Interim Report to the Master Project WS 2019/20 | Zwischenbericht zum Masterprojekt WS 2019/20

Introduction / Summary

Motivation

A large number of companies are renewing their customer service in order to quickly bring their range of offers to potential buyers. Digitalization is a useful tool for bringing information to interested parties. The chatbot plays an important role here. Chatbots are dialogue systems that communicate via voice or text messages. Chatbots are used in various areas and present a variety of offers to inform users. There are also other categories, such as chatbots, which provide specific information about the weather. The Beuth University of Applied Sciences in Berlin offers its students, employees, scientific staff and teachers various services. The focus is on important questions such as when the opening hours of Beuth University are. For students, the opening hours of the library, the study administration, the dean's offices, the study and recreation rooms are also important. For these reasons Professor Thomas Ziemer proposes to develop a chatbot for the university.

Target group

The chatbot is aimed primarily at students, teachers and visitors to Beuth University. It helps the above mentioned groups to quickly get information about the learning rooms, Mensaplan and other services of the university. The chatbot also provides information about the weather.

Scope

Beuth University has an interest in offering a service that leads through the university. This service is intended to help new students find their way around Beuth University. This includes, among other things, that students have knowledge of exam dates and the teaching staff's consultation hours in order to better organize their studies. The chatbot also answers questions about the Mensaplan. The Mensa's offer is varied, e.g. the Chatbot answers to inquiries, when there is vegetarian or vegan food. It has other functions as well: So it can answer questions about the next week's menu and can consider hints from users, such as the request of a vegetarian.

Software Architecture

Table of content

Overview

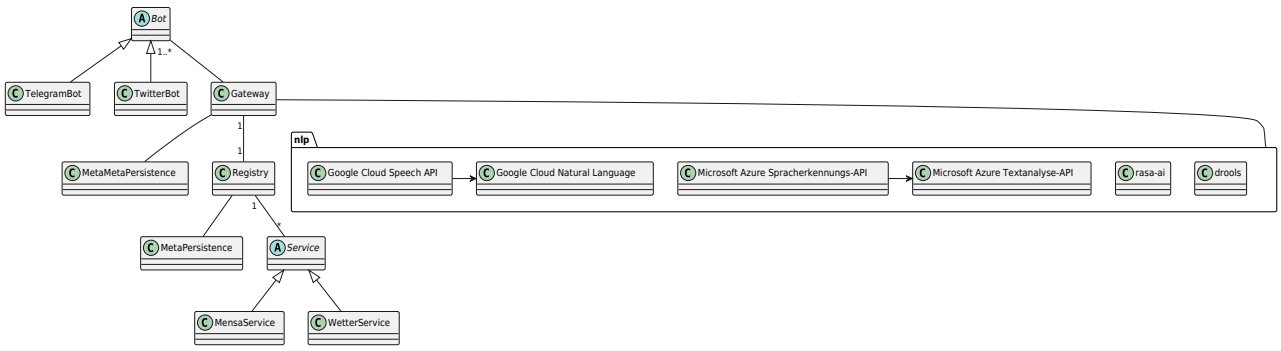
BeuthBot consists of many interwoven *Microservices*. Evey Microservice uses our basic API to communicate with other Microservices. This approach enables us to change parts of the system easily at any time or to introduce new Microservices, all they need to do is to implement our API.

Basic Structure

Our application is basically composed of the following four components.

