Zwischenbericht WS2020/21

BeuthBot Projektgruppe

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

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 Studierenden, für Studierende. Somit können nicht nur die Mitarbeiter des Studierendenservice entlastet, sondern auch kleine bis große Fragen der User schnell beantwortet werden. Ein intelligenter Chatbot, der per Text und Sprache bedient werden kann, mittels künstlicher Intelligenz antwortet und viele Features anbietet, wie z.B. die automatische Erinnerung an wichtige Termine der Universität.

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 privat von Herrn Ziermers 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. Discord bietet neben der Instant-Messenging Funktion auch noch die Möglichkeit persistente, themenbasierte Unter-Chats bzw. Räume zu eröffnen, so dass besprochene Inhalte trotz der formlosen Kommunikation leicht und dauerhaft auffindbar sind.

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.

Kernfuntionalitäten, die Jira abbildet sind die wöchentlichen Sprints in Form eines Kanban Boards, der Backlog und Zeitaufschreibung zur Kontrolle des Arbeitsumfangs durch den Dozierenden

Jira hat außedem eine App, die es schafft alle Funktionen (die bisher gesucht wurden) in das Mobile Interface zu überführen, so dass es möglich wird unterwegs PM Tasks zu erledigen wie die wöchentliche Sprint Planung.

GitHub

GitHub ist ein netzbasierter Dienst zur Versionsverwaltung für Software-Entwicklungsprojekte. Er wird für die Zusammenführung der Arbeitsergebnisse des Projektteams genutzt.

Durch die Nutzung von Github wird (im Gegensatz zum Einsatz des Beuth-Gitlabs) ermöglicht, dass die Arbeitsergebnisse dieses Projekts auch für außenstehende verfügbar werden, wodurch ein aktiver Beitrag zur **Open Source** Bewegung geleistet wird. Durch die Beteiligung an diesem Proejkt erhalten Projektteilnehmende auf diese Art auch eine gewisse Art von öffentlicher Reputation.

Technische Dokumentation wird immer am entsprechenden Projekt-Teil, also dem entsprechenden Repository angelegt. Dadurch findet sich die technische Dokumentation des Bots vor allem auf Github

Ziemer's Wiki

Im Ziemer's Wiki, welches auf DokuWiki basiert, 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
- WebStorm / PHPStorm (Jetbrains Produkte)

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.

NPMJS

npmjs.com ist das Repository für nodejs Anwendungen. Javascript Code, der von mehreren Services / Modulen implementiert werden soll, wird daher über npmjs verteilt.

Organisation

Das Projektteam arbeitet teilweise nach der agilen Vorgehensweise, genauer gesagt nach Scrum. Ziel bei der agilen Vorgehensweise ist, dass schon nach kurzer Zeit ein fertiges Arbeitspaket bzw. ein Prototyp abgegeben wird. So erhält das Projektteam schon frühzeitig ein Feedback des Dozenten und kann mögliche Änderungen zeitnah im Projektverlauf berücksichtigen.

Dafür nutzt das Projektteam unter andrem Jira. Dort wird das Product Backlog gepflegt. Das Product Backlog enthält alle Anforderungen und kann sich während des Projektverlaufs stetig ändern, da Anforderungen hinzukommen, sich ändern oder wegfallen können. Die Anforderungen werden in einen einwöchigen Sprint aufgenommen. Ein Sprint ist die zur Verfügung gestellte Zeit, um die gewählten Anforderungen zu bearbeiten. Am Ende jedes Sprints sollten die Aufgaben umgesetzt sein. So hat das Projektteam dann immer ein kleines Arbeitspaket vorzuweisen.

Nach jedem Sprint gibt es dann das Sprint Review. Im Sprint Review trifft sich das Projektteam mit dem Dozenten, um die erbrachte Leistung des abgeschlossenen Sprints zu präsentieren, Feedback einzuholen, mögliche Unklarheiten zu klären und Ziele für die folgende Woche zu besprechen. Das Sprintreview findet jede Woche (Donnerstag 10:00) in Form eines Videochats mittels Jitsi statt. 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 sind. 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.

