Zwischenbericht WS2020/21

BeuthBot Project Group

- Lukas Danke
- Robert Halwaß
- Rim Khreis
- Alexis Popovski
- Dennis Walz

Gliederung

- Einleitung
- Tools
- Organisation
- Vorgefundener Stand
- Aktueller Stand
- Bisher aufgetretene Probleme
- Geplante Aufgaben

Einleitung

Chatbots sind ein logisches Produkt der heutigen Automatisierungsära. Sie können theoretisch für fast jedes Unternehmen nützlich sein, da ein erheblicher Teil von Benutzeranfragen ähnlich sind und wiederholt gestellt werden. Mit der schnellen Entwicklung des maschinellen Lernens, stellen Chatbots ein relevantes und aktuelles Thema dar, welches immer beliebter und nachgefragter wird. Weshalb also nicht auch ein Chatbot der Beuth Hochschule für Technik Berlin? Entwickelt von Studenten, für Studenten. Somit können nicht nur die Mitarbeiter des Studierendenservice entlastet, sondern auch kleine bis große Fragen der Studenten schnell beantwortet werden. Ein intelligenter Chatbot, der per Text und Sprache bedient werden kann, mittels künstlicher Intelligenz antwortet und vielen Features, wie z.B. sich vom Chatbot an wichtige Abgabetermine erinnern zu lassen.

Tools

In diesem Kapitel werden alle Tools aufgelistet und beschrieben, welche während des Projektverlaufes zum Einsatz kommen.

Jitsi

Jitsi ist ein OpenSource Video-Conferencing-Tool, welches im Rechenzentrum der Beuth-Hochschule gehostet wird. Dadurch ist der Datenschutz gewährleistet. Das Tool wird für die regelmäßigen wöchentlichen digitalen Treffen zwischen dem Betreuer und dem Projektteam zum Austausch von Informationen und Fragen genutzt.

Telegram

Telegram ist ein kostenloser, verschlüsselter Text- und Sprachnachrichten-Dienst, welcher sowohl auf mobilen Geräten als auch Desktop-Ansicht funktioniert. Auch über Telegram kommuniziert das Projektteam über Textnachrichten mit dem Betreuer, falls, in den Tagen vor oder nach dem Meeting in Jitsi, Fragen aufkommen, welche schnellstmöglich beantwortet werden müssen.

Discord

Discord ist ebenfalls ein kostenloser Onlinedienst mit Chat-, Sprachkonferenz- und Videokonferenz-Funktion. Das Projektteam nutzt Discord, um sich, nach den wöchentlichen Treffen mit dem Dozenten, nochmal untereinander auszutauschen und auch an anderen Wochentagen zusammenzutun um sich zu organisieren.

Jira

Jira ist eine von Atlassian entwickelte Webanwendung, welche dem Projektmanagement bzw. Aufgabenmanagement dient. In Jira verfasst das Projektteam die Anforderungen des Projekts in kleineren Tasks, um diese wöchentlich in konstanten Schritten abzuarbeiten. Außerdem werden dort bislang die Stundenaufwände der einzelnen Projektmitglieder dokumentiert.

Ziemer's Wiki

Im Wiki befindet sich eine große Sammlung an Berichten, Dokumentationen und Information zum BeuthBot und die Arbeit der vorherigen Semester an diesem. Diese große Ansammlung dient dem Projektteam zur Einarbeitung in das Projekt bzw. dem BeuthBot, aber auch zur Inspiration, wie sie bestimmte Dinge angehen und gestalten können. Des Weiteren hält das Projektteam in Ziemer's Wiki allgemeine Notizen zum Überblick, aber auch natürlich den Zwischenbericht und andere Dokumentationen fest.

IDEs

Im folgenden werden die verschiedensten Entwicklungsumgebungen aufgelistet, welches jedes Projektmitglied nach seinen Vorlieben und Präferenzen nutzt, um am Code des BeuthBot's zu arbeiten und die Anforderungen umzusetzen.

Visual Studio Code

IntelliJ

WebStorm

Sonstiges

Postman

Postman ist ein Programm zum Entwickeln, aber auch zum Testen von bereits bestehenden REST API's. Mit diesem Tool kann das Projektteam die Antwort auf ihre Anfragen überprüfen.

Organisation

Jede Woche (Donnerstag 10:00) findet zwischen den Studierenden und dem Dozenten eine Rücksprache in Form eines Videochats mittels Jitsi statt. Bei diesem Treffen werden die Ergebnisse der letzten Woche besprochen, mögliche Unklarheiten geklärt und Ziele für die folgende Woche gesetzt.

Zur allgemeinen Kommunikation außerhalb der wöchentlichen Rücksprachen zwischen Studierenden und Dozent wird Telegram verwendet, um kleinere Fragen zu klären, welche nicht bis zum nächsten Rücksprachetermin warten können. Da der BeuthBot zum Start des Projektes nur Telegram unterstütze, erwies sich dies als logische Wahl.

Zur privaten Kommunikation unter den Studierenden wird Discord verwendet. Discord bietet den Vorteil, dass sowohl Text-Chat als auch Voice- und Video-Chat möglich ist. Dies bietet viel Flexibilität bei der Kommunikation innerhalb des Teams. Der Server wird privat gehostet, dadurch ist der Datenschutz hier ebenfalls gewährleistet. Viele der Studierenden haben auch bereits Erfahrung bei der Verwendung von Discord, wodurch keine lange Einarbeitungszeit notwendig war. Zusätzlich wurde sich auch dafür entschieden, den BeuthBot ebenfalls über Discord zugänglich zu machen.