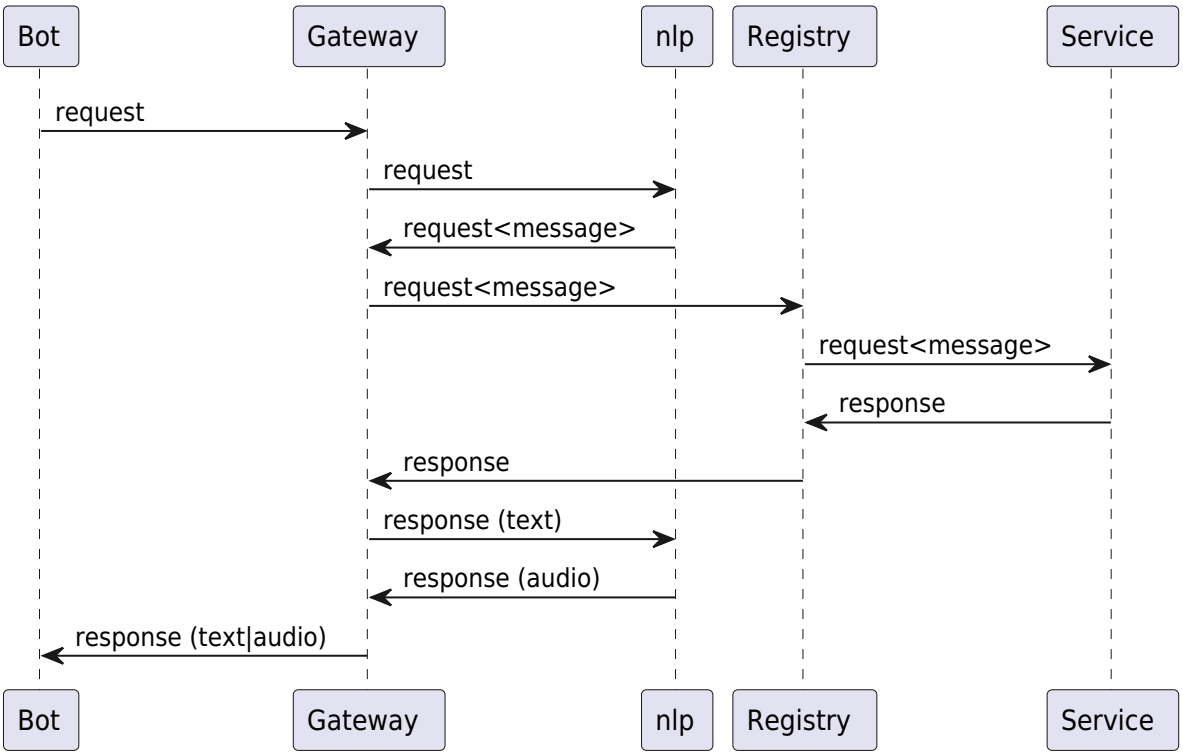
> Bot ⇔ Gateway ⇔ Registry ⇔ Service

Following diagram shows that in more detail:



A user can write the *Bot* to request informations, the meaning of the message is extracted and a fitting *Microservice* is choosen to retrieve the necessary data. A response is build from that data and distributed back up to the bot which answers the users request.

Following sequence diagram further illustrates that:



Bot

This is an abstraction for the available chatbots, e.g. a *Bot* for *Telegram* and another *Bot* for *WhatsApp*.

The user interacts with this *Microservice*, here she can request information and gets answers from *BeuthBot*.

Gateway

The *Gateway* is the centerpiece of *BeuthBot* one could say.

The *Bot* notifies the *Gateway* with the message it got from the user.

The *Gateway* then uses NLP (Natural Language Processing) *Microservices* to get the meaning and intention of the user. Here we try to extract what the user wants from *BeuthBot*, to notify the right service and present a fitting answer to our user.

Registry

After obtaining the intention of our user, the *Gateway* notifies the *Registry*, to get the information the user requested.

The Registry distributes the request to the correct *Service*, that takes care of retrieving the right informations.

Service

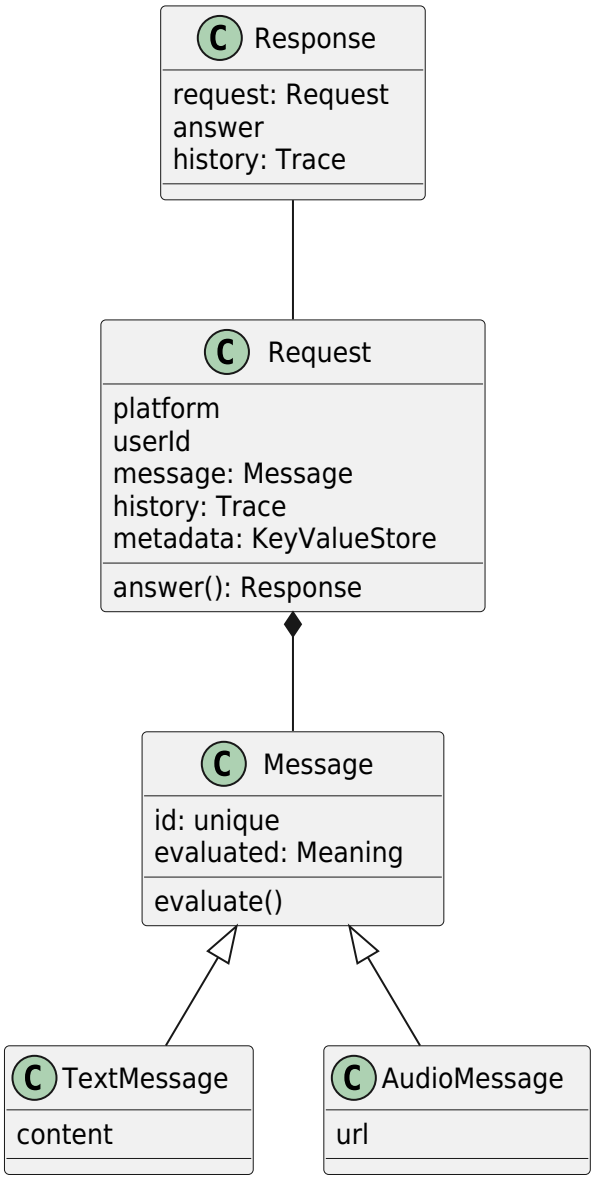
Service is an abstraction for the implemented *Microservices* that retrieve the necessary data we need to answer users requests. E.g. the *MensaService* is a *Microservice* that can give informations about the current menu, filtered by a number of parameters, e.g. a vegan user.

API

Because of the complexity of the single *Microservices*, every single *Microservice* implements its own, distinct, API.

But to answer a users request we use a unified, comprehensive API. Its basic idea is to pass a *Response*-Object trough the individual *Microservices*, which consists of the initial request, an answer as a response to the users request and informations about the user.

