Nabu

Chatbot-Assistent für Ausbildungszwecke

SA/BA HS 2020

OST Rapperswil-Jona

Version: 1.0

Autor (BA):	**Christoph Streiff**	Betreuer:	Stefan Keller
Autor (SA):	**Alexandra Diener**	Themengebiet:	SW Engineering
Autor (SA):	**Julia Fritsche**	Studiengang:	Informatik
		Gegenleser:	Andreas Steffen

Erstellt am: **22. September 2020**

Letzte Änderung am: **06. Januar 2021**

Abstract

Problemstellung

Chatbots sind ein immer weiter verbreitetes Mittel, um Nutzern das schnellere Erlernen eine Applikation, einfacheren Zugriff auf Informationen sowie eine natürlichere Interaktionserfahrung zu ermöglichen.

Diese Arbeit soll untersuchen, ob eine prototypische Implementation eines solchen Chatbots inklusive der notwendigen Umgebung auch für einen Hochschulkontext möglich ist. Studierende sollen über den Chatbot schneller und einfacher Zugriff auf wichtige Informationen erhalten, insbesondere auf fachliche und organisatorische Daten zu Vorlesungen.

Ziel der Arbeit

Ziel dieser Arbeit ist die Implementation eines Chatbots zur Unterstützung des Lernprozesses, der sich nach den Bedürfnissen der OST richtet. Dieser Chatbot soll zweierlei Aufgaben übernehmen können:

- Die Beantwortung von fachlichen Abfragen zu Unterrichtsmaterial unter Angabe von Quellen
- Das Bereitstellen von administrativen Informationen zu Ereignissen wie Testat-Abgaben oder Prüfungsterminen

Zur Nutzer-Interaktion mit dem Chatbot sind Client-Applikationen nötig, die ein Chat-Interface zur Verfügung stellen. Diese sollen ebenfalls im Rahmen dieser Arbeit erstellt werden.

Darüber hinaus soll eine Umgebung zur Verfügung gestellt werden, welche Dozierenden das Erfassen von Informationen aus Vorlesungsunterlagen und das Bereitstellen im Chatbot erlaubt, damit sie abgefragt werden können.

Ergebnis

Primäres Resultat der Arbeit ist ein funktionstauglicher Chatbot, genannt Nabu, der fachliche und administrative Anfragen zu Vorlesungen beantworten kann. Die zugrunde liegende Technologie ist MindMeld, ein auf Machine-Learning basierendes Konversations-Framework von Cisco, das in Python geschrieben und als Open-Source-Software verfügbar ist.

Zu diesem Chatbot gehört eine vollständige Daten-Pipeline, welche die Erfassung von Informationen aus Vorlesungsunterlagen im Keyword-Verfahren sowie von organisatorischen Daten zu wichtigen Terminen erlaubt.

Zusätzlich wurden zwei Chatbot-Frontends erstellt: Das eine ist in Moodle, die von der OST verwendete Learning-Management-Plattform, integriert. Beim anderen handelt es sich um einen Bot für die Messenger-Applikation Telegram. Über diese Clients können die erfassten Daten abgefragt werden.

Obwohl grundsätzlich voll funktionsfähig, mangelt es der Applikation noch an einigen sicherheitsrelevanten Aspekten, weswegen sie noch nicht zum unmittelbaren Deployment bereit ist.

Inhalt

1 Aufgabenstellung im Original

Nabu - Chatbot-Assistent für Ausbildungszwecke

Experimente mit Chat-Bots und Crawler für den Hochschulbetrieb, insbesondere für Informatik-Lehrveranstaltungen auf OST.ch / HSR.ch.

- Studienarbeit oder Bachelorarbeit (SA+BA): Team Alexandra Diener (SA), Julia Fritsche (SA) plus Christoph Streiff (BA), Herbstsemester 2020/2021
- Betreuer: Prof. Stefan Keller, Institut für Software / Abteilung Informatik der HSR, sowie Mitarbeiter vom Institut für Software
- Externer Partner: (-)
- Weitere Beteiligte: (-)
- Organisatorisches (wöchentl. Meetings): Jeden Mittwoch
- Dokumentation: Word via OneDrive
- Arbeitszeiten: Jeweils Mittwochmorgen plus Meeting

Hintergrund

Mit dem Fernunterricht werden immer mehr auch Chats verwendet und da könnten Text-Chat-Bots ganz nützlich sein.

Dies kann beispielsweise in den Übungen sein oder aber für Fragen zur Vorlesung (Skript) oder für organisatorische Fragen allgemein (von "remote").

Aufgabe

Grundsätzlich

- Evaluieren bestehender Chat-Bots (Open Source) sowie ggf. Search Engines und Crawlers.
- Adaptieren der geeignetsten Lösung einerseits ggf. durch Pull-Requests und andererseits durch Eigenentwicklung.
- Resultat: Webapp(s), API/Services, Plugins für Moodle/MS Teams oder weitere Chattechnologien (Open Source)

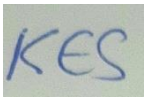
Themen

1. Chatbot beantwortet organisatorische Fragen zur Vorlesung.
2. Chatbot beantwortet während Vorlesung automatische Antworten auf Basis des Skripts und weitere Materialien (automatisch aus Moodle kopiert oder explizit pro Vorlesung konfiguriert).

Technologien

Stichworte: Artificial Intelligence, Information Retrieval, Webtechnologien (HTML5, JS), ggf. Python.

Integration des Chatbots in Moodle und/oder MS Teams (also evtl. auch beides!)



2 Management Summary

2.1 Ausgangslage

Chatbots

Immer mehr Firmen, Dienste und Websites setzen auf Chatbots und Konversationsschnittstellen, um ihren Usern eine angenehmere und natürlichere Erfahrung zu bieten [1]. Eine Konversationsschnittstelle ist ein Programminterface, das ein Gespräch von Person zu Person nachahmt und dem User dabei als Ersatz für ein traditionelleres, graphisches Interface dienen kann. Interaktionen mit Konversationsschnittstellen nehmen entweder eine stimm- oder textgesteuerte Form an und erinnern somit entweder an eine Art Telefongespräch oder eine schriftliche Chatkonversation, wie man sie aus Messenger-Apps kennt. In beiden Fällen ist das Ziel, den Aufwand für den Benutzer zu verringern und schnellen Umgang mit einer neuen Applikation zu ermöglichen.

