Abschlussbericht zum Masterprojekt SS 2020

Inhalt

Abstract

Uebersicht / Inbetriebnahme

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 Microservice

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

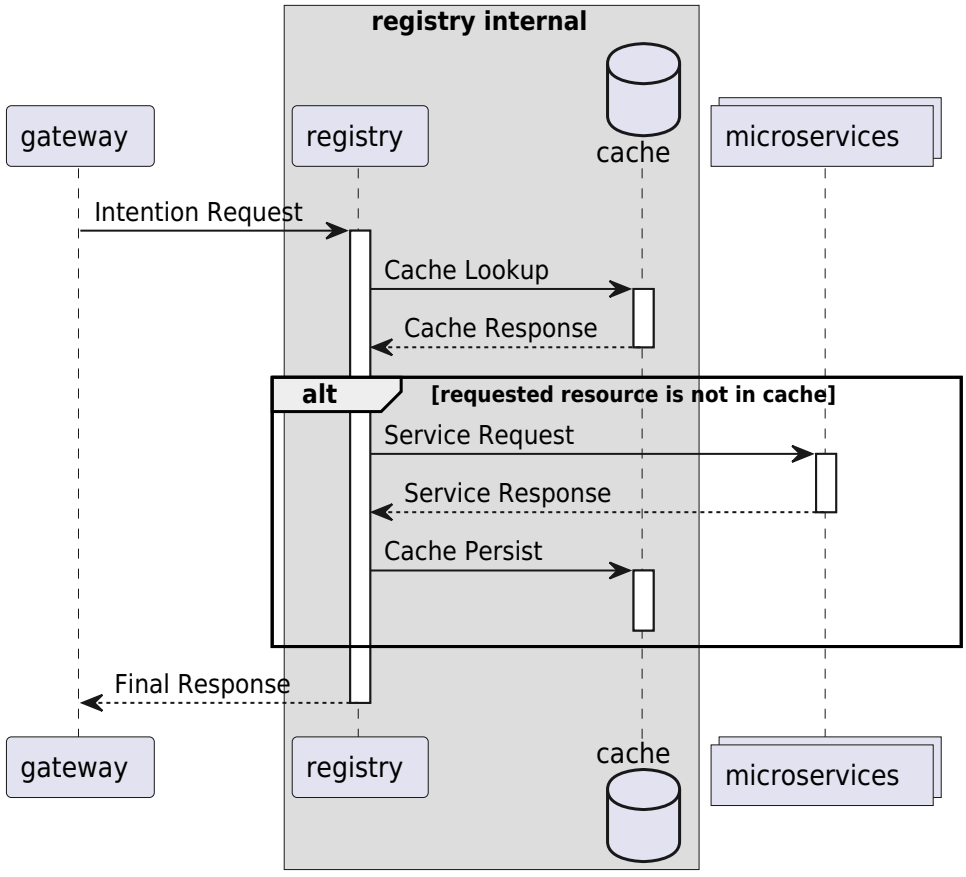
Für Node.js existieren mehrere Caching Lösungen. Bei den ersten recherchen fielen die npm packages “memory-cache” und “node-cache” auf. Da “memory-cache” seit drei Jahren kein Update bekommen hat, haben wir uns letzten endes für “node-cache” entschieden.

“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

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.

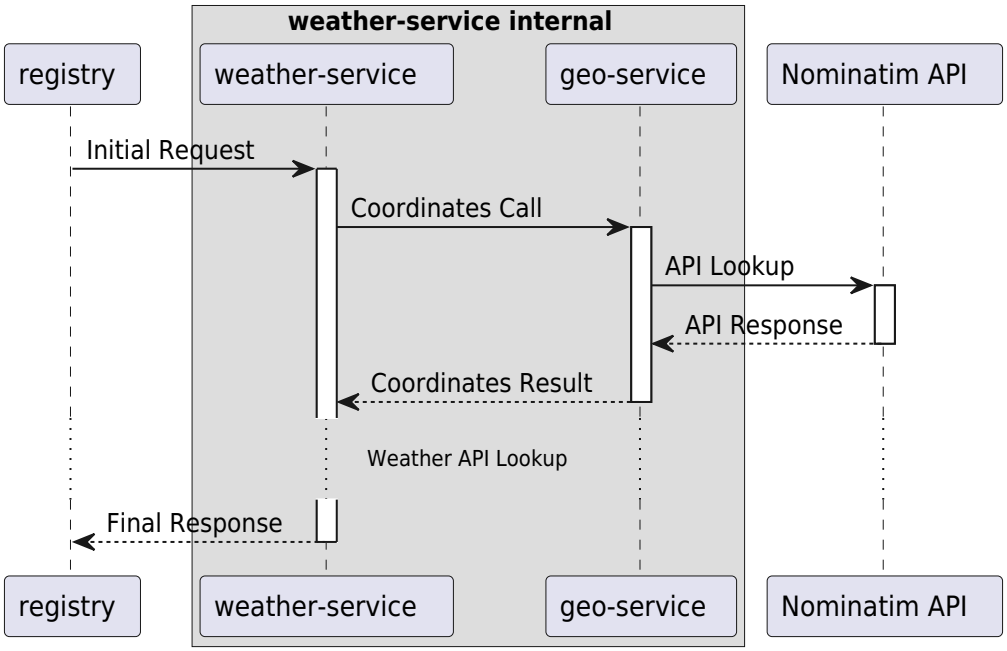
“Nominatim” ist eine Geocoding Lösung, mit der man einen beliebigen String in Koordinaten umwandeln kann. Der String kann dabei zum Beispiel aus einem Stadtnamen oder einem

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