Abschlussbericht zum Masterprojekt SS 2020

Inhalt

1. Inhalt
2. Abstract
3. Uebersicht / Inbetriebnahme
4. Deconcentrator-JS
5. Database
6. Database Microservice
7. RASA Trainieren
8. Cache
9. Update Wetter Microservice
10. Geoservice Erweiterung des Wetter Microservices

Abstract

Uebersicht

Die folgende Liste zeigt die Erweiterungen, welche in diesem Semester umgesetzt wurden.

- BeuthBot Master Repository erstellt um die Organisation des Projektes zu vereinfachen.
- Dokumentation des Master Repository, Erweiterung der bestehenden Dokumentation.
- Ersetzen des Deconcentrators durch Deconcentrator-JS.
- Einbauen einer Persistenz:
    - Realisierung eines passenden Models für die Persistierung der Daten.
    - Implementierung des Database-Controllers welcher eine MongoDB nutzt um die User Daten zu speichern.
    - Implementierung des Database-Microservice welcher Datenbank-Anfragen auflöst und sie durchsetzt.
5. Einbau Cache in Registry.
6. Umbau / Verbesserung des Weather-Microservice:
    - Formatierung der Antworten
    - Refactoring des bestehenden Codes
    - Einbau von zeitbezogenen Daten
    - Einbau von ortsbezogenen Daten

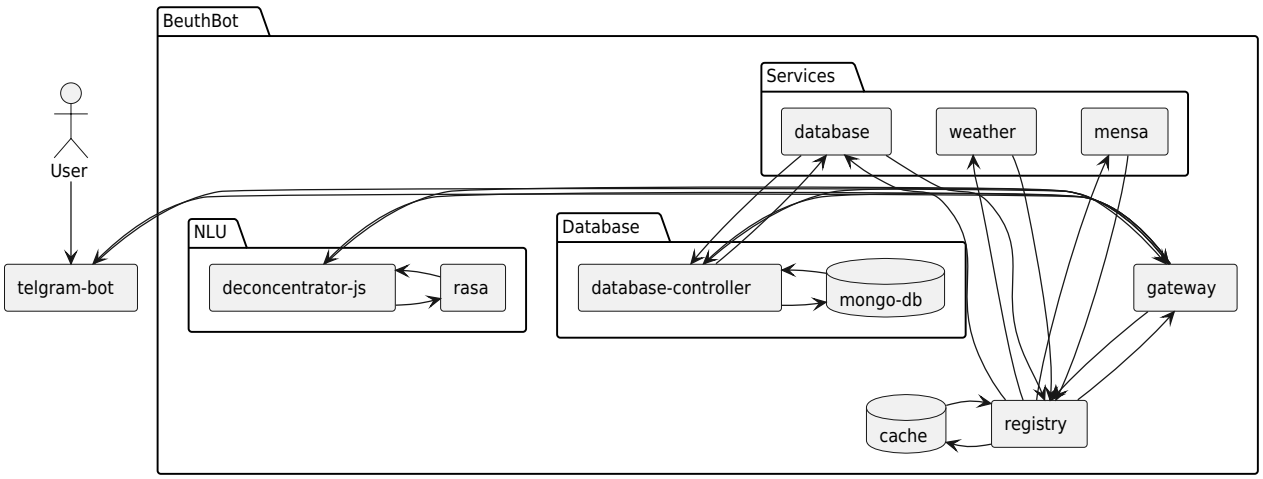
BeuthBot Master Repository

Wie dem Zwischenbericht entnommen werden kann, hatten wir am Anfang ziemliche Schwierigkeiten das gesamte Projekt in Betrieb zu nehmen. Jedes Repository musste einzeln gecloned werden und danach jede Komponente einzeln gestartet werden. Aus dieser Not haben wir ein Master-Repository erstellt, welches die einzelnen Module als Git-Submodules beinhaltet. So ist es nun möglich mit einem einzigen Befehl das gesamte Projekt zu laden. Das Projekt kann unter dem Link https://github.com/beuthbot/beuthbot angeschaut werden.

Inbetriebnahme

```
# clone project
$ git clone --recursive https://github.com/beuthbot/beuthbot.git

# or with ssh
$ git clone --recursive git@github.com:beuthbot/beuthbot.git

# change into directory
$ cd beuthbot

# edit environment file
$ cp .env.sample .env && vim .env

# start BeuthBot
$ docker-compose up -d

# check whether the gateway is running on port 3000
$ curl http://localhost:3000          # prints: Hello from BeuthBot
Gateway
```

Komponenten des BeuthBot



Deconcentrator-JS

Der Deconcentrator-JS übernimmt die gleiche Aufgabe wie der Deconcentrator und ersetzt diesen. Die Entscheidung den Deconcentrator auszutauschen kam daher, dass es uns am Anfang des Semester nicht möglich war, den Deconcentrator aus dem vorherigen Semester in Betrieb zu nehmen. Auch nach langer Beschäftigung und der Hilfe eines Studenten aus dem letzten Semester blieben Erfolge aus. Da dieses Projekt noch weiter entwickelt werden und somit ein einfacher Einstieg und eine überschaubare Komplexität gewährleistet werden soll, erschien es uns also sinnvoll den Deconcentrator durch den neuen Deconcentrator-JS auszutauschen. Ein weiterer Faktor bestand darin, dass der alte Deconcentrator in Python geschrieben war. Eine Vorgabe des Projekts ist aber die Verwendung der Programmiersprache

JavaScript. Mehr Informationen über den Deconcentrator-JS befinden sich hier im Wiki. Das Projekt kann unter dem Link https://github.com/beuthbot/deconcentrator-js angeschaut werden.