Situation an der OST

Das gezielte Auffinden von Informationen an der OST Rapperswil-Jona kann unter Umständen mühselig sein, insbesondere wenn der oder die fragliche Studierende noch nicht viel Erfahrung mit dem Skripte-Server, der Moodle-Plattform oder unter Umständen sogar vorlesungsspezifischen Drittsystemen hat und nicht genau weiss, wo gesucht werden soll.

Aus diesem Grund besteht Bedarf nach einer niederschwelligen, hoch verfügbaren Methode, um schnell herauszufinden, wo mehr Informationen zu einem Thema verfügbar sind.

Lösungsansatz

Anhand einer prototypischen Implementierung einer möglichen Lösung soll untersucht werden, ob und wie sich der Chatbot-Ansatz zur Lösung dieser Probleme und somit zur Lernförderung und -vereinfachung im Hochschulumfeld eignet.

Idealerweise könnte ein solcher Chatbot die erste Anlaufstelle werden, an welche sich Studierende wenden können, wenn sie eine sachliche oder organisatorische Frage haben.

Dies würde eine schnellere Antwort auf diese Fragen ermöglichen, da ein Chatbot viel konsistenter verfügbar sein kann als Kommilitonen oder Dozierende. Darüber hinaus ist die Schwelle zur Nutzung sehr tief, da der Bot über natürliche Sprache angesprochen werden kann. Ausserdem verringert sich durch die wegfallenden Fragen der Studierenden der Arbeitsaufwand der Dozierenden und deren Mitarbeitenden.

2.2 Vorgehen

Ansatz

Im Rahmen dieser Arbeit soll die Erstellung eines Chatbot-Prototyps zur Lernförderung und des zu seiner Verwendung notwendigen Umfelds erarbeitet werden. Dieser Chatbot trägt den Namen Nabu nach dem altmesopotamischen Gott der Weisheit. Er soll prinzipiell zwei Aufgaben übernehmen können: Das Beantworten von fachlichen Fragen einerseits und organisatorischen Fragen andererseits. Unter erstere Kategorie fällt beispielsweise die Fragen «Was ist eine Transaktion im Fach Datenbanken 1?». Ein Beispiel für eine Frage aus der zweiten Kategorie wäre «Was muss ich in Analysis 2 während dem Semester abgeben?».

Umfeld

Zum oben erwähnten Umfeld für den Chatbot gehört einerseits ein Weg, die angesprochenen organisatorischen und fachlichen Daten für die Abfrage zur Verfügung zu stellen. Es muss also ein Tool erstellt werden, mit dem es möglich wird, Vorlesungsunterlagen zu erfassen und in die Chatbot-Wissensbasis hochzuladen.

Darüber hinaus muss eine Methode bestehen, um Abfragen am Chatbot zu machen. Dazu wird ein Chat-Interface gebraucht, über welches bidirektionale textuelle Kommunikation erfolgen kann.

Technologien

Zur Implementation des Chatbots wird das Open-Source-NLP-Framework MindMeld [2] [3] verwendet, welches von Cisco in der Programmiersprache Python erstellt wurde und als Open-Source-Software verfügbar ist. Es beinhaltet eine Wissensbasis, in welche die erfassten fachlichen und organisatorischen Daten hochgeladen werden können und so abfragbar werden.

Als Interaktionsmedium für die Chatbot-Clients wird einerseits die Learning-Management-Plattform Moodle [4], andererseits die Mobile-Messenger-App Telegram [5] verwendet, um Studierenden zwei verschiedene Möglichkeiten zu geben, auf den Chatbot-Dienst zuzugreifen. Dabei fokussiert sich Moodle eher auf eine Desktop-Anwendung, während Telegram besser auf die Anwendung von Mobilgeräten aus zugeschnitten ist.

2.3 Ergebnisse

Erreichte Ziele

Die zu Projektbeginn gesteckten Ziele wurden zum Grossteil erreicht. Primäres Resultat des Projekts ist Nabu, ein funktionstüchtiger Chatbot-Prototyp, der die oben erklärten fachlichen und administrativen Abfragen beantworten kann.

Dazu kommt die zu Nabu gehörende Datenerfassungs-Pipeline, über welche es möglich wird, Daten aus Vorlesungsunterlagen, Foliensätzen und Übungen zu erfassen, in Nabus Wissensdatenbank hochzuladen und sie so über einen Chat-Client abfragbar zu machen.

Solche Chat-Clients wurden, wie oben bereits erwähnt, für Moodle und Telegram implementiert. Der Moodle-Client richtet sich dabei eher an die Verwendung aus einer Desktop-Umgebung, während der Telegram-Client sich eher an Nutzer richtet, welche von ihrem Mobilgerät Abfragen machen möchten.

Nicht erreichte Ziele

Bedingt durch die knapp bemessene Zeit und den Charakter des Projektes als Prototypisierung mussten gewisse Kompromisse bei der Implementierung eingegangen werden. Aus diesem Grund fehlen in der Nabu-Umgebung gewisse sicherheitsrelevante Aspekte, wie beispielsweise die verschlüsselte Datenübertragung über HTTPS und die Authentifizierung und Autorisierung von Uploads in die Wissensdatenbank. Somit ist Nabu noch nicht bereit, in einer Produktionsumgebung eingesetzt zu werden.

Ausblick

Die nächste Stufe der Weiterentwicklung für das Nabu-Projekt ist der Einbau der oben erwähnten sicherheitsrelevanten Zusätze, wonach ein erster Testlauf in einer produktiven Umgebung durchgeführt werden könnte.

Danach würde sich der Einbau von verschiedenen hochschulspezifischen Features wie beispielsweise die Abfrage des Kantinenmenüs oder das An- oder Abmelden von Prüfungen als nächste Ziele anbieten.

3 Technischer Bericht

3.1 Zielsetzung

Chatbot

Primäres Ziel dieser Arbeit ist der Entwurf und die Konstruktion eines Chatbots, der sich zur Einbindung in Ausbildungsumgebungen (z.B. der Moodle-Webseite einer Hochschule) eignet und dort Nutzer beim Lernvorgang im Alltag unterstützen kann. Der Chatbot soll rein schriftlich kommunizieren können, eine Sprachsteuerung ist im Rahmen dieses Projekts nicht vorgesehen.

