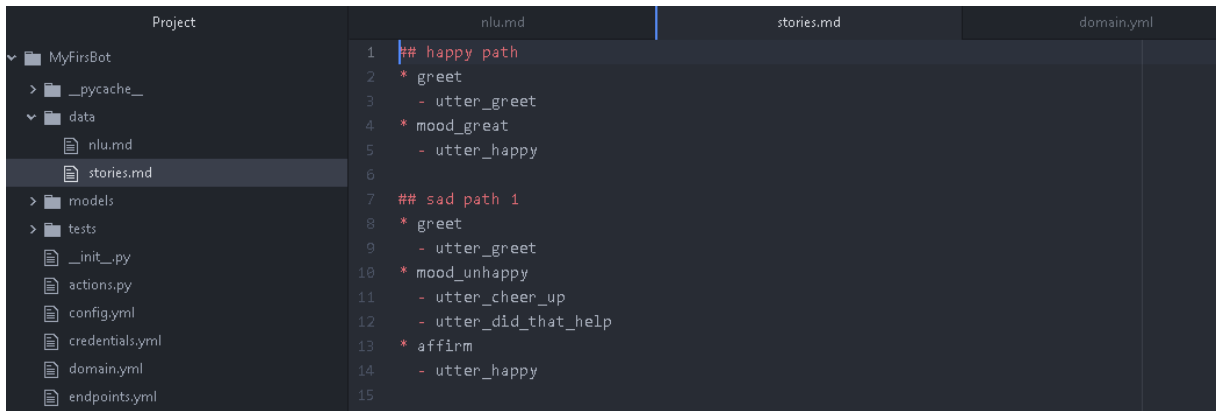
In der Domäne werden die Absichten, Entitäten, Slots, Antworten, Formulare und Aktionen des Assistenten definiert, über die der Chatbot Bescheid wissen sollte. Zusätzlich werden hier auch Konfigurationen für die Sitzung vorgenommen. Am Dateianfang sind alle in der *nlu.md* definierten Intents sichtbar. Zu den Intents werden in diesem Beispiel nun die jeweiligen Antworten des Assistenten festgelegt.



```
1 intents:
2   - greet
3   - goodbye
4   - affirm
5   - deny
6   - mood_great
7   - mood_unhappy
8   - bot_challenge
9
10 responses:
11   utter_greet:
12     - text: "Hey! How are you?"
13
14   utter_cheer_up:
15     - text: "Here is something to cheer you up:"
16       image: "https://i.imgur.com/nGF1k8f.jpg"
17
18   utter_did_that_help:
19     - text: "Did that help you?"
20
21   utter_happy:
22     - text: "Great, carry on!"
```

Stories.md

Stories (Geschichten) sind eine Art von Trainingsdaten, die dazu verwendet werden, das Dialogmanagementmodell des Assistenten zu trainieren. Mit anderen Worten: Es werden theoretisch mögliche Dialogbeispiele samt Antworten definiert. Dabei wird pro im Laufe des Dialoges aufgerufenen Intent immer nur eine passende Antwort aus der *domain.yml* zugeordnet, wie in der folgenden Abbildung zu erkennen ist.



Beispiel:

Zu dem Intent *greet* bestehend aus den Utterances *{hey, hallo, hay, ...}* welche innerhalb der *nlu.md* definiert sind geben wir eine Antwort *utter_greet*, welche auf den String „*Hey! How are you?*“ auflöst, wie in der *domain.yml* beschrieben. Enthält eine Antwortreferenz mehrere potentielle Antworten, so wird die auszugebende Antwort zufällig aus allen möglichen enthaltenen Antworten ausgewählt.

An der Stelle ein kleiner Test:

Wenn ein Benutzer den Chatbot mit einer der folgenden Utterances aus dem Intent *greet* aufruft, wird dieser immer die Antwort „*Hey! How are you?*“ zurückgeben, da diese als einzige Antwort für diesen Intent definiert wurde.

