Otto-von-Guericke University Magdeburg

Faculty of Computer Science
IIKT-Mobile Dialog Systems

Digital Engineering Project

A Chatbot for International Students' support

Authors:
Apoorva Chandrashekar
Gulandam Jabiyeva
Kiran Babu Thatha
Priyanka Byahatti

July 28, 2021

Supervisor
Jun.-Prof. Dr.-Ing. Ingo Siegert

Faculty of Computer Science
Otto-von-Guericke University
Universitätsplatz 2
39106 Magdeburg, Germany

Eleanor Jabiyeva, Apoorva Chandrashekar, Kiran Babu Thatha, Priyanka Byahatti:
A Chatbot for International Students' support
Digital Engineering Project, Otto-von-Guericke University
Magdeburg, 2021.

Contents

Abstract

The idea of the project was to identify a topic and build a chatbot for international students that allows them to engage with interactive content that helps answer typical questions related to the content. The chatbot was developed in Google Dialogflow platform which was used to develop and train using the inbuilt Natural Language Model. Along with Dialogflow, the Jovo framework was used to train the bot and also to connect with external databases to fetch dynamic information to fulfill a conversation. The chatbot functionalities were hosted on the google cloud platform as a cloud function and then finally published on the Slack Platform. To build the bot, many qualitative measures like usability and acceptance tests were taken into consideration. A structure was created to perform these analyses during the early stages of development so that we could come up with a bot that is efficient and easy to use. Post-deployment of the app, preliminary acceptance, and attractiveness tests were conducted using Questionnaires via Google forms to get feedback from users which were useful for improvising the bot functionality and for future research. The students generally approved of the final product due to its usability and well-structured data.

Acknowledgements

Our team would like to express gratitude to our supervisors Jun.-Prof. Dr.-Ing. Ingo Siegert and Matthias Busch for their extensive support. They have put tremendous effort to guide us in the right direction and gave us insightful information. We thank them for providing us necessary facilities that were required for the completion of project. During the unexpected times our supervisors equipped us with not just the facilities but they also provided huge moral support.

1 Introduction

1.1 Introduction

Chatbots are conversation bots that are used for several purposes such as gaining information, processing action, or advising the user through communication with customers in a digital environment. Nowadays, many advanced types of chatbots have entered our lives which can artificially replicate patterns of human communication making the conversations effective.

The chatbot built by our team serves all the international students with answers to all the questions, provides required information regarding the Foreigner's Office in Magdeburg (DE: Ausländeramt Magdeburg). Our main objective in selecting this topic was to help the students in understanding the documentation process in Germany, considering that they are new to this system including all the regulations and steps needed to be taken. Some students are new to Germany and can find the rules, documentation processes overwhelming with their ongoing studies. As international students, we have also been through such difficulties and we believe this chatbot would be able to facilitate and ease this process.

The content of the chatbot also covers different types of regulations specific to a country of origin or the current study status of students. Certain rules in the documentation process (required documents, fee amount, etc.) can vary depending on the origin of the student (European / Non-European). The rules can also vary based on the student status whether they are currently studying or have questions about the required steps after graduation. One of our main objectives was to adjust any possible situation to the chatbot using Dialogflow and Jovo Framework and provide relevant information using web-scraped data that is stored in Firestore Database. More detailed information about the tools we have used in building our chatbot will be provided in Chapter 2 of the report. To this extent, the chatbot gathers basic information about students through communication with users and subsequently answers the following questions. To ease this process and to make the chatbot user friendly, we used multiple methods of asking questions, such as multiple-choice questions or suggestions about content via visual content (buttons) that allows users to directly send mail, make an appointment online or get information about contact details in one click.

At the moment, the chatbot can provide necessary information and answers questions about following topics:

- Office Hours of Foreigner's Office

- Location / Address along with Contact details
- Means of transport to get to the office
- Work permission and tax information
- Immigration rules (specific to each user depending on their origin)
- Appointment
- Residence permit (general information about why it is required, what to do when it has been lost and what steps do users need to take depending on the case whether it is their first time of having one or they need extension of present residence permit card)
- List of required documents and application forms according to each documentation process

2

Tools and frameworks

This chapter includes tools and frameworks that we used to build our chatbot, in a more elaborate fashion. It contains information about the structure and working principles for each, and also discussed their principal role and practical application on our project. Furthermore, the reasons and justification about why especially those tools are selected is explained by highlighting thier advantages.

2.1 DialogFlow

Dialogflow[GOOGLE_DOCS_1] is an AI chatbot framework owned by Google, which has built-in features like machine learning capabilities, natural language features, and also integrations with many popular communication platforms. All these features help us to create intelligent bots which are capable of handling interesting and engaging conversations with the users. Dialogflow is supported by Google's Cloud Natural Language which makes it easier for the bot to understand the input from the users in the form of text or audio and respond to the human sentiments and emotions in the form of text or speech output.

Dialogflow offers two distinct virtual agent services known as Dialogflow EX and Dialogflow CS, each with its type of agents, API, user interfaces. In our project, we used the standard agent form (DailogFlow ES) that is appropriate for small to medium-sized and easy to moderately complex agents. Apart from the features mentioned above, the reasons for choosing the Dialogflow Platform - ES is as follows.

1. Price - Standard Edition is free and it helps us with all the features required for building a simple to moderately complex bot.
2. Delivers natural and rich conversational experiences - In comparison to platforms that operate on predefined queries, Dialogflow Agents are effective and provide better user experience through Natural Language Processing(NLP).
3. Offers cros-platform support - Dialogflow assists in the creation of a device-agnostic chatbot. It can be integrated with the most common messaging apps, including Facebook Messenger, Slack, Twitter, Line, Skype, Telegram, and Twilio with a single click.

Following are the features of the Dialogflow that we have used to develop the bot

2.1.1 Agents

A Dialogflow agent is a software agent that interacts with the end-users. The agents consist of Intents which in turn handles the logic behind the conversations. The agent is trained in a way that handles the phrases input by the user and returns the desired output.

2.1.2 Intents and Entities

An Intent is a specific action that the user can invoke by using one of the phrases that are defined in the Dialogflow console. It consists of training phrases - which are the ones the end-user might say, Action - Once the intent is matched, the agent provides the action which triggers certain events in the system, Parameters - Dialogflow can extract certain information from the user's phrases which are captured using entity type (For example, in the input phrase the terms apply and extend as shown in the image below where custom entities were created in Jovo), Responses - The responses to be given out to the user for a respective intent are given in the form of text, speech, or visual responses.

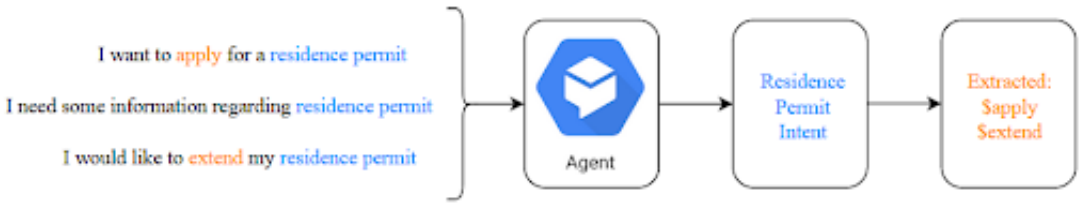


Figure 2.1: Agent detection[GOOGLE_DOCS_2]

2.1.3 Fulfillment for Integration

The agent gives the static response which is mentioned under each intent by default. Fulfillment can be used to provide a dynamic response to the user concerning the user's query. Each intent has an option to enable this fulfillment option. If the agent invokes an intent that doesn't have the fulfillment enabled, it uses the static response that is defined for a given intent.

When an input expression of the user is matched with an intent that has the fulfillment enabled, Dialogflow sends a request to the webhook service with details of the matched intent. In OttoBot, a POST request is sent to the webhook service which is the Google cloud function which then performs the required action. The detailed functionality of the webhook service is explained further in Section 5.1 with an example.

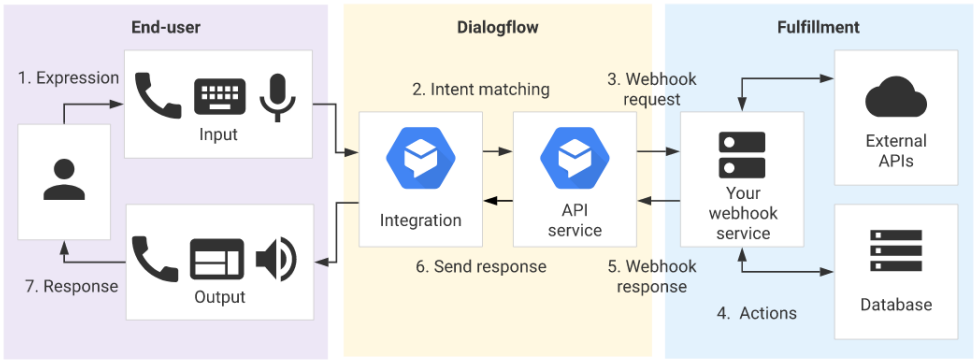


Figure 2.2: Webhook calling[GOOGLE_DOCS_3]

2.2 Jovo Framework

Jovo Framework[JOVO_DOCS_1] is an open-source framework that allows the creation of Alexa Skill and Google Action by building just one single Javascript application, which means it is not required to create both codes in two separate codebases which would make it difficult to maintain them. Jovo framework works with many different services, each of which has its own schema to be applied to their model's training. These are platforms and NLU (natural language understanding) providers that turn spoken or written language into structured meaning.

There are two main elements in a Jovo project:

1. Project Files
2. Source Files

2.2.1 Project Files

Project Files are the root folders which contain the overall project-related files i.e the project configuration, language models, and platform specific files.

The **project.js** file stores all the necessary information like the platforms the application can be hosted on, language model and the project files for deployment. As seen in the below Figure 2.3 (a), dialogflow object which is the root of the language model is added as a natural language understanding tool. A unique Jovo Webhook URL is automatically added as an endpoint which is used to reach the app. Deployment details can also be mentioned in the file however, in the case of our bot deployment is done via the Dialogflow platform.

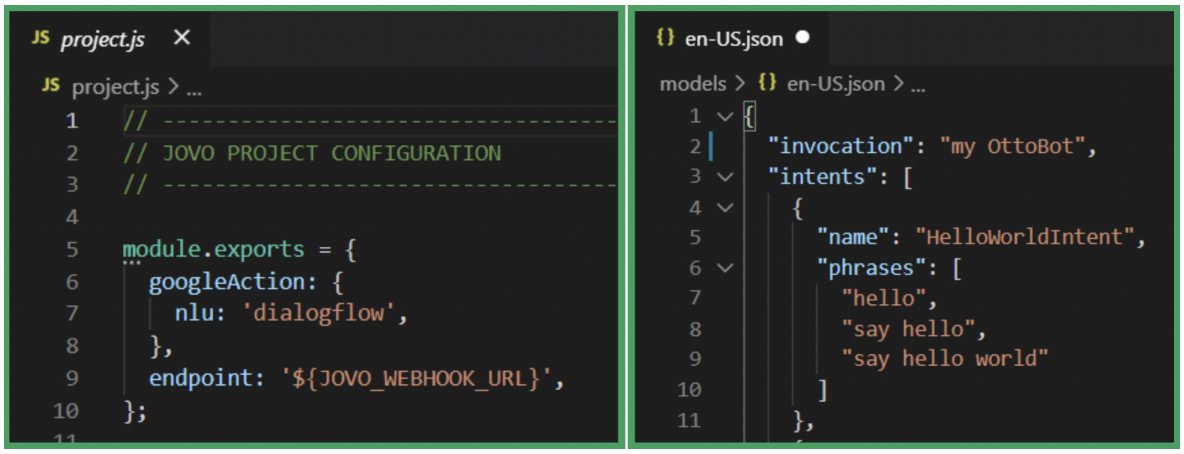


Figure 2.3: a) Project.js b) en-US.json

Models folder **(en-US.json)** is used to create and update platform-specific language models. Instead of having to go to the platform developer consoles individually, the aim is to maintain a single language model (Jovo Language Model) locally. Every language in the models folder gets a file, such as the en-US.json file shown in the above Figure 2.3 (b). There are several elements in Jovo Model as mentioned below:

1. **Invocation** is used by platforms as the name of the app ("my OttoBot") to access it.
2. Similar to Dialogflow, **Intents** can be created which are used to route users through the app, that includes name, phrases and its synonyms.
    - The intent name defines how it is going to be called on the platforms.
    - Phrases, also known as utterances contain an array of example phrases that are used in training the language model.
    - Inputs: Entities in the form of phrases that include specific variable input. The expected user inputs are declared as "values" within "inputTypes" array lists.