Zu der dazu benötigten Funktionalität zählt in erster Linie:

- das Beantworten von fachlichen Fragen zum Stoff durch Angabe von Quellen
- das Beantworten von organisatorischen Fragen zum Schulalltag

Im Falle der fachlichen Abfrage kann sich der Nutzer beim Chatbot melden und diesem seine Frage schildern. Der Chatbot durchsucht daraufhin seine interne Wissensdatenbank und gibt daraufhin die gefundenen Ergebnisse inklusive des Links zu den korrekten Unterlagen und der Seitenzahl mit der korrekten Antwort zurück.

Damit wird dem Nutzer die Zeit, die er selbst für die Durchsuchung der Unterlagen aufwenden muss, erheblich verkürzt und der Lern- und Arbeitsprozess effizienter gestaltet.

Eine organisatorische Abfrage hingegen umfasst beispielsweise Testat-Abgaben, Prüfungen oder obligatorische Gastvorlesungen – alles Ereignisse aus dem Studienalltag, über die die Studierenden im Bilde sein sollten, damit sie nicht verpasst werden. Der Chatbot soll diese Informationen ebenfalls zur Verfügung stellen können.

Umgebung

Um solche Abfragen zu ermöglichen, ist das Anlegen einer Knowledge Base notwendig, welche auf das spezifische Themengebiet zugeschnitten ist und vom Chatbot abgefragt werden kann. Ebenfalls Teil der Zielsetzung dieser Arbeit ist daher die Erstellung eines Tools, mit dem aus Rohdaten wie Vorlesungsunterlagen, Artikeln oder Foliensätzen eine solche Datenbank erstellt werden kann, die dem Chatbot als Grundlage für seine Antworten dient. Konkret gehört dazu:

- Das Parsen von Informationen aus PDFs, Word-Files und anderen Dateien, die den Studierenden zur Verfügung gestellt werden
- Das Filtern dieser Informationen nach wichtigen Begriffen und Kontext
- Das Hinterlegen dieser Informationen in einer durch den Chatbot abfragbaren Datenbank

Rahmenbedingungen

In erster Linie erfolgen die Konzipierung und Implementation des Projekts gemäss den Bedürfnissen und Umständen der OST Rapperswil-Jona, kommuniziert durch den Betreuer dieser Arbeit. Dies hat insbesondere einen Einfluss auf die verwendeten Technologien und die damit zusammenhängenden Designentscheidungen.

Ziel ist allerdings eine möglichst breite Anwendbarkeit zur Lernvereinfachung auch abseits der OST Rapperswil-Jona im Spezifischen und des Hochschulkontexts im Allgemeinen. Aus diesem Grund wird Wert auf die Verwendung von Open-Source-Technologien und die gute Anpassbarkeit der Wissensdatenbank gelegt, um möglichst vielen Standorten die einfache Adaptierung und das Deployment der Lösung zu ermöglichen.

Explizit soll der im Rahmen dieses Projektes entstandene Source Code selbst Open Source gemacht werden.

3.2 Stand der Technik

Training von Chatbots

Gemäss heutigem Stand der Technik müssen Chatbots für jeden Einsatzzweck und jedes Wissensgebiet eigens trainiert werden. Ein General-Purpose-Chatbot, der vollautomatisch Daten zu beliebigen Themen sammeln und mit Nutzern interagieren kann, existiert (noch) nicht.

Aus diesem Grund gibt es auch keinen frei verfügbaren Chatbot, der das Kernproblem dieser Arbeit ohne Weiteres lösen kann, und erst recht keinen, der eigens auf die Bedürfnisse der OST Rapperswil-Jona zugeschnitten ist. Damit geht einher, dass ein solcher Chatbot im Rahmen dieser Arbeit neu entwickelt werden muss.

Existierende Frameworks

Im Unterschied zu konkreten Chatbots existieren allerdings eine Reihe an Open-Source-Bibliotheken und Frameworks, welche die Implementierung eines solchen Chatbots ungemein vereinfachen, indem sie dem Entwickler die komplexesten Teile der Entwicklung (wie beispielsweise das Parsen von natürlicher Sprache und das Erkennen der Absicht des Nutzers) abnehmen. Auf der Basis eines solchen sogenannten NLP-Frameworks lässt sich dementsprechend ein auf spezifischere Bedürfnisse zugeschnittener Chatbot entwickeln.

Da das Designen und Erarbeiten einer neuen, eigenen NLP-Plattform den Rahmen dieser Arbeit bei Weitem sprengen würde, wurde bereits früh in der Elaboration-Phase die Entscheidung gefällt, ein solches bestehendes Framework zu verwenden (siehe Kapitel 4.2.2: Chatbot-Framework).

Defizite

Mit der Tatsache, dass jeder Chatbot in Bezug auf Wissen und sprachliche Fähigkeiten eigens trainiert werden muss, geht einher, dass es einen grossen Aufwand darstellt, einen Chatbot zu kreieren, der viele verschiedene Domains kompetent abdecken kann. Jeder Chatbot, der für einen neuen Zweck erstellt wird, muss praktisch wieder von vorne beginnen. Aus diesem Grund beschränkt sich die Zielsetzung dieses Projekts auch auf administrative und organisatorische Abfragen als Einstiegspunkt, wobei eine Erweiterung zu einem späteren Zeitpunkt denkbar wäre.

3.3 Lösungsansatz

Drei-Stufen-Konzept

Aufgrund der vielen zu Projektbeginn vorhandenen Unbekannten, insbesondere im Zusammenhang mit den verwendeten Frameworks und Technologien, wurde nach einem Drei-Stufen-Konzept vorgegangen, welches das Resultat der Arbeit in drei aufeinanderfolgende Schritte gliedert. Es handelt sich dabei um:

- **Proof of Concept** (PoC): Das Proof of Concept stellt eine Art Machbarkeitsstudie dar. In diesem Schritt gilt es zu beweisen, dass ein über das Internet erreichbarer Chatbot mit dazu erstellter Wissensdatenbank bereitgestellt werden kann, mit dem eine Remote-Abfrage dieser Datenbank möglich ist. Dazu gehören:
    - Eine Recherche, Evaluation und Selektion der für das Projekt zu verwendenden Technologien, Standards und Ressourcen.