- *hey*
- *hello*
- *hi*
- *good morning*
- *good evening*
- *hey there*

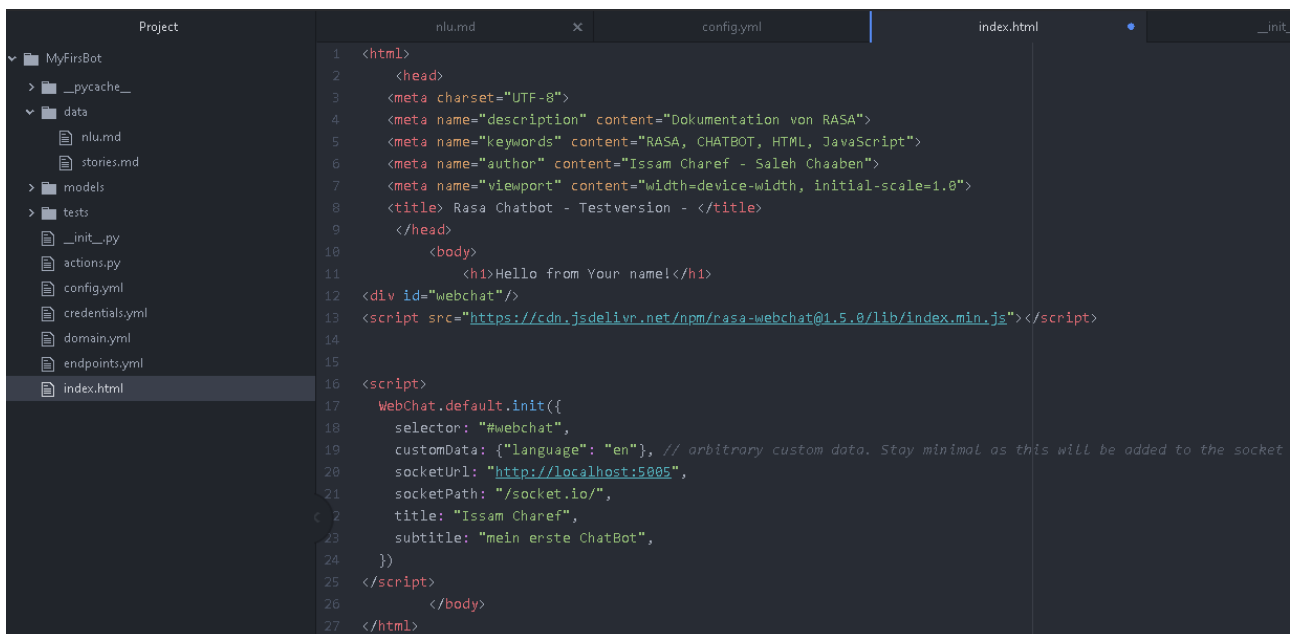
```
Your input -> hey
Hey! How are you?
Your input -> hallo
Hey! How are you?
Your input -> hay
Hey! How are you?
Your input ->
```

config.yml

In der Datei können die Sprache, Pipelineschlüssel, Richtlinien und weitere Konfiguration geändert werden.

Integration von RASA in eine Webanwendung

Um den Chatbot auf *localhost* auszuführen wurde eine einfache HTML-Seite erstellt und die RASA-Anwendung über ein JavaScript-Snippet verknüpft, wie in der Abbildung zu sehen.



```
1 <html>
2 <head>
3   <meta charset="UTF-8">
4   <meta name="description" content="Dokumentation von RASA">
5   <meta name="keywords" content="RASA, CHATBOT, HTML, JavaScript">
6   <meta name="author" content="Issam Charef - Saleh Chaaben">
7   <meta name="viewport" content="width=device-width, initial-scale=1.0">
8   <title> Rasa Chatbot - Testversion - </title>
9 </head>
10 <body>
11   <h1>Hello from Your name!</h1>
12 <div id="webchat">
13   <script src="https://cdn.jsdelivr.net/npm/rasa-webchat@1.5.0/lib/index.min.js"></script>
14
15   <script>
16     WebChat.default.init({
17       selector: "#webchat",
18       customData: {"language": "en"}, // arbitrary custom data. Stay minimal as this will be added to the socket
19       socketUrl: "http://localhost:5005",
20       socketPath: "/socket.io/",
21       title: "Issam Charef",
22       subtitle: "mein erste ChatBot",
23     })
24   </script>
25 </div>
26 </body>
27 </html>
```

