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.

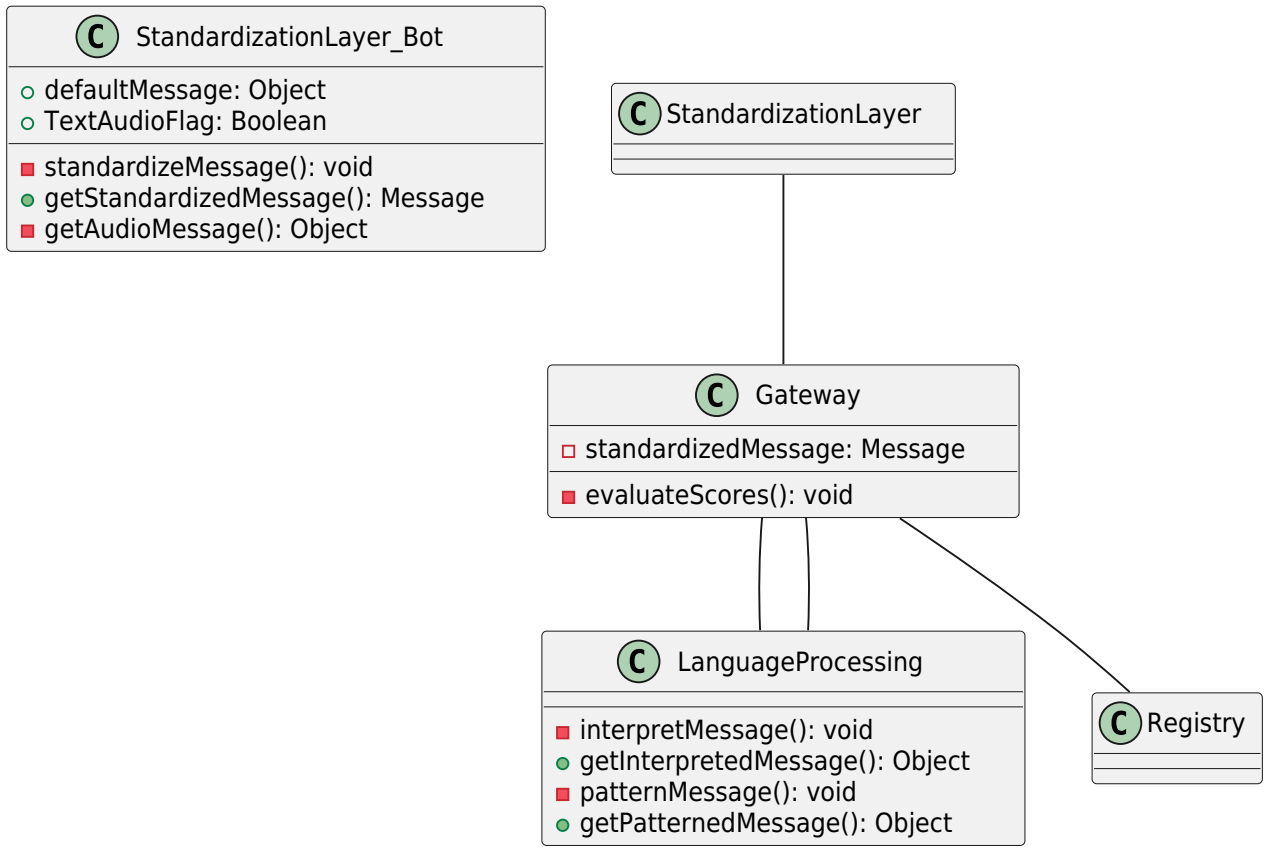
Overview

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:

Gateway - Class Diagram



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 GitLab for versioning.

Authors

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

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