  - Das Implementieren einer kleinen Testapplikation, die aus Front- und Backend besteht und aufzeigt, dass die gewählten Technologien für das Erreichen der weiteren Ziele hinreichend sind.

Ausdrücklich nicht Ziel dieses Schrittes ist das Erstellen einer Wissendatenbank auf Basis der Kursdaten der OST Rapperswil-Jona.

- **Minimum Viable Product** (MVP): Das MVP umfasst die eigentliche Zielsetzung dieser Arbeit. Es handelt sich dabei um all jene Features, die mindestens vorhanden sein müssen, um die Applikation in einer Produktionsumgebung einsetzen zu können. Im Spezifischen sind dies:
  - Ein Chatbot-Backend («Nabu»), das über das Internet Nachrichten entgegennehmen und verarbeiten sowie auf Basis dieser Nachrichten Antworten aus der Wissensdatenbank liefern kann.
  - Ein Backend-Modul («Librarian»), welches die fachlichen und organisatorischen Informationen der OST Rapperswil-Jona entgegennimmt und in ein Format bringt, das als Wissensdatenbank für den Chatbot dienen kann. Somit wird die Datenbank der OST für den Chatbot durchsuchbar gemacht.
  - Ein dazu gehörendes kommandozeilenbasiertes Frontend für den Librarian, mit dem Daten zur Persistierung an den Librarian geschickt werden können.
  - Ein Frontend für die Moodle-Plattform, die von der OST Rapperswil-Jona verwendet wird, welches zur Kommunikation mit dem Chatbot dient.
  - Ein Frontend für die Telegram-Nachrichtenapplikation, mit welcher ebenfalls mit dem Chatbot kommuniziert werden kann.
- **Stretch Goals** (SG): Bei den Stretch Goals handelt es sich um zur Funktionstüchtigkeit des Resultats nicht zwingend notwendige Teile des Projektes, die mit grossem zusätzlichem Aufwand verbunden sind und aus diesem Grund nicht zum MVP gehören. Die Stretch Goals werden dann in Angriff genommen, wenn die Implementation des MVP schneller als erwartet fortschreitet. Dazu gehören:
  - Implementation eines Autorisierungsmanagement-Moduls für Zugriffe auf die Wissensdatenbank.
  - Implementation eines grösseren Frontends (z.B. Website) für das Librarian-Modul als nächste Ausbaustufe des CLI-Frontends.
  - Implementation von Deutsch als zweiter verfügbarer Sprache für den Chatbot.

Deliverables

Die untenstehende Tabelle zeigt die Deliverables auf, die anhand des oben erwähnten Drei-Stufen-Konzeptes während der Elaborationsphase in Zusammenarbeit mit dem Betreuer dieses Projektes definiert wurden.

ID	Deliverable	Format
PoC1	Lauffähiges Proof of Concept	Unspezifiziert
MVP1	Chatbot-Backend	Docker-Image
MVP2	Librarian-Applikation	Docker-Image
MVP3	Moodle-Frontend	Moodle-Plugin
MVP4	Telegram-Frontend	Docker-Image
MVP5	Librarian-Frontend	Python-Applikation
SG1	Erweitertes Librarian-Frontend	Unspezifiziert

Tabelle 1: Deliverables

3.4 Resultate

Zusammenfassung

Das Minimum Viable Product wurde zu grossen Teilen erreicht und die Funktionalität der gesamten Chatbot-Pipeline vom Erfassen und Speichern der Daten bis zur Abfrage wurde erstellt. Aus Zeitgründen mussten allerdings gewisse Kompromisse bei den nicht-funktionalen Anforderungen eingegangen werden.

Eine genauere Analyse davon, welche Ziele erreicht und nicht erreicht wurden, sowie einen Ausblick auf mögliche Weiterentwicklung und konkrete Ansätze dazu findet sich in Kapitel 4.4.

4 Projektdokumentation

4.1 Anforderungsspezifikation

Definition

Die Festlegung der ersten Iteration der Anforderungsspezifikation erfolgte in der Elaborationsphase als Teil des Analyse- und Prototypisierungsprozesses. Dabei entstanden Use Cases, nichtfunktionale Anforderungen und Diagramme zur grundlegenden Architektur des Projektes. Insbesondere letztere wurden danach in verschiedenen Iterationen überarbeitet und verfeinert.

4.1.1 Use Cases

Use-Case-Diagramm

Als Übersicht über die drei aus der Problemstellung hergeleiteten Use Cases dient das folgende Use-Case-Diagramm:

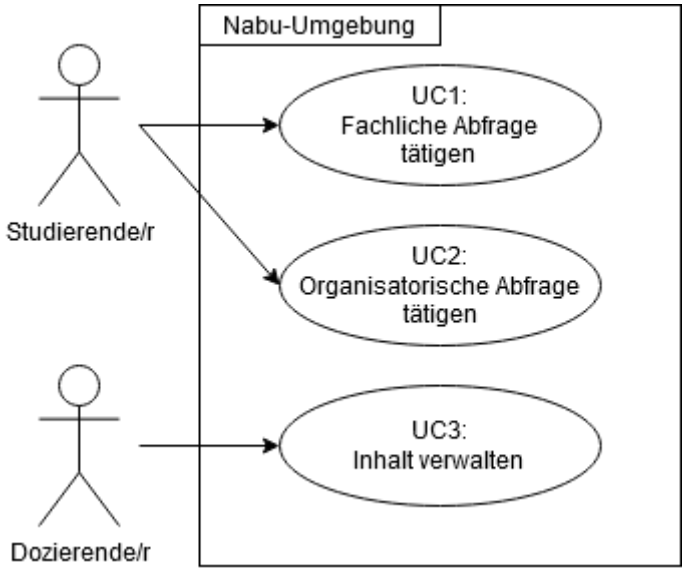


Abbildung 1: Use-Case-Diagramm

Actors

Im Rahmen der Use-Case-Analyse wurden zwei Actors erfasst. Dabei handelt es sich um:

- **Studierende/r:** Im Scope dieser Arbeit wird als Studierender derjenige Nutzer aufgefasst, der Abfragen auf dem Chatbot machen möchte, sprich derjenige, dessen Lernprozess unterstützt werden soll. Diese Namensgebung ist unabhängig von der tatsächlichen Verwendung des Chatbots im oder ausserhalb des Hochschulkontexts.
- **Dozierende/r:** Als Dozierender wird derjenige Nutzer verstanden, der Zugriff auf das Backend des Chatbots hat und sich (alleine oder mit anderen Dozierenden) um die Erstellung und Aufrechterhaltung der Wissensdatenbank kümmert. Wiederum ist die Namensgebung unabhängig davon, ob der Chatbot im Kontext einer Hochschule angewandt wird oder nicht.