Vorgefundener Stand

Der Beuthbot besteht aus mehreren ineinandergreifenden Microservices, die über eine umfassende API miteinander kommunizieren. Durch diesen gewählten Ansatz lassen sich jederzeit weitere Microservices integrieren. Die Basis stellen folgende 4 Komponenten dar:

- Bot
- Gateway
- Registry
- Services

Bot

Hierbei handelt es sich um eine Abstraktion der verfügbaren Chatbots unterstützter Messaging-Dienste. Der Nutzer interagiert mit diesem Microservice, indem er Anfragen stellt und Antworten des Beuthbots erhält.

Gateway

Das Gateway stellt das Herz des BeuthBots dar. Der Bot informiert das Gateway mit der vom User empfangenen Nachricht, nutzt dann NLP Microservices, um die Bedeutung und Absicht des Nutzers zu erkennen und informiert den entsprechenden Service, um dem Nutzer eine adäquate Antwort zu liefern.

Registry

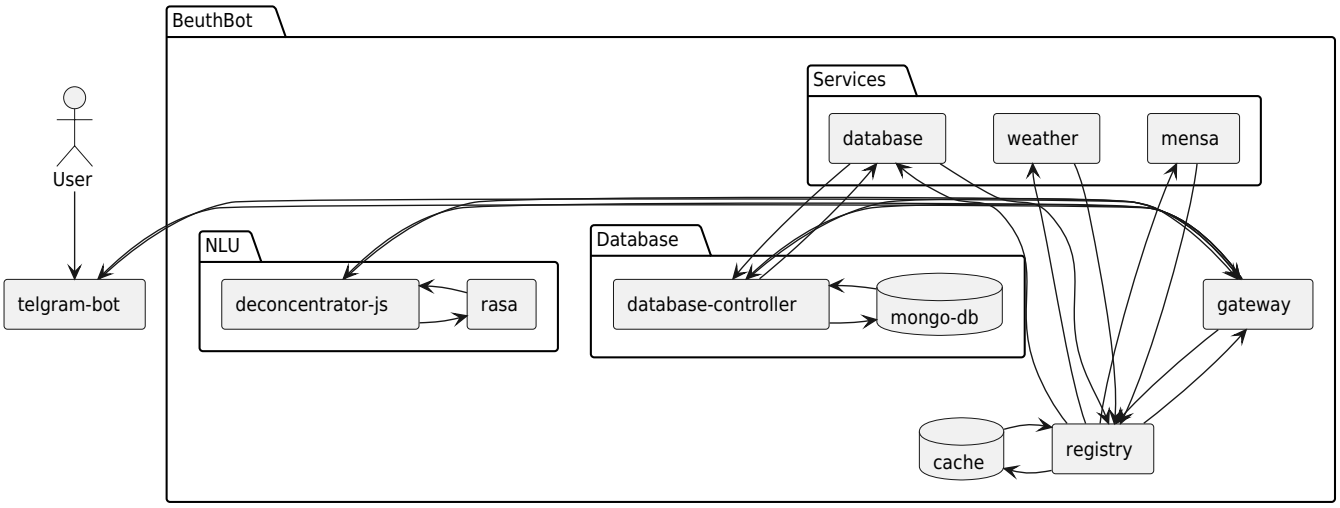
Nachdem die Absicht des Nutzers analysiert worden ist, informiert das Gateway die Registry, um die Informationen zu erhalten, die der Nutzer benötigt. Darauffolgend verteilt die Registry die Anfrage an den entsprechenden Service.

Service

Die Services liefern die seitens des Nutzers angefragten Daten. So liefert bspw. Der MensaService Informationen zu aktuellen Menüs, welche via diverser Parameter gefiltert warden (etwa nur vegetarische Gerichte).

API

Die Mikroservices kommunizieren untereinander mittels einer API. Sie basiert auf einem Response-Objekt, das die einzelnen Microservices durchläuft. Es besteht aus der anfänglichen Anfrage des Nutzers, seiner Daten und der ihm erstellten Antwort.



Aktueller Stand

Der Bot war ständig nicht erreichbar

Der Bot läuft via docker-compose in einer VM. Immer wenn der Bot nicht erreichbar war, startete er neu sobald sich jemand in die VM einloggte und war dann auch wieder erreichbar. Der Grund dafür war, dass docker-compose so konfiguriert war, dass die Container zwar neu starten sollten, aber nicht sofort wenn sie abstürzten (sondern in diesem fall dann eben beim Login durch einen docker-user).

Der Lösung bestand entsprechend in der Änderung der Configuration nach „restart-always“.
https://github.com/beuthbot/beuthbot/pull/3

Gateway funktionierte nicht ohne Telegram-ID

Bei ersten Experimenten ist aufgefallen, dass wenn eine Nachricht an das Gateway geschickt wird und diese keine valide Telegram-ID enthielt, wurde die Nachricht ignoriert. Dieses war entgegen der Dokumentation, welche die Telegram-ID als optional definierte. Da dies für Testzwecke sehr hinderlich ist und im Projektverlauf zwei weitere Messenger (Discord und eigene Webseite) hinzugefügt werden sollen, galt dieses als eines der ersten Probleme die behoben werden sollten.

Durch eine Anpassung des Gateways bei der User-Abfrage wird eine valide Telegram-ID nicht vorausgesetzt. https://github.com/beuthbot/gateway/pull/2

Continous Deployment

Zuvor musste man um eine neue Version des BeuthBots zu deployen, musste manuell die neuste Version von GitHub gepullt werden und der Docker-compos-Befehl ausgeführt werden.