Über Discord findet auch die Sprint Retrospektive und das Sprint Planning jeden Donnerstag nach dem Sprint Review statt. In der Sprint Retrospektive spricht das Projektteam über die Ereignisse, die sie im Sprint gut oder schlecht fanden, die beibehalten, wegfallen oder mit denen sie beginnen sollten. Es bietet für jedes Teammitglied eine Möglichkeit, Feedback an das gesamte Team zu geben. Im Sprint Planning werden dann die Anforderungen ausgewählt, die das Team im nächsten Sprint bearbeiten wird.

Während des Sprints findet auch das Daily Scrum statt. Das Team kommt beim Daily Scrum ein weiteres Mal in der Woche, abgesehen vom Donnerstag, zusammen, um zu besprechen, was jedes Mitglied gemacht hat, noch machen wird oder welche Hindernisse es hat.

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

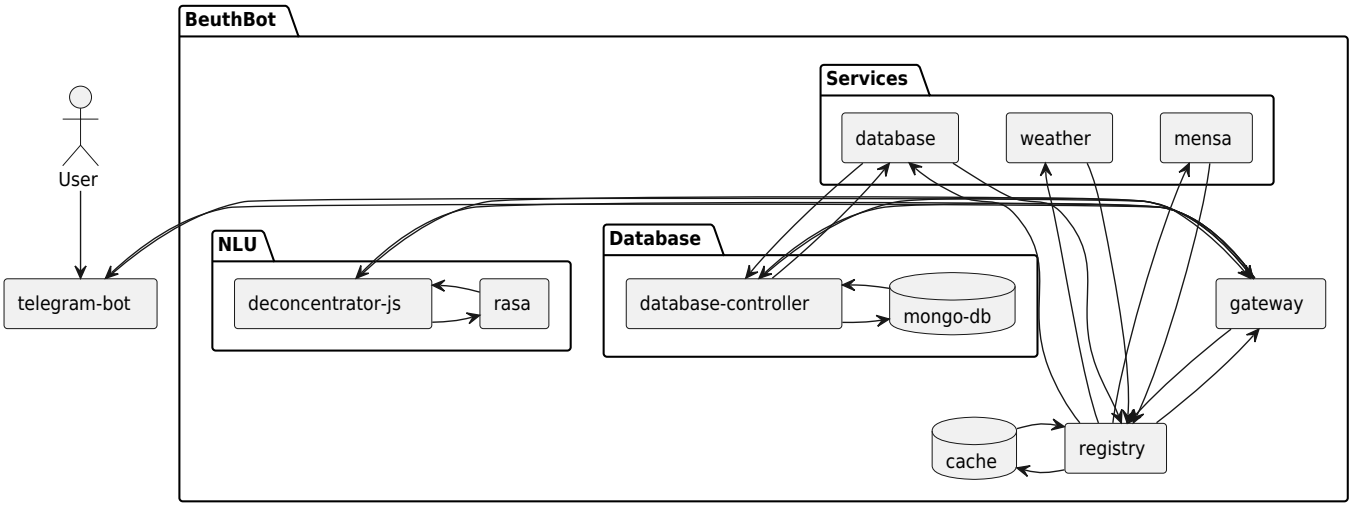
Nachdem die Absicht des Nutzers analysiert worden ist, informiert das Gateway die Registry, um die Informationen zu erhalten, die der Nutzer benötigt. Darauffolgend verteilt die Registry die Anfrage an den entsprechenden Service.

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

Continous Integration & Deployment ist ein wichtiger Pfeiler für ein stabiles Production Environment. Durch eine CI/CD Pipeline kann sichergestellt werden, dass das Deployment nachvollziehbar, zuverlässig ausgeführt wird und bietet zugleich die Möglichkeit Qualitätssicherungs-Mechanismen in der Pipeline zu manifestieren.

Zu Beginn des Semesters wurde das Deployment manuell ausgelöst. Es gab mehrere Scripte, die diesen Vorgang unterschiedlich angingen. Die Updatestrategie ist grundsätzlich „alle Repositories updaten via git pull“ - leider gab es hier allerdings flaws, die dazu führten dass das lokale Repository „unrein“ wurde und nicht automatisch aktualisiert werden konnte.