UC1: Fachliche Abfrage tätigen

Hauptactor:

- Studierende/r

Interessen:

- Studierende/r will schnelle und korrekte Beantwortung der gestellten Frage.

Bedingungen:

- Der fragliche Stoff ist in der Knowledge Base erfasst und somit für die Konversationsschnittstelle abfragbar.

Haupterfolgsszenario:

1. User formuliert eine fachliche Frage, die für ihn im Moment von Interesse ist.
2. User öffnet Chat-Interface zur Konversationsschnittstelle und eröffnet die Konversation.
3. User legt fest, zu welcher Learning Unit seine Frage gehört.
4. User stellt seine Frage an die Konversationsschnittstelle und erhält von Schnittstelle Antwort.
5. User beendet Konversation.

Erweiterungen:

- 3a: User überspringt Unit-Festlegung und stellt Frage direkt.
    1. Chatbot versucht aus Fragestellung die Learning Unit zu extrapolieren.
    2. Chatbot fragt nach, ob die Unit korrekt identifiziert wurde.
    3. User kann klarstellen, ob die Unit korrekt ist oder die richtige Unit angeben.
- 4a: User will weitere Fragen zur selben Learning Unit stellen.
    1. User stellt nach Beantwortung der vorherigen Frage eine neue Frage.
    2. User erhält von Schnittstelle Antwort auf die neue Frage.
- 4b: User will Frage zu einer anderen Learning Unit stellen.
    1. User äussert den Wunsch, eine andere Unit abzufragen.
    2. Chatbot passt Kontext der Konversation entsprechend an.
    3. User kann Fragen zur neuen Unit stellen.
- 5a: User will nach fachlicher Abfrage eine organisatorische Abfrage tätigen.
    1. Abfrage fährt gemäss UC2 fort.

Häufigkeit:

- Sehr häufig.

UC2: Organisatorische Abfrage tätigen

Hauptactor:

- Studierende/r

Interessen:

- Studierende/r will schnelle und korrekte Beantwortung der gestellten Frage.

Bedingungen:

- Die fraglichen organisatorische Daten sind in der Knowledge Base erfasst und somit für die Konversationsschnittstelle abfragbar.

Haupterfolgsszenario:

1. User formuliert eine organisatorische Frage, die für ihn im Moment von Interesse ist.
2. User öffnet Chat-Interface zur Konversationsschnittstelle und eröffnet die Konversation.
3. User legt fest, zu welcher Learning Unit seine Frage gehört.
4. User stellt seine Frage an die Konversationsschnittstelle und erhält von Schnittstelle Antwort.
5. User beendet Konversation.

Erweiterungen:

- 3a: User überspringt Unit-Festlegung und stellt Frage direkt.
    1. Chatbot versucht aus Fragestellung die Learning Unit zu extrapolieren.
    2. Chatbot fragt nach, ob die Unit korrekt identifiziert wurde.
    3. User kann klarstellen, ob die Unit korrekt ist oder die richtige Unit angeben.
- 4a: User will weitere Fragen zur selben Learning Unit stellen.
    1. User stellt nach Beantwortung der vorherigen Frage eine neue Frage.
    2. User erhält von Schnittstelle Antwort auf die neue Frage.
- 4b: User will Frage zu einer anderen Learning Unit stellen.
    1. User äussert den Wunsch, eine andere Unit abzufragen.
    2. Chatbot passt Kontext der Konversation entsprechend an.
    3. User kann Fragen zur neuen Unit stellen.
- 5a: User will nach organisatorischer Abfrage eine fachliche Abfrage tätigen.
    1. Abfrage fährt gemäss UC1 fort.

Häufigkeit:

- Häufig.

UC3: Inhalte verwalten

Hauptactor:

- Dozierende/r

Interessen:

- Dozierende/r will effektive und effiziente Erfassung von fachlichen und/oder administrativen Daten zu Learning Unit.

Bedingungen:

- Fachliche und/oder organisatorische Daten haben kompatibles Format und sind aktuell.

Haupterfolgsszenario:

1. User kompiliert Liste von zu erfassendem Material in verfügbaren Formaten.
2. User übergibt Materialliste an Applikation.
3. Applikation parst Liste Datei für Datei und fügt sie in Knowledge Base ein.

Erweiterungen:

- 3a: Einzelne Datei ist nicht von kompatiblem Format.
    1. Applikation meldet, dass Datei übersprungen wird.
    2. Parsing-Vorgang fährt nahtlos bei nächster Datei weiter.
- 3b: Es sind keine Daten in der Datei vorhanden.
    1. Applikation meldet, dass keine Daten vorhanden sind.
    2. Parsing-Vorgang fährt nahtlos bei nächster Datei weiter.
- 3c: Die Daten konnten nicht in der Librarian-Datenbank abgelegt werden.
    1. Applikation meldet, dass ein Fehler aufgetreten ist.
    2. Parsing-Vorgang fährt nahtlos bei nächster Datei weiter.

Häufigkeit:

- Gelegentlich.

4.1.2 Nichtfunktionale Anforderungen (NFRs)

Überblick

Bedingt durch den mehrteiligen Aufbau der Applikation wurden die nichtfunktionalen Anforderungen pro Modul erfasst. Die Erfassung erfolgte auf der Basis der in ISO/IEC 9126 [6] definierten Kriterien.