For instance, in the example provided below Figure 2.4 (a), we used phrases expected from the user for "ResidencePermitIntent", as well as inputs (rp) that take values such as "lost", "renew", "apply", "issuance" and inputTypes with their possible synonyms as depicted below in Figure 2.4(b). This is similar to the entity types feature in Dialogflow which was explained in the Section 2.1.2

Figure 2.4: en-US.json

The **platforms** folder gets created upon building the project using the "jovo build" command. This helps to bind all the project files into a zip folder which can then be imported onto the Dialogflow platform so that the intents and the models get uploaded. The integration process in Explained in Chapter 3 under Section 3.2

2.2.2 Source Files

Source Files can be found in the src folder and is where the app logic is run and it contains the actual code of the Jovo App. It includes the app.js, config.js, and index.js files.

These two building blocks are connected in an endpoint where the app logic is executed in the code. An interaction pair with a request (incoming data from the platform to the endpoint) and a response (including speech output, visual output, or session data that is sent back to the platform after request) builds each interaction in the jovo application and the endpoint is used by the platform to send these requests and receive responses.

The Jovo CLI is the center of application development within the Jovo Framework using which creation of projects containing language models and deploying them to run the apps locally is easy for prototyping and testing. The reasons for choosing Jovo Platform to build our bot are as follows.

- Jovo is a **Cross-Platform framework** and contains a single code base that works across various platforms like Google Assistant, Alexa, and many more.
- Jovo being an **open source** tool helps us to host it on any cloud platforms and it can also be extended with plugins, integrations, and hooks which makes it more **flexible**.
- Jovo uses an integrated unit testing system to ensure that the applications are **robust**. Also, it can be integrated onto any workflows which also includes staging features.

2.3 Cloud Firestore

Firestore[FIREBASE_DOCS_1] Database is **NoSql database model** hosted from the cloud which can be used for mobile, Web, and Server application developments from Google and Firebase. Nosql Data Model of firestore stores data in the form of Documents consisting of fields supporting many data types mapped to their respective values as shown in the Figure 2.5. Different documents are stored together in a container called collections and each document can contain sub-collections of documents.[FIREBASE_DOCS_3] This collection and document structure allow us to build a hierarchical structure over the data.

The main reasons for choosing the Cloud Firestore database over other database models are as follows :

- Firestore provides **seamless integration** with Google products like Dialogflow, Google Cloud Function.
- Data access only possible by Authentication keys.
- Firestore data can be accessed from **multiple environments** like Node.js, Python, Java, web applications.

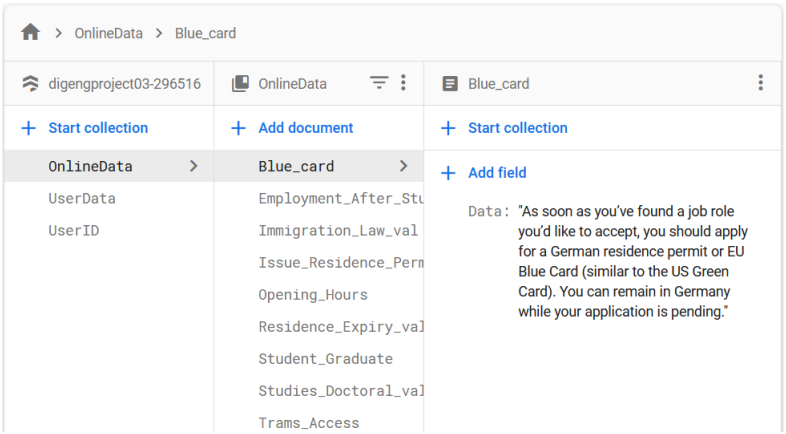


Figure 2.5: Firestore Database[FIREBASE_DOCS_2]

Cloud Firestore is used in both Node.js and Python SDK's files. Since our goal is to build a chatbot for the Foreigners' office, we built a script using Python language to scrape the data from the Ausländerbehörde website which returns all the data in the form of a JSON array. The data is then parsed and the string is then stored in the database which is explained in detail in Section 2.5

2.4 Google Cloud Function

Google Cloud Functions[GOOGLE_DOCS_4] allow us to trigger the code from google cloud or to invoke it directly from any web application, mobile or backend application via HTTP trigger.

With this HTTP trigger URL we use, the webhook functionality of Dialogflow and Cloud function receives a POST request from Dialogflow in the form response to a query request matched by intents with webhook enabled .

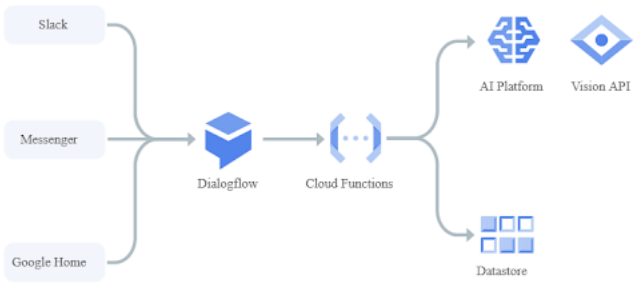


Figure 2.6: Cloud Function[GOOGLE_DOCS_4]

2.5 Python Beautifulsoup

To provide students with required information related to Ausländerbehörde(foreigner's office), we have scrapped the data from Ausländerbehörde Magdeburg official web page and OVGU student portal using **BeautifulSoup**[TAGLIAFERRI] library.

ScrapLoad.py python script hosted on google cloud function has the required code that scrapes data from websites. The scraped data is then loaded onto the Cloud Firestore database. The function is also set up to detect any changes that occur in the website. It then automatically runs the code and updates the data present in Firestore. Apart from BeautifulSoup, we have also used other libraries like **requests** and **firebase admin** to connect to the database.

Requests library which is an HTTP library for Python is used to get the website's HTML data. The below code snippet performs an http request to the given URL and retrieves the HTML data response sent by the server and stores it in the python response object.

```
import requests
from bs4 import BeautifulSoup
url="website's url"
page = requests.get(url)
soup = BeautifulSoup(page.content, 'html.parser')
# Get the url and parse it using BeautifulSoup
def get_soup(url):
    page = requests.get(url)
    soup = BeautifulSoup(page.content, 'html.parser')
    return soup
```

Figure 2.7: Scraping using Requests and BeautifulSoup packages

The HTML response is parsed using BeautifulSoup library to pick up relevant data. BeautifulSoup object created as shown in Figure 2.7, takes the HTML content as input and instantiated object can be used with appropriate methods to pull the intended data.

The function **get_soup()** takes the URL of the website as input and returns the BeautifulSoup object as output.this object is then used with the **parser methods** (find(),findNext(),..etc) to pull the data from HTML response and is stored in a variable or an array.

2.6 Slack

Slack is our platform of choice for communication since it is commonly used for organizational level communications around the world. Also, it has excellent support for adding bots through its APIs. A chatbot workspace and a slack application for the bot was first created. Integration of the dialogflow with slack was done on the agent by enabling the 'integrations' button, choosing slack as the platform and picking the bot that was created for testing.

The reason for choosing slack was, it has gained huge popularity and has been used by several organizations. Also, Dialogflow's Slack Integration makes it easy to create bots and to train them.

3 Integration

This chapter explains the integrations of all tools and frameworks used in the Chabot development that were discussed in the previous chapter. Along with the steps required for integration we have also included some parts of our code to make it clearer and more unambiguous.

3.1 Google Cloud Function - Jovo code

As a connective layer of logic,[GOOGLE_DOCS_5] Google Cloud Function was integrated with Jovo code and a function was created which was used as a trigger. The function was created by following the steps:

1. Execute the "npm run bundle" command in the Jovo console which will create a bundle.zip file containing all the src files.
2. A Google Cloud function should be created with the below-mentioned settings and the bundle.zip is uploaded onto the cloud function.
    - Trigger type as **HTTP** and Authentication option as **Required authentication.**
    - Choose the runtime environment as **Node.js**.
    - Jovo source code(index.js) has the the below settings and hence the entry point in the Cloud Function is set as **handler**

```
exports.handler = async (req, res) => {
  await app.handle(new GoogleCloudFunction(req, res));
};
```

Figure 3.1: Entry point in the index.js file (Jovo)

3.2 Dialogflow-Jovo

Jovo code and Dialogflow were integrated by a webhook[JOVO_DOCS_2]. Jovo being hosted by the Google Cloud function will receive a POST request from Dialogflow. This request is received in response to a user query matched by intents wherein the webhook is enabled.

To be able to use to the intents and entities created in Jovo on the Dialogflow Platform, also without having to create them explicitly in Dialogflow the code was imported using the below steps:

- The command **"jovo build"** is executed in the Jovo console which creates/updates platform-specific integration models using the Jovo model. It uses the files in the /models folder and converts them into files in the /platform folder.
- Now, the command **"jovo deploy -p googleAction"** is executed to create a platform (Dialogflow) specific zip file (dialogflow_agent.zip) with all the information regarding Intents and Entities created.
- On the Dialogflow platform, the **dialogflow_agent.zip** file can be uploaded using the Restore from ZIP/Import from ZIP option.

Once the file was uploaded, all the Jovo intents along with entities were available on the Dialogflow Platform.

3.3 Jovo- Firestore

The Private key file was generated via the Firestore console and was included in the Config.js file[JOVO_DOCS_3] in Jovo as shown in the Figure 3.2

```
db: {
  Firestore: {
    credential: require('./firebase_connection_file.json'),
    databaseURL:'https://digengproject03-296516-default-rtdb.europe-west1.firebasedatabase.app/'
  }
},
```

Figure 3.2: Firestore Database configuration in the config.js file

3.4 Python-Firestore

After scraping the data the variable that contains the data needs to be stored onto the Cloud Firestore for which the firebase authentication needs to be done. **firebase_admin**[FIREBASE_DOCS_4] library is used to access the firestore database.

The credentials library is used to connect to Firestore using the keys file. The app is then initialized with a service account and the admin privileges are granted.

```
# connection to firestore Database
try:
  app = firebase_admin.get_app()
except ValueError as e:
 # add the credentials
  cred = credentials.Certificate('./firebase_connection_file.json')
  firebase_admin.initialize_app(cred)
db=firestore.client()
```

Figure 3.3: Authentication and DB object

3.4.1 Read from and Write to the database

Once the data is stored, **get()** method is used to retrieve the collection by querying the documents as shown in Figure 3.4

When the data is added to the folder in firestore, it generates collections and records it implicitly. It is not necessary to construct collections or documents explicitly. The write operation **(set() method): trams_access** as shown in Figure 3.4, is the variable having the data and is stored in a field named 'Data' on the Firestore in the form of a key-value pair.

```
# Write to Firestore
        doc_ref=db.collection('collection name').document('document name ')
        doc_ref.set({'Data':trams_access})
# Read from Firestore
        db_trams_access=db.collection('collection name').document('document name').get().to_dict()
        db_trams_access['Data']
# Update function
        if(db_trams_access['Data']!=trams_access) :
            doc_ref=db.collection('UserData').document('TramAccess')
            doc_ref.set({ 'Data':trams_access})
```

Figure 3.4: Read, write and update actions on the database

3.4.2 Update the database upon change in the website

In order to check and update data we have used a simple if statement where we compare the stored and scraped data. If there is a difference in the latest scraped data it will overwrite the data stored in Firestore with the updated information.

3.5 Slack-Dialogflow

In order to integrate the bot, a bot workspace was created on Slack using the instructions provided in the **Link**. Single click integration method was used on the Dialogflow end to integrate the Agent with the bot. The Event Requests URLs generated from Dialogflow were added on the Slack Developer Console.