Following class diagram further illustrates that:



Requirement Analysis BeuthBot

Functional requirements

/F100/ The system must allow the user to enter requests by text or language
/F101/ The system should be able to learn from errors from incoming messages
/F102/ The system must understand user input
/F103/ The system must be able to respond contextually to user input
/F104/ The system must persist messages in a database anonymously
/F105/ The system must be able to persist and retrieve specified preferences for users
/F200/ The system must be able to retrieve the Beuth Mensa menu for a specific day from the OpenMensa API
/F201/ The system must be able to forward the menu from the OpenMensa API
/F202/ The system must be able to filter and probe the menu according to the user's specifications
/F203/ The system must be able to cache the food plan

/F300/ The system must be able to access the learning rooms of Beuth University of Applied Sciences Berlin
/F301/ The system must be able to forward where the learning rooms are located.

/F400/ System must be able to remind user of appointments
/F401/ The system must have access to the user's appointment calendar

/F500/ The system must be able to call up the opening hours of the Beuth University buildings.
/F501/ The system must be able to cache opening hours

/F600/ The system must be able to retrieve the current weather for Berlin via a Weather API
/F601/ The system must be able to forward the current weather
/F602/ The system must be able to cache the current weather

/F700/ The system must be able to call up the examination dates for exams at the Beuth University for Applied Sciences
/F701/ The system must be able to forward the test dates
/F702/ The system must be able to filter and probe the examination dates according to user specifications
/F703/ The system must be able to cache the test dates

/F800/ The system must be able to call up the winding rooms at the Beuth University for Applied Sciences.
/F801/ The system must be able to forward where the winding rooms are located.
/F802/ The system must be able to cache the winding rooms

Non-functional requirements

/NF100/ The system must respond to a message within 3 seconds
/NF101/ The system must retrieve data from the microservices within a few milliseconds
/NF102/ The system must be able to process and evaluate a message within 1.5 seconds
/NF103/ The system must have enough memory for persistence of data from ~13k students

/NF200/ Service downtime (NLP component, microservices, gateway) should be less than 1%
/NF201/ ref. /NF100/
/NF202/ ref. /NF101/
/NF203/ ref. /NF102/
/NF204/ Database downtime should be less than 1%

/NF300/ The system should be as modular as possible
/NF301/ The system should be easily scalable
/NF302/ The system should contain easily replaceable components
/NF303/ The system should store understandable error messages

/NF400/ The system should be easily portable to other systems

/NF500/ The system should comply with DSGVO guidelines
/NF501/ The system should be based on security standards
/NF502/ Databases should be protected from unwanted access
/NF503/ The databases should be password protected
/NF504/ The databases should be based on security standards

/NF600/ The system should restart the service independently in the event of a service failure

/NF700/ The system should be well documented
/NF701/ The system should be easy to understand

Use cases

In the following we present three usecases in detail, which exemplarily describe our functional requirements.

Use case /F103/

Title: Responding to user input

Short description: User sends a message to the chatbot via text or speech and the bot replies to it.

Actor: User

Preconditions: The chatbot, NLP component, gateway, registry and microservices are running

Basic flow: The user writes a message to the bot via telegram. This message is processed and evaluated by the NLP component, then the message, including the evaluation of the NLP component, is persisted in the database and forwarded to a corresponding microservice, which then generates a response and sends it back.

Effects: The user gets a reply from the chatbot, which refers to his message.

Use case /F200/

Title: User asks for today's menu of the mensa

Short description: User sends a request to the chatbot that he would like to know what there is to eat in the mensa today.

Actor: User

Preconditions: The chatbot, the NLP component, the Mensa micro service, the gateway and the registry are running.

Basic flow: The user writes a message to the bot via telegram. The NLP component recognizes that the user wants to have today's menu of the mensa. The evaluated message is forwarded to the mensa microservice. The microservice reads out what is required and asks the OpenMensa API for the mensaplan for the Beuth University of Applied Sciences. An answer is generated from the object which the microservice receives from the API and sent back to the user.

Effects: The user gets an answer from the chatbot containing today's menu of the mensa.

Use case /F300/

Title: Output learning spaces

Short description: The user wants to know which learning rooms there are and where they are, the chatbot gives him the information.

Actor: User

Preconditions: The chatbot, NLP component, gateway, registry, and learning room service are running.

Basic flow: User writes to the chatbot that he wants to know which learning rooms there are. The system processes the message and forwards it to the learning room microservice. If the learning rooms have not yet been cached, the service uses web scraping to search for the required information on the corresponding website, generates a response from it and sends it to the user.

Effects: The user receives an answer from the chatbot containing the required information.

Bot Documentation

> Telegram Bot build for the *BeuthBot*-Project, with easy extensibility and customization in mind.

Bla bla

Table of content

Getting Started

These instructions will get you a copy of the project up and running on your local machine for development and testing purposes.

Prerequisites

You will need a current version of node & npm.

Installing

After cloning the repository, install the dependencies. You can then run the project.

```
# install dependencies
> npm install
```

```
# serve at localhost:8000
> npm start
```

Overview

The bot is basically a *Node-Express*-Backend. Incoming requests are checked and specifically handled.

Structure

The bot is separated into two files. `index.js` contains the fundamental logic. The bot get created with its token and waits for incoming events. For example an incoming message. The bot then calls a handler function.

These handlers can be found in the second file, `commands.js`. This file contains the available commands as an Object. Furthermore does ist contain functions to determine if a message contains a commands and to answer the several requests a user can make.