Hinzu kam das Problem, dass die Ordnerstruktur unterschiedlichen Usern gehörte, immer denjenigen, die das Update ausgeführt haben, bei dem die Dateien erstmals im Repository auftauchten. Dadurch scheiterten Updates zusätzlich, wenn „der falsche user“ das update versuchte bzw. „die falschen dateien“ im update aktualisiert würden.

Wir haben via Github-Actions eine CI/CD Pipeline erstellt, die den Deployment Prozess in 3 Stages ausführt: Build, Test, Deploy. Die Test-Stage ist via Makefile angebunden, so dass EntwicklerInnen neue Tests simpel in eine zentrale Stelle eintragen können. Das Makefile ist nun Single Point of Truth. Die teilweise widersprüchlichen Scripte von vorher wurden aufgeräumt

Code Änderungen im Repo (Pull Request): https://github.com/beuthbot/beuthbot/pull/4

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. Der Runner führt die Aktualisierung auf der VM des Bot durch.

Doku zur Runner Config:
https://github.com/beuthbot/beuthbot/blob/master/.documentation/github-runner.md

Public Domain

Es gab keine Public Domain zum Telegram Gateway. Diese brauchen wir aber um a) eine Landing Page für den Bot zu hosten und b) das Gateway von Chatbots ansprechen zu können, die nicht in der VM gehosted sind. Damit dies funktioniert wurde ein Proxy-Pass für die Default-Domain von **https://beuthbot.ziemers.de/** angelegt. Damit wird jetzt folgender Curl Möglich: *$ curl https://beuthbot.ziemers.de/message -X POST -H „Content-Type: application/json“ –data „{\“text\„:\“Wie wird das Wetter morgen?\„}“*

Text To 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 To Text Recherche

In diesem Projekt soll es ermöglicht werden, dass Nutzer ebenfalls Sprachnachrichten an den BeuthBot schicken können um mit diesem zu interagieren. Um dieses umzusetzen ist eine sogenanntes Speech-To-Text-Programm erforderlich, welche Sprachnachrichten in Text umwandelt. Diese umgewandelten Nachrichten können dann wie normale Textnachrichten vom BeuthBot weiterverabeitet werden. Da es sich hierbei um ein äußerst kompliziertes technisches Problem handelt, bei dem Ansätze mit statischen Algorithmen nicht anwendbar sind, werden ausschließlich Ansätze des DeepLearning angewendet. Neben vielen Cloud-Lösungen von namenhaften Anbietern wie Amazon und Google gibt es ebenfalls eine Reihe von OpenSource-Lösungen, welche privat gehostet werden. Dieses bietet mehrere Vorteile. Zum einen, fallen keine Gebühren für die Nutzung an, da alle Berechnungen lokal auf dem BeuthBot-Server ausgeführt werden. Zum anderen ist Datenschutz leichter umzusetzen, da alles lokal verarbeitet wird und keine Daten an externe Services weitergegeben werden. Der Recherche ergab eine Vielzahl an Lösungen, jedoch sind nur drei für das Projekt geeignet, da nur für diese ein vortrainiertes Modell für die deutschen Sprache verfügbar ist. Diese sind:

Mozilla Voice STT (DeepSpeech)

- https://github.com/mozilla/DeepSpeech
- https://github.com/AASHISHAG/deepspeech-german
- Entwickler: Mozilla
- Opensource
- Offline nutzbar
- Viel Dokumentation
- Deutsches Modell
- WER: 15%
- Zukunft ungewiss (https://www.ghacks.net/2020/08/24/the-future-of-mozillas-speech-to-text-project-deepspeech-is-uncertain/)

Kaldi

- https://github.com/kaldi-asr/kaldi
- http://kaldi-asr.org/doc/about.html
- http://zamia-speech.org/asr/
- Entwickler: Kaldi
- Opensource
- Offline nutzbar
- Deutsche Modelle
- WER: 8,44%

Wav2Letter

- https://github.com/facebookresearch/wav2letter

- http://zamia-speech.org/asr/
- Entwickler: Facebook Research
- Opensource
- Offline nutzbar
- Deutsche Modelle
- WER: 3,97%

Während des Projekts gilt es, diese drei Lösungen zu testen, miteinander zu vergleichen und darauf basierend die beste Lösung auszuwählen und im BeuthBot zu implementieren.