Database

database

Table of Content

1. database
2. Table of Content
3. Motivation
4. Requirements
    a. Functional
    b. Non Functional
5. User Stories
6. Use Cases
7. Klassendiagramm User
8. Technologies
9. Integration
    a. Sequenzdiagramm mit angesteuertem Service
    b. Sequenzdiagramm nur Datenbank betreffend
10. Getting Started
    a. Windows
11. API
    a. Request all Users
    b. Request Users
    c. Add / Change Detail
    d. Delete all Details
    e. Delete Detail

Motivation

Die Motivation hinter einer Datenbank im BeuthBot Projekt kommt durch das Problem, dass Benutzer ihre Wünsche immer wieder komplett ausführen müssen.
Als Beispiel: Wenn der Benutzer die Mensa nach veganen Gerichten anfragt, dann muss er das bei der nächsten Anfrage wiederholen.
Die Datenbank soll das Problem beheben und den Benutzern die Möglichkeit bieten ihre Vorlieben zu persistieren, ohne dass diese sich einen extra Account anlegen müssen.
Dabei muss darauf geachtet werden, dass in der Zukunft noch neue Services dazu kommen können. Die Architektur und die Datenbank sollten so konzipiert werden, dass neue Details die zu neuen Services gehören gespeichert werden können, ohne dass die Datenbank dazu angepasst werden muss.

Requirements

 # Was soll die DB können?

Functional

- /DBF100/ The system must be able to store details about a user.
- /DBF101/ The system must be able to add a detail related to a user.
- /DBF102/ The system must be able to change a detail related to a user.
- /DBF103/ The system must be able to delete a detail related to a user.
- /DBF104/ The system must be able to load all details about a user.
- /DBF105/ The system must be able to delete all entries related to a user.
- /DBF106/ It must be able to add new services and store related user details without modifying the database.
- /DBF200/ The database must store data related to a user.
- /DBF201/ The database must be able to store a nickname related to a user.
- /DBF202/ The database must be able to store a new detail about a user without scaling the schema.
- /DBF203/ The database must have a capacity of N.

Non Functional

- /DBNF200/ The database must be easily scalable.
- /DBNF201/ The database must be easy replaceable.

Referenz WS2019:

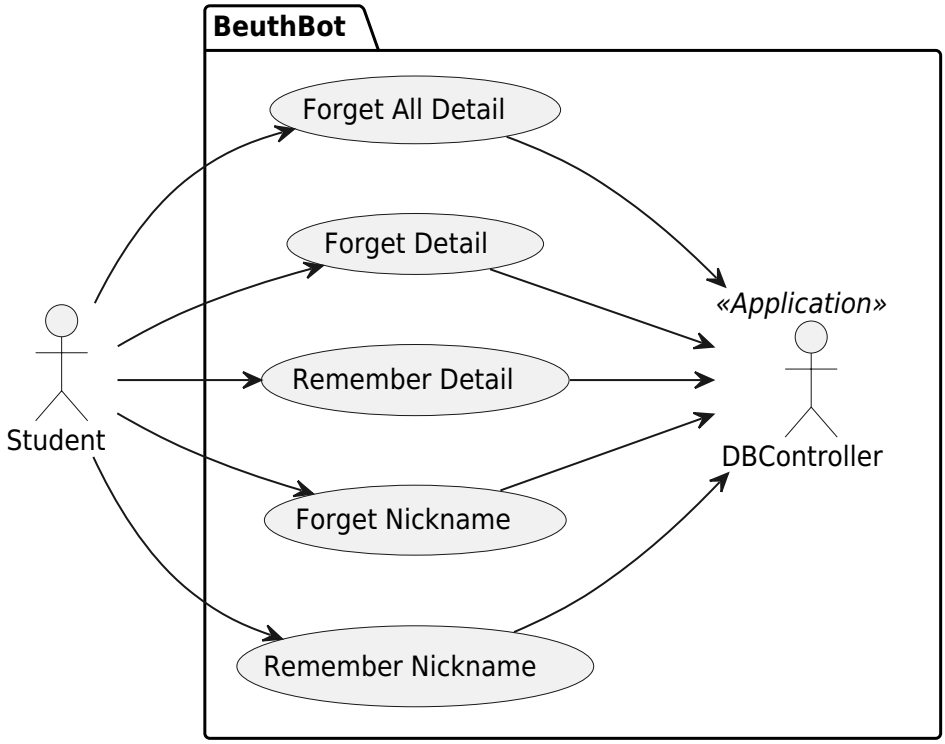
- /NF500/ The system should comply with DSGVO guidelines
- /NF501/ The system should be based on security standards
- /NF502/ Databases should be protected from unwanted access
- /NF503/ The databases should be password protected
- /NF504/ The databases should be based on security standards
- /NF600/ 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

User Stories

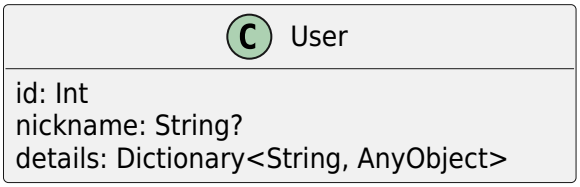
```
"Als <Rolle> möchte ich <Ziel/Wunsch>, um <Nutzen>"
```

- /DBUS100/ Als Student möchte ich meine Vorlieben speichern, damit ich sie nicht immer wieder ausschreiben muss.
- /DBUS101/ Als Student möchte ich nach einigen Anfragen, dass ich gefragt werde ob ich meine Vorlieben speichern möchte, damit ich sie nicht immer wieder ausschreiben muss.
- /DBUS102/ Als Student möchte ich, dass der Bot mich wiedererkennt ohne einen extra Account anlegen zu müssen, damit ich keine zusätzlichen persönlichen Informationen preisgeben muss.
- /DBUS103/ Als Student möchte ich dem Bot sagen können, dass er meine ALLE meine Daten löschen soll.
- /DBUS103/ Als Student möchte ich dem Bot sagen können, dass er ein Detail über mich löschen soll.