Functionalities

User Requests

The bot supports three different kinds of user requests:

- **Message**: A user sends a message to the bot. We then check if the message contains a command. Commands are declared with a prefixed '/' in *Telegram*.

- **Callback Queries**: The bot can answer with a question, providing the user a simple interface, using a button matrix. When the user clicks on of these buttons we get a *callback query*.

- **Inline Queries**: Users can call our bot from within another chat by prefixing the botname with an '@'. The user can then send a text to the bot, which results in an inline query. The result that the bot gives back is inserted in the chat, where the user called the bot from.

Commands

The `commands.js`-file contains an `commands`-object. Every entry of this object is a supported command. The Key is always the command string, prefixed with '/', eg: '/help'. The value for these keys is an object containing an description, and options object and the reference to the function that renders the answer for the specific command.

```
const commands = {
    '/help': {
        answer: renderHelpString,
        description: 'Get a helpful list of all available commands
and functionalities',
        options: {
```

```
            parse_mode: 'Markdown'
          }
      },
    '/date': {
      answer: (message => 'What date format do you prefer?'),
      description: 'Get the current timestamp in a chooseable
format',
      options: {
        parse_mode: 'Markdown',
        reply_markup: {
          // this initiates a callback query
          // by giving the user two buttons to answer with
          inline_keyboard: [
            [
              {
                text: 'Zulu',
                callback_data: JSON.stringify({
                  command: 'date',
                  payload: 'zulu'
                })
              },
              {
                text: 'German',
                callback_data: JSON.stringify({
                  command: 'date',
                  payload: 'german'
                })
              }
            ]
          ]
        }
      }
    }
}
```

Functions

The 'commands.js'-file provides several functions. Eg. functions to check if a message contains an command and to find out if the requested command is in the 'commands'-object, which means it is an supported command.

Further are functions provided to handle Messages (containing normal *Commands*), *Callback Queries* and *Inline Queries*.

The bot has the following functionalities, that a user can request and use:

- getTimestamp: Get the timestamp of the moment the message containing this command was send.
- getFormatedTimestamp: Renders the timestamp in Zulu or German format, this is a function used to answer a *Callback Query*.

- renderHelpString: Iterates over the *commands*-object and prints all available commands and there description.
- supportedMarkdown: This function gives the User a list of supported Markdown markup by Telegram.

BotFather

The BotFather allows so configure our bot. You can just write the *BotFather* with Telegram and the bot will guide you through everything. The *BotFather* enables you among others to change the profile picture, description and about text of your bot.

Further you can register the commands and inline queries your bot supports. This allows a cleaner user experience since the bot will then suggest commands and inline queries while the user types. So absolutely do register them!

The neccessary commands are:

- '/setcommands' - '/setinline'

Further Development

New commands can simply added to the 'commands'-object but have to follow the presented structure under commands.

Further Reading

- Telegram Bot API

Built With

- Node.js
- Express.js
- Node-Telegram-Bot-API

Versioning

We use SemVer for versioning. For the versions available, see the tags on this repository.

Authors

- **Tobias Klatt** - *Initial work* - GitHub

See also the list of contributors who participated in this project.

Gateway

The **gateway** itself is the core microservice of our application. It represents the top (first) layer in our system architecture and has a direct bidirectional interface to Telegram. The main functionality of the gateway is to receive and handle all incoming user requests. Once a user is interacting with our bot - doesn't matter whether the user communicates via text or voice message - all requests are going to be passed on to the gateway.

Table Of Content

Getting Started

The following sections will give an overview how the gateway was created. It is strongly recommended to read Telegrams bot introduction for developers to get a better insight what we are talking about in this context.

Every time a Telegram bot receives a message, the bot forwards this message in form of an API call to a corresponding server that handles all incoming messages. Once this is done, the server processes the request and a response will be generated that will go back to the user. In general there are two ways to get notified about incoming messages:

1. *Long polling*
2. *Webhooks*

Within this project we are going to use webhooks.

Prerequisites

Since the gateway is built from scratch there are no specific requirements or dependencies.

[*Appendum:* We decided to establish the server using node.js. That's why an installation of node and npm is necessary.]

Telegram Bot

As mentioned here our gateway is directly connected to the bot. Therefore the creation of a Telegram bot is necassary before it comes to the actual implementation. For test purposes an onw Telegram has been created as part of preparing the gateway implementation. It is reachable via `cbeuthbot` on Telegram. The created bot does not have any kind dependecies to the productive `BeuthBot` and is completely autonomous. This means that the system architecture is intended to be as flexible as possible to enable a simple addition or removel of different types of bots.

Set Up

Once a Telegram bot has been created and configured, we started to initialize a local project in a first step. Therefore a project directory has been set up as well as a `> npm init` has been executed in this directory. After this step a `package.json` has been created automatically. On top of that, `express`, `axios` and `body-parser` have been installed via `> npm install`. In this context `express` is our application server, `axios` is an HTTP client and `body-parser` helps to parse the response body received from each request. As soon as these components have been succesfully installed we created our actual **gateway** - first simply named `index.js`.

The content of this file was looking very rudimentary in the beginning. It simply repsents a 'Ping-Pong' service at this point. This means, if a user writes a message that includes e.g. the word 'ping' our gateway creates a response with the word 'pong'. The answer will be sent back to the user by using the chat-id. Additionally we established `3000` as our port for communicating.