Die STT-Programme ohne verfügbares deutsches Modell sind folgende:

Espresso

- https://github.com/freewym/espresso
- Entwickler: Freewym
- Opensource
- Offline nutzbar
- Kein deutsches Modell

OpenSeq2Seq

- https://github.com/NVIDIA/OpenSeq2Seq
- Entwickler: NVIDIA
- Opensource
- Offline nutzbar
- Kein Deutsches Modell

Word Error Rate (WER)

Um die Qualität eines STT-Modells zu messen, wird der sogenannte Word Error Rate (WER) Wert verwendet. Dieser Wert gibt an, basierend auf dem Testdatensatz, wie viele Wörter prozentual falsch erkannt werden. Zum Beispiel, wenn bei einem Satz, welcher 100 Wörter enthält, 10 Wörter falsch erkannt werden, dann beträgt der WER-Wert 10%.

Unten befindet sich eine Auflistung von WER-Werten von kommerziellen STT-Diensten für die englische Sprache aus dem Jahre 2017. Darunter befindet sich ebenfalls die WER-Werte der recherchierten OpenSoruce-Lösungen. Da alle Dienste unterschiedliche Datensätze zum Training und Test verwenden, sind diese Ergebnisse nicht komplett vergleichbar, aber sie bieten eine grundsätzliche Übersicht über die Performance der OpenSource-Lösungen.

- Google (8%)
- Microsoft (5.9%)
- IBM (5.5%)
- Apple (5%)
- Baidu (16%)
- Hound (5%)
- Mozilla Voice STT (15%)
- Kaldi (8,44%)
- Wav2Letter (3,97%)

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

DokuWiki Plugins

edittable

Standartmäßig werden im Ziemer's-Wiki alle Tabellen mittels MarkDown angelegt. Da dieses jedoch besonders bei großen Tabellen sehr fehleranfällig ist, wurde das edittable-Plugin installiert. Dieses erlaubt es mittels einer grafischen Benutzeroberfläche Tabellen anzulegen und zu bearbeiten. So entstandene Tabellen werden dann als normale MarkDown-Tabellen im Wiki abgelegt. Dieses erleichterte das Arbeiten mit Tabellen ungemein.
https://www.dokuwiki.org/plugin:edittable

PageBreak

Die finale Abgabe des Zwischenberichtes sollte in Form eines PDFs abgeben werden. Das Ziemer's-Wiki hatte bereits das DW2PDF-Plugin installiert, welches es auf einfache Weise ermöglicht jede beliebige Wiki-Seite als PDF zu exportieren. Hierbei ergab sich jedoch das Problem, dass alle Kapital ohne große Abstände hintereinander in das PDF geschrieben wurden, welches die Übersichtlichkeit stark beeinträchtigt hat. Um dieses Problem zu lösen, wurde das PageBreak-Plugin im Ziemers-Wiki installiert. Dieses erlaubt es, mittels des pagebreak-Tags, dem DW2PDF-Plugin mitzuteilen wann ein Seitenumbruch passieren. Damit konnten wir nach jedem Kapitel und Feature-Tabelle einen Seitenumbruch hinzufügen. Dies hat die Übersichtlichkeit des Zwischenberichtes deutlich erhöht.
https://www.dokuwiki.org/plugin:pagebreak

Vorüberlegung zu Features: 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.

Geplanter Stand

Präambel

Neben einigen Wünschen der Projektleitung konnte die Projektgruppe eigene Schwerpunkte in die Feature Planung einbringen. Die kommende Feature-Planung ist ein Resultat der folgenden Überlegungen

Content First

Der BHT-Bot besteht bisher aus lediglich zwei Services, die Content zur Verfügung stellen: Mensa- und Wetter-Service. Aufgrund der Covid19-Pandemie finden keine Präsenzveranstaltungen an der Hochschule statt und in Folge dessen hat die Mensa geschlossen, der Bot-Service ist entsprechend auch eingestellt. Defakto kann der BHT-Bot damit derzeit ausschließlich das Wetter ansagen. Für uns ist es daher umso wichtiger dieses Semester neuen Content zu erzeugen bzw. nutzbare Features in den BHT-Bot zu integrieren, damit dieses Projekt überhaupt eine Daseinsberechtigung bekommt.