Der nächste Schritt besteht daraus Anpassungen in der *credentials.yml* vorzunehmen, um eine Verbindung zwischen dem Chatbot und Facebook, Slack, usw. herzustellen. Für diesen Anwendungsfall wurde an der Stelle *SocketIO* verwendet, wie zu erkennen ist.

```
> tests
  _init_.py
  actions.py
  config.yml
  credentials.yml
  domain.yml
  endpoints.yml
  index.html

8
9
10 #facebook:
11 # verify: "<verify>"
12 # secret: "<your secret>"
13 # page-access-token: "<your page access token>"
14
15 #slack:
16 # slack_token: "<your slack token>"
17 # slack_channel: "<the slack channel>"
18
19 socketio:
20   user_message_evt: user_uttered
21   bot_message_evt: bot_uttered
22   session_persistence: true
23
24 #mattermost:
25 # url: "https://<mattermost instance>/api/v4"
26 # token: "<bot token>"
27 # webhook_url: "<callback URL>"
28
29 # This entry is needed if you are using Rasa X. The entry represents credentials
30 # for the Rasa X "channel", i.e. Talk to your bot and share with guest testers.
31 rasa:
32   url: "http://localhost:5002/api"
```

Dabei ist der Flag `session_persistence` gesetzt, um die Unterhaltung des Benutzers zu speichern und bei der nächsten Sitzung wieder anzuzeigen.

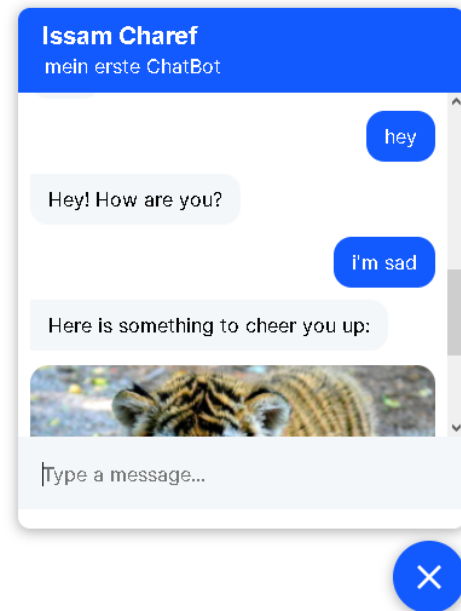
In einem letzten Schritt muss nur noch das folgende Kommando ausgeführt werden, um den Chatbot in der Webanwendung zum Laufen zu bekommen. Anschließend befindet sich die RASA-Anwendung im Indexverzeichnis.

```
(chatbotenv) C:\Users\chare\OneDrive\Desktop\MyFirsBot>rasa run --cors "*" --debug
2022-01-10 04:19:35 DEBUG rasa.cli.utils - Parameter 'endpoints' not set. Using default lo
2022-01-10 04:19:38 DEBUG rasa.cli.utils - Parameter 'credentials' not set. Using default
2022-01-10 04:19:39 DEBUG rasa.model - Extracted model to 'C:\Users\chare\AppData\Local\Te
2022-01-10 04:19:42 DEBUG sanic.root - CORS: Configuring CORS with resources: {'/*': {'ori
low_headers': ['.*'], 'expose_headers': None, 'supports_credentials': True, 'max_age': None, '
rces': {'/*': {'origins': ['*']}}}, 'intercept_exceptions': True, 'always_send': True}}
2022-01-10 04:19:42 DEBUG rasa.core.utils - Available web server routes:
/webhooks/rasa GET custom_webho
/webhooks/rasa/webhook POST custom_webho
/webhooks/rest GET custom_webho
/webhooks/rest/webhook POST custom_webho
/socket.io GET handle_reque
/ GET hello
/webhooks/socketio GET socketio_web
```

Das finale Resultat des integrierten Chatbots sieht dann wie folgt aus:

Dokumentation von RASA -

Issam Charef & Saleh Chaaben



Anmerkung:

Das Kommando `cls()` kann dazu verwendet werden, um das Anaconda-Terminal zu leeren. Außerdem können alle möglichen Anaconda-Kommandos über die Option `-h` abgefragt werden.