Use Cases



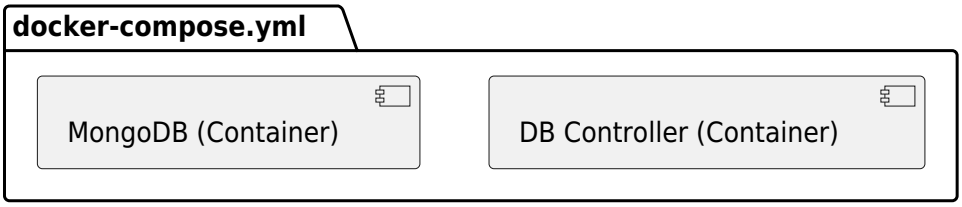
Klassendiagramm User



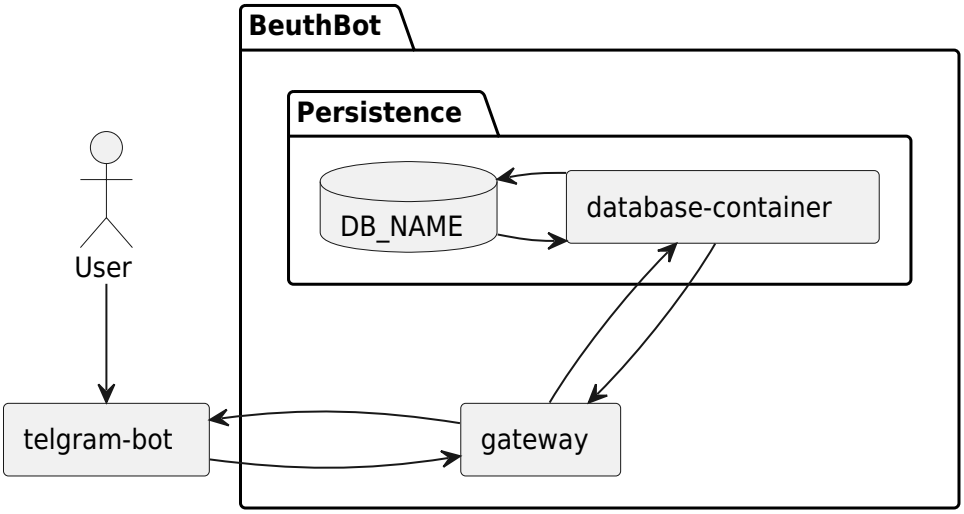
Technologies

Durch die Anforderung, dass die Details, die zu einem User gespeichert werden sehr variabel sein können, ist von einer relationalen Datenbank wie MySQL o.ä. abzuraten. MongoDB ist eine dokumentenorientierte NoSQL-Datenbank. Mit ihr können Sammlungen von JSON-ähnlichen Dokumenten erstellt und verwaltet werden. So können wir die Daten zu einem User in komplexen Hierarchien verschachteln und erweitern ohne uns Gedanken zu einem Tabellen-Schema machen zu müssen.

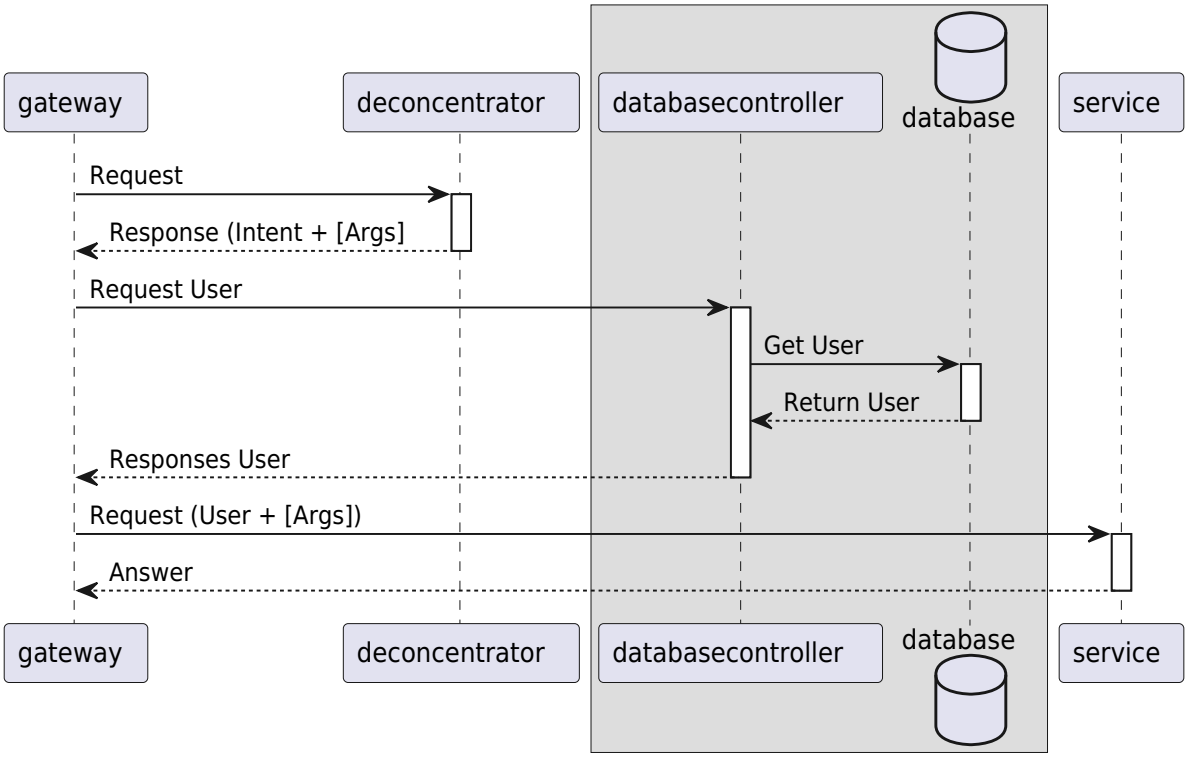
- MongoDB Link
- MongoDB Docker Image Link