Chatbot

- **Effizienz**:
  - Zeitverhalten:
    - Für eine inhaltliche Abfrage (gemäss UC1) darf die Wartezeit nicht länger als 4 Sekunden betragen, gemessen mit einem einzelnen Zugriff (nicht unter Last). Die Wartezeit wird definiert als Zeit zwischen Eingang der Anfrage und Ausgang der Antwort ans Netzwerk.
    - Für eine organisatorische Abfrage (gemäss UC2 darf die Wartezeit nicht länger als 2 Sekunden betragen, gemessen mit einem Zugriff zur gleichen Zeit. Die Wartezeit wird definiert als Zeit zwischen Eingang der Anfrage und Ausgang der Antwort ans Netzwerk.
- **Zuverlässigkeit**:
  - Fehlertoleranz:
    - Bei nicht klar bearbeitbaren Abfragen fragt der Chatbot beim User nach, ob die Anfrage korrekt verstanden wurde und gibt dem User die Möglichkeit, seine Eingaben zu präzisieren.
    - Bei Fehlern in der Verarbeitung der Anfrage, die zum endgültigen Fehlschlag führen, wird der User entsprechend benachrichtigt.
- **Übertragbarkeit**:
  - Anpassbarkeit:
    - Die dem Chatbot hinterlegte Wissensdatenbank ist vollständig vom Hoster verwaltbar und anpassbar. Als vereinfachendes Werkzeug dafür existiert der Librarian.
  - Installierbarkeit:
    - Die Applikation läuft in einem Docker-Container und ist entsprechend als Image installierbar.

Librarian

- **Effizienz**:
  - Zeitverhalten:
    - Eine Datei von 1MB Grösse zu parsen, darf nicht länger dauern als 30 Sekunden.
    - Dem User wird angeboten, Dateien in eine Queue zu setzen und sequentiell zu parsen, um die Wartezeit handhabbar zu machen.
- **Zuverlässigkeit**:
  - Fehlertoleranz:
    - Bei beschädigten oder nicht lesbaren Dateien wird beim Einlesen eine Warnung angezeigt, dass ein Parsen nicht möglich war und die Datei übersprungen wird.

Moodle-Frontend

- **Effizienz**:
  - Zeitverhalten:
    - Die Antwort vom Chatbot (egal ob die Frage organisatorischer oder fachlicher Natur war) sollte innerhalb von 5 Sekunden nach Absenden im Nachrichtenfenster angezeigt werden.

- **Funktionalität**:
  - Sicherheit:
    - Die Daten, die zwischen Front- und Backend ausgetauscht werden, sind mit HTTPS verschlüsselt
  - Konformität:
    - Das Format des Plugins ist mit der von der Moodle-Plattform vorgegebenen Plugin-Struktur konform.
- **Übertragbarkeit**:
  - Installierbarkeit:
    - Die Applikation (inkl. Moodle) läuft standalone in einem Docker-Container und ist entsprechend als Image installierbar.
    - Sollte ein bestehender Moodle-Server vorliegen, so kann stattdessen ein Plugin-Folder zur Installation verwendet werden.

Telegram-Frontend

- **Effizienz**:
  - Zeitverhalten:
    - Die Antwort vom Chatbot (egal ob die Frage organisatorischer oder fachlicher Natur war) sollte innerhalb von 5 Sekunden nach Absenden im Nachrichtenfenster angezeigt werden.
- **Funktionalität**:
  - Sicherheit:
    - Die Daten, die zwischen Front- und Backend ausgetauscht werden, sind mit HTTPS verschlüsselt.
- **Übertragbarkeit**:
  - Installierbarkeit:
    - Die Applikation läuft im einem Docker-Container und ist entsprechend als Image installierbar.

4.1.3 Weitere Anforderungen

Natürlichkeit der Interaktion

In erster Linie soll der Nabu-Chatbot die in Use Cases 1 und 2 definierten Aufgaben erledigen und dem Nutzer Antwort auf seine Fragen geben können. Bedingt durch den Charakter einer konversationellen Schnittstelle, also dem Vortäuschen eines Gespräches oder Chats, ist es ein sekundäres Ziel, dem User eine gewisse Natürlichkeit des Gesprächsverlaufs anzubieten. Dies ist zur primären Zielerfüllung zwar nicht strikt notwendig, wurde aber trotzdem als Anforderung definiert.

Dazu gehört beispielsweise die Möglichkeit, den Nutzer zu grüssen oder sich zu verabschieden, wenn die Interaktion vorbei ist. Ebenfalls darunter fällt das Definieren von verschiedenen, äquivalenten Antworten (zum Beispiel «Alright, you can ask me about Datenbanken 1 now» und «Got it, we're talking about Datenbanken 1»), um den Verlauf des Gesprächs von Abfrage zu Abfrage ein wenig anders zu gestalten und zu personalisieren.

4.1.4 Domain-Modell

Überblick

Die primär zu verwaltenden Objekte der Applikation sind die Einträge in der Wissensdatenbank des Chatbots. Das Domain-Modell an und für sich beschränkt sich dementsprechend auf diese Daten. In untenstehender Abbildung ist ein Überblick des Aufbaus der Struktur ersichtlich, nachfolgend folgt eine Erklärung der einzelnen Bestandteile.