Hands-On & Dokumentation

Wenn man neu in ein Projekt kommt gibt es viel Dokumentation aufzuarbeiten, als auch undokumentierte Zustände zu entdecken. Wir hatten Gelegenheit den BHT-Bot in einem Hands-On Workshop von Lukas Dankwerth aus SoSe2020 vorgestellt zu bekommen. Im Workshop haben wir uns neben all dem Guten, was die letzten Semester geschaffen haben (danke vor allem für die docker-compose zentralisierung der infrastruktur!) beispielsweise angeschaut a) Warum der Bot eine Telegram-ID braucht in allen Requests b) Dass der Bot derzeit zwei fast identische Datenbankservices hat, die zusammengeführt werden könnten c) Dass die Services derzeit inkonsistenzen aufweisen, die bis zu essentiellen Debugging-Strategien wie der Request-History reichen, die nicht gepflegt wurde d) Dass Dokumentation auf verschiedene Projekte und Plattformen verteilt wurde.

Ohne einen persönliche Hands-On-Workshop ist eins der größten Probleme um ins Projekt zu finden, dass es verschiedene Stellen gibt, an denen Dokumentation, Code, Infrastruktur, etc. verteilt ist, aber keine zentrale Stelle, die als „Single Point of truth“ registerartig auf die weiteren Quellen verweist.

Wir wollen diese Situation verbessern, indem wir falsche und für uns nicht-nachvollziehbare Dokumentation verbessern, Wikistrukturen bereinigen, eine Landing-Page für den Bot unter einer zentralen Bot-Domain verfügbar machen und Abläufe und Strukturen in Code gießen.

Redundanter Code / Wartbarkeit der Microservices

Beim Studium der einzelnen Services wurde sichtbar, dass fast ausnahmslos jeder Service des BHT-Bot die gleiche Grundstruktur hat: Alle Services stellen eine JSON-REST-Schnittstelle zur Verfügung, die via NodeJS + Express Framework implementiert ist. Vergleicht man die Services, sieht man, dass insbesondere Services, die Inhalte ausspielen sollen einer identischen Struktur folgen (müssen), dies aber individuell handhaben. Dadurch ist der Boilerplate Code für jeden Service unnötig hoch und zugleich ist das Warten der Services bei Änderung von globalen Schnittstellen mit hohem individuellen Aufwand verbunden. Wir wollen diese Common-Funktionalitäten identifizieren und in zentralen Bibliotheken bzw. Frameworks bündeln.

Bei allem was wir tun und Planen gilt die alte Pfadfinder-Regel: „Hinterlasse das Camp immer aufgeräumter, als du es vorgefunden hast“. Das beinhalte tbeispielsweise auch die Wartung bzw. Updates von bestehenden Dependencies wie express-js oder Rasa-NLP.

Qualitätsmanagement

Bisher sind alle Services lose miteinander verbunden, jeder Service implementiert die Kommunikation für sich selbst, die Integration erfolgt durch manuelles Deployment, es gibt keine Typisierung, keine (automatischen) Regeln zur Collaboration, keine Versionierung, kein Unit-Testing, kein Monitoring, .. Kurz gesagt: Der BHT-Bot hat bislang keine Form von Qualitätssicherung. Dadurch ergeben sich natürlich viele Baustellen, die wir, insbesondere auch angesichts unserers Anspruchs „Inhalte in den Bot zu bringen“ kaum befriedigend erfüllen können. Wir wollen dennoch einen Beitrag zur Verbesserung der aktuellen Situation erbringen:

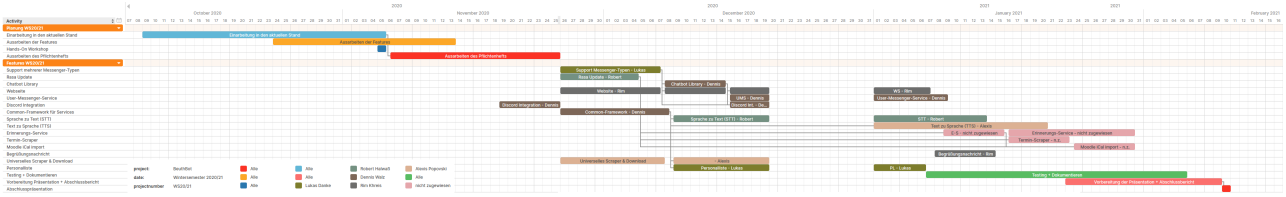
1. Wir schreiben typisierte Libraries, die eine zentrale Dokumentation der Kommunikation mit dem Bot darstellt.
2. Alle Libraries und Frameworks werden durch Linting-Regeln in ein, für alle Entwickler einheitliches, Format gebracht
3. Alle Libraries und Frameworks haben automatische Unit-Tests, als auch Prüfung der Testabdeckung, wir verlangen mindestens 80% Testabdeckung, Ziel ist 100% zu erreichen
4. Wir wechseln vom bisherigen manuellen Deployment zu einer automatischen CI/CD Pipeline. Diese Pipeline beinhaltet auch eine Testing-Stage, in die wir dieses Semester mindestens einen Service anbinden werden
5. Releases werden durch Versionierte Git Tags erzeugt (und dann automatisch deployed) - Die Tags werden semantisch versioniert
6. Neue Features werden via Pull Request in das Repository übernommen. Nach Möglichkeit werden die Requests durch ein Projektmitglied reviewd und freigegeben

Übersicht geplante Tasks und Abhängigkeiten

ID	Name	Priorität	Initiale Aufwandsschätzung (in Tage)	Abhängigkeiten	Wird verticketet von

ID	Name	Priorität	Initiale Aufwandsschätzung (in Tage)	Abhängigkeiten	Wird verticketet von
BOT-16	Support mehrerer Messenger-Typen durch Umbau der Benutzererkennung	1	1		Lukas Danke
BOT-12	Rasa Update auf 2.0	1	1		Robert Halwaß
BOT-30	Chatbot Library: Vereinheitlichung der Kommunikation von Javascript Chatbots mit dem Gateway	2	1	BOT-16	Dennis Walz
BOT-74	Webseite	2	3	BOT-16, BOT-30	Rim Khreis
BOT-49	User-Messenger-Service: Nachricht proaktiv, requestunabhängig an Clients senden	2	2.5		Dennis Walz
BOT-37	Discord Integration	2	2	BOT-16, BOT-30	Dennis Walz
BOT-43	Erstellung eines Common-Frameworks für (Content-)Services	2	1.5		Dennis Walz
BOT-13	Komponente zur Umwandlung von Sprache zu Text (STT)	3	3	BOT-43	Robert Halwaß
BOT-23	Komponente zur Umwandlung von Text zu Sprache (TTS)	3	3	BOT-43	Alexis Popovski
BOT-55	Erinnerungs-Service: Behandelt „erinnere mich“ Befehle und erinnert bei Fälligkeit autonom	3	3	BOT-12	Dennis Walz
BOT-82	Termin-Scraper, der automatisch Erinnerungen aus öffentlichen Quellen bezieht	3	1	BOT-12, BOT-55	Dennis Walz
BOT-89	Moodle iCal import als Erinnerungen	3	1	BOT-12, BOT-55	Dennis Walz
BOT-75	Begrüßungsnachricht	4	1		Rim Khreis

ID	Name	Priorität	Initiale Aufwandsschätzung (in Tage)	Abhängigkeiten	Wird verticketet von
BOT-11	Universelles Scraper & Download	4	3	BOT-43	Alexis Popovski
BOT-15	Personalliste der Beuht-Hochschule im Beuthbot abrufbar machen	4	2	BOT-12, BOT-43	Lukas Danke