Mithilfe eines Selfhosted-Runners welcher auf dem BeuthBot-Server installiert wurde, ist es jetzt möglich diesen Prozess zu automatisieren. Sobald ein Git-Commit mit einem Versions-Tag gepusht wird, wird dies vom Runner erkannt und der Deploy-Prozess wird angestoßen.
https://github.com/beuthbot/beuthbot/pull/4

Text 2 Speech Recherche

- Say.js
  - https://www.npmjs.com/package/say
  - https://github.com/marak/say.js/
2. Web Speech API
   - https://wiki.selfhtml.org/wiki/JavaScript/Web_Speech
3. Text2Speech
   - https://www.npmjs.com/package/text-to-speech-file

Speech 2 Text Recherche

Mozilla Voice STT (DeepSpeech)

https://github.com/mozilla/DeepSpeech https://github.com/AASHISHAG/deepspeech-german

- Opensource
- Offline nutzbar
- Viel Dokumentation
- Deutsches Modell
- WER: 15%
- Zukunft ungewiss

Kaldi

https://github.com/kaldi-asr/kaldi http://kaldi-asr.org/doc/about.html

- Opensource
- Offline nutzbar
- Deutsche Modelle
- WER: 8,44%

Wav2Letter

https://github.com/facebookresearch/wav2letter

- Opensource
- Offline nutzbar
- Deutsche Modelle
- WER: 4%

Espresso

https://github.com/freewym/espresso

- Opensource
- Offline nutzbar
- Kein deutsches Modell

Nvidea OpenSeq2Seq

https://github.com/NVIDIA/OpenSeq2Seq

- Opensource
- Offline nutzbar
- Kein Deutsches Modell

WER Vergleich 2017

- Google (8%)
- Microsoft (5.9%)
- IBM (5.5%)
- Apple (5%)
- Baidu (16%)
- Hound (5%)

Quelle:
https://askwonder.com/research/current-voice-recognition-word-error-rates-google-amazon-microsoft-ibm-apple-5b88trj0t

Bisher aufgetretene Probleme

Vorüberlegung: Save-Storage / Moodle Integration

Eine initiale Feature-Idee für dieses Semester war es, dem Bot eine Moodle-Integration zu implementieren, durch die der Nutzer Ereignisse in Moodle mitgeteilt und an Abgabetermine erinnert werden kann. Moodle bietet hierfür eine REST-API: https://docs.moodle.org/dev/Web_service_API_functions.

Problem 1: Login

Damit user-bezogene Daten abgerufen werden können muss der User sich in Moodle via username + passwort einloggen. Dieser Login kann nicht über die Chatbot-Funktionalitäten erfolgen, da Messenger in der Regel keine Passwort-Eingabe ermöglichen, was darin mündet, dass die Credentials im Cleartext im Chatlog landen.

Lösung: Es benötigt ein Webformular, was den Moodle-Login sicher zentralisiert. Der Bot darf im Chat nur den Link zum Login ausspielen.

Problem 2: Speicherung des Tokens

Der BeuthBot ist ein Studierenden-Projekt. Es gibt derzeit keinen Production-Server, der nicht von Studierenden eingesehen werden kann. Die hohe Fluktuation an „Administratoren“ erzeugt ein hohes Risiko für das „Leaken“ von gespeicherten Credentials.

Gefahren:

1. Böswilliger Entwickler (programmiert daten-abfluss)
2. Gutwilliger Entwickler (macht Programmierfehler)
3. Böswilliger Admin (transferiert/liest persistierte daten)
4. Gutwilliger Admin (dupliziert/transferiert daten als backups)
5. Böswilliger Nutzer (nutzt sicherheitslücken / programmierfehler)

Lösung 1: Das User-Token wird nur im RAM abgelegt

Das Token wird so nie auf die Festplatte geschrieben

1. [x] Böswilliger Entwickler (programmiert daten-abfluss)
2. [✓] Gutwilliger Entwickler (macht Programmierfehler)
3. [✓] Böswilliger Admin (transferiert/liest persistierte daten)
4. [✓] Gutwilliger Admin (dupliziert/transferiert daten als backups)
5. [✓] Böswilliger Nutzer (nutzt sicherheitslücken / programmierfehler)

Nachteil Lösung 1: Die Usability leidet stark, wenn der User sich ständig neu einloggen muss, weil der Bot die Credentials beim Neustart vergisst. Vor allem für Erinnerungs-Features ist das ein Showstopper.

Lösung 2: Das User-Token wird nur persistiert, wenn die Datenbank geschrieben wird.

Die Daten werden beim Speichern mit einem One-Time-Token verschlüsselt. Beim nächsten Start des Dienstes werden die Daten entschlüsselt und die persistente Kopie gelöscht

1. [x] Böswilliger Entwickler (programmiert daten-abfluss)
2. [✓] Gutwilliger Entwickler (macht Programmierfehler)
3. [x] Böswilliger Admin (transferiert/liest persistierte daten)
4. [✓] Gutwilliger Admin (dupliziert/transferiert daten als backups)
5. [✓] Böswilliger Nutzer (nutzt sicherheitslücken / programmierfehler)