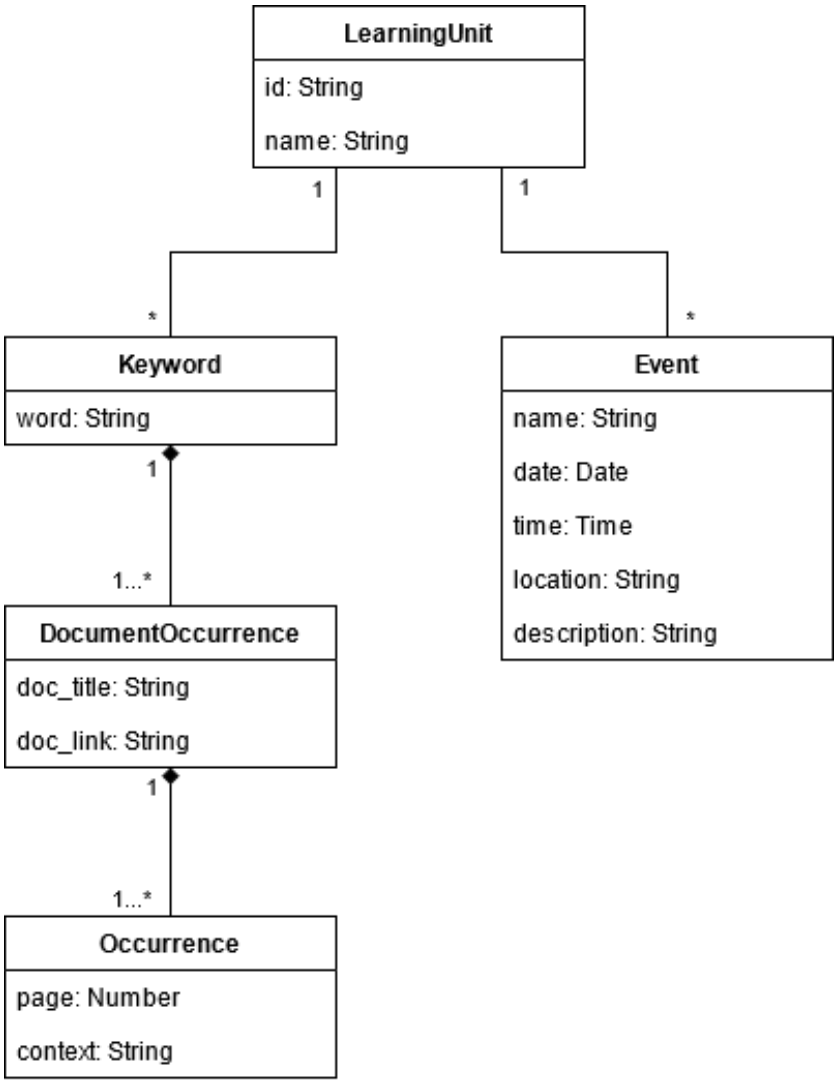


Abbildung 2: Domainmodell

Learning Unit

Die Learning Unit entspricht im engeren Sinne einem Kurs an einer Hochschule, der von Studierenden belegt werden kann und von einem oder mehreren Dozierenden abgehalten wird, beispielsweise «Datenbanken 2» oder «Webdesign 1». Im weiteren Sinn kann eine Learning Unit als Sammlung von zusammenhängenden fachlichen und organisatorischen Informationen zu einem bestimmten Themengebiet verstanden werden, die gebündelt abgefragt werden können.

Um diesen Zweck zu erfüllen, muss die Learning Unit zwei Aufgaben übernehmen können:

- Einerseits muss eine Learning Unit ihren Inhalt auf einen Blick für Abfragen verständlich repräsentieren können. Dazu dient das Namensattribut, das in der Praxis mit Werten wie «Datenbanken 2» oder «Einführung Geodaten» abgefüllt werden kann, die zum schnellen Überblick über den Inhalt dienen.

- Andererseits muss eine Learning Unit eindeutig identifizierbar sein, wozu das Attribut ID dient. Dieses kann zum Beispiel den String «DBS1» (Datenbanken 1) oder «BuRe1» (Business und Recht 1) beinhalten.

Fachliche Daten

Die linke Seite des Domänendiagramms repräsentiert die fachlichen Daten, welche die eine Hälfte der Learning Unit ausmachen. Dabei geht es in erster Linie um die Abfrage von Schlüsselwörtern (Keywords), den Verweis auf Dokumente, wo diese vorkommen und das Etablieren ihres Kontexts. Keywords werden absichtlich nur im Rahmen einer Learning Unit gespeichert, da gleiche Wörter in anderen Learning Units eine andere Bedeutung haben können (z.B. hat das Wort «Transaktion» im Zusammenhang mit Datenbanken und Wirtschaft eine andere Bedeutung). Zwischen Learning Unit und Keyword existiert eine 0-zu-n-Beziehung. Dies begründet sich darin, dass eine Learning Unit beliebig viele Keywords halten kann, aber nicht zwingend überhaupt fachlichen Inhalt haben muss – es könnten beispielsweise auch nur Events abgespeichert werden.

Ein Keyword speichert in erster Linie nur das suchbare Wort an und für sich. Darüber hinaus hält es eine Liste von DocumentOccurrences, die beschreiben, in welchen Dokumenten das Wort vorkommt und darüber hinaus einen Link zu dieser Quelle zur Verfügung stellen. Dies ermöglicht eine rasche Prüfung, ob ein Dokument ein bestimmtes Keyword enthält oder ob es beim Suchen und Zusammenstellen einer Antwort vollständig ignoriert werden kann. Zwischen Keyword und DocumentOccurrence besteht eine 1-zu-1…n-Beziehung, da ein Keyword in beliebig vielen Dokumenten vorkommen kann. Dabei ist aber jeweils nur ein Eintrag pro Dokument nötig (vorhanden oder nicht vorhanden), und wenn ein Keyword in keinem Dokument vorkommt, wird es gar nicht erst gelistet.

Innerhalb einer DocumentOccurrence gibt es eine Liste von Occurrences. Diese speichern die genaue Seitenzahl des Auftretens im Dokument. Zusammen mit der in der Hierarchie darüberstehenden DocumentOccurrence kann damit ein eindeutiger Verweis auf ein Vorkommen eines Keywords erstellt werden. In der Occurrence wird darüber hinaus der Kontext mitgespeichert, also die unmittelbare textuelle Nachbarschaft des Keywords innerhalb eines Satzes oder grösseren Textbausteins. Dies soll darüber Auskunft erteilen können, was der Zusammenhang der jeweiligen Occurrence ist beziehungsweise welcher Informationsgehalt ihr entnommen werden kann. Aus Occurrences mit mehr Kontext lässt sich direkter eine Antwort auf die ursprüngliche Frage ablesen als aus solchen, die nur das Keyword selbst enthalten.

Organisatorische Daten

Auf der rechten Seite des Domänendiagramms findet sich die Repräsentation der organisatorischen Daten, um die eine Learning Unit ergänzt werden kann. Diese sollen eingesetzt werden können, um verschiedenste Anlässe zu repräsentieren, wie beispielsweise das Abgabedatums den Beginn eines bestimmten Seminars oder die zu einer Prüfung zugehörigen Informationen (Datum, Zeit, Ort). Wie bei den Keywords besteht hier eine 1-zu-0…n-Beziehung, da eine LearningUnit nicht zwingend Events aufweisen muss.