4

Pipeline

The general information about implemented tools, their application, and integration have been made clear from former chapters. In this chapter, we have included a visual representation of process flow between all tools once they are integrated which can be seen in the below Figure 4.1

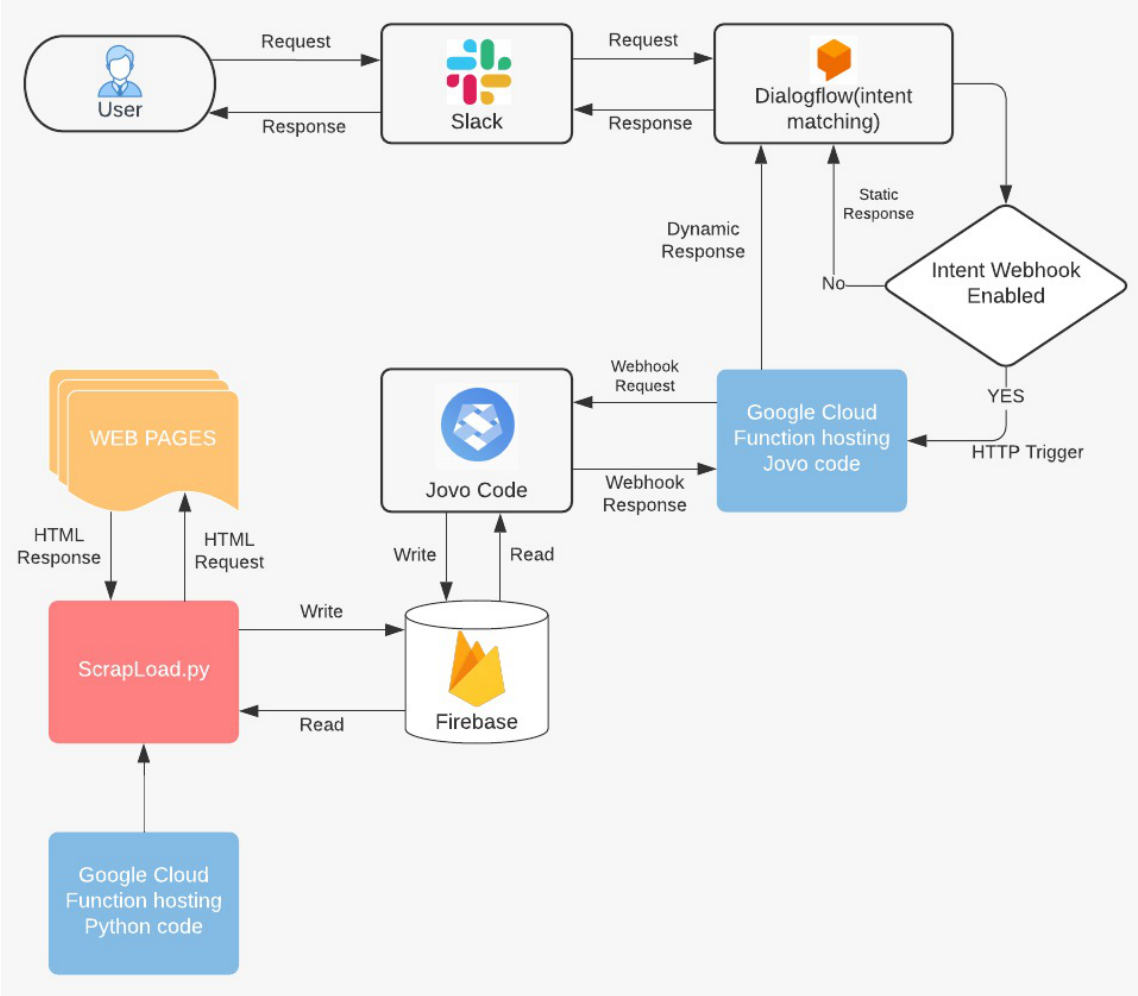


Figure 4.1: Pipeline Flow Diagram

User sends the query request to **Slack** and Slack sends this query request to **Dialogflow**. At Dialogflow the Agent executes the NLP process and tries to match the user's query request

phrase with the phrases of the available Intents. When an Intent is matched with the user's query phrase one of the two actions takes place depending on whether the Intent has **Webhook call** enabled or not . If the Webhook call is not enabled for the matched Intent, **Static response** present at Dialogflow is sent to the user at Slack and if the Webhook is enabled for the matched Intent, then the Dialogflow sends **Webhook Request** message to the Webhook Service(**Google Cloud Function**).This Webhook request message contains information about the matched Intent, the Action, the parameters and the response defined by the intent. **Jovo Framework** code which is running on the Google Cloud Function takes the request information and executes the matched Intent. Depending on the code flow Jovo extracts the required data from **Firestore Database** or writes User related data to the Firestore Database. The matched intent response is sent back to Dialogflow as **Webhook Response** also known as **Dynamic Response** from the Google Cloud Function. Dialogflow sends this dynamic response to the user at Slack. **ScrapLoad.py** which is also running on google cloud function, scrapes the data from the web pages and writes them to the database.

5 Conversation Flow

In this chapter, as the name suggests conversation flow between user and chatbot is argued. That is, how the chatbot communicates with the user, including the greeting phase, and how the user data (Name and Category) that is useful in various cases about the user is stored when the conversation is initiated. Two different scenarios have also been included to provide a more comprehensible understanding.

5.1 User Data collection

When the bot is launched the user initiates the conversation with a 'Hello' and a request is sent from the slackbot. The request is then handled by the Dialogflow agent which executes the natural language processing of the input phrase. The agent tries to match the input phrase with the phrases of the available intents. Once the intent is matched, it then checks if the webhook is enabled for a given intent and performs one of the below mentioned actions.

1. If the webhook is enabled for the matched Intent, the Dialogflow sends a Webhook request message to the webhook Service(Google Cloud Function). This webhook request message contains information about the matched Intent, the Action, the parameters and the response defined by the intent. Jovo Framework code which is running on the Google Cloud Function picks up the request information and executes the matched Intent. Depending on the code flow, Jovo extracts the required data from Firestore Database or writes user related data to the Firestore Database. The matched intent response is sent back to Dialogflow as a webhook response from the Google Cloud Function. Dialogflow outputs this dynamic response to the user at the slackbot end.
2. If the webhook call is not enabled for the matched Intent, a static response present in Dialogflow is sent to the user at the Slackbot end.

When the user types 'hello', it is matched with the 'HelloWorldIntent" in dialogflow. For this intent, the webhook is enabled due to which the handle is passed to jovo to execute the code of the intent as follows:

5.1.1 HelloWorldIntent

The **HelloWorldIntent** uses the **getId()** method which pulls the userID information from the webhook message. This userID is then used by the function **test(userId)** to lookup in the

database. If the user information is already present, it fetches the name and sends the response statement using this.tell() method *'Hello ' + name + '!' + 'Welcome back, How can I help you today?'*. If the user's information is not present, the bot sends the response statement asking the user to enter his/her name using the this.ask() method *"Hello, My name is Otto! I'd like to know a few details about you. Could you please tell me your name?"* and the user is finally routed to the **NameState** using **this.followUpState()** method which further captures the user's name as an input.

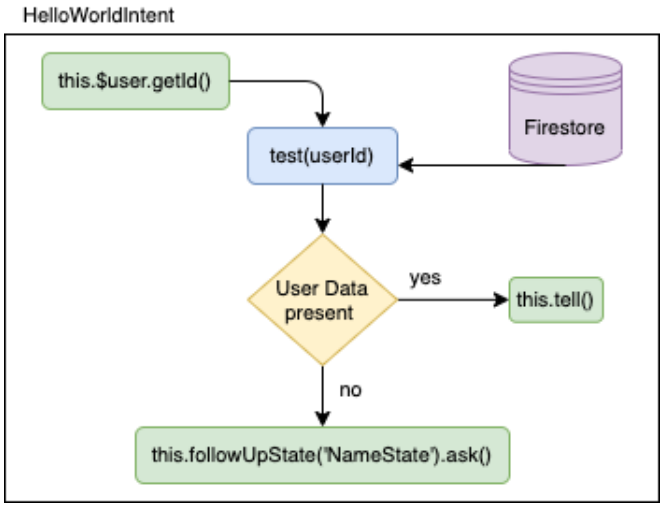


Figure 5.1: Flow Diagram of HelloWorldIntent

The difference between the above mentioned two methods is that this.tell() is used to end the session and is only invoked when the user triggers the intent again. In the case of this.ask() method, the session is kept active and will await for the user's input to activate the respective intent/state.

5.1.2 MyNameIsIntent

The reason for using **state** functionality here was to make sure the bot is able to capture the user's response as an input name and also to make sure it doesn't pick any other arbitrary intent which is available outside the state. When the user enters his name, **MyNameIsIntent** is triggered. This intent captures the input name as a parameter using **this.$inputs.name.value** and also fetches the userID information.

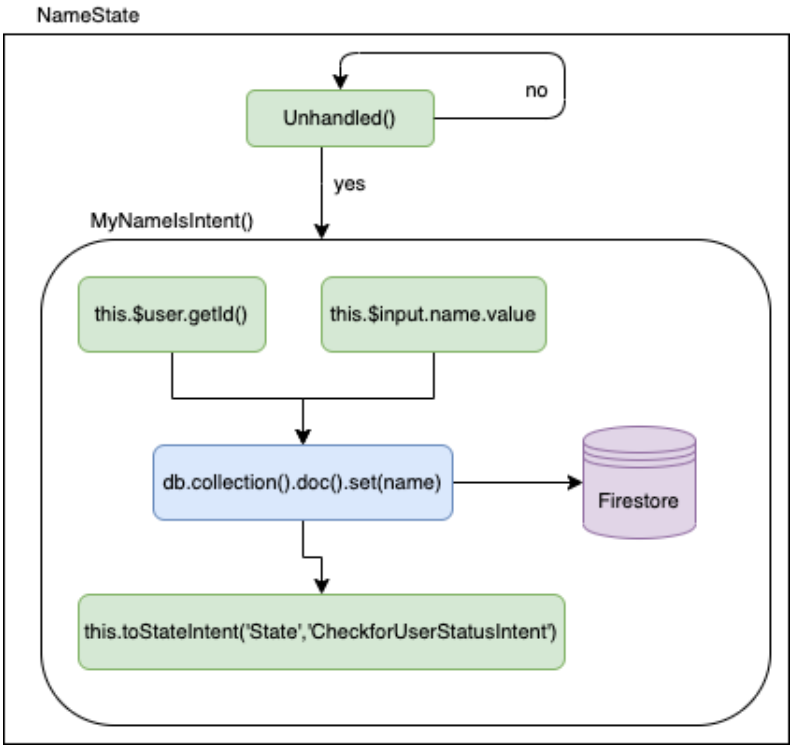


Figure 5.2: Flow Diagram of MyNameIsIntent

The userID value and the input name is stored onto the firestore database using the function **db.collection('UserData').doc(userId).set(name)**. In the firestore database, it creates a collection named 'UserData' and a document named 'userId' which stores the respective user 'name'. Since name is one of the mandatory parameters, we have also used a custom **Unhandled** intent in this state to catch the request and remain in the current state to make sure the chatbot does not pick any other intent outside the state. Without the unhandled Intent, the system would jump out and look for other intents if it matches with the user input.

Once the user data is stored, the user is finally routed to **CheckforUserStatusIntent** which is present inside a **State** using **this.toStateIntent()** method. It then prompts the user to choose a category he/she belongs to.

5.1.3 CheckforUserStatusIntent

The intent was created inside a State since the user's category was one of the mandatory parameters and to make sure the intent session doesn't terminate without the parameter.

