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

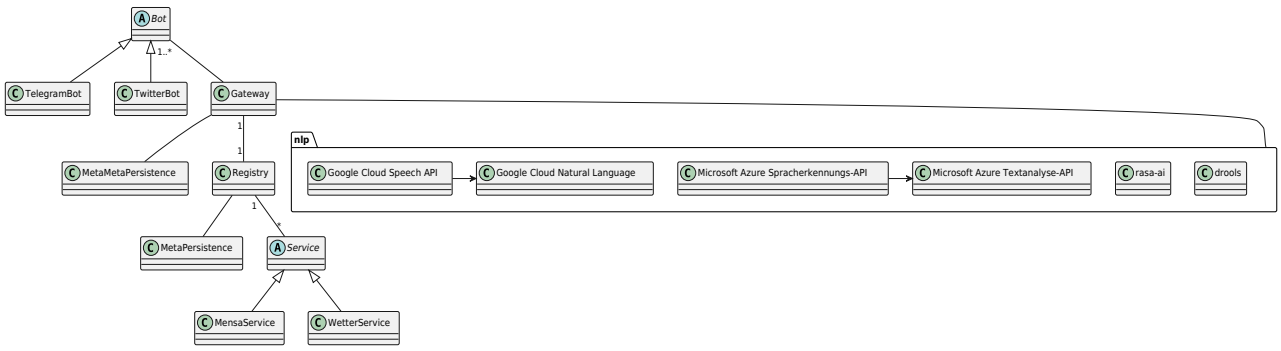
BeuthBot consists of many interwoven *Microservices*. Evey Microservice uses our basic API to communicate with other Microservices. This approach enables us to change parts of the system easily at any time or to introduce new Microservices, all they need to do is to implement our API.

Basic Structure

Our application is basically composed of the following four components.

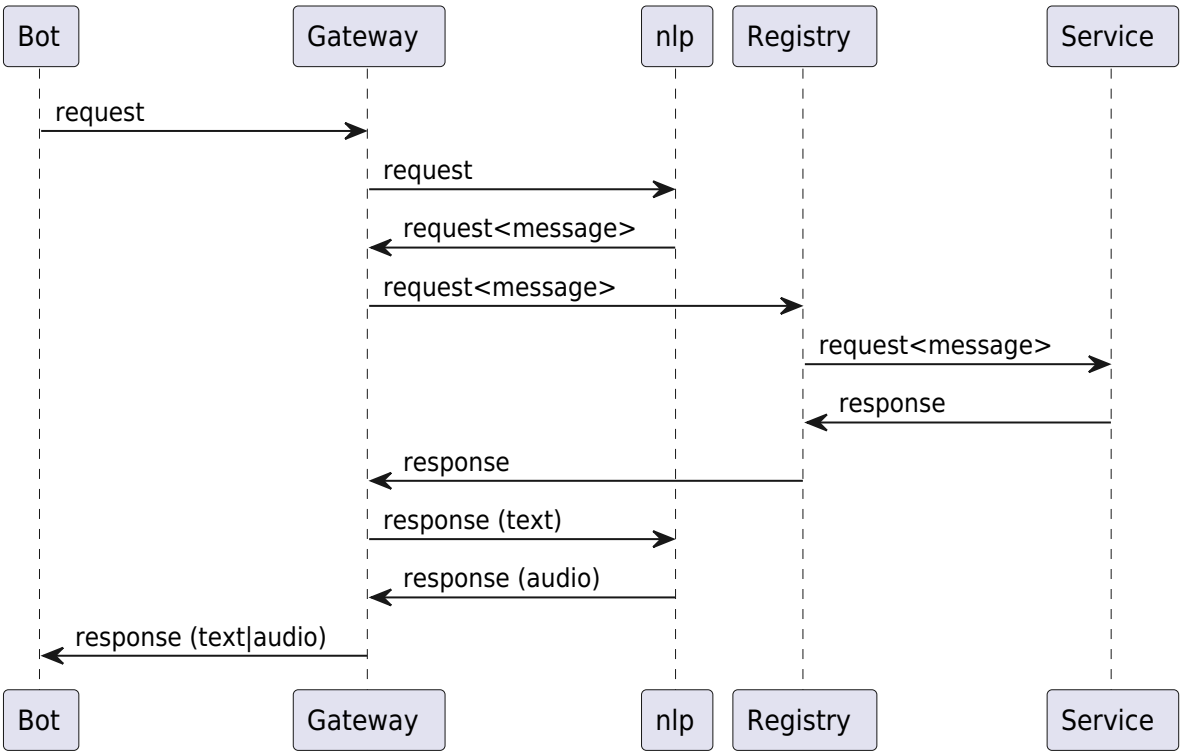
> Bot ⇔ Gateway ⇔ Registry ⇔ Service

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

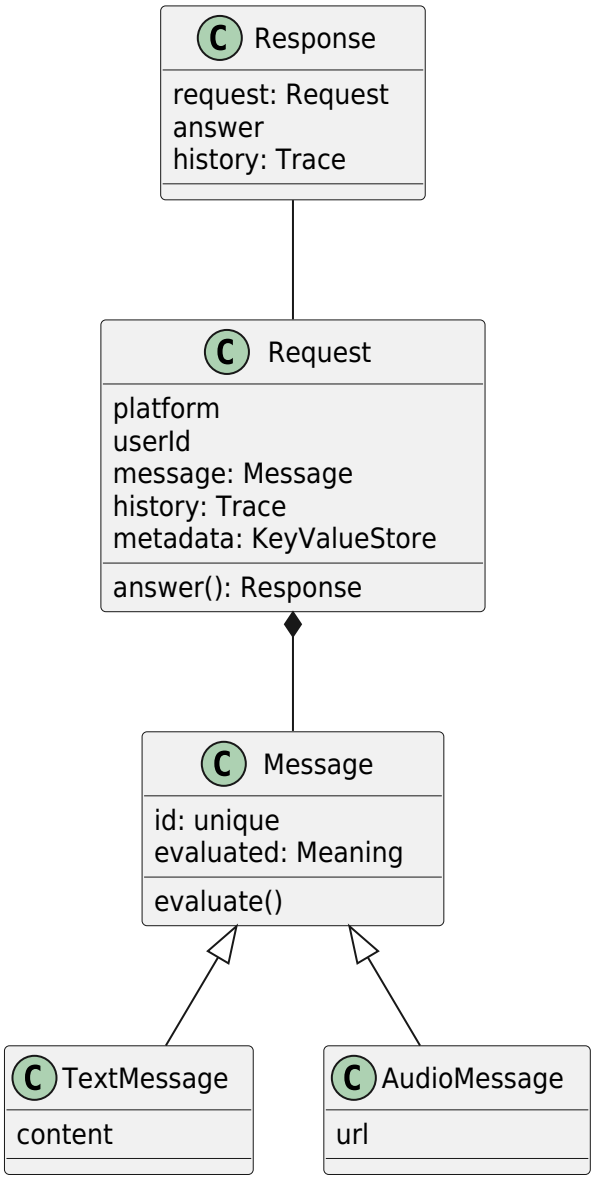
Service is an abstraction for the implemented *Microservices* that retrieve the necessary data we need to answer users requests. E.g. the *MensaService* is a *Microservice* that can give informations about the current menu, filtered by a number of parameters, e.g. a vegan user.

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.