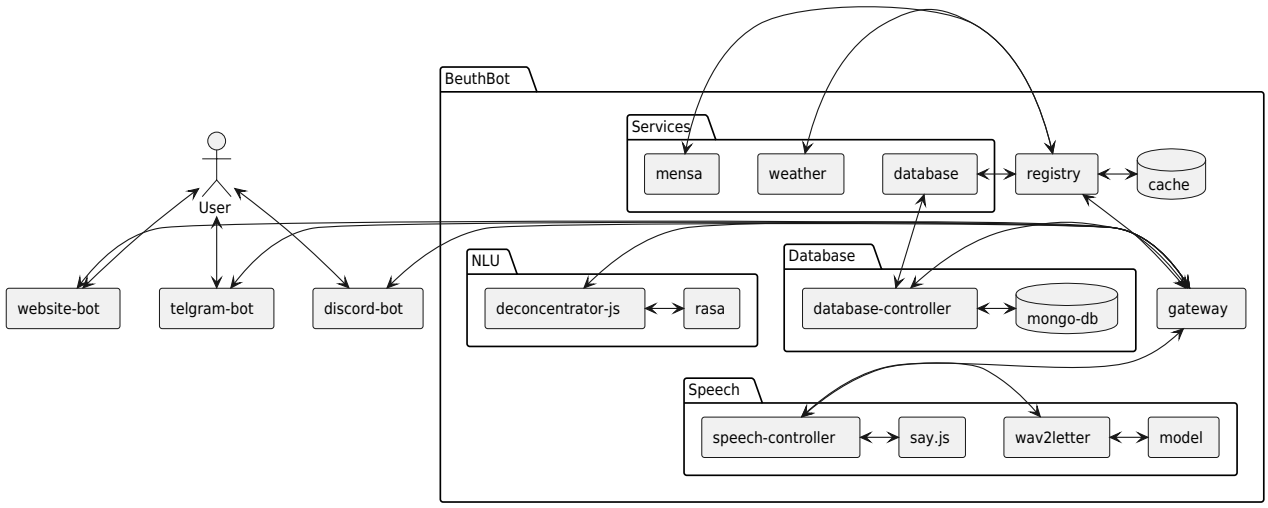
Nachteil Lösung 2: Der One-Time-Key muss auch irgendwie gespeichert werden. Da dieser auch für die Anwendung zugänglich sein muss liegt er gewissermaßen neben der verschlüsselten Datei. Ein cleveres One-Time-Verfahren kann hier zwar dafür sorgen, dass fremder Zugriff auf die Daten nicht unbemerkt bleibt - Gerade Backups können auf diese Art ganz gut abgesichert werden indem der Key dort nicht gespeichert wird. Ein echter Schutz der Daten ist aber nicht gegeben, spätestens der böswillige Admin wird einen Weg finden das Verfahren zu manipulieren

Fazit: Wir haben uns gegen eine Speicherung von User-Credentials entschieden, Solange es kein (professionell administriertes und zugriffsbeschränktes) Produktiv-Environment gibt.

Geplante Aufgaben

ID	Name	Priorität	Initiale Aufwandsschätzung (in Tage)	Abhängig von	Wird verticketet
1	Fix: Gateway funktioniert nicht ohne Telegram-ID	1	1		Lukas Danke
2	Rasa Update auf 2.0	1	1		Robert Halwaß
3	Common Definitions und Funktionen für Bot-Messenger-Clients in Library zusammenfassen	2	1	1	Dennis Walz
4	Discord ChatBot Implementation	2	2	1, 4	Dennis Walz
5	ErinnerungsFeature Command & Erinnerung	2	3	2	Dennis Walz
6	ErinnerungsFeature - Scrape Beuth Termine	3	1	6	Dennis Walz
7	Speech To Text	2	3		Robert Halwaß
8	Text To Speech	2	3		Alexis Popovski
9	ErinnerungsFeature - Scrape Moodle iCal export (https://lms.beuth-hochschule.de/calendar/export.php)	3	1	6	Dennis Walz
10	Cross Platform Erkennung + Datenbank “telegram_id” zu one:many relation	3	3		Lukas Danke
11	Begrüßungsnachricht in Chatbot-Clients (listen auf erst-kontakt) + Nachricht von Server	4	1		Rim Khreis

ID	Name	Priorität	Initiale Aufwandsschätzung (in Tage)	Abhängig von	Wird verticketet
12	Website mit Präsentation aller BeuthBot-Resourcen (Wiki, Telegram, Discourse, Github) und Implmentation Chatbot	4	3	1, 4	Rim Khreis
13	Universelles Scraper & Download	4	3		Alexis Popovski
14	Personalliste Gesamtverzeichnis	4	2	2	Lukas Danke



BOT-12: Rasa 2.0 Update

Um die neusten Funktionen und Fixes von Rasa zu benutzen, ist ein Update von Version 1.6 auf 2.0 notwendig.

Zustänidigkeit	Robert Halwaß
Initiale Schätzung	3 Tage
Programmiersprachen	* Python
Frameworks/Libraries	* Rasa
Services	* NLU
Abhängigkeiten	keine
Anforderungen	* Kompatibilität mit bestehenden NLU-Trainingsdaten erhalten * Mögliche JSON- und MarkDown-Dateien in YAML-Datein umwandeln

Tasks	* BOT-124: docker-compose.yml anpassen * BOT-61: Chatito Kompatibiltät testen * BOT-62: Config Anpassen * BOT-63: Duckling updaten * BOT-64: Modell neu trainieren * BOT-116: Performance zwischen Version 1.6 und 2.0 vergleichen

BOT-30: Chatbot Library: Vereinheitlichung der Kommunikation von Javascript Chatbots mit dem Gateway

Problem: Derzeit muss jede Anwendung, die den BHT-Bot als Chatbot implementieren möchte selbst implementieren, wie die Kommunikation zwischen Anwendung und Gateway aussieht, als auch die Schnittstellen Parameter in Anfrage und Antwort.
Um diese Implementationsredundanz zu verhindern, wird die Kommunikation und Typedefinitionen in einer zentralen Javascript Bibliothek zusammengefasst.
Dies ermöglicht auch weitere geplante Funktionalitäten (wie der asymmetrische Kommnuikationskanal zur Requestunabhängigen Server → Client Kommunikation) zentral entwickelt und mittels Dependency Management schnell in die ChatClients überführt werden.