At this point we were able to run our server locally by typing in `> node index.js`. But a local server implies that the bot cannot call an API. It is desperate need of a public domain name. This means we have to deploy our application with ZEIT.

Once this is done we have to let telegram know that our bot has to talk to this url whenver it receives any message in a last step. This get managed through cURL.

References

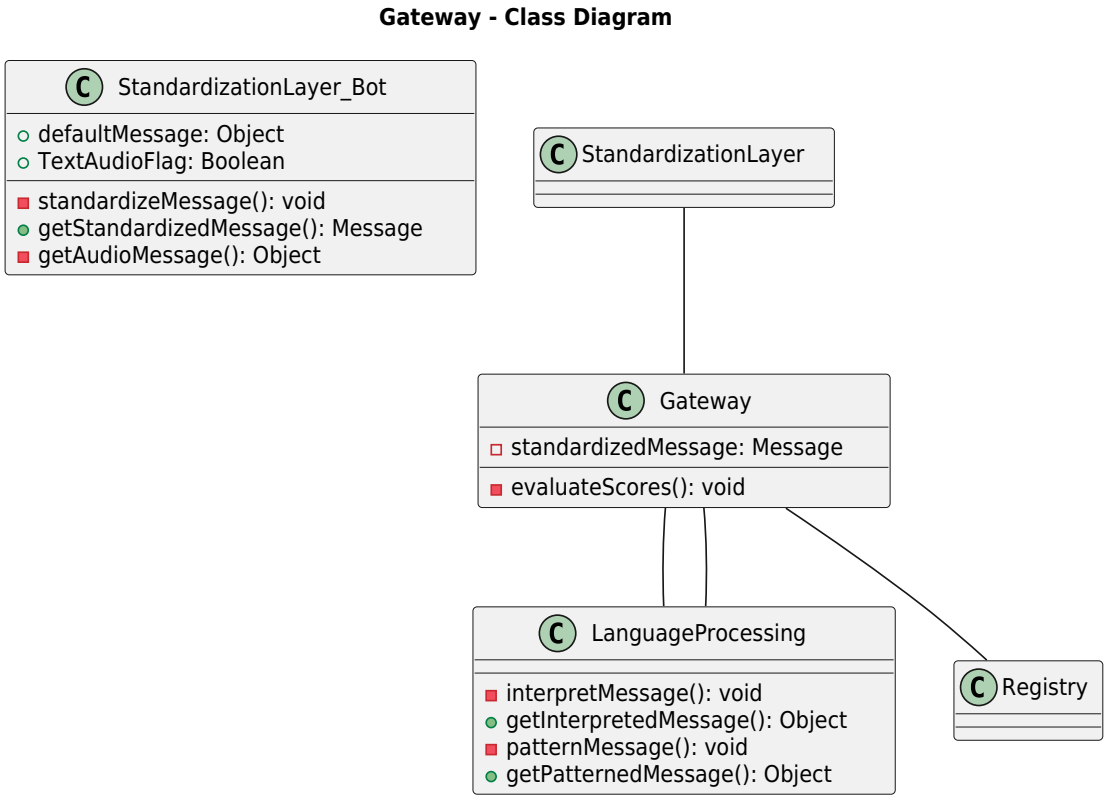
During the implementation of the **gateway** we used this manual as a kind of orientation.

2. Overview

The **gateway** we built is able to receive incoming messages from our bot and also standardizes (since there is no guarantee for uniform requests, all incoming messages are getting standardized in a very first step) all requests. Once this is done, the **gateway** calls one or more of our NLU interfaces to evaluate the message text. This is done via `HTTP-POST` and `json`. The evaluation of our message (score determining) can be done separately or together with the text analysis. E.g. when using `Microsoft Azure Cognitive Services` we transfer our messages with all relevant parameters and as a result out `HTTP-POST` delivers the score, entities, key words etc. in form of a `json` object. With this result we continue to call the API of our „next“ microservice (in this case the `registry`) and pass on all relevant values.

Structure

To give a better overview of how the gateway is built up, the following class diagram has been created:



This class diagram shows the structure around the gateway. Here it is important that there is a StandardizationLayer beforehand, which standsardize the incoming messages. The gateway then directs the message to an NLU service where we get the evaluated object back and compare the scores. The best evaluated message is then forwarded to the registry.

Functionalities

Variables

The defined variables are based on `express`, `body-parser` and `axios`:

```
var express = require('express')
var app = express()
var bodyParser = require('body-parser')
const axios = require('axios')
```

API-Call

Each time our bot is mesaged in the chat, the message will be passed on to the gateway. This is mapped via cURL. All incoming messages use the route `\message-in`. If the message has no content, the response is empty. The code for the described behaviour looks as follows:

```
app.post('/message-in', function(req, res) { // This is the
route the API will call
    const { message } = req.body
        if (!message || message.text.length < 1) { // In case
a message is not present, or if our message is empty, do
nothing and return an empty response
            return res.end()
        }
```