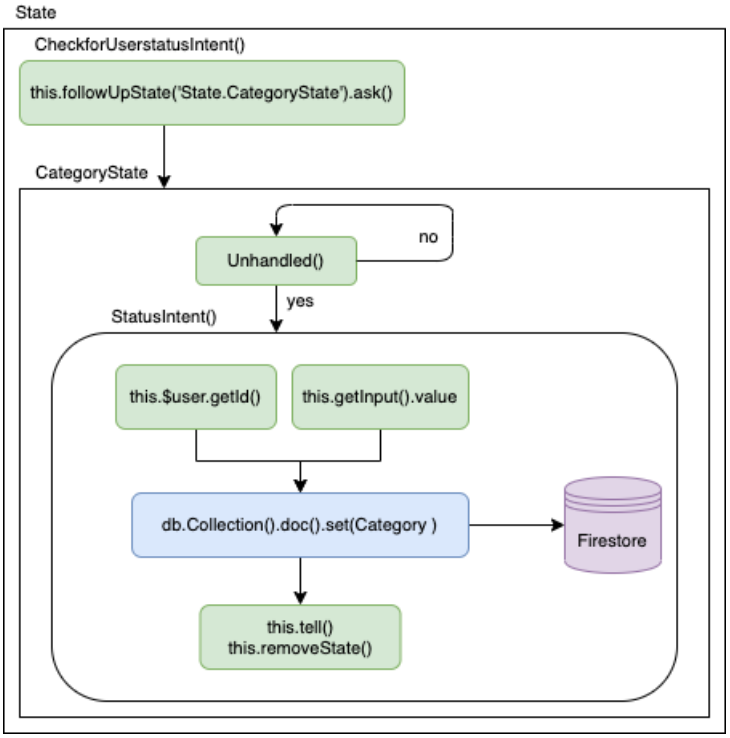


Figure 5.3: Flow diagram of UserStatusIntent having nested states

Once the handle is passed to **CheckforUserStatusIntent()**, it uses this.ask() method and prints the statement *Could you please choose category that you belong to: a) EU Student, b) EU Graduate, c) Non-EU Student, d) Non-EU Graduate.* To capture the category which will be input by the user, the handle is passed to **StatusIntent** which captures the category information using the **this.getInput().value** method and also the user id information using **this.$user.getId()** method. The userID value and the category is stored onto the firestore database using the function **db.collection('UserData').doc(userId).set(Category)**. In the firestore database, since the collection is already present it creates a field called 'Category' which stores the information. Once the action is performed it uses **this.tell()** method and prints *Thanks for providing us the information. Let me know what I can do for you?* and exits the session.

The intent also uses **this.removeState()** method to move outside to any other global intents for the next conversation. The bot now has all the mandatory information needed and the user will be free to ask any questions of his/her interest.

5.2 Residence Permit Intent

Residence permit was one of the main and prominent topics covered among all the other bot functionalities. During development, there were a couple of scenarios which were to be considered while querying about residence permit. All the possible scenarios handled by the bot are explained below with the flow diagram in Figure 5.4.

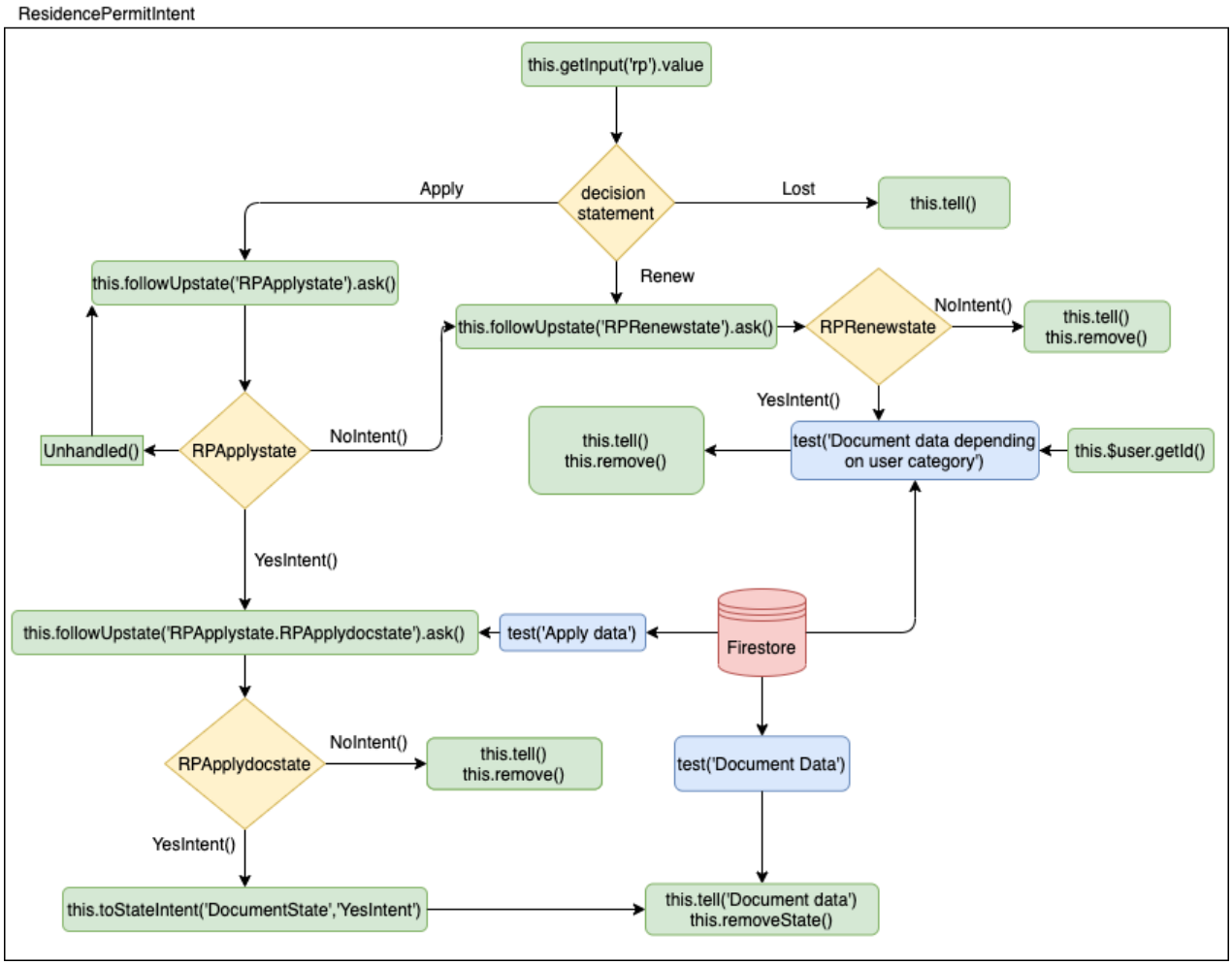


Figure 5.4: Flow diagram of ResidencePermitIntent

Whenever the user asks a question related to the residence permit **ResidencePermitIntent** is triggered. Depending on the keyword(rp value) present in the question respective action takes place as per the logic in the ResidencePermitIntent Intent.

Action depending on Phrase		
Phrases	**rp value**	**Action**
"residence","permit"	null	User is prompted to reply if he wants to apply or renew his residence permit.
"residence","permit","apply"	"apply"	1
"residence","permit","renew"	"renew"	2
"residence","permit","lost"	"lost"	3

Actions 1, 2 and 3 are defined below

1. Since the rp value is **apply**, this.followUpState('RPApplystate').ask() is triggered and the user will get a dynamic response as *Is this your first-time issuance?* and control goes to

'RPApplystate' state. If the user responds to the question with a **"No"** action 2 will take place and if the user responds as **"Yes"**, then the YesIntent() of **'RPApplystate'** will be triggered which will execute this.followUpState('RPApplystate.RPApplydocstate').ask(). The function **test('Apply Data')** pulls all the information related to the application process of residence permit from the firestore and is delivered to the user in the form of a response and the control is then passed to 'RPApplystate.RPApplydocstate' where the user is asked if he would like to know the documents required for the application.

(a) If the user responds **"Yes"** then the YesIntent() of 'RPApplydocstate' will be triggered which will execute this.toStateIntent ('DocumentState', 'YesIntent') where **YesIntent()** of **'DocumentState'** state is triggered which uses the function **test('Document Data')** to pull the information from the firestore database and the user gets the needed information in the form of dynamic response through **this.tell('Document Data')**.

(b) If the user responds with a **"No"**, then NoIntent() of RPApplydocstate is triggered which responds the user with *Alright, Please type your query for any further assistance or type bye to end the conversation* using the function this.tell().

2. If the rp value is **renew**, this.followUpState('RPRenewstate').ask() is triggered and the user will get information about the renewal process of residence permit along with a question if he/she wants to know the documents required for the renewal process?

   (a) If the user responds **"Yes"**, then the YesIntent() of 'RPRenewstate' is triggered. Here the test() function uses the **this.$user.getId()** statement to check the user category (if student/graduate) and pulls the category-specific data from Firestore and returns the data in the form of response to the user.

   (b) If the user responds with a **"No"**, then NoIntent() of 'RPRenewstate' is triggered which responds the user with *Alright, Please type your query for any further assistance or type bye to end the conversation* using the function this.tell().

3. If the rp value is **Lost**, the user will get information on what to do if the residence permit is lost through this.tell().

5.3 Conversation Scenarios

The following section includes scenarios having user personas and explains how the bot handles the conversations for all the topics related to Ausländerbehörde. When any user initiates the conversation for the first time, the bot requests the user to enter the 'username' and 'category', stores the data as mentioned in the Section 5.1. Following are the possible conversations the bot is able to handle via Dialogflow and JOVO.

5.3.1 Intents handled by Dialogflow

Intents handled via Dialogflow		
Phrase input by the user	Intent picked after matching the phrase	Output response
Can I contact the office via phone?	ContactIntent	Custom Payload Response
When can I go to the office?	OfficeHoursIntent	Static Response
How do I reach the office location?	TramIntent	Custom Payload Response
Where is the office located?	LocationIntent	Static Response
How much tax do I pay?	TaxIntent	Static Response

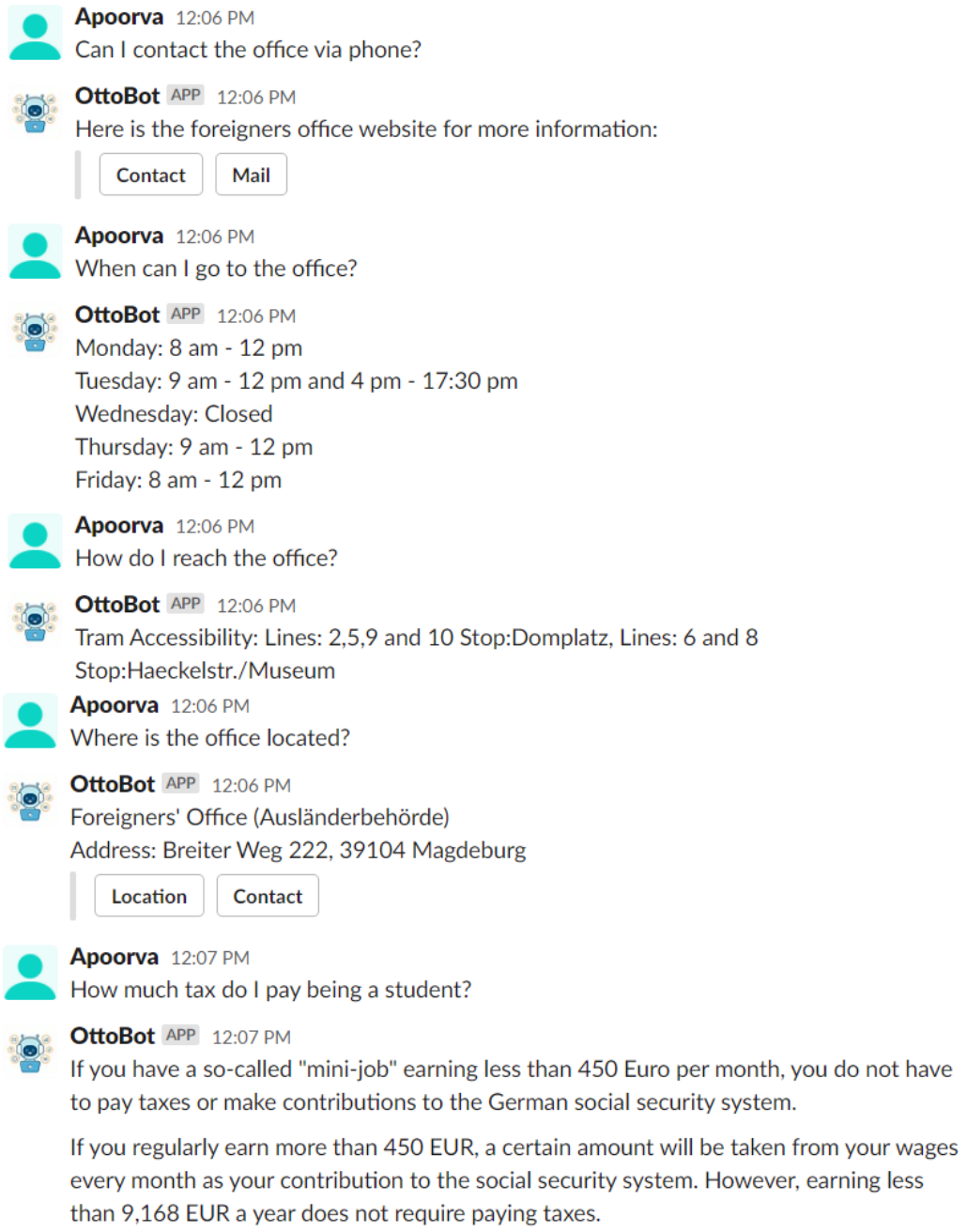


Figure 5.5: Conversation with the bot querying about generic information of Ausländerbehörde.