Initiale Schätzung	1
Technologien	* Javascript * Typescript
Abhängigkeiten	keine
Anforderungen	* Die Library lässt sich in Node und Browser Javascript einbinden * Die Library nutzt semantische Versionierung zur Ermöglichung von Non-Breaking-Updates * Die fertige Library lässt sich via Dependency-Management (npm/yarn/webpack) userseitig einbinden und updaten * Die Library enthält typisierte (typescript) Entitäten für Common Request und Response Format(e) * Die Library enthält Unit-Tests für essentielle Funktionen und Typen * Die Library ist dokumentiert, sowohl was Nutzung, als auch Contribution angeht * Die Library verbessert die Collaboration mittels Linting-Regeln und Workflow-Scripten
Tasks	* BOT-33 Library Usage Dokumentieren * BOT-34 Library in Discord Bot integrieren * BOT-35 Library in Telegram Bot integrieren * BOT-36 Library in Website integrieren * BOT-31 Common Funktionalität / Use Cases identifizieren * BOT-32 Typescript Library für Bot erstellen

BOT-37: Discord Integration des BHT-Bot

Discord ist eine weit verbreitete Kommunikatinsplattform, auf der Nutzer sich in „Servern“ vernetzen und dort meist thematisch organisiert kommunizieren können. Die Projektgruppe des WS2020 ist selbst Teil der Zielgruppe des Discord-Messengers, wodurch sich diesese Plattform besonders eignet um einen weiteren Chatservice (neben Telegram) an den BHT-Bot anzubinden.
Eine Implementation des Chatbots innerhalb der Discord-Struktur steigert somit zum Einen die Verbreitung(smöglichkeit) des BHT-Bot und bietet gleichzeitig eine gute Möglichkeit Debug-Bot-Instanzen im präferierten Messenger zu betreiben.

Initiale Schätzung	2

Technologien	* Javascript * Docker
Abhängigkeiten	* BOT-30
Anforderungen	* Der Discourse Bot benutzt die zu entwickelnde zentralisierte Library zur Gateway Kommunikation um Coderedundanz mit dem Telegram Bot zu verhindern * Der Discourse Bot kann (direkte) Nutzer-Nachrichten mittels Gateway verarbeiten und antwortet dem User entsprechend * Wenn keine Verbindung mit dem Gateway besteht oder Fehler bei Anfragen auftreten reagiert der Chatbot durch Präsentation einer hilfreichen Fehlermeldung * Der Discourse Bot wird äquivalent zum Telgram Bot in das BHT-Bot Universum mittels Docker + compose integriert * Der Discourse Bot ist nicht Teil des BHT Bot, er wird als paralleler, unabhängiger Service betrieben * Alle Credentials und Urls/Ports werden aus dem Environment bezogen, es gibt keine hard-coded Referenzen zu Strukturen des BHT-Bot Gateways
Tasks	* BOT-38 NodeJS Chatbot erstellen * BOT-39 Docker Container + Compose für Container erstellen * BOT-40 Bot Usage dokumentieren * BOT-41 Bot Account anlegen für release (https://discord.com/developers/applications) * BOT-42 Bot Container in Beuth-Docker-Netzwerk einbinden (release)

BOT-43: Erstellung eines Common-Frameworks für (Content-)Services

Services im BHT-Bot kommunizieren alle über REST-Schnittstellen. Diese sind alle als Express-Anwendung mit JSON-Kommunikation implementiert, was für viel Code-Redundanz führt. Gleichzeitig benutzt kein Service strukturierte Darstellungen der Schnittstellen, wie Anfragen und Antworten zum Gateway, User Daten oder Rasa-Intents.
Ein spezieller, repititiver, Unterfall der Microservices sind solche, die Content bereitstellen. Diese erhalten alle Nachrichten vom Gateway und senden ihre antworten auch dort wieder zurück. Request- und Response sind durch das Gateway definiert, die resultierende Struktur ist entsprechend für alle Contentservices prinzipiell Identisch.
Zur Vermeidung von Code-Redundanzen und Erleichterung des „Kick-Off“ eines neuen Content-Services sollen die Common Funktionen und Entitäten in ein Framework gegossen werden

Initiale Schätzung	1
Technologien	* Javascript * Typescript * Dockerfile
Abhängigkeiten	* BOT-37

Anforderungen	* Das Framework implementiert eine NodeJS Express REST-API, äquivalent zu den existierenden Content-Services * Das Framework lässt sich in NodeJS Anwendungen via Dependency-Management einbinden (npm/yarn) * Das Framework abstrahiert den (Express) Server und dessen Routing, so dass ein Contentservices nur noch die Response implementieren muss * Das Framework implementiert die Schnittstelle zum Datenbankservice um a) Userdaten zu speichern/abzurufen und b) eigene Daten zu speicher und abzurufen * Das Framework füllt die “debug-history” der Requests so, dass ein Service dieses Feature zwangsweise implementiert / nutzt * Requests, Responses, User, Rasa-Intents und ggf. weitere Entitäten werden durch das Framework als typisierte (typescript) Objekte definiert * Das Framework wird in allen bestehenden und geplanten Content-Services implementiert: Wetter, Mensa, Reminder * Die Nutzung des Frameworks ist verständlich dokumentiert * Die Contribution wird mittels Linting, Dokumentation und Build-Scripts erleichtert * Das Framework nutzt types aus der (noch zu entwickelnde) BHT-Bot Library um Redundanzen zwischen Client-Bibliothem und Service-Framwork zu vermeiden
Tasks	* BOT-44 Common Code, Features, Entitäten identifizieren → Use Case ableiten * BOT-45 Typisiertes Javascript Framework erstellen * BOT-46 Framework einbinden in Weather Service * BOT-47 Framework einbinden in Mensa Service * BOT-48 Framework einbinden in Reminder Service * BOT-108 Framework dokumentieren * BOT-117 Request History implementieren - Nutzung enforcen

BOT-49: User-Messenger-Service: Nachricht proaktiv, requestunabhängig an Clients senden

Der BHT-Bot kann bisher nur passiv auf Anfragen warten und diese dann beantworten. Zur Implementierung von asymmetrischer bzw. request-unabhängiger Kommunikation benötigt der Bot einen neuen Service, der als Schnittstelle für diese Art von Kommunikation dient.