Chatbot für 4Free

Der Chatbot wurde in eine Webanwendung integriert und beantwortet allgemeine Fragen rund um *4Free* bzw. den zugehörigen Alexa-Skill. Der Chatbot ist unter folgendem Link erreichbar:

- [Link](#)

Die Webanwendung wird auf einem öffentlich zugänglichen Server gehostet und kann von jedem mit einem Internetbrowser verwendet werden.

Probleme mit RASA

RASA hat keine direkte Möglichkeit, um den Chat wieder zu schließen. Über eine API wie zum Beispiel „Kong“ kann das Problem ggf. gelöst werden.

NLU

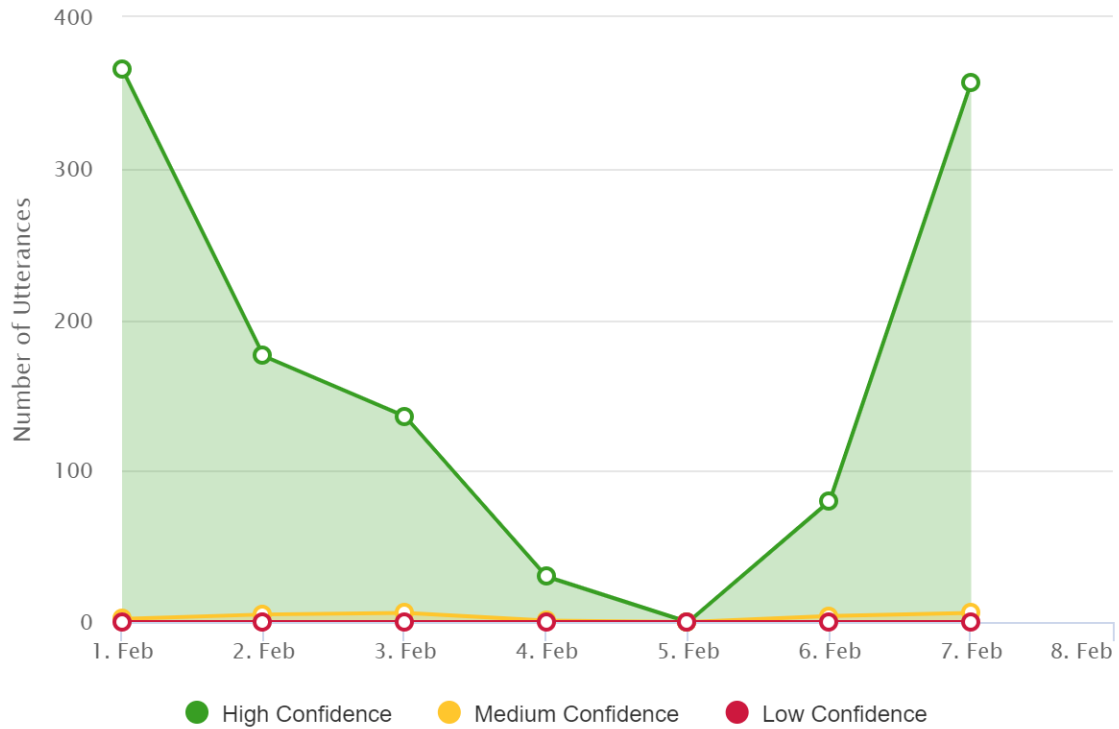
Dokumentation zur Genauigkeitserkennung (NLU Vergleich)

Ein direkter Vergleich der Genauigkeit der NLU-Komponente von Alexa und RASA ist nicht umsetzbar, da der Analytics-Tab der Alexa-Developer-Console keinen Aufschluss über den jeweiligen Konfidenzwert einzelner Intents zulässt. Somit sind keine aussagekräftigen Vergleiche der Erkennungsgenauigkeiten zwischen den Intents aus Alexa und denen aus RASA möglich. Als weiteres Hindernis erweist sich die grobe Kategorisierung der Konfidenzwerte in lediglich