Microsoft Azure - Cognitive Services - Headers

Microsot Azure predetermines its specific header that should be used for HTTP-POST. The header looks like this:

```
    const options = {
        headers: {
            'Host': 'northeurope.api.cognitive.microsoft.com',
        'Content-Type': 'application/json',
        'Ocp-Apim-Subscription-Key':
'********************************'
        }
    }
```

HTTP-POST

This section of code shows a request to the Azure service and generates a response which is sent directly to the bot. The code is shaded like this because Axios

processes the messages asyncronous and we have to ensure a response has already been received. The following code snippet shows this more in detail:

```
axios.post('https://northeurope.api.cognitive.microsoft.com/text/analytics/v2.1/sentiment', {
        "documents": [{
            "language": "en",
        "id": message.chat.id,
        "text": message.text
        }]
    }, options).then(function (response) {
        message_out = "[" + message.chat.id + "]: " + "Hi, your score is " + response.data.documents[0].score + "."
axios.post('https://api.telegram.org/bot:<token>/sendMessage', {
            chat_id: message.chat.id,
            text: message_out
        }).then(response => {
            // We get here if the message was successfully posted
            console.log('Message posted')
            res.end('ok')
        })
    })
})
```

Server

The server is listening on port 3000:

```
// Finally, start our server
app.listen(3000, function() {
    console.log('Telegram app listening on port 3000!')
})
```

Further Development

In the short term, we are considering replacing Azure with Rasa to test the modular requirements. It is later considered that we will connect an NLU adapter that compares the two services and takes the best results.

Further Reading

To get a deeper insight into the technical components of our gateway, we recommend to follow up with some of the topics that are mentioned here or here.

Built With

- Telegram Bot API
- Node.js
- Express.js
- Axios
- Body-Parser
- ZEIT
- cURL
- Microsoft Azure

Versioning

We use GitHub for versioning.

Authors

- **Christopher Lehmann** - *Development* & *Documentation* GitHub

- **Timo Bruns** - *Development* GitHub \\

See also the list of contributors who participated in this project.

Rasa NLU

> Rasa is an open source solution for developing „AI assistants“ or chatbots. Rasa provides a stack consisting of the modules „Rasa NLU“ and „Rasa Core“. With the help of „Rasa NLU“ the user intention is determined from the received text message (Intent Recognition) and afterwards the NLU returns all intentions of the message sorted according to the „Confidence Score“. Training data is required to record the user's intentions. Furthermore, Rasa NLU allows „entity recognition“ to extract relevant terms from the text. The Rasa Core is a dialog engine that uses machine-learning trained models to decide which response to send to the user, such as greet the user. Furthermore, the core allows „session management“ as well as „context-handling“. Within the project only the component „Rasa NLU“ will be used, because only the functionality is needed to capture entities from a text message and to determine the user intention.

Table Of Content

Getting Started

The following instructions are intended to help the user run the Rasa-NLU-component on the local machine for development.

Understanding Rasa-NLU

Rasa NLU allows the processing of natural language to classify user intentions and extract entities from text.

e.g.

`„Wie ist das Wetter übermorgen?“`

The user intention is then determined from the text. In the figure below, the response to the message is displayed. The user intention („Wetter“) and the date for tomorrow are illustrated.

```
{
    "intent": {
```

```
        "name": "wetter",
        "confidence": 0.9518181086
    },
    "entities": [
        {
            "start": 19,
            "end": 29,
            "text": "übermorgen",
            "value": "2020-01-20T00:00:00.000+01:00",
            "confidence": 1.0,
            ...
        }
    ]
    ...
}
```

Training data is needed so that Rasa can identify the intention of a text. For this purpose, training data can be created in the form of Markdown or JSON. You can define this data in a single file or in multiple files in a directory.

To create a trained model for Rasa from the Markdown or JSON, Rasa offers a REST API. An alternative to creating trained models is to install Rasa on your local machine and then create the model using the command „rasa train nlu“. Rasa creates the training model (tar.gz) from the Markdown or JSON.

Furthermore Rasa NLU is configurable and is defined by pipelines. These pipelines define how the models are generated with the training data and which entities are extracted. For this, a preconfigured pipeline with „supervised_embeddings“ is used. „supervised_embeddings“ allows to tokenize any languages.

Perform Rasa locally

You need the local installation of Rasa to create and test training models. For this, you use the directory „training“.

Basic requirements

The following installations must be made:
- Pip
- Python (Version 3.6.8)
- Tensorflow
- Making further installations (Rasa Installation)
- If necessary, further installation via pip (depending on the message of the compiler)

Project structure

The following files and directories are important for the project to configure Rasa, customize training data and create appropriate training models.

- config.yaml:
contains the configuration of the NLU e.g. specification of the pipeline (how the trained model is generated)
- /data (directory):
contains training data in the form of JSON (Markdown would also be possible)
- /models (directory):
contains the trained model in the form of tar.gz.files The model is needed to capture entities and the user intent of a message.

Commands

You need to run the following commands in the directory '.training'.