Initiale Schätzung	2.5
Technologien	* Javascript * Websockets * Docker
Abhängigkeiten	* BOT-30

Anforderungen	* Der User-Messenger-Service kann eine Nachricht an einen User (via Client-Unabhängiger User-ID) senden * Wenn ein User mehrere Clients benutzt, wird die Nachricht an alle Clients gesendet * Wenn ein User (über längere Zeit) nicht erreichbar ist wird die Kennung entfernt * Wenn eine Nachricht nicht an einen User gesendet werden kann bekommt der auslösende Service diese Information unterscheidbar zwischen a) Der Nutzer ist gerade nicht erreichbar b) Der Nutzer ist dauerhaft nicht erreichbar (gelöscht) c) Der Dienst ist generell unhealthy * Der Service wird als Docker Image via docker-compose in die BHT-Bot Infrastruktur integirert. Er ist Teil des BHT-Bot Repositories * Die Funktionalität des Services wird auf Clientseite in die Common-Chatbot-Library (BOT-30) * Alle bestehenden ChatBot-Services werden an den Messenger Service angebunden (Telegram, Discord)
Tasks	* BOT-50 Websocket Registry für ChatBotClients * BOT-51 REST-Service für Nachrichtenversand * BOT-52 Implementation der Websocket-Registrierung in Common-Library für Chatbots * BOT-53 Implementierung der Common-Library-Websocket-Registrierung in TelegramBot * BOT-54 Implementierung der Common-Library-Websocket-Registrierung in DiscordBot * BOT-56 Dokumentation Usage Service * BOT-110 Deployment / Release

BOT-55: Erinnerungs-Service: Behandelt „erinnere mich“ Befehle und erinnert bei Fälligkeit autonom

Erinnerungen schedulen zu können ist ein typischer, weil praktischer, Anwendungsfall in beliebten (Business) Kommunikations-Services wie Slack oder Mattermost. In beiden Fällen wird diese Funktionalität durch die hauseigenen Reminder-Bots zur Verfügung gestellt.
Um die Featuredichte des BHT-Bot zu erhöhen wird ein Reminder-Service erstellt, durch den identische Funktionalität wie bei genannten Diensten zur Verfügung stellt. Durch die Multi-Messenger-Fähigkeit des BHT-Bot wird dieses Feature somit auch für User angeboten, deren Kommunikations-Plattform keinen eigenen Reminder-Bot anbietet.
BeispielAnfragen:
* Erinnere mich am 22.10. an die Klausur in Mathe
* Erinnere mich jeden Donnerstag um 18 Uhr an den Ballettkurs
* Erinnere mich jedes Jahr am 01.01 an den Geburtstag meiner Mutter.
* Erinnere mich in 10 Tagen das Probeabonnement zu kündigen
* <Erinnere> <Zeitpunkt/Zeitspanne/Interval> <Thema>

Initiale Schätzung	3
Technologien	* Javascript * Docker * Rasa * MongoDB * Cronjob

Abhängigkeiten	* BOT-43 * BOT-30 * BOT-12 * BOT-49
Anforderungen	* Erfolgreiche „erinnere“-Anfragen werden vom Dienst durch Bestätigung der erkannten und persistierten Daten beantwortetoder * Fehlerhafte “erinnere”-Anfragen werden durch ein Mini-Usage-Tutorial beantwortet, damit der User seine Anfrage korrigieren kann * Erinnerungen werden auf user-ebene (clientunabhängig) gespeichert, so dass ein User die gleichen Erinnerungen in allen genutzten Clients zur Verfügung hat * Erinnerungen werden bei Fälligkeit einmalig (an alle clients des users) ausgespielt * Wiederkehrende Erinnerungen werden ausgespielt und anschließend an Hand des Intervals neu terminiert * Der Nutzer kann Erinnerungen löschen * Der Nutzer kann seine Erinnerungen anzeigen lassen * Der Service wird als Docker Container via docker-compose verwaltet und in das BHT-Repository integriert * Der Service nutzt das (noch zu schaffende) Content-Service-Framework * Wenn der Reminder-Service nach einem Ausfall wieder aktiv wird erinnert er nicht (einzeln) an alle Termine, die zwischenzeitig fällig waren * Der Serive ist in Hilfe-Texten des (Chat) BHT-Bots integriert
Tasks	* BOT-57 Rasa Anbindung „Erinnere“-Direktive * BOT-58 Service Endpoint speichert Reminder und Antwortet auf Probleme * BOT-59 Scheduler/Cronjob prüft und sendet regelmäßig fällige Erinnerungen * BOT-60 Dokumentation Service Usage * BOT-109 Deployment / Release * BOT-112 Hilfe-Texte in (Chat)BHT-Bot help-command / willkommensnachricht

BOT-82: Termin-Scraper, der automatisch Erinnerungen aus öffentlichen Quellen bezieht

Es gibt Termine, an die wollen alle Studierenden typischerweise erinnert werden, als auch Termine, von denen die Universität möchte, dass die Studierende sich daran erinnern. Typische Beispiele hierfür sind Rückmeldefristen und (Beuth-eigene) Feiertage.
Durch einen Webscraper sollen solche Termine aus (öffentlichen) Quellen automatisch bezogen und dann an alle Studierenden ausgespielt werden.
Auch wenn dieses Feature für die meisten Studierenden interesssant sein dürfte, muss es eine Möglichkeit zum Opt-Out geben. Ggf. ist sogar angezeigt, dass ein Opt-In erfolgt, was allerdings den Nutzen des Features, vor allem aus Universitätssicht, mindern würde.