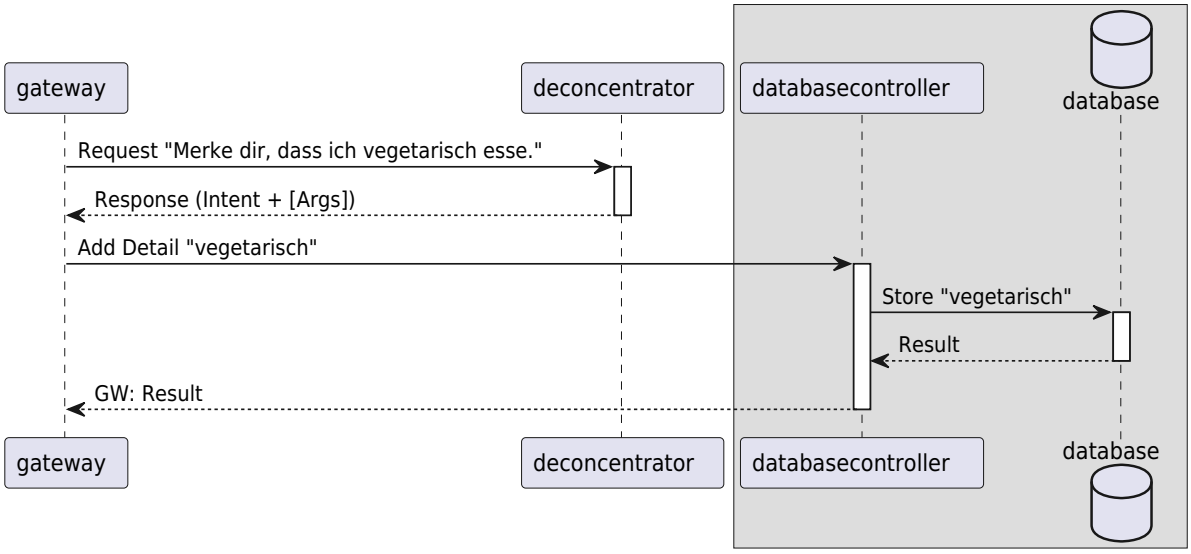
Integration



Sequenzdiagramm mit angesteuertem Service



Sequenzdiagramm nur Datenbank betreffend



Nächsten Schritte

- Projekt Ordner und GitHub Repository erstellen
- DB dem BeuthBot Projekt hinzufügen
- Geignete Dockerfile formulieren mit MongoDB als Abhängigkeit
- Datenbankcontroller erstellen, welcher ADD, REMOVE, CHANGE Befehle für Details entgegen nimmt
- Trainingsmodell für RASA für die Datenbank erstellen
- Erste versuche mit dem Trainingsmodell von RASA

Getting Started

Die Datenbank wurde mit Docker erstellt. Um diese zum laufen zu bringen müssen folgende Befehle ausgeführt werden:

```
# clone the repository
git clone https://github.com/beuthbot/database.git

# go to the folder
cd database

# start the docker container to run the mongodb and its
corresponding database microservice
docker-compose up
```

Windows

Damit es auf Windows funktionieren kann müssen folgende Zeilen in der docker-compose.yml Datei geändert werden:

```
...
    volumes:
      - mongodata:/data/db # needed for me to run container on
Windows 10
      #- ./../.database:/data/db # For Mac/Linux
...
# needed for me to run container on Windows 10
volumes:
  mongodata:
```

Außerdem muss ein shared Folder existieren, welcher beispielsweise 'mongodb' genannt werden muss, worin sich der Ordner 'data' mit den Unterordnern 'db' und 'configdb' befindet. Die Ordnerstruktur sollte nun wie folgt aussehen:

```
E:\mongodb
    └───data
          ├───configdb
          └───db
```

API

Request all Users

Requests all Users in the collection

```
GET http://localhost:27000/users
```

Response

```
{...},
{
  "id": 12345678,
  "nickname": "Alan",
  "details" : {
    "eating_habit" : "vegetarisch",
    "city" : "Berlin"
  }
},
{...}
```

Error

```
{
  "error": ...
}
```

Request User

```
GET http://localhost:27000/users/<id>
```

Reponse

Request a single user with the given id.

```
{
  "id": 12345678,
  "nickname": "Alan",
  "details" : {
    "eating_habit" : "vegetarisch",
    "city" : "Berlin"
  }
}
```

Error

```
{
  "error": ...
}
```

Add / Change Detail

Add/Change a Detaile to/from the User with the given id.

```
POST http://localhost:27000/users/<id>/detail
```

Request Body

```
{
  "detail": "eating_habit",
  "value": "vegetarisch"
}
```

Reponse

If the operation was successful the error will be set to null and the success will be set to true. If the operation failed an error message will be set and the success will be set to false.

```
{
```

```
  "error": null,
  "success": true | false
}
```

Delete all Details

Deletes all Details from the User with the given id

```
DELETE http://localhost:27000/user/<id>/detail?q=<value>
```

Reponse

If the operation was successful the error will be set to null and the success will be set to true. If the operation failed an error message will be set and the success will be set to false.

```
{
  "error": null,
  "success": true | false
}
```

Delete Detail

Deletes one Detail from the User with the given id.

```
DELETE http://localhost:27000/user/<id>/detail?q=<value>
```

Reponse

If the operation was successful the error will be set to null and the success will be set to true. If the operation failed an error message will be set and the success will be set to false.

```
{
  "error": null,
  "success": true | false
}
```

Database Microservice

Database Microservice

Inhaltsangabe

a. Motivation
b. Technologien
c. Funktionsweise
d. API

Motivation

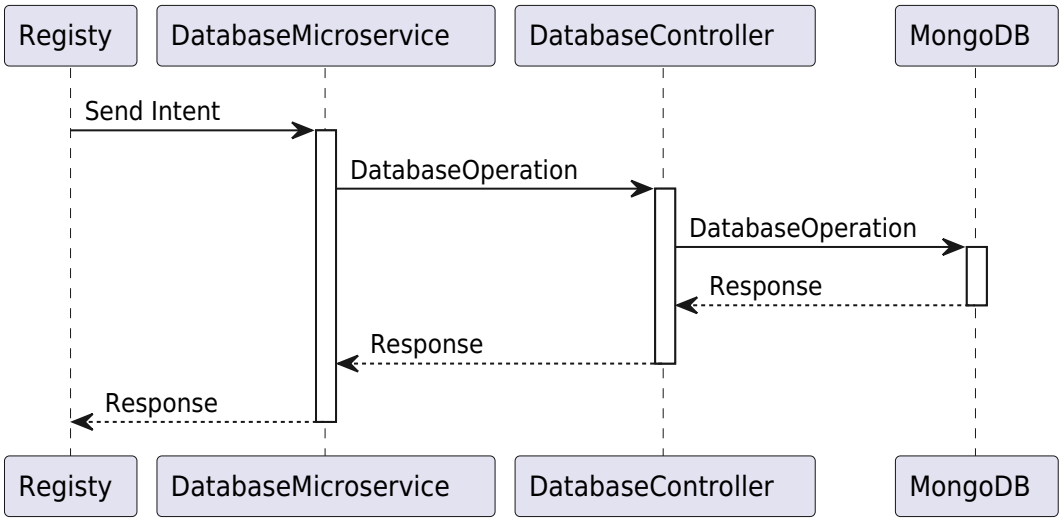
Um die Datenbank unabhängig von den anderen Microservices zu machen, mussten die Datenbank Operationen ausgelagert werden. Das führte dazu, dass die Intents von Rasa aufgelöst werden mussten, damit die richtigen Datenbank Operationen ausgeführt werden können.