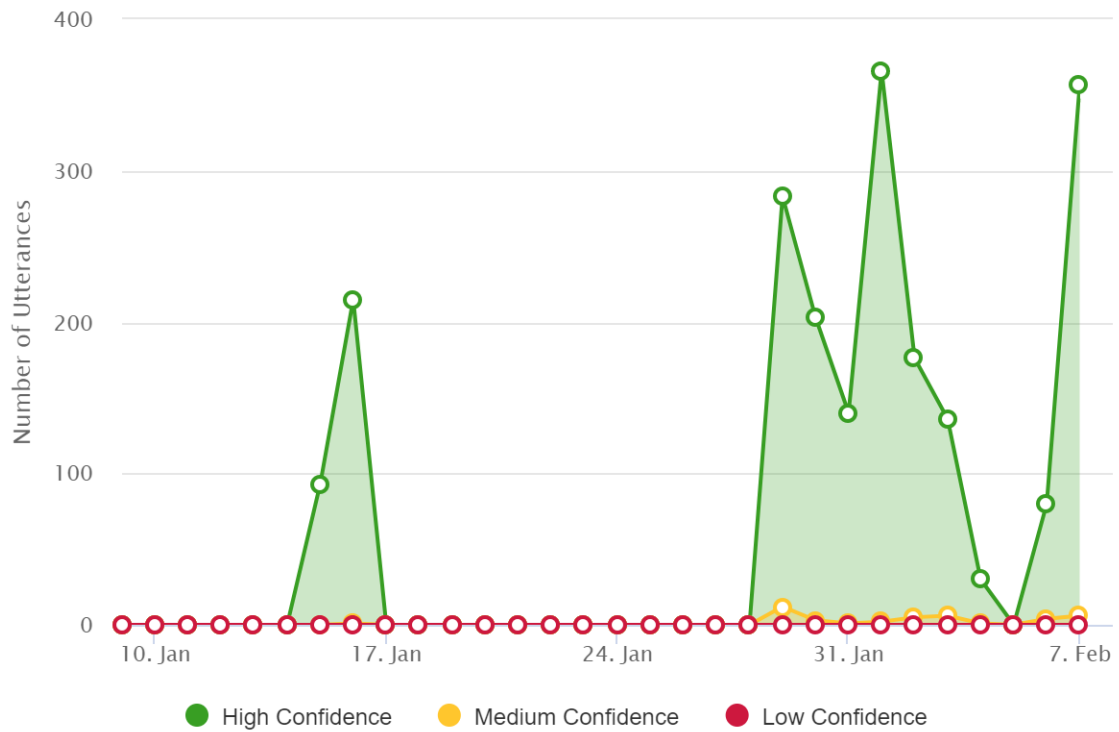
- *Low Confidence*
- *Medium Confidence*
- *High Confidence*

im Analytics-Bereich von Alexa. Welche genauen Konfidenzwerte sich hinter den Begriffen verstecken ist dabei nicht erkennbar.

Die einkategorisierung der zusammengefassten *Intent Confidence* hängt davon ab, wie viele der Utterances an dem jeweiligen Tag der Verwendung des Skills den jeweils richtigen Intents zugeordnet werden konnten. An einem Tag an dem der Skill kaum genutzt und Intents häufig falsch zugeordnet wurden, würde sich für den Tag ein niedriger Konfidenzwert ergeben.



Intent Confidence von 4Free über die Laufzeit vom 01.02.21 – 07.02.21 (eine Woche)



Intent Confidence von 4Free über die Laufzeit vom 10.01.21 – 07.02.21 (ungefähr ein Monat)

Aus den obigen Graphen lassen sich drei Sachverhalte ableiten

1. Der Skill wurde an vereinzelten Tagen sehr intensiv genutzt
2. Die Konfidenzwerte waren fast durchgehend im hohen Bereich
3. Ein geringer Konfidenzwert wurde zu keinem Zeitpunkt erreicht

Insgesamt funktioniert die Intent Erkennung für *4Free* in Alexa also sehr gut und es wird im Schnitt ein Konfidenzwert im hohen Bereich erzielt.