Initiale Schätzung	1
Technologien	* Javascript * Docker * HTML-DOM

Abhängigkeiten	* BOT-55
Anforderungen	* Relevante Termine werden regelmäßig, automatisch bezogen und als Erinnerung gespeichert * User können “globale Erinnerungen” aktivieren oder deaktivieren * Das Feature ist resistent gegen Änderungen an den Domains oder deren Struktur → Fallen gescrapete Dienste länger aus wird dies reported * Das Feature wird als nicht-eigenständig in den Reminder Service integriert * Das Feature ist bzgl. Opt-in oder Opt-out in den Hilfe-Texten/Begrüßungsnachrichten des BHT-Bot dokumentiert
Tasks	* BOT-83 Prüfen ob Opt-In (nötig ist) oder Opt-Out (möglich ist) * BOT-84 Quellen mit relevanten Terminen zusammentragen - auf Scrapebarkeit achten * BOT-85 Scraping (mittels Scapring-Service) implementieren * BOT-86 Termine aus Scraping in Erinnerungs Datenbank überführen * BOT-87 Rasa Direktive Opt-In oder Opt-Out implementieren * BOT-88 Error-Reporting implementieren, bei Misslungenen Scraping (über gewisse Zeit hinweg) * BOT-114 Opt-In oder Opt-Out in Hilfe-Texten / Willkommensnachricht dokumentieren

BOT-89: Moodle iCal import als Erinnerungen

Moodle ist die zentrale Online-Lernplattform der Beuth Hochschule. Kurse, die in Moodle verwaltet werden erhalten in der Regel Abgabetermine für Aufgaben, die während des Semesters fällig werden. Oftmals stehen diese Abgabetermine bereits zu Beginn des Semesters in Moodle fest und können dort in einer Kalenderansicht betrachtet werden. Moodle bietet außerdem eine Funktion zum Export der Eintragungen im eigenen Kalender: https://lms.beuth-hochschule.de/calendar/export.php
Der User kann hier einen Link erzeugen, über den eine iCal Datei bezogen werden kann. Diese Datei enthält die Semestertermine und kann entsprechend in Erinnerungen des Bots umgewandelt werden

Initiale Schätzung	1
Technologien	* Javascript * iCal * Rasa
Abhängigkeiten	* BOT-55

Anforderungen	* Der User kann dem Bot seinen Moodle-Kalender-Export-Link senden um einen Import auszulösen * Die iCal Datei hinter dem Export-Link wird in Erinnerungs-Einträge des Erinnerungs-Service umgewandelt * Erinnerungen an Abgabetermine erfolgen zu Beginn des Tages / zeitlich versetzt vor der Fälligkeit der Abgabe, nicht zum im Kalender angegebenen Zeitpunkt * Das Moodle-Import Feature wird nicht-eigenständig in den Reminder-Service integriert * Das Feature (und seine Nutzung) ist in der BHT-Bot Hilfe gelistet
Tasks	* BOT-90 Import Moodle Rasa Direktive * BOT-91 Moodle iCal Download via zentralem Scaper Service * BOT-92 iCal Parsing und Persistierung als (sinnvolle) Erinnerung * BOT-111 Feature Release * BOT-113 Hilfe-Texte in (Chat)BHT-Bot help-command / willkommensnachricht

BOT-13: Speech To Text (STT)

Es soll ermöglicht werden, dass Benutzern neben Textnachrichten auch mittels Sprachnachrichten mit dem BeuthBot kommunizieren können. Dabei sollen die Sprachnachrichten mittels eines neuen Services in Text übersetzt werden und dann wie andere Textnachrichten verarbeitet werden. Hierzui sollen 3 bekannte STT-Frameworks (Kaldi, Mozilla Voice STT und Wav2Letter) getestet und vergleichen werden. Basierend darauf soll eine Entscheidung getroffen werden, welches Framework schlussendlich in der Production-Environment verwendet werden soll. Das Framework wird dann in Form eines neuen Micro-Services in den BeuthBot integriert.

Zuständigkeit	Robert Halwaß
Initiale Schätzung	3 Tage
Programmiersprachen	* Python (Mozilla Voice STT) * C++ (Kaldi, WAV2Letter)
Frameworks/Libraries	* Kaldi * Mozilla Voice STT * WAV2Letter
Services	* Speech
Abhängigkeiten	* BOT-43: Erstellung eines Common-Frameworks für (Content-)Services

Anforderungen	* Die Übersetzung soll mittels neuronaler Netzte geschehen * Nur Sprachnachrichten auf Deutsch sollen übersetzt werden * Das verwendete Framework muss OpenSource sein und Lokal auf dem BeuthBot-Server ausführbar sein * Es soll keine Model-Adapation durchgeführt werden

Tasks

BOT-23: Komponente zur Umwandlung von Text in Sprache (TTS)

Neben der bereits vorhandenen Funktion Textnachrichten vom Beuthbot zu erhalten, sollen Nutzer die Möglichkeit bekommen ebenfalls Sprachnachrichten zu empfangen. Hierfür wird eine Komponente zur Konvertierung von Text in Sprache (Eng: „Text-To-Speech (TTS)) benötigt. Dieses Feature soll dem Nutzer in künftigen, dem Beuthbot hinzugefügten, Messenger-Diensten zur Verfügung stehen. Zur Umsetzung soll optimalerweise von einer Library Gebrauch gemacht werden, welche den Anforderungen gerecht wird.

Initiale Schätzung	1
Technologien	Javascript
Abhängigkeiten	keine
Anforderungen	*Support für die deutsche Sprache *Sprachnachrichten lassen sich als MP3- und WAV-Dateien exportieren *Das Feature ist bzgl. Opt-in oder Opt-out in den Hilfe-Texten/Begrüßungsnachrichten des Beuthbots dokumentiert
Nice to Haves	Sprachgeschwindigkeit und Stimme des Sprechers sind konfigurierbar
Tasks	* BOT-24 Recherche nach geeignetem Tool (TTS) * BOT-25 Eigene Implementierung (TTS) * BOT-98 Integration in Beuthbot

BOT-75: Begrüßungsnachricht

Neue Benutzer des BeuthBots sollen mit einer Begrüßungsnachricht empfangen werden. Diese Nachricht soll die Features des BeuthBots vorstellen und einen Shortcut nennen, welchen die Benutzer verwenden können, wenn sie Hilfe benötigen. Mit dem Shortcut listet der BeuthBot nochmals all seine Features auf.