Technologien

Aufgebaut wurde dieser Microservice als REST-Server mit JavaScript. Die verwendeten Technologien dafür sind:

- NodeJS
- ExpressJS
- Axios

Funktionsweise



Da dieser Microservice nur eine Route besitzt, über welche der Intent gesendet wird, muss der Microservice dazu in der Lage sein, diesen Intent aufzulösen, sodass die richtige Datenbank-Operation ausgeführt wird. Der Intent kommt ursprünglich von Rasa. Dieser

sieht wie folgt aus:

```
{
   "user": {
      "id": 12345,
      "telegram-id": 12345,
      "nickname": "Al",
      "details": {
         "home": "Bonn",
         "birthday": "23.06.1912",
         ...
      }
   },
   "intent":{
      "name":"database-set",
      "confidence":0.9998944998
   },
   "entities":[
      ...,
      {
         "start":26,
         "end":36,
         "value":"krebstiere",
         "entity":"allergen",
         "confidence":0.9999893608,
         "extractor":"CRFEntityExtractor"
      },
      {
         "start":37,
         "end":51,
         "value":"alergisch bin.",
         "entity":"detail-allergic",
         "confidence":0,
         "extractor":"CRFEntityExtractor"
      },
      ...
   ],
   "text":"Merke dir, dass ich gegen Krebstiere alergisch bin.",
   ...
}
```

Durch den Intent, erfährt man, welche Operation ausgeführt werden soll und in den Entities steht drin, was hinzugefügt/gelöscht/ausgelesen werden soll, sowie von welchem User diese Operation ausgeführt werden soll. Die Entity mit dem höchsten Confidence-Score ist die gewollte Anfrage an die Datenbank.

API

Request

```
POST https://localhost:<PORT>/resolve
```

oder

```
POST https://localhost:<PORT>/database
```

Response

```
{
  answer: {
    content: 'Deine Daten:\n' +
      '\n' +
      'Nickname: **DennySchumann**\n' +
      'Vorname: **Denny**\n' +
      'Nachname: **Schumann**\n' +
      '\n' +
      'home: **köln**\n',
    history: [ 'intent-resolve' ]
  }
}
```

RASA Trainieren

Neue Funktionen oder Microservices müssen dem BeuthBot „beigebracht“ werden. Konkret für Rasa heißt das, dass es neue Trainings-Daten braucht aus denen das Model generiert werden kann. Dieses Model nutzt Rasa zur Laufzeit um Anfragen zu interpretieren. Am Anfang des Semesters haben wir bereits ein funktionierendes Model und die dazugehörigen Trainings-Daten vorgefunden. Die Trainings-Daten wurde mit Hilfe von Tracy generiert. Tracy wird mit einem Web-Interface bedient. Man kann dort Sätze und Entities eingeben aus denen sich Trainings-Daten generieren lassen welche dann exportiert werden können. Die Daten wurden damals manuell eingegeben. Als wir das Model erweitern wollten, hätten wir diese Daten wieder manuell eingeben und ergänzen müssen. Da dies nicht praktikabel erschien haben wir nach alternativen Lösungen gesucht und eine gefunden. Chatito ist ein Tool mit dem wie bei Tracy Trainings-Daten generiert werden können. Der Unterschied ist das bei Chatito die Daten nicht manuell über ein Web-Interface eingeben werden, sondern mit einer DSL (Domain Specific Language) in *.chatito-Dateien definiert werden. Chatito generiert dann aus einer beliebigen Anzahl gegebener .chatito-Dateien die Trainings-Daten welche dann von Rasa genutzt werden können um das Model zu erstellen. Die Chatito Dateien liegen im Rasa Projekt im Ordner /training/app/input.

Eine Anleitung zum Trainieren eines neuen Models ist hier in diesem Wiki. Die gleiche Anleitung und mehr Informationen befinden sich in der TRAINING.md Datei des Rasa Projektes.

Cache

cache

Motivation

Der Cache soll vorrangig die Microservices entlasten, indem er die Response zwischenspeichert und der Registry für eine gewisse Zeit zur Verfügung stellt. Insbesondere der Service Weather ist davon betroffen, da dieser eine API von OpenWeatherMap nutzt, welches pro Tag 4000 Wettervorhersagen treffen kann, sonst wird dieser Kostenpflichtig bzw. kann dann keine Requests mehr entgegen nehmen.

Free	Startup **40 USD** / month	Developer **180 USD** / month	Professional **470 USD** / month	Enterprise **2.000 USD** / month
60 calls/minute **1,000,000 calls**/month	**600 calls**/minute **10,000,000 calls**/month	**3,000 calls**/minute **100,000,000 calls**/month	**30,000 calls**/minute **1,000,000,000 calls**/month	**200,000 calls**/minute **5,000,000,000 calls**/month
Current Weather Minute Forecast 1 hour∗ Hourly Forecast 2 days∗ Daily Forecast 7 days∗ Historical weather 5 days∗ Climatic Forecast 30 days Bulk Download	Current Weather Minute Forecast 1 hour∗∗ Hourly Forecast 2 days∗∗ Daily Forecast 16 days Historical weather 5 days∗∗ Climatic Forecast 30 days Bulk Download	Current Weather Minute Forecast 1 hour Hourly Forecast 4 days Daily Forecast 16 days Historical weather 5 days Climatic Forecast 30 days Bulk Download	Current Weather Minute Forecast 1 hour Hourly Forecast 4 days Daily Forecast 16 days Historical weather 5 days Climatic Forecast 30 days Bulk Download	Current Weather Minute forecast 1 hour Hourly Forecast 4 days Daily Forecast 16 days Historical weather 5 days Climatic Forecast 30 days Bulk Download
Basic weather maps Historical maps	Basic weather maps Historical maps	Advanced weather maps Historical maps	Advanced weather maps Historical maps	Advanced weather maps Historical maps
Weather triggers	Weather triggers	Weather triggers	Weather triggers	Weather triggers
Weather widgets	Weather widgets	Weather widgets	Weather widgets	Weather widgets
Uptime 95%	Uptime 95%	Uptime 99.5%	Uptime 99.5%	Uptime 99.9%