5.3.2 Scenario 1- Work Permit

Ahmed is a Non EU student who is studying in his 2nd semester DE course at OVGU. He would like to work alongside studies and wants to know if it's possible. Ahmed enters the input phrase *Can I work as a student?*. The bot matches the phrase with the intents in Dialogflow and picks the **WorkPermitIntent**. Since the webhook is enabled for this intent, the handle is passed to Jovo to execute the code. It checks for the user's category that is stored in the Firestore database. Ahmed is a Non EU Student, hence it pulls the data accordingly from the database and the response is sent back to the user. The conversation with the bot is as shown in the below Figure 5.6.

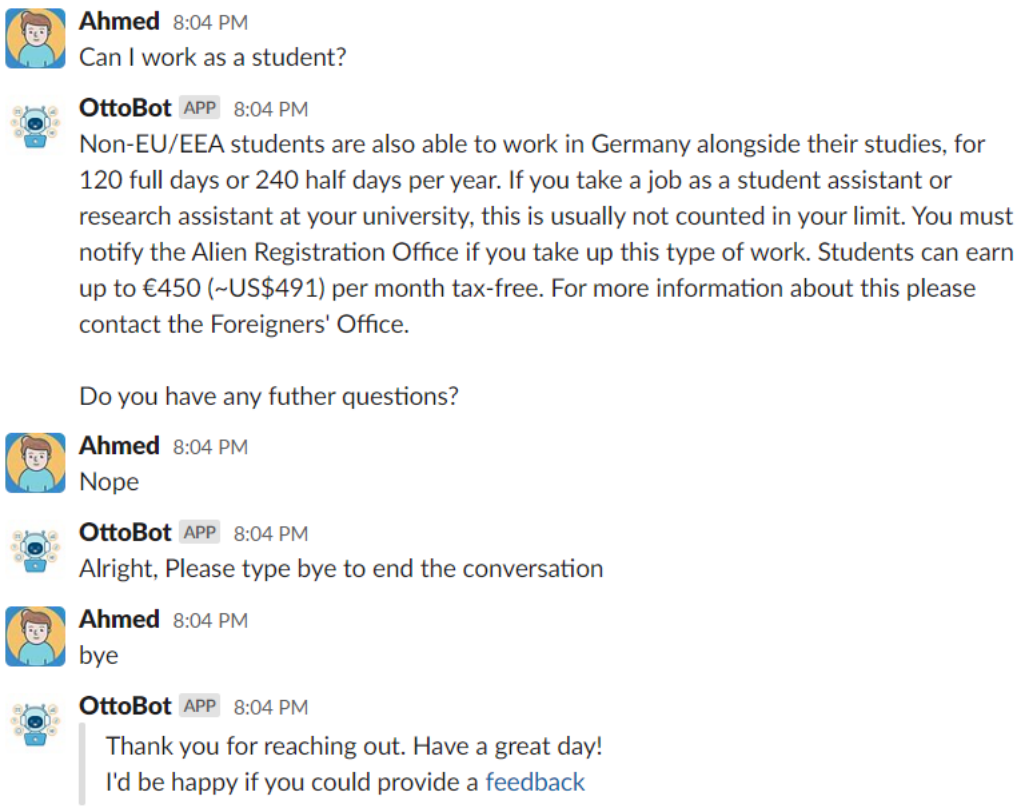


Figure 5.6: Conversation with the bot querying about work permit information

5.3.3 Scenario 2 - Request for application form

Ela is a Non EU Graduate who has finished her Masters Degree from OVGU. Ela would like to apply for a bluecard and needs to know the application form for the same. Ela enters the input phrase *I would like to get the application form to apply for a bluecard.* The bot matches the phrase with the intents in Dialogflow and picks the **LinksIntent**. Since the webhook is disabled for this intent, the response is sent back from Dialogflow itself. A custom payload which contains a visual output functionality was designed to be sent as a response to the user on the Slack end. The custom payload contains buttons with links, gifs and images which was created using the **Block bit builder**[SLACK_DOCS] UI framework. The conversation along with the visual output is as shown in the below Figure 5.7.

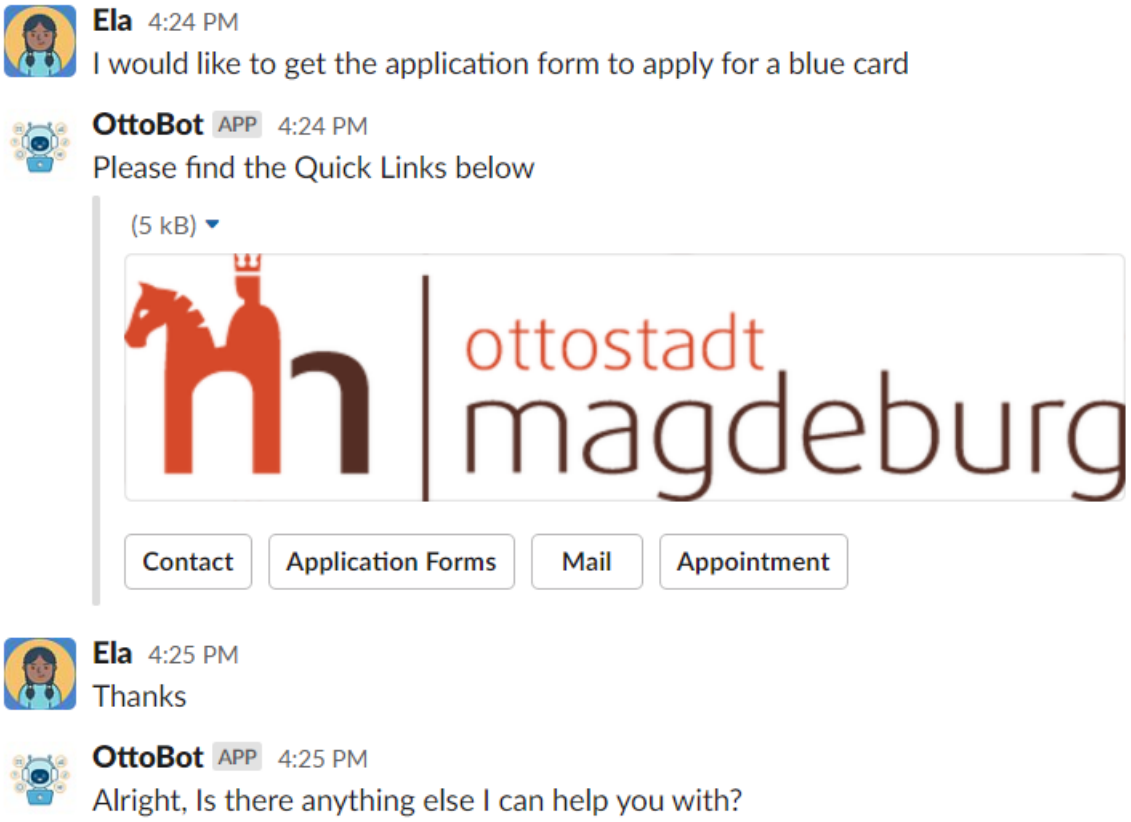


Figure 5.7: Conversation with the bot querying about application forms

5.3.4 Scenario 3 - Documents required for visa extension and appointment details

Priyanka is a Non EU Student who is studying in 1st Semester at OVGU. She wants to know the documents required for visa extension and also wants to know how to take an appointment. Priyanka enters the input phrase *I would like to know the documents required for visa extension.* The bot matches the phrase with the intents in Dialogflow and picks the **DocsIntent**. Since the webhook is enabled for this intent, the handle is passed to Jovo to execute the code. It checks for the user's category that is stored in the Firestore database. Priyanka is a Non EU Student, hence it pulls the data accordingly from the database and the response is sent back to the user. Priyanka also wants to know how to take an appointment. She enters the phrase *Please tell me how do I take an appointment*. The bot matches the phrase with the intents in Dialogflow and picks the **AppointmentIntent**. A custom payload designed in the Dialogflow is sent to the user as a response to the query. The conversation along with the visual output is as shown in the below Figure 5.8.

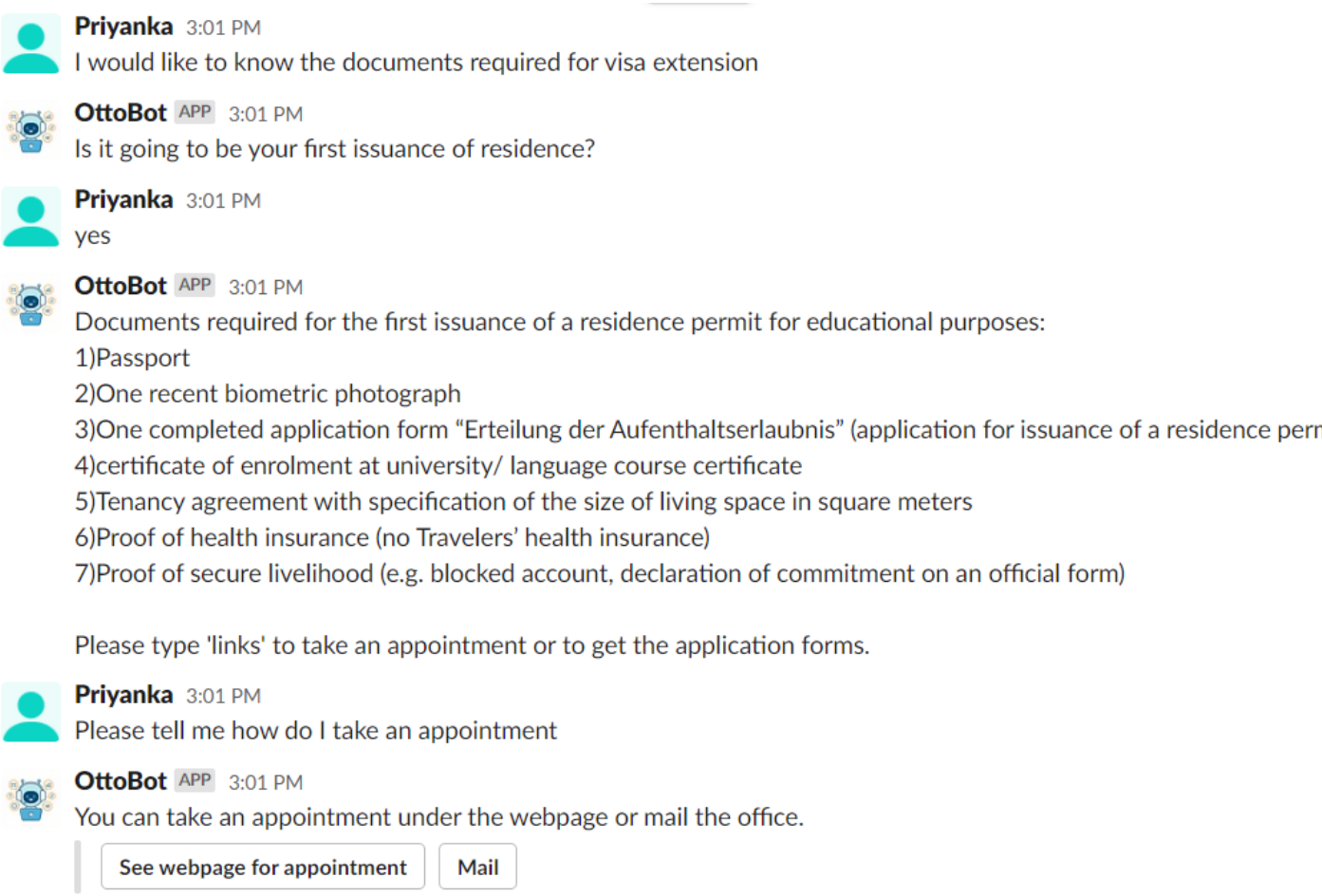


Figure 5.8: Conversation with the bot querying about documents required and appointment link

5.3.5 Scenario 4 - Immigration rules and chatbot FAQs

Anna is a Non EU Student who is willing to come to Germany for her Masters. She would like to know the immigration rules. Anna enters the input phrase *What are the immigration rules while travelling to Germany?* The bot matches the phrase with the intents in Dialogflow and picks the **ImmigrationIntent**. Since the webhook is enabled for this intent, the handle is passed to Jovo to execute the code. It checks for the user's category that is stored in the Firestore database. Anna is a Non EU Student, hence it pulls the data accordingly from the database and the response is sent back to the user. The bot also asks the user if she has any further questions. Anna would like to know what are the bot functionalities. She enters the phrase *Yes, sure. How can you help me?*. The bot matches the phrase with the intents in Dialogflow and picks the **FAQIntent**. A custom payload designed in the Dialogflow is sent to the user as a response to the query. The conversation along with the visual output is as shown in the below Figure 5.9.