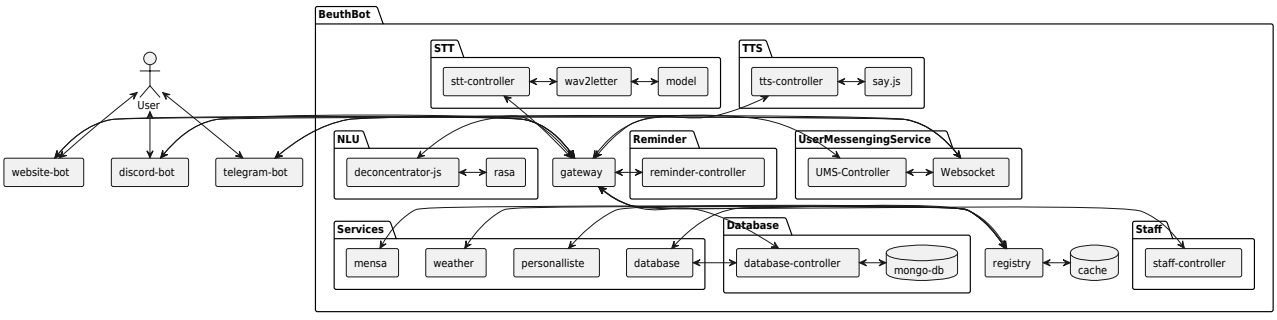


Architektur

Die Architektur-Entscheidungen orientieren sich an der bisherigen Infrastruktur. Durch die Verwendung der gleichen Programmiersprachen und Technologien wird Wartbarkeit gewährleistet und insbesondere die Abstraktion von Common-Funktionalitäten in Libraries begünstiget.

Dies bedeutet für den BHT-Bot:

- Alle Features werden als Microservice-Architektur konzipiert
- Microservices werden via Docker Container betrieben
- Docker Container werden via docker-compose orchestriert
- Microservices kommunizieren via REST-Schnittstellen innerhalb des Docker-Network
- Services werden in Javascript geschrieben
- Libraries und Frameworks werden in Typescript geschrieben



2. Aufgaben Spezifikation

BOT-16: Support mehrerer Messenger-Typen durch Umbau der Benutzererkennung

Aktueller Stand:

Der BeuthBot unterstützt aktuell nur eine Verwendung über Telegram. Die Telegram-ID des Benutzers wird in der Datenbank gespeichert.
Diese wird dann verwendet, um individuelle Informationen zu dem Benutzer abrufen zu können.

Geplanter Umbau:
In Zukunft sollen für den BeuthBot mehrere Kontaktmöglichkeit zur Verfügung gestellt werden. Damit ein Benutzer aber auch unabhängig vom Messenger erkannt wird muss der BeuthBot angepasst werden:
1. Umbau der Datenbank damit mehrere Messenger-IDs gespeichert werden können
2. Umbau der Erkennung des Messengers
3. Vereinfachen der Erkennung, damit ein zukünfiger Support von neuen Messengern einfach und schnell erfolgen kann.

Initiale Schätzung	3 Tage
Technologien	* Javascript * mongodb
Abhängigkeiten	keine
Anforderungen	* einzelne Speicherung der Telegram-ID aus der Datenbank entfernen * Neue Spalte zur Verwaltung aller Benutzer-IDs * vereinfachter Einbau von neuen Messengern gewährleisten * Abhängigkeit von der Telegram-ID entfernen * Mehrere Messenger verfügbar machen * Anmeldung eines neuen Benutzers * Löschen eines Benutzers/Messengers * Anmeldung eines neuen Messengers für ein bestehenden Benutzer
Tasks	* BOT-20 - Umbau der Datenbank - Speicherung von mehreren Accounts für einen Benutzer * BOT-21 - Anmeldung eines neuen Accounts für einen Benutzer * BOT-22 - Löschen eines bestehenden Accounts für einen Benutzer * BOT-97 - Methodik zur einfachen Erweiterung der unterstützten Messenger

2.

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.

Initiale Schätzung	3 Tage
Technologien	* Python * Rasa
Abhängigkeiten	keine
Anforderungen	* Kompatibilität mit bestehenden NLU-Trainingsdaten erhalten * Mögliche JSON- und MarkDown-Dateien in YAML-Datein umwandeln
Tasks	* BOT-62: Config Anpassen * BOT-63: Duckling updaten * BOT-64: Modell neu trainieren * BOT-116: Performance zwischen Version 1.6 und 2.0 vergleichen

2.

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 Tag
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

2.

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 Tage
Technologien	* TypeScript * JavaScript
Frameworks/ Libraries	* Angular * Bootstrap
Abhängigkeiten	* BOT-10 * BOT-30
Anforderungen	* Jeder Ressource wird ein Abschnitt gewidmet, welcher Infos & einen Link zu der jeweiligen Ressource enthält * Anschauliches, Einheitliches und responsive 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-130 Notifications wenn Nachricht vom Chatbot * BOT-131 Responsive

2.