In RASA werden im Vergleich zu Alexa die Konfidenzwerte der einzelnen Intents auf der Kommandozeile genau angegeben, was die Auswertung dieser erheblich erleichtert. Um zumindest einen groben Vergleich mit dem Alexa Skill vornehmen zu können wurden ausgewählte Intents und zugehörige Utterances aus Alexa in RASA migriert, darunter:

- *EntwicklerInfoIntent*
- *InseratErzeugenIntent*
- *AbfrageEigeneInserateIntent*

Für den *EntwicklerInfoIntent* ergibt sich nach einem Test folgende Auswertung.

```
(chatbotvenv) C:\Users\chare\OneDrive\Desktop\RASA\ForFree>rasa shell nlu
2022-02-05 07:03:34.148821: E tensorflow/stream_executor/cuda/cuda_driver.
UNKNOWN ERROR (303)
NLU model loaded. Type a message and press enter to parse it.
Next message:
entwicklerinfointent
{
  "intent": {
    "name": "entwicklerInfoIntent",
    "confidence": 0.9752056002616882
  },
  "entities": [],
  "intent_ranking": [
    {
      "name": "entwicklerInfoIntent",
      "confidence": 0.9752056002616882
    }
  ],
}
```

RASA Kommandozeilenausgabe mit Konfidenzwert des *EntwicklerInfoIntent*

Mit knapp 97,52% befindet sich der Konfidenzwert in einem sehr hohen Bereich.

Für den folgenden Test für *InseratErzeugenIntent* ergibt sich mit 99,99% ein sogar noch höherer Konfidenzwert.

```
(chatbotenv) C:\Users\chare\OneDrive\Desktop\RASA\ForFree>rasa shell nlu
2022-02-05 07:27:58.232657: E tensorflow/stream_executor/cuda/cuda_driver.cc:351] failed
UNKNOWN ERROR (303)
NLU model loaded. Type a message and press enter to parse it.
Next message:
inseratErzeugenIntent
{
  "intent": {
    "name": "inseratErzeugenIntent",
    "confidence": 0.9999921917915344
  },
  "entities": [],
  "intent_ranking": [
    {
      "name": "inseratErzeugenIntent",
      "confidence": 0.9999921917915344
    }
  ]
}
```

RASA Kommandozeilenausgabe mit Konfidenzwert des InseratErzeugenIntent

Dies lässt sich damit begründen, dass der Konfidenzwert umso höher ist, je weniger Überschneidungen eine Utterance mit der eines anderen Intents hat bzw. je einzigartiger eine Utterance für den jeweiligen Intent ist.

Der letzte Test wurde für den *AbfrageEigeneInserateIntent* durchgeführt.

```
Next message:
abfrageEigeneInseratIntent
{
  "intent": {
    "name": "inseratErzeugenIntent",
    "confidence": 0.9521728754043579
  },
  "entities": [],
  "intent_ranking": [
    {
      "name": "inseratErzeugenIntent",
      "confidence": 0.9521728754043579
    }
  ]
}
```

RASA Kommandozeilenausgabe mit Konfidenzwert des AbfrageEigeneInserateInten

Zwar hatte der Test im direkten Vergleich zu den vorherigen mit 95,22% den niedrigsten Konfidenzwert, jedoch immer noch einen sehr hohen.

Abschließend lässt sich formulieren, dass die Intenterkennung von RASA insgesamt hervorragend funktioniert, jedoch aufgrund der unterschiedlichen Maßzahlen kein aussagekräftiger Vergleich mit einem Skill auf der Alexa Plattform gemacht werden kann. Trotz allem liefert auch der Alexa-Skill im Bezug auf seine Maßzahlen gute Werte für die Intenterkennung, was einerseits auf gut formulierte Intents und andererseits auf eine gute NLU schließen lässt.