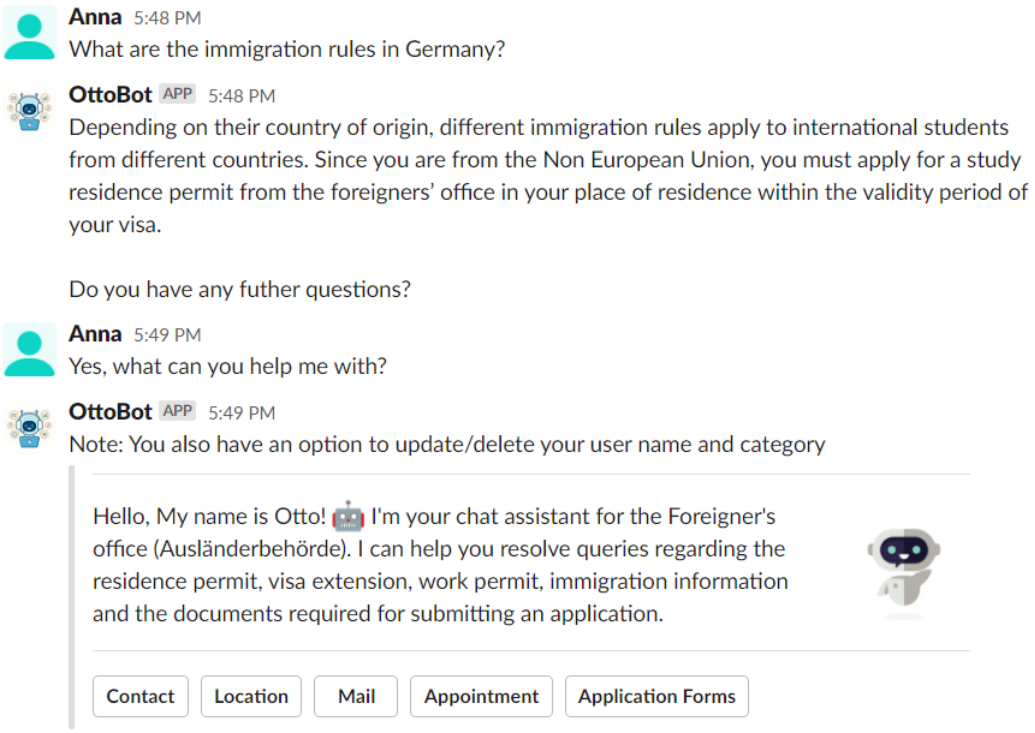


Figure 5.9: Conversation with the bot querying about Immigration rules and Bot FAQs

5.3.6 Scenario 5 - Delete User data

Anna is an EU Graduate who conversed with the bot and got all the information required. She would like to delete her information that was provided to the bot when she initiated the conversation (user name and category). Anna enters the input phrase *Can you please delete my data?* The bot matches the phrase with the intents in Dialogflow and picks the **DeleteIntent**. Since the webhook is enabled for this intent, the handle is passed to Jovo to execute the code. It prompts the user with *Are you sure you would like to delete the user data that is stored?* to confirm if the user wants to delete it. Upon confirmation the bot deletes the data that is stored in the Firestore database. The conversation is as shown in the below Figure 5.10.

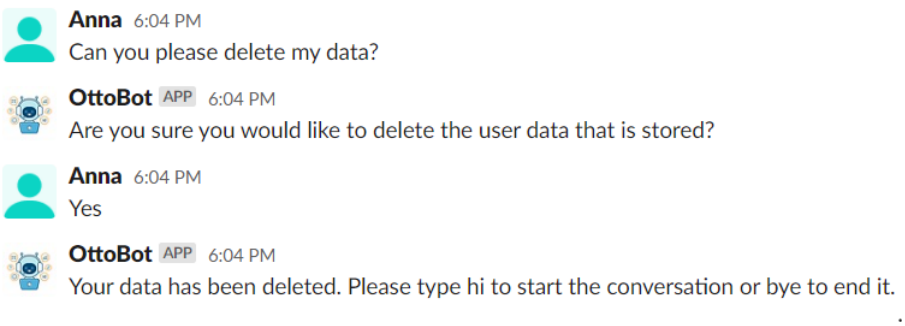


Figure 5.10: Conversation with the bot to delete user data

6

Testing and Evaluation

During the development of our chatbot, we also implemented a testing process for quality assurance. Testing is the process of evaluating an entire system or its components separately in order to determine if it meets the required specifications and to identify software bugs before deploying it to production. Testing was split into two commonly used predominant test functionalities called Whitebox testing and Blackbox testing, details of which will further be explained in this chapter

6.1 Whitebox Testing

It is one of the software testing techniques that test the internal structure of the code and verifies the input/output flow in order to improvise the design. In order to test the functionality of our chatbot, we used **Botium Box**[BOTIUM_DOCS] to conduct the performance testing along with quality assurance and modified the bot to get the best results. Botium allows to run defined conversations that consist of user inputs, expected bot responses, and also to work with variables by utilizing Scripting Memory. We also used the integration feature of Botium with DialogFlow for better performance and quicker results.

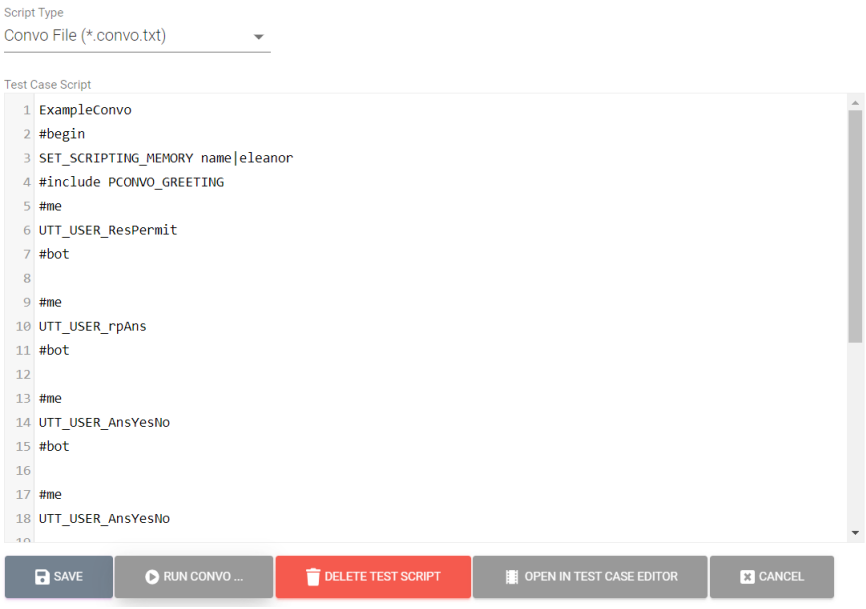


Figure 6.1: Convo file containing the script of the conversation

Conversations can be created as text files, also known as **BotiumScript**, including different types, such as Convos, Partial Convos, Utterances, and Scripting Memory. Convo files were created to show the conversation flow by inserting phrases (or) questions from the user and testing how the bot will respond. For example, the above Figure 6.1 shows the greeting between the user and the chatbot which is scripted in a convo file

Here, we took an advantage of **reusable Partial Convo** files that we can use in both convo files and utterances. The most efficient way to use Pconvo files would be for the start and end of the conversation so that they can be used several times as a part of different scenarios in convo files.

There can also be alternative methods of phrases, messages, or questions expected by the user, it would be useful to separate conversation structure from conversation content. Such separation can be done with **Utterances**.

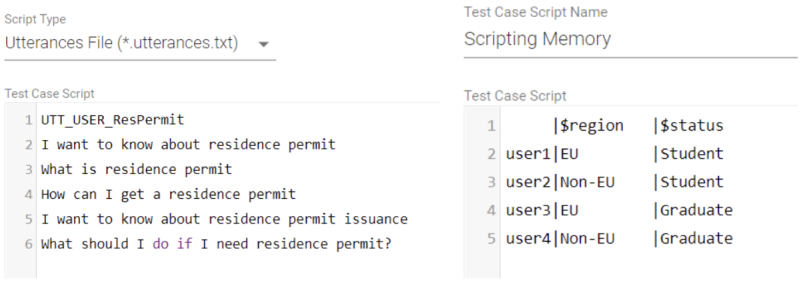


Figure 6.2: Utterances File and Scripting Memory

To make the testing process more dynamic, we have also used **Scripting memory** as shown in Figure 6.2 that allows us to reuse later the variables that are pushed to memory. Figure 6.1 shows us a part of the convo file that includes Utterances, Scripting Memory files. The result obtained once the script is run, is shown in Figure 6.3

Figure 6.3: Conversation result post running the script

Botium also gives a chance to directly integrate with Dialogflow and check automatically. The intents on the Dialogflow are uploaded onto Botium and test cases are built and run for all possibilities of input phrases. The test results are obtained for all the test cases which are scripted in the convo files as shown in Figure 6.4

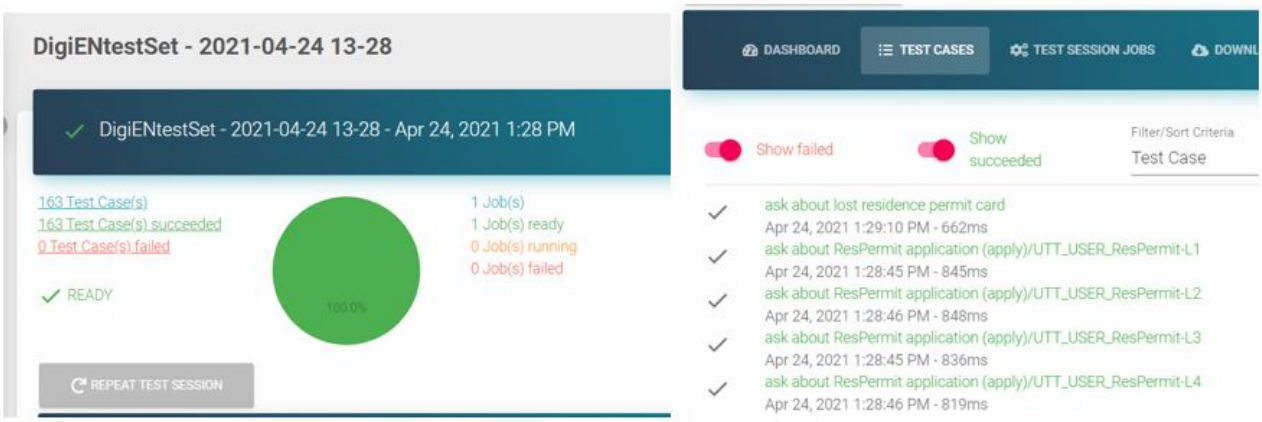


Figure 6.4: Test results of all the convo files

The test results are demonstrated including all succeeded and failed cases as shown in Figure 6.5. While testing the performance of the chatbot, we also encountered a few failed test cases which demonstrated that, either the answer of the chatbot was different from expected or the bot was not responding at all. This was because of some phrases that were input by the user confused the chatbot and was creating conflict among several intents. Using the results obtained from these failed cases, the input phrases of the intents were modified and adjusted to get the correct results. This way Botium was helpful to tune the input phrases of the bot during development phase, so as to capture all the possible phrases to invoke a given intent.