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 Tage
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

2.

BOT-37: Discord Integration

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 Tage
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)

2.

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 Tag
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

2.

BOT-13: Komponente zur Umwandlung von Sprache zu 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.

Initiale Schätzung	3 Tage
Programmiersprachen	* Python (Mozilla Voice STT) * C++ (Kaldi, WAV2Letter) * Kaldi * Mozilla Voice STT * WAV2Letter
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-69 WAV2Letter testen * BOT-70 Mozilla Voice testen * BOT-73 Kaldi testen * BOT-71 Framework aussuchen * BOT-122 Micro-Service erstellen * BOT-123 Sprachnachricht in WAV umwandeln

2.

BOT-23: Komponente zur Umwandlung von Text zu 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 Tag
Technologien	Javascript
Abhängigkeiten	* BOT-43: Erstellung eines Common-Frameworks für (Content-)Services
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

2.

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 Tage
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

2.

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 Tag
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

2.

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 Tag
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

2.

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 Tag
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-132 Server fragt alle anderen Server was sie können/ machen und gibt das dann aus

2.

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 Tag
Technologien	Javascript
Abhängigkeiten	* BOT-43: Erstellung eines Common-Frameworks für (Content-)Services
Anforderungen	*Import von HTML- und XML-Dateien *Daten lassen sich im JSON-Format exportieren *Datensätze sind per HTML-Tags und CSS-Selektoren extrahierbar *Dateien einer Webseite lassen sich downloaden
Tasks	* BOT-26 Recherche nach geeignetster Methode (HTML-JSON)

2.

BOT-15: Personalliste der Beuth-Hochschule im BeuthBot abrufbar machen

Dem Bot wird eines neues Feature hinzugefügt. Dieses Feature soll dem Benutzer des BeuthBots ein Abfrage von Informationen über das Personal der Beuth Hochschule(BHT) ermöglichen

Ablauf:

Der Benutzer teilt dem Bot über einen Befehl mit (“Wer ist Max Mustermann?”, “Welche Person hat die E-Mail mail@mail.com?”, “Welche Personen sitzen in Raum B001?”) , dass er Informationen über eine oder mehrere Personen erhalten möchte. Der Bot prüft dann die mitgegebenen Informationen und gibt dann aus:

* Wenn er passende Daten in der Datenbank findet:
- Auflistung der angefragen Daten
* Wenn er keine passenden Daten finden kann:
- Meldung, das die Suche erfolglos war

Weitere Informationen:

Die Informationen über das Personal werden in einer Tabelle in der Datenbank gespeichert und von dort abgerufen. Diese Informationen können jederzeit aktuallisiert werden.

Spätere Erweiterungsmöglichkeiten:

Zunächst werden nur Entwickler Zugriff auf das Bearbeiten der Daten besitzen, für eine spätere Ausbaustufe ist aber eine Verwaltung der Daten mittels dafür berechtigter User vorstellbar.
Die Einbindung der Personaldaten kann über einen Scrapper auf der Seite der Personalliste auch persepektivisch komplett automatisiert werden.
Die Daten in der Datenbank können mit neuen Informationen erweitert werden. (Persönliches, Sprechzeiten, etc.)

Initiale Schätzung	3 Tage
Technologien	* Javascript * mongodb
Abhängigkeiten	keine
Anforderungen	* Auslesen der Personalliste der BHT * Einbau einer Speichermöglichkeit in der Datenbank * Speichern der ausgelesenen Informationen in der Datenbank * Schnittstelle zur Bearbeitung der Daten erstellen * Anlegen eines neuen Service „Personalliste“ im Bot * Service zur Verwendung für die Benutzer abrufbar machen
Tasks	* BOT-17 - initiales Auslesen der Personalliste * BOT-18 - Abrufen der Information aus der Personalliste * BOT-19 - Erkennung der Benutzeranfrage zur Personensuche * BOT-96 - Bearbeiten der Daten der Personalliste * BOT-125 - Erstellen einer neuen Tabelle „Personalliste“ in der Datenbank