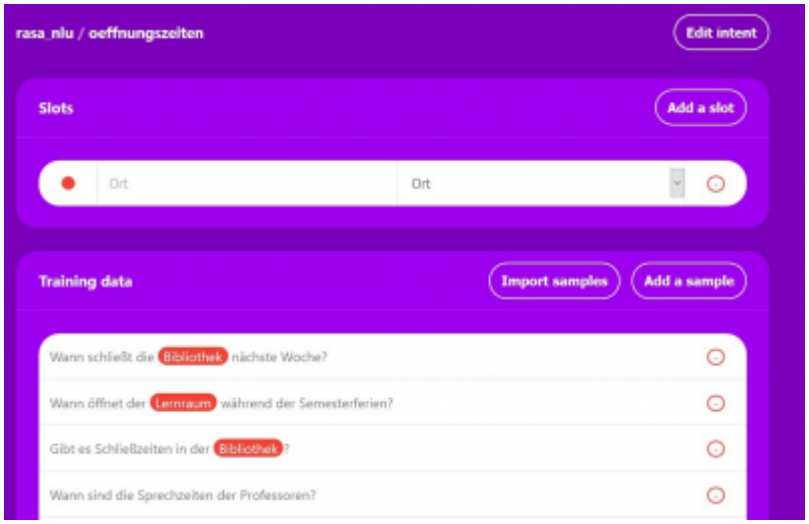
```
# Create training-model:
rasa train nlu

# Communicating with Rasa NLU on the command line:
rasa shell nlu –m models/name-of-the-model.tar.gz
```

Running Duckling:

After using the command „rasa train nlu“ a model is generated. When communicating with Rasa via the shell („rasa shell nlu …“) the component „Duckling“ is not addressed. With Duckling you can identify and resolve dates. To use Duckling you can add the trained model in the path „docker\rasa-app-data\models“. Then you can run Rasa and Duckling as docker-containers and query them using the Rest API. Running Rasa and Duckling as docker-containers are explained in a later section.

How to generate training datasets

In this project we write training data in the form of JSON, because JSON offers the possibility to extract entities from a text message. For this purpose the data was generated with the tool Tracy. In the image below, Tracy is shown with „Öffnungszeiten“. Entities are added as „slots“, such as „Ort“. Training data follows in the lower part of the picture. As training data, you can specify messages, which the user can send to the „chatbot“. Currently the three user intentions „Mensa“, „Wetter“ and „Öffnungszeiten“ are supported.



Add new Model for Rasa-Container (Docker)

You have to add the generated model (tar.gz) under the path „rasa-app-data\models“.

Use of Docker

You will need to install Docker in order to use the Docker-Compose-file for running the application.

Installation instructions for Docker

Installing

After the repository has been cloned and the prerequisites have been fulfilled, you can run the Docker-Compose-file. The docker commands must be executed in the 'docker'-directory.

! Fix Me! Fix Me! Fix Me!

```
# build and start Containers && serve at localhost:5005 (rasa)
and at localhost:8000 (duckling)
docker-compose up

# stop and remove rasa-containers, volumes, images and
networks
docker-compose down

# do the same steps as "docker-compose down"
# additionally remove declared volumes in Docker-Compose-file
docker-compose down -v

# lists running containers
```

```
docker ps

# connect to the container with a bash
docker exec -it <Container-ID> bash
```

Overview

As part of the chatbot-project, microservices are supposed to run in Docker-Containers. In order to start several different services in containers at the same time, a Docker-Compose-File should be created. A Docker-Image is used for the Rasa NLU. Duckling has also been added as an Docker-Image for capturing date entries and allows to parse dates in a structured text.

```
version: '3.0'
services:
  rasa:
    image: rasa/rasa:1.6.0-spacy-de
    ports:
      - 5005:5005
    volumes:
      - ./rasa-app-data:/app
    command:
      - run
      - --enable-api
      - --cors
      - "*"
  duckling:
    image: rasa/duckling:0.1.6.2
    ports:
      - 8000:8000
```

The most important file for Rasa is the machine learning trained model (.tar.gz), which is written in the volume of the docker container. When executing the Rasa container, the model is needed to recognize user intentions.

HTTP-API

Rasa offers several REST APIs to provide server information, training models, etc. The Rasa features used in the project are listed here:

- **Serverinformation**:

You can query the Rasa-server whether it is still running or which Rasa version is available. You can also check which model Rasa is currently using.

- **Model**:

You can send requests via the Rest API of the Rasa server to create a trained model

or load the model into Rasa. You can also send text to the server and Rasa will then determine the user's intention and the confidence score.

Links:

- HTTP-API (Retrieved 12.12.2019)
- OpenAPI-specification (Retrieved 12.12.2019)

Further Development

For further development, it is important that the existing training data be expanded and improved.

Further Reading

- Rasa Documentation (Retrieved 12.12.2019)
- Running Rasa with Docker (Retrieved 12.12.2019)

Built With

- Docker-Compose (Retrieved 12.12.2019)
- Docker Hub Rasa (Retrieved 12.12.2019)

Author

- **Abirathan Yogarajah**

References

- https://rasa.com/ (Retrieved 12.12.2019)
- https://botfriends.de/botwiki/rasa (Retrieved 12.12.2019)
- https://www.artificial-solutions.com/wp-content/uploads/chatbots-ebook-deutsche.pdf (Retrieved 12.12.2019)
- https://docs.docker.com/ (Retrieved 12.12.2019)

I. Microsoft Azure

Microsoft Azure is a Cloud computing platform from Microsoft. Our primary used service is the cognitive services text analytics. For the Beuthbot we use the student version off Azure.We chose this provider because we did not need to provide a credit card here.