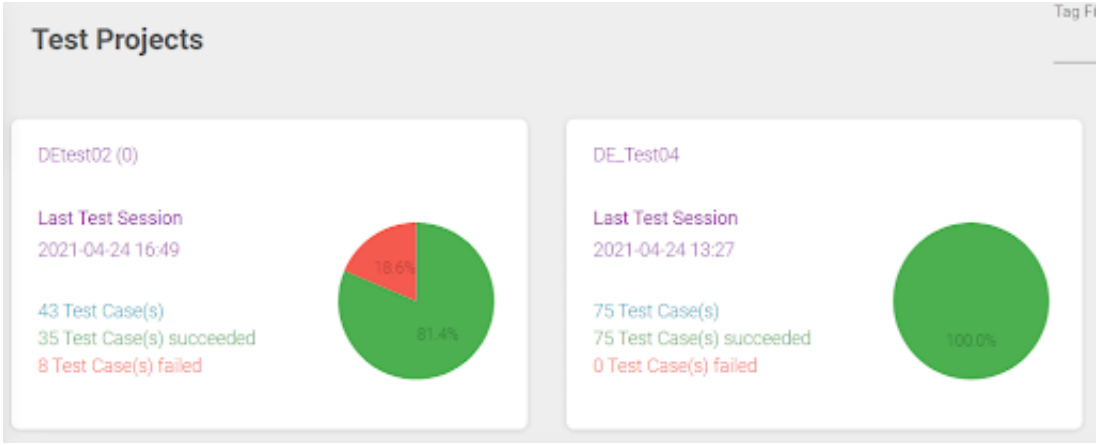


Figure 6.5: Test results of all the test sets created

6.2 Blackbox Testing

This software testing technique is used to check the functionality of an application without going too deep into the internal structure of code or program. For the users, it is not typical to see or read the code while using a software application. The users can be unaware of how the software is working, however they can determine what the application is capable of and what it does.

A questionnaire (google form) was created before developing the bot which helped us to know the questions users might expect the chabot to answer. It gathered the following information from the users:

- Which semester are you in?
- What are the problems you faced when you came to Magdeburg for the first time?
- Were there any questions for which you did not find the answer on the Foreigner's Office website? If so, please mention.
- Do you think the chatbot will be helpful?
- If yes, what are the features you would suggest to improvise the chatbot?

This helped us to know the potential queries users might ask the bot and helped us focus on them during the development of the chatbot.

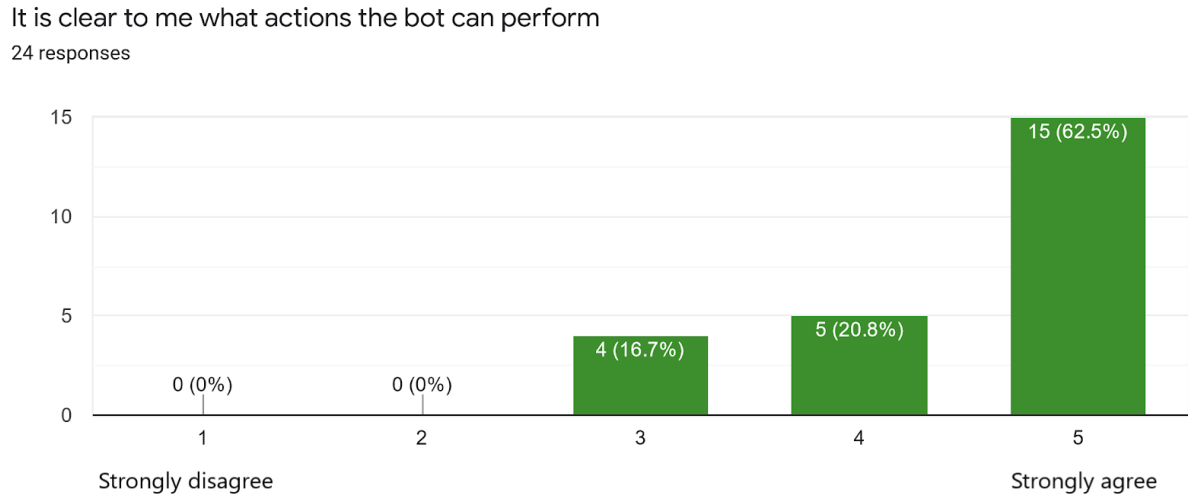
Post development of the bot, in order to examine whether our chatbot understands the user questions and responds properly, we decided to prepare a questionnaire(google form) and directly apply it on international students (users). Since our target user group is international students, it came handy with this method to determine whether the conversation steps and responses were relevant. To get feedback from the real potential users – from international students would definitely provide us a chance to evaluate the quality of our application and to improve it accordingly by analyzing the user answers from questionnaires.

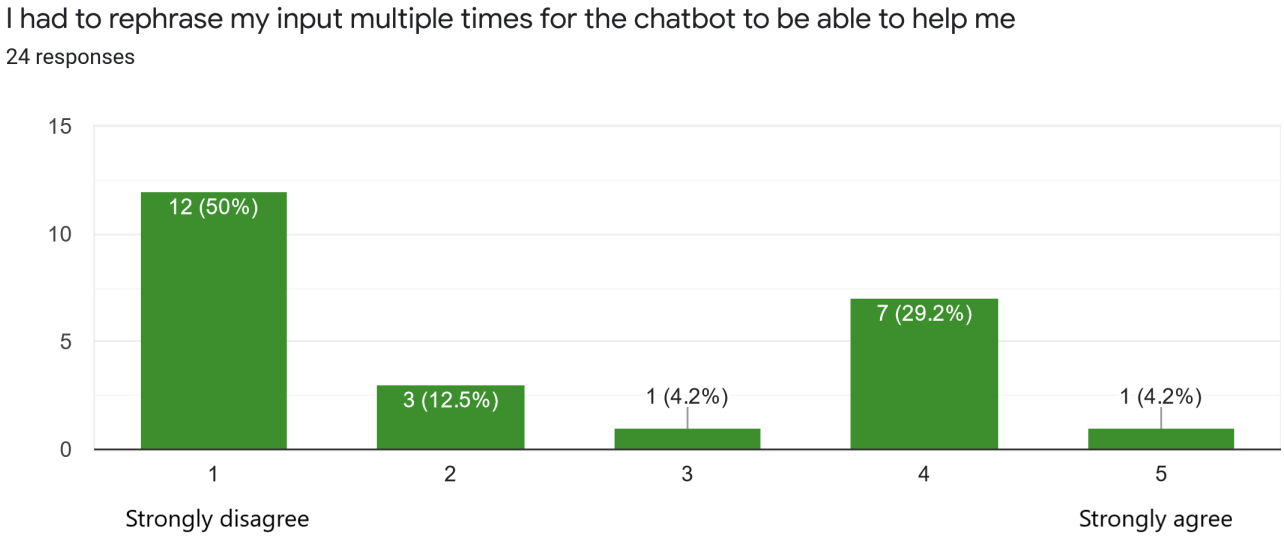
6.2.1 Evaluation

This evaluation method in the form of a Questionnaire contains quality attributes on a 1-5 Likert scale which defines a final score that reflects the Quality of Experience (QoE) of a chatbot. The Questionnaire was divided into three categories - functionality, content and response quality, and overall user satisfaction. 24 participants took the survey after using the bot which includes students as well as graduates.

The first section of the questionnaire was the functionality evaluation. This section helped us to understand if the users were able to understand the features of the bot and all the functionalities it offers. It also helped us to understand if the bot was able to understand and process the user queries. The collected usage statistics are an indication of how well the chatbot performs. About 62.5% of the users agreed that they were able to understand what functionalities the bot performs. However, there were a few users who partially agreed to it. Also, we found

that there were a couple of phrases that the bot was not able to understand hence the user had to rephrase the input multiple times. These were identified and fixed post evaluation.

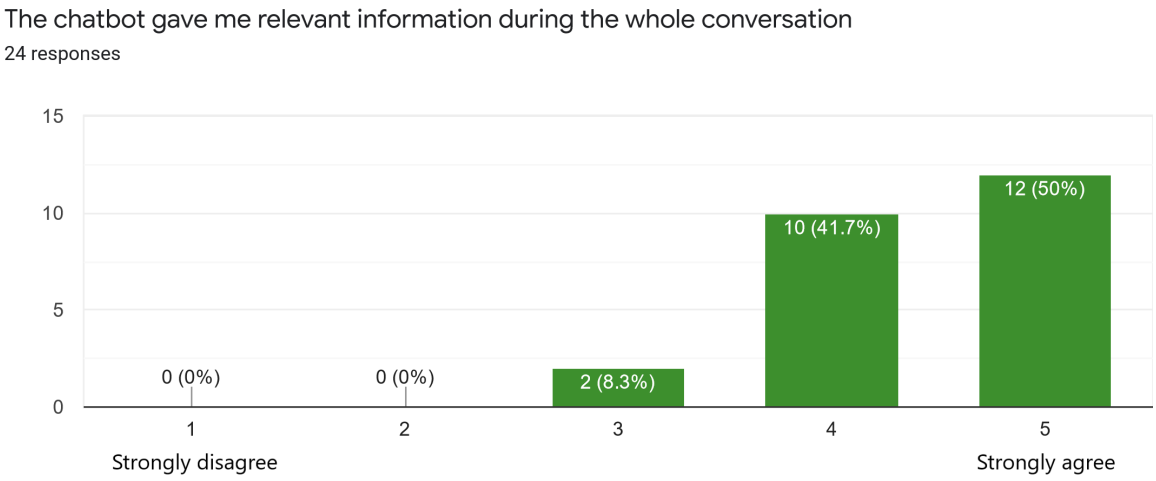


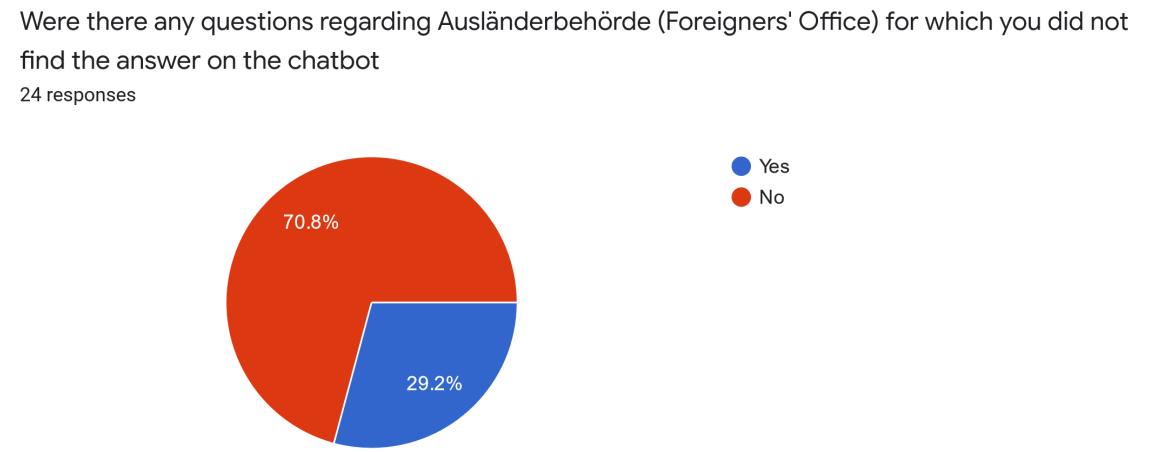


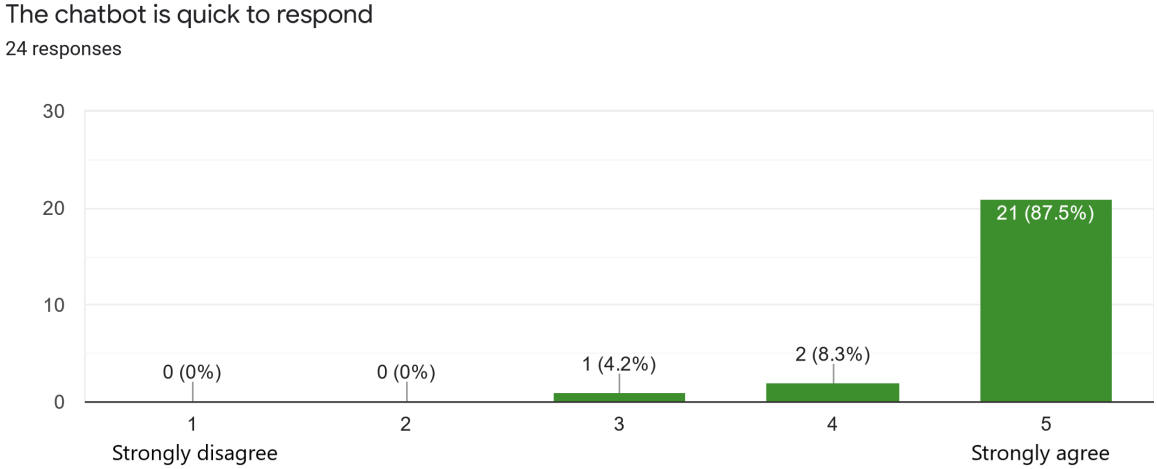
The second section of the questionnaire was the content and response quality. This helped us to understand if we covered all the possible questions the users might ask the bot and provide the relevant information in the form of a response. The collected usage statistics are an indication that the bot gave relevant information to the users and how quick the response is. About 29.2% of the users said that they did not find answers for some of the questions which were as mentioned below:

- Will the visa office be open on public holidays?
- The bot was not providing information on family reunion.
- No info about blocked account/academic pause/sign out from city move to another city.