Initiale Schätzung	1
Technologien	* JavaScript
Abhängigkeiten	Keine

Anforderungen	* Die Begrüßungsnachricht erscheint nur für neue Benutzer * Das System sollte einen Shortcut zur Wiedervorstellung der Features bereitstellen, falls Benutzer Hilfe brauchen * Das System muss in der Lage sein, auf die Hilfeanfrage. des Benutzers mit Hilfe des Shortcuts innerhalb von 1,5 Sekunden zu antworten * Das System sollte gut dokumentiert sein * Das System sollte leicht zu verstehen sein
Tasks	* BOT-93 Client * BOT-94 Server

BOT-74: Webseite

Um eine komplette Übersicht für alle genutzten BeuthBot-Resourcen zu haben, soll eine Webseite zur Präsentation dieser Ressourcen erstellt werden. Zu den Ressourcen zählen das Ziemer's Wiki, Telegram, Discord, Github und die Implementation des Chatbots.

Initiale Schätzung	3
Technologien	* TypeScript * JavaScript
Abhängigkeiten	* BOT-10 * BOT-30
Anforderungen	* Jeder Ressource wird ein Menüpunkt gewidmet, welcher Infos & einen Link zu der jeweiligen Ressource enthält * Anschauliches & Einheitliches Design der Webseite * leicht austauschbare Komponenten * Das System sollte eine Ansicht innerhalb von 3 Sekunden laden * Das System sollte gut dokumentiert sein * Das System sollte leicht zu verstehen sein
Tasks	* BOT-76 Webseite Einrichten * BOT-77 Infos zum Wiki * BOT-78 Infos zu Telegram * BOT-79 Infos zu Discord * BOT-80 Infos zu GitHub * BOT-81 Implementation Chatbot

BOT-11: Universeller Scraper & Download

Der Beuthbot soll einen „universellen“ Web-Scraper beinhalten, der als Grundlage für künftige Features dienen soll, die für konkrete Scraping-Funktionalitäten vorgesehen sind. Aufgrund der hohen Diversität an Datenstrukturen unterschiedlicher Webseiten, soll dieser möglichst abstrakte Funktionalitäten zur Extrahierung von Datensätzen bieten.

Initiale Schätzung	1
Technologien	Javascript

Abhängigkeiten	keine
Anforderungen	*Daten lassen sich im JSON- und XML-Format ausgeben *Datensätze sind per HTML-Tags und CSS-Selektoren extrahierbar *Dateien einer Webseite lassen sich downloaden
Tasks	* BOT-26 Recherche nach geeignetster Methode (HTML-JSON)

BACKLOG BOT-106: Monitoring bzw. Alerting Features

Aktuell gibt es keine Stelle durch die sich der Bot im Falle von Problemen bemerkbar machen kann.
So fallen Ausfälle von ganzen Servicen ebenso wenig auf, wie der Ausfall von Teil-Logiken.
2 aktuelle Beispiele:
1) Der Bot stürzt regelmäßig ab, der Container startet neu und funktioniert wieder. Kein Admin kann derzeit mitbekommen oder analysieren warum das so ist.
2) Services die bspw. Webscraping einsetzen werden zwangsweise irgendwann unfunktional, weil sich die genutzten Endpunkte / Strukturen ändern ohne dass dies von Admin/Entwicklerseite bekannt ist. Solche Dienste fallen ggf. vom Service sogar richtig behandelt aus, so dass der User ein Fehler-Feedback bekommt, Admins bekommen diese Information aber wiederum nicht

BACKLOG BOT-115: History Feature prüfen

Im Gespräch mit Lukas aus dem SoSe2020 wurde bekannt, dass das „History Feature“ letztes Semester etwas vernachlässigt wurde. Es handelt sich um ein Array, das bei Requests weitergereicht und von jedem Service mit eigenen Einträgen befüllt wird, so dass zu jedem Zeitpunkt ersichtlich wird welche Services bereits Kontakt mit dem Request hatten
Da dieses Feature nützlich beim Debuggen ist, aber nur helfen kann, wenn genügend Informationen darin gesammelt werden, muss die Konsistenz überprüft und mindestens in neuen Services wieder hergestellt werden

BACKLOG BOT-119: Mock-Option für Deconcentrator

Aus Gespräch mit Lukas aus SoSe2020 hat sich ergeben, dass die Sprachverarbeitung in früheren Stadien des BHT-Bot gemocked wurde. Dieses Feature ist nicht mehr funktionsfähig, für die Entwicklung könnte eine solche Implementation allerdings hilfreich sein:
Wenn ein Service entwickelt wird kann dieser nur durch User angesprochen werden, wenn der Concentrator User-Requests korrekt parsed. Dies beinhaltet die Weitergabe und Analyse des Requests mittels Sprachananalyse Services, aktuell insbesondere durch Rasa.
Um einen Service unabhängig von Rasa-Training und NLP testen/entwickeln zu können wird ein Mock-Feature für den oder als Ersatz für den Concentrator entwickelt.

BACKLOG BOT-121: Database und Database_micrososervice zusammenführen

Es gibt derzeit 2 Services, die im Grunde das gleiche zu machen scheinen:
https://github.com/beuthbot/database_microservice
https://github.com/beuthbot/database/
Der Unterschied scheint lediglich in der bereitgestellten Funktionalität zu liegen: Während der „database“ Service lediglich userbezogene Abfragen behandelt kann der „database_mircoservice“ Service arbiträre Datensätze der Services persistieren. Im Kern dienen beide Services aber lediglich dazu eine zentrale Schnittselle zur gleichen Mongo-DB herzustellen
Diese dopplung der Datenbankservices ist nicht dokumentiert, oder zumindest ist diese Dokumentation nicht sichtbar geworden. Daher gilt es die beiden Datenbanservices zu einer logischen Einheit zu vereinen, oder zumindest sicherzustellen, dass nachfolgende EntwicklerInnen erkennen können dass es hier eine „unlogische“ Service-Redundanz gibt