Table of content

Student Account

The Student Account is an free to use Account from Microsoft Azure. Here we get some limittet free acess to different services of the azure programm. Details you can find under: https://azure.microsoft.com/de-de/free/students/

Because we have problems when creating the account, we have designed a following short manual:

A. click on „activate now“ on https://azure.microsoft.com/de-de/free/students/
B. login with an privat microsoft account(no University Mail!)
C. after login you must verify your Student Account with University Mail
D. at last you must ident you by name, mail address and phone number here you get an activation code what you must

The important knowledge what we get from this registration is that we need private microsoft accounts and that the microsoft support is not very helpful.

Cognitive Services Text Analytics

To use the ervice we create in oure Azure account an BeuthBot Projekt(resourcesgroup). In this we create an Cognitive Services Text Analytics in North Europe. Now we get an Api End Point from Microsoft where we can do „Post „request with our messages. We have four options to ask for: the language, analyze sentiment, Extract key phrases and Identify linked entities.

More informations about the API you can find her: https://docs.microsoft.com/de-de/azure/cognitive-services/text-analytics/.

For the moment we use the service direktly in our gateway later we want to outsource the service in an extra NLU Request Service(Adapter for NLU).

A Post Request Object in JavaScript should look like the following example:

```
POST
https://beutbot.cognitiveservices.azure.com/text/analytics/v3.0-preview.1/languages HTTP/1.1

Host: beutbot.cognitiveservices.azure.com

Content-Type: application/json

Ocp-Apim-Subscription-Key: ••••••••••••••••••••••••••••••••


{
  "documents": [
    {
      "id": "",
      "text": ""
    }
  ]
}
```

The response body has following structure:

```
{
   "documents": [
      {
         "id": "",
         "detectedLanguages": [
            {
               "name": "",
               "iso6391Name": "",
               "score":
            }
         ]
        }
    ]
 }
```

To interpret the response we have to write a parser for all operations what we ask by the API.

Knowledge

Very quickly, we realized that is it not possible to operate a chatbot with the text analytics services without paying for the service. Because the free limited acess of the service is consumed very quickly in our own tests. So, while we can show that it is possible to use a chatbot with this service, it will not be usable that way. We allready found out, that IBM has an also free NLU Service with more request than azure but there we will have the same problem but we try to implement this service also, so that we cann show that our NLU Adapter work and we have different

options for services to request.

I. Pricing

Here we show a cost calculation for various NLU services. The latest figures from the Beuthhochschule were taken. For students it was expected that you would use all services and that they would be there 4 days a week. All of this multiplies the requests per week. Employees were expected to use only two services, but they also spent 5 days in college. In the case of lecturers, it was calculated that they use 2 services, but are only there 1 day a week.

	Students	**Employee**	**external lecturer**
Number of people	12667	787	600
Number of services used	4	5	1
Number of days present	6	2	2
Inquiries per week	304008	7870	1200
Total requests per week	313078		
Total requests per month	1252312		

After we have calculated the maximum total number of inquiries **for one month**, we have downsized accordingly the percent of inquiries. These were calculated with the prices of the providers according to the price lists. With some providers it was not possible to determine the exact price and it may be that the price becomes even more expensive.

providers	**Cost in €**	**100%**	**80%**	**60%**	**40%**
Azure	0,844	1057,532€	845,688€	634,688€	422,844€
IBM	0,0009	2254,1616€	1803,32928€	1352,49696€	901,66464€
Google	0,9	1127,7€	901,8€	676,8€	1127,7€
AWS	0,00054	676,24848	540,998784€	405,749088€	270,499392€

This table clearly shows that AWS offers the cheapest service. However, we tend to Azure because there is a search query with more characters available.

I. Implemented Product Features

Based on the previous documentation for the Microservices has already given an impression that we have some services ready. Each of these services works self-contained. In order to be able to test all the services together, there are still a few services that are planned to be implemented in the next few weeks. In addition, existing ones may need to be extended to ensure full coverage of all components planned in the architecture.
Below is a list of which of the defined requirements have already been implemented. Here a distinction is made between partially implemented or fully implemented requirements.

Table Of Content

Partially implemented Requirements

The following requirements are partially met, since the microservice is ready for this, but there is still no full connection to the system.

`/F100/` The system must allow the user to enter requests by text ~~or language~~

> Here are some NLU services that interpret the text. In order to generate an answer, we still lack a rule machine.We do not yet support voice input.

`/F200/` The ~~system~~ must be able to retrieve the Beuth Mensa menu for a specific day from the OpenMensa API
`/F201/` The ~~system~~ must be able to forward the menu from the OpenMensa API
`/F202/` The ~~system~~ must be able to filter and probe the menu according to the user's specifications

> Here we already have the microservice but since the system is not ready the requirement is not fully supported yet.

Architecturally, we have already started with some microservices, some of which are not yet full functional. An overview of the existing functions has hopefully given you the pre-recorded documentation.

Fully implemented Requirements

The following Requirements should be fully implemented our plant with our

archtiecture.

/NF300/ The system should be as modular as possible
/NF301/ The system should be easily scalable
/NF302/ The system should contain easily replaceable components

After assembling it, we have found that our concept works and our implemented services work. The bot responds to corresponding requests.