Some of these topics were taken into consideration and programmed post evaluation. However, a few of them have been considered for future work since the bot mainly focused on queries concerned with Ausländerbehörde







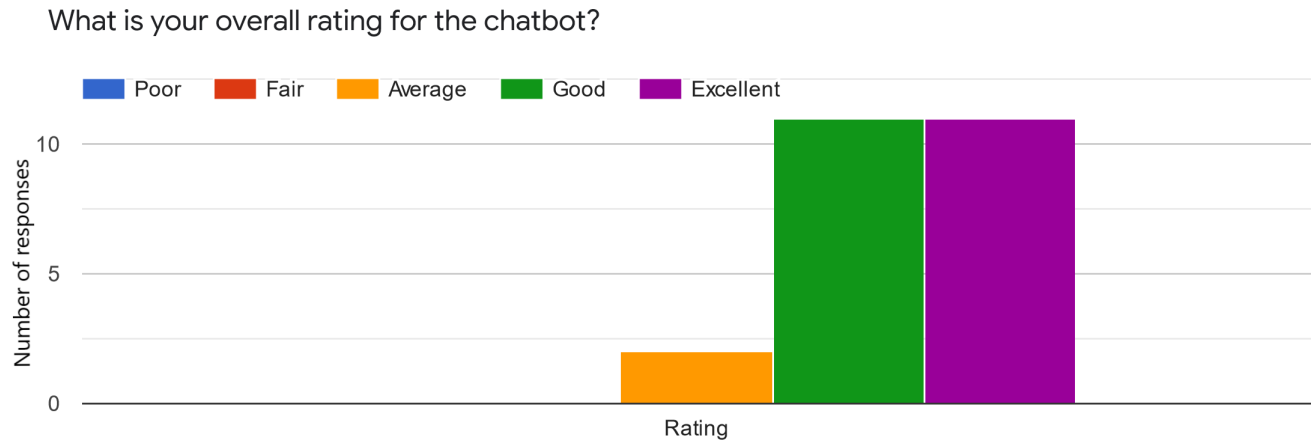
The third section of the questionnaire was the overall rating and feedback for the application. We also asked the users to choose other platforms on which they might find the bot useful. As per the statistics, 98% of the users said that the bot was functioning Good/Excellent and we got a few positive feedbacks from the users. However, 2% of the users said that the bot performed on average where there is room for improvement. Also when asked the users what other platforms would they want the bot on, 87.5% of users suggested the bot to be implemented on Whatsapp application as it is more convenient. To sum up, some of the overall feedbacks provided by the users are as follows.

Positive feedbacks:

- Chatbot is very clean looking. Actions of the bot and required actions are easy to understand. For the "contact" of the Ausländerbehörde, the bot itself could even provide the required information when clicking the button, as its trustworthy and clean design is even more intuitive than the website itself (which loads in german language, at least in my case)
- Data privacy is well handled, I was able to delete my details collected by chatbot
- It is very useful for new students like me especially during this pandemic time
- There was no need for me to search for questions online as the bot was able to get answers for me instantly. Good work!

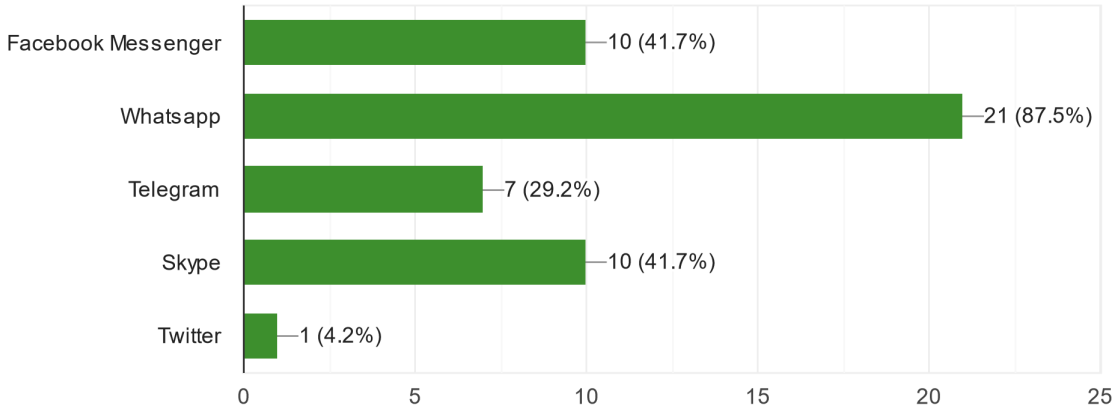
Feedbacks suggesting scope for improvement

- Chatbot is not able to understand the question sometimes and the same answers are being given by the chatbot.
- Overall chatbot performed good. Maybe in future it can cover more topics like working hours of student, and allow user to directly download documents instead of going to the websites, etc.



Which other platforms would you like the bot to be available on?

24 responses



7 Issues and fixes

Building an interactive and intelligent chatbot did not come up without challenges. However, through motivation and co-operation, we have fixed the issues and added improvements throughout the development process. This chapter explains those issues and how they were fixed.

7.1 Db object creation

Initially, the default firebase setup for Jovo allowed the application to only execute implicit read and write operations as shown in Figure 3.2 in section 3. But Jovo didn't have a default option to create a database object for explicit read and write operations. A database object was then created using the below code[JOVO_DOCS_3].

```
const admin = require("firebase-admin");
const serviceAccount = require('./firebase_connection_file.json');
admin.initializeApp({
 credential: admin.credential.cert(serviceAccount),
 databaseURL: 'https://digengproject03-296516-default-rtdb.europe-west1.firebasedatabase.app/'
});
const db = admin.firestore();
```

Figure 7.1: DB Object creation

This database object "db" was used to perform explicit read and write operations as mentioned below.

```
# Read operations
        db.collection(collection).doc(document).get()
# Write Operations
        db.collection(collection).doc(document).set()
        db.collection(collection).doc(document).update()
```

Figure 7.2: DB object operations

7.2 Using await and async functions

As we know that JavaScript is a single thread code executer and can only perform series execution of the code in a synchronous manner. However, we needed asynchronous calling for handling database read and write operations, and hence we chose to use the Async/Await[LINDSTRÖM] process. Creating a function using async keywords makes it asynchronous, which means when the function is called a promise is returned and normal code execution will take place. Await keyword used inside the async function blocks the execution of the Javascript code until awaiting code execution is completed.

```
# Async function using await keyword
async BlueCardIntent() {
    let blue_val = await  db.collection(collection).doc(document).get();
            this.followUpState('ConfirmState').ask(blue_val.Data.Blue_card+' For more information about
                    this please contact the Foreigners\' Office.'+'\n \n Do you have any further questions?'); },
```

Figure 7.3: Await and Async functions

7.3 Name issue

In the Dialogflow Agent, a default entity called **@sys.name** was used to recognize the names in the **MyNameIsIntent**. It was able to recognize only the English names and was not able to recognize any of the regional names.
To resolve the issue, we used an entity called **@sys.any** which can recognize any text entered by the user and will pick it up as a name. A downside of using **@sys.any** was that whenever a user enters any text, **MyNameIsIntent** would get triggered. To avoid this, the intent was included in a state along with an unhandled function.
Now, when a user is asked to enter his name, the call goes to **NameState** which has only one intent - **MyNameIsIntent** in it. Jovo triggers this intent only when the call is inside the NameState and the unhandled function makes sure that the call doesn't leave the state until the user enters his name.

7.4 Button response

After the Slack-Dialogflow integration when we tried to test the button functionality. The button responses from Slack were not getting recognized by the Dialogflow agents. The following configuration was done to be able to capture the response from the Slack.

- In **SlackAPI** select the Interactivity Shortcuts tab and enable Interactivity
- Enter the Request URL from Dialogflow-Slack integration setup at Dialogflow in Slack-API
- Click the Save Changes button.

8 Future work

There are many ongoing research topics on improvising chatbots, and how they could be made virtual assistants. Concerning our project, there are numerous ways to make our chatbot more informative by adding features after user study.

- First off, there are several students concerned about family reunion queries, job-seekers visas, and related documentation process, which currently is not covered by the bot.
- Another feature that could make a huge difference is to inculcate multilingualism in the bot, where it could handle multiple languages. This was one of the feedbacks got post evaluation.
- There is also so much room for improvement when it comes to building a visual bot. A visual bot could be trained on images/videos and it will be able answer the questions based on an image, for example identifying if the picture depicts the foreigners' office of Magdeburg or not.
- Another future work includes one of the features that we tried to implement in our bot - repromting the user with a phrase/question after a certain duration of timeout when there is no response back from the user. This was tried on the Jovo platform using similar concept of speech builder. However, this feature worked only with Google Actions and not on Slack. We tried to reach out to Jovo team however, failed to implement this feature due to middleware issues. [JOVO_FORUM_QUERY]
- Future work could also include taking 'responses' to the next level, by adding button responses or even visual responses. DialogFlow is able to recognise the button response from the Slack platform whereas the current Jovo platform is having middleware issues in recognising these button responses. Enhancements in Jovo fixing this middleware issue will give us functionality to trigger the Intents present at Jovo from button actions. This can make the conversations more visually aesthetic.
- NLP algorithms can be largely leveraged in chatbot functionality to be able handle multilingual queries, wildcard queries (in case there are typos in user's requests), topic modelling, summarisation, sentiment analysis (understand the mood of the user) etc.

9 Conclusion

Chatbots are evolving continuously. The state of art in Natural Language Processing (NLP) is growing rapidly. NLP is directly applicable in designing Chatbots. And that means there is a huge room for improvement when it comes to Chatbots, which are currently limited in their capabilities. Additionally, AI and machine learning algorithms can be implemented in designing the bot which will make it closer to virtual assistants. The ultimate goal behind chatbot development is to have natural and seamless personal conversation with it. While it may take time for research topics to become commercially viable, there are some exciting leaps in a couple of years as the field becomes more accessible for non-programmers.

A

List of Figures

B
Bibliography

[GOOGLE_DOCS_1] **Dialogflow Documents**. link.

[GOOGLE_DOCS_2] **Agent detection**. link.

[GOOGLE_DOCS_3] **Webhook calling**. link.

[JOVO_DOCS_1] **Jovo Documents**. link.

[FIREBASE_DOCS_1] **Firestore Documents**. link.

[FIREBASE_DOCS_3] **Data structure**. link.

[FIREBASE_DOCS_2] **Diagram Reference**. link.

[GOOGLE_DOCS_4] **Cloud function Documents**. link.

[TAGLIAFERRI] **BeautifulSoup and requests**. link.

[GOOGLE_DOCS_5] **Cloud Function and Jovo Integration**. link.

[JOVO_DOCS_2] **Dialogflow and Jovo Integration**. link.

[JOVO_DOCS_3] **Firestore and Jovo Integration**. link.

[FIREBASE_DOCS_4] **Firestore and Python Integration**. link.

[SLACK_DOCS] **Block bit builder UI**. link.

[BOTIUM_DOCS] **BotiumScript Concepts**. link.

[LINDSTRÖM] **Await and Async**. link.

[JOVO_FORUM_QUERY] **Jovo Forum Query**. link.

Declaration of Academic Integrity

We hereby declare that we have written the present work ourselves and did not use any sources or tools other than the ones indicated.

Datum: .....

(Signature)