∗ - 1,000 API calls per day by using One Call API

∗∗ - 2,000 API calls per day by using One Call API

Requirements

Die Registry soll Requests von dem Deconcentrator entgegennehmen und überprüfen, ob diese Request innerhalb einer fest definierten Zeit bereits eine Response erhalten hat. Ist dies der Fall guckt die Registry in den Cache, um sich die dort Zwischengespeicherte Response zu holen und diese an den Sender der Request zu leiten. Dabei “ersetzt” der Cache den angesprochenen Microservice. Ist dies allerdings nicht der Fall wendet sich die Registry weiter an den angesprochenen Microservice und speichert dessen Response in den Cache.

Functional

- `/CAF100/` The system must check if the requested resource is available in the cache before relaying the request to a microservice.
- `/CAF100/` The system must place the response of a microservice in the cache.
- `/CAF200/` The cache must offer an option to save a response of a microservice.
- `/CAF201/` The cache must offer an option to retrieve a saved response.
- `/CAF202/` The cache must automatically delete a saved response if the given timeout has been exceeded.

Non Functional

- `/CANF100/` The system must answer faster with a cached response than if a request is relayed to a microservice.
- `/CANF200/` The cache must save at least 1000 Responses.
- `/CANF201/` The cache must answer in at least 5ms.

User Stories

- `/CAUS100/` Als Betreiber möchte ich Anfragen die das selbe Ergebnis erzeugen abfangen und damit die Microservices entlasten.
- `/CAUS101/` Als Betreiber möchte ich die Anfragen an die verschiedenen APIs reduzieren um nicht in ein teureres Preispaket zu fallen.

Use Cases

Technologies

Für Node.js existieren mehrere Caching Lösungen. Bei den ersten recherchen fielen die npm packages “memory-cache” und “node-cache” auf. Da “memory-cache” seit drei Jahren kein Update bekommen hat, haben wir uns letzten endes für “node-cache” entschieden.

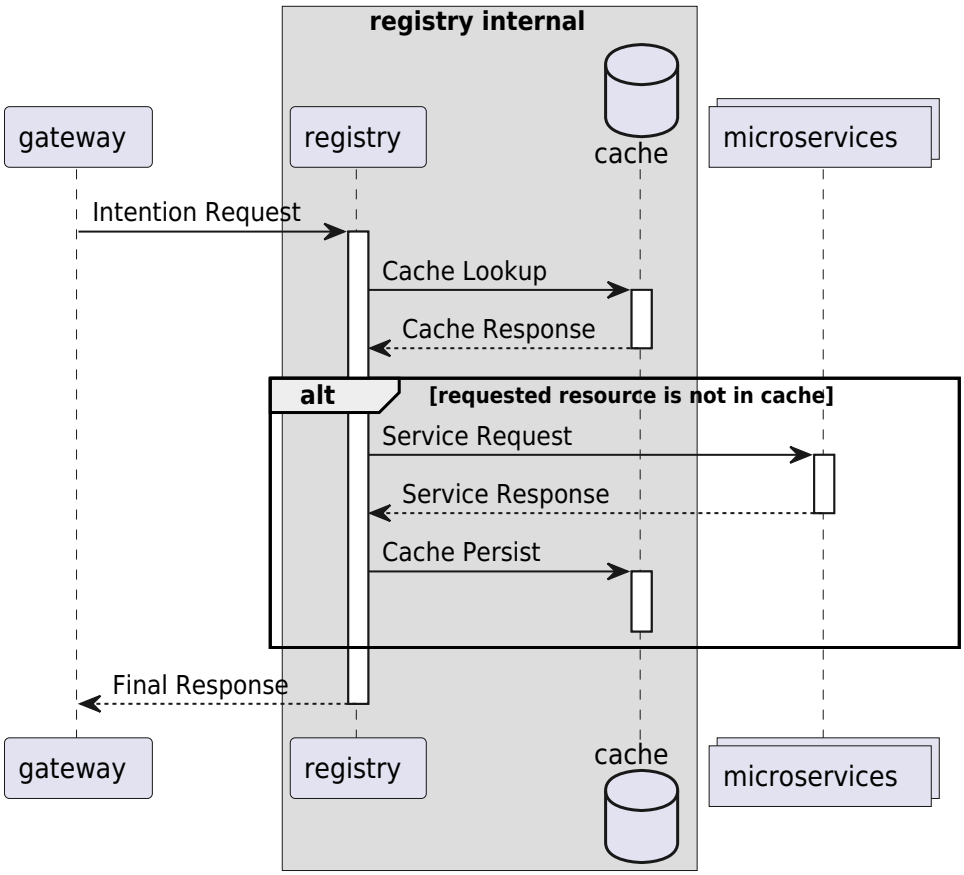
“node-cache” ist eine simple Caching Lösung, die nach dem Key-Value prinzip funktioniert. Der Funktionsumfang besteht dabei aus den Methoden “set”, “get” und “delete”, wobei die Methode “set” einem zusätzlich erlaubt noch einen Timeout (“ttl” bzw. “time to live” genannt) zu übergeben. Ist der Timeout überschritten, wird der Eintrag

automatisch aus dem Cache gelöscht. Der Nachteil dieser Lösung ist, dass nur eine Millionen einträge pro Cache Instanz eingetragen werden können. Da aber gleich viel in den Cache eingetragen wird, wie die Anzahl angebotener Funktionen aller Microservices, wird dieser Nachteil nicht eintreffen.

Integration

Der Cache wird wie schon in Technologies beschrieben in Node.js verwendet. Spezifisch wird der Cache dem „registry“ Server hinzugefügt.

Das Resultat (veranschaulicht im folgenden UML diagram) besteht darin, dass registry versucht die angefragte Ressource aus dem Cache zu holen und gegebenfalls eine Anfrage an den entsprechenden Microservice zu stellen, falls die Ressource nicht im Cache vorhanden ist.



Resultate

Die momentane Implementierung ist wie zuvor beschrieben umgesetzt. Der einzige Unterschied besteht darin, dass die Microservices die Möglichkeit besitzen, einen ttl mitzuschicken. Wird kein ttl vom Microservice mitgeschickt, so wird ein Standard ttl (momentan 30 Minuten) verwendet.

Wenn ein Microservice einen ttl mitschicken möchte, so muss dem „answer“ Object lediglich ein integer namens „ttl“ hinzugefügt werden. Dieser repräsentiert die Anzahl an Sekunden, wie lang zwischengespeichert werden soll.

Update Wetter Microservice

Beschreibung

Der Wetterservice ist ein Dienst, welche aktuell über den Telegram Client angesprochen wird. Dabei kann der Service per Chat einfach angesprochen werden ohne das der User im Vorfeld etwas einrichten muss. Dieser Dienst antwortet dabei auf Fragen, welche das Wetter im Allgemeinen, aber auch im Stündlichen betreffen, ebenso wie das vergangene Wetter. Aktuell ist eine Anfrage von bis zu 7 Tage in der Zukunft, sowie 5 Tage in der Vergangenheit möglich. Ab 47 Stunden in der Zukunft kann der Bot nur noch mit dem allgemeinen Wetter antworten, sprich keine stündliche anfrage.

Verwendete Technologien

Node.js(https://nodejs.org/en/) Express.js(https://expressjs.com/de) Axios.js(https://www.npmjs.com/package/axios) OpenWeatherMap(https://openweathermap.org)

Open Weather Api

Open Weather Map ist eine Api, welche einem die Möglichkeit bietet Wettervorhersagen abzurufen. Dabei bietet sie 4 verschiedene Arten von Calls: /weather → Aktuellen Wettervorhersage /onecall → Aktuellen Wettervorhersage, bis 7 Tage in die Zukunft /forecast → Wettervorhersage bis zu 5 Tage in die Zukunft /history → Wettervorhersage bis zu 5 Tage in die Vergangenheit In dieses Service wurden /onecall und /history verwendet, da /forecast nur 5 Tage in die Zukunft geht und auch nur in 3 Stunden abständen. Ebenso wurde /weather nicht genommen, das nur das aktuelle Wetter zurückgibt, welches in /onecall schon enthalten ist. Um diese Api nutzen zu können muss erst einmal ein Key auf der Seite generiert werden, welcher dann am Ende jeweils als Parameter an die URL angehängt werden muss.

Geoservice Erweiterung des Wetter Microservices

Geo Service

Motivation

Die OpenWeatherMap API akzeptiert nur Koordinaten (sprich Longitude & Latitude). Daher muss die von Rasa übergebene Entity namens „city“ in Koordinaten umgewandelt werden. Ursprünglich wollten wir Rasa um diese Funktion erweitern, aus verschiedenen Gründen haben wir uns letztenendes dafür entschieden, die Umwandlung im Wetterservice durchzuführen.

Requirements

Der Wetterservice soll um einen Service erweitert werden, der die „city“ Entity von Rasa in Koordinaten umwandelt. Ist keine „city“ Entity vorhanden, soll der Wohnort des Users („home“ Wert aus der Datenbank) verwendet werden. Falls weder Entity noch Wohnort vorhanden ist, soll „Berlin“ als Standardwert genutzt werden. Um diesen Wert umzuwandeln muss der Service dann diesen Wert an die Nominatim API von OpenStreetMap schicken und soll danach die Response auswerten. Erhält der Service ein leeres Array (sprich keine Koordinaten) oder einen Fehler, so soll eine Fehlermeldung zurückgegeben werden.

Technologies

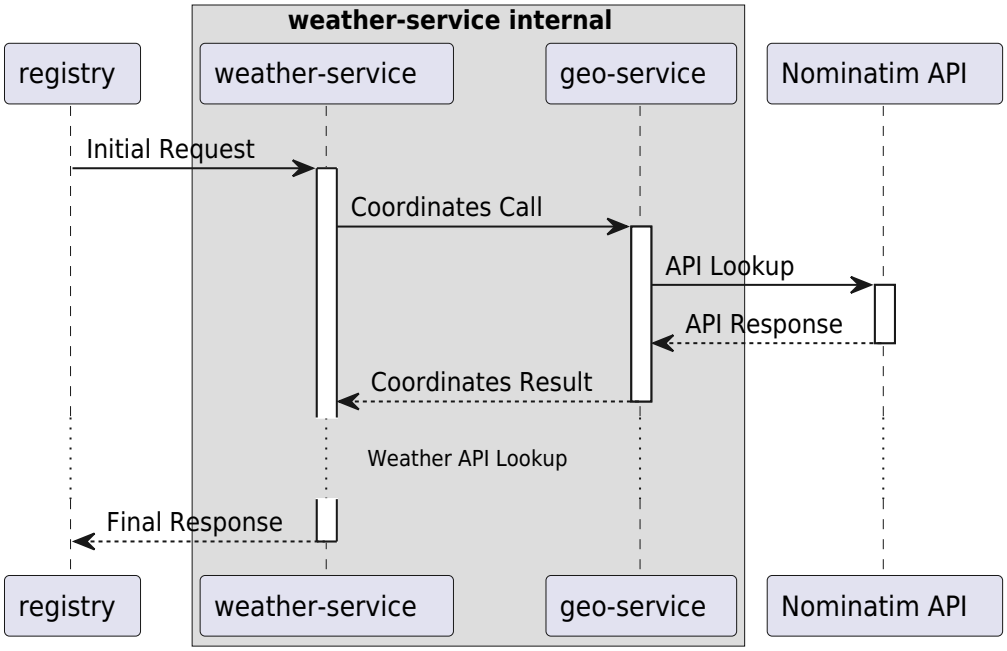
Um einen String in Koordinaten umzuwandeln, existieren mehrere Lösungen. Bei den Recherchen stachen vorallem „Maps“ von Google und „Nominatim“ von OpenStreetMaps hervor. Im Endeffekt haben wir uns für Nominatim entschieden, da diese eine Open-Source Alternative zu Maps darstellt, durch die Offenheit einen niedrigschwelligen Einstieg gewährt und damit ein unkompliziertes Nutzen der API ermöglicht. Des weiteren bietet Nominatim zusätzlich einen Docker Container an und kann damit auch Lokal benutzt werden.

“Nominatim” ist eine Geocoding Lösung, mit der man einen beliebigen String in Koordinaten umwandeln kann. Der String kann dabei zum Beispiel aus einem Stadtnamen oder einem Firmennamen bestehen. Der Service sucht anhand des Strings alle relevanten Koordinaten aus deren Datenbank und gibt diese in einem Response zurück. Des weiteren, kann in der Anfrage bestimmt werden, in welchem Datenformat die Koordinaten verpackt werden sollen. Wenn anhand des Strings keine Einträge in der Datenbank gefunden wurden, übergibt die API ein leeres Array.

Integration

Wie schon in Motivation beschrieben, wird der Wetterservice Server um den Geoservice erweitert.

Das Resultat (veranschaulicht im folgenden UML diagram) besteht darin, dass der Wetterservice einen String an die Nominatim API schickt und den Response auswertet.



Further Reading

- Nominatim API

weather.js

Hier wird die Request vom Service registry aufgenommen und eine Response erstellt und gesendet. Als erstes wird die Request in message und Ort aufgeteilt. Daraufhin wird der Ort, welcher in der Request mitgesendet wurde in Längen.- und Breitengrad via eines Geoservices umgerechnet und wieder gegeben. Dies muss passieren, da die OpenWeatherMap Api bei einigen Anfragen nicht die Ortschaft selbst, sondern nur die Latitude und longitude nimmt. Ist kein Ort in der Request vorhanden wird standardmäßig Berlin oder der durch die Persistenz abgespeicherte Wohnungsort genommen. Daraufhin wird geprüft, ob die Request eine Uhrzeit enthält, um festzustellen ob eine allgemeine oder eine zeitspezifische Anfrage gestellt wurde. Bevor die Bearbeitung anfängt, wird vorher festgestellt, ob das Angefragte Datum auf dem Rahmen fällt, sprich z.B. zu weit in der Zukunft. Ist alles Regelkonform, wird festgelegt, in welchem Zeitraum das Wetter angefragt wird, welches dann an den richtigen Codeblock weitergeleitet wird. Im Block angekommen wird als erstes immer ein Api Call an OpenWeatherMap über die Datei weatherService mit dem Datum und den Koordinaten getätigt, welches dann in der Variable weather gespeichert wird. Danach werden weather(Response vom api call), date(Datum) und city(Stadtname) an generatedMessage und die jeweilige Methode weitergeleitet, welche dann die fertige Nachricht zurückgibt. Diese wird letztendlich als Response an des User weitergeleitet.

weatherService.js

Diese Datei besitzt zwei Methoden, welche jeweils mit axios einen Api call an OpenWeatherMap machen und diesen zurückgeben. Beide Funktionen benötigen jeweils die Koordinaten des gefragten Bereiches. Die Methode getForecast gibt ein JSON zurück in

welchem das aktuelle, das stündliche und das tägliche Wetter befindet, welche dann später in generatedMessage.js wiederverwertet werden. Die Funktion getHistory hingegen benötigt noch die spezifische Zeit als long Wert und gibt ebenfalls ein JSON zurück, allerdings nur mit dem exakt gefragten Moment, sowie ein Objekt mit den Stündlichen vergangenen Wetterdaten.

generatedMessage.js

Diese ist die zuletzt aufgerufene Datei, in welcher drei Methoden existieren, welche den Response Text für Telegram zusammenstellen. Jede Funktion benötigt die Parameter weather(Response vom weatherService.js api call), date(Datum) und city(Stadtname). Es müssen diverse Zeitformate erstellt, bzw. angepasst werden, welche dann in der Response eingefügt werden. Ebenso befindet sich in der Response von OpenWeatherMap ein Icon String, welches dann in folgende Text Icons umgewandelt wird: ☀, □, ☁, □, □, □, ❄, □. Diese ganzen umgewandelten Daten werden daraufhin in einen fest definierten String integriert und als messageText zurückgegeben.

Telegram Weather Request and Response

Allgemeine Wetteranfrage

Dies ist eine Chatanfrage ohne dabei einen spezifischen Ort zu nennen. Wenn keine Ort genannt wird, wird standardmäßig der Sitzt der Beuth Hochschule, also Berlin genommen oder aber der User hat seinen Wohnort gesetzt, dann wird dieser genommen, es seiden die Geo Api kenn diesen nicht.

Mit spezifischer Uhrzeit

Bei dieser Chatanfrage wurde zusätzlich die Uhrzeit mit angegeben. Wenn es sich um die nächsten 47 Stunden handelt, dann wird dementsprechend ein Response mit der Uhrzeit gesendet. Überschreitet das Datum allerdings 47 Stunde, so wird es wie eine allgemeine Anfrage gehandhabt.

Ort ohne Uhrzeit

Wir ein Ort, allerdings keine Uhrzeit mit geschickt, so handhabt der Bot die Anfrage wir im ersten Bild, nur das er denn mitgesendeten Ort nimmt, es seiden die Geo Api kennt diesen nicht, dann wir standardmäßig Berlin genommen.

Ort mit Uhrzeit

Wird neben der Uhrzeit noch ein Ort gesendet und überschreitet es nicht 47 Stunden in die Zukunft, kann der Bot da Wetter an dem Ort mit der spezifisch mitgesendeten Uhrzeit zurückgeben, vorausgesetzt die Geo Api kennt diesen auch.

Complete Sequence Diagram