Da Events eine grosse Bandbreite an tatsächlichen Terminen und Vorkommnissen repräsentieren können, wurden sie bewusst relativ vage gehalten, damit sie diese Vielseitigkeit widerspiegeln können. Wichtig ist insbesondere, dass die Felder «Time», «Location» und «Description» nicht zwingend ausgefüllt werden müssen, da sie unter gewissen Umständen schlicht keinen Sinn machen (ein Testabgabetermin z.B. hat keinen definierten Ort, wo er stattfindet).

4.2 Design

Externe Dokumentationen und Verweise

In diesem Kapitel wird wiederholt auf Technologien eingegangen, die für dieses Projekt verwendet oder in Betracht gezogen wurden. Die Wiedergabe aller technischen Facetten und Details dieser Technologien kann allerdings nicht im Rahmen dieser Dokumentation liegen. Deshalb wird, wo nötig, ein Konzept kurz erklärt und für weitere Ausführungen auf die Konsultation der offiziellen Dokumentation zu den betreffenden Frameworks verwiesen.

4.2.1 Namensgebung

Ursprung des Namens

Der Name «Nabu» sowohl für das Projekt als auch für den Chatbot selbst wurde zum Ende der Elaborationsphase gewählt. Es handelt sich dabei um den altmesopotamischen Gott der Belesenheit, der Wissenschaften, der Weisheit und der Schreiber.

Von Beginn weg war ein kurzer, einprägsamer Name gefragt, mit dem der Bot angesprochen werden könnte und der idealerweise mit Wissen, Lernen oder Weisheit assoziiert ist. Andere Kandidaten waren beispielsweise Pallas oder Minerva. Doch zu den meisten römischen beziehungsweise griechischen Gottheiten existiert bereits eine Technologie mit diesem Namen, weshalb die Wahl schlussendlich auf Nabu fiel.

4.2.2 Chatbot-Framework

Begründung

Schon früh in der Elaborationsphase war klar, dass das Erstellen einer eigenen NLP-Verarbeitungskette einerseits den Rahmen dieser Arbeit sprengen würde und andererseits eine enorm tiefe Einarbeitung in computerlinguistische Prinzipien und Machine Learning erfordern würde, für welche die Zeit schlicht zu knapp bemessen wäre.

In dieser Tatsache begründet sich die Entscheidung, stattdessen ein existierendes Chatbot-Framework zu verwenden und in Kauf zu nehmen, dass sich die Architektur des Projektes den durch dieses Framework vorgegebenen Umständen (Format der Datenhaltung, Aufbau und Ablauf einer Konversation, Serverstruktur) anpassen muss.

Kriterien

Bei der Auswahl des Chatbot-Frameworks waren die folgenden Kriterien relevant:

- **Open-Source**: Für das gesamte Projekt wurde eingangs definiert, dass die verwendeten Technologien möglichst open-source sein sollten. Dies begründet sich damit, dass das Resultat so möglichst vielseitig verwendet werden kann (an Hochschulen, in Unternehmen, privat…), ohne dass die Lizenzierung zum Problem wird.
- **Gute Dokumentation**: Um ein effizientes Einarbeiten zu ermöglichen und die Zeit, die mit dem Testen von Lösungen im Trial-and-Error-Prinzip verloren geht, zu minimieren, wurde von Anfang der Evaluation an darauf geachtet, dass in Betracht kommende Lösungen gut dokumentiert sind.
- **Erweiterbarkeit**: Durch den Fokus auf die Möglichkeit der Verwendung über das Umfeld der OST Rapperswil-Jona hinaus wurde darauf geachtet, dass es möglich ist, die Wissensbasis einfach auszutauschen, im Idealfall sogar nach dem «Plug-and-Play»-Konzept.
- **Skalierbarkeit**: Um das Resultat des Projektes auch grossen Institutionen zur Verfügung stellen zu können, ist die Skalierbarkeit ebenfalls ein wichtiger Aspekt.

- **Mehrsprachigkeit**: Da die OST Vorlesungen sowohl in Deutsch als auch in Englisch anbietet, wurde die Mehrsprachigkeit zusätzlich als Kriterium aufgenommen.

Die Kriterien wurden jeweils auf einer Skala von 1 bis 5 bewertet (wobei 5 die beste Note ist), mit Ausnahme des Open-Source-Status der evaluierten Technologie, welcher ein Killerkriterium darstellte. Lösungen, die es nicht erfüllten, wurden nicht weiter in Betracht gezogen. Darunter fielen zum Beispiel Amazon Lex [7] oder Azure Bot Service [8].

Evaluierte Technologien

Die folgenden Technologien wurden evaluiert:

- **MindMeld** [2]: Das MindMeld-Framework ist ein etabliertes Chatbot-Framework aus dem Hause Cisco, das zu Beginn der Arbeit auch vom Betreuer als Beispiel aufgeführt wurde. Zu Beginn noch proprietär, wurde es 2019 von Cisco als Open-Source-Code freigegeben und fiel somit in die engere Auswahl dieses Projektes. MindMeld basiert auf der Programmiersprache Python und nutzt im Hintergrund das Elasticsearch-Framework [9], um die interne Wissensdatenbank zu durchsuchen.
  MindMeld überzeugt insbesondere durch die gute Dokumentation und die verfügbaren offiziellen Tutorials [10], die den Aufwand für den Einstieg in ein doch recht umfangreiches Framework zumindest zum Teil reduzieren können.
  Darüber hinaus bietet MindMeld bietet dem Nutzer sehr viele Freiheiten, um den Ablauf von Konversationen und die Handhabung der Daten selbst festzulegen. Insbesondere sticht die Knowledge Base hervor, deren Anlegung und Struktur zum Grossteil vom Programmierer bestimmt werden kann. Sie basiert auf Daten-JSONs, die zur Laufzeit von Elasticsearch indexiert und geladen werden. Damit wird das Austauschen der Wissensbasis zu einer einfachen Verschiebung von Dateien. Ebenfalls ein Pluspunkt bezüglich der Erweiterbarkeit ist das programmatische Implementieren von Konversationsabläufen in Python, mit dem eine sehr detaillierte Kontrolle über den Verlauf der Konversation sowie die Interaktion mit anderen Komponenten möglich wird. Insbesondere ermöglicht es dieser Ansatz, den Code um beliebige eigene Konstrukte zu erweitern.
  Aus diesem Grund eignet sich MindMeld auch für skalierende Deployments, da die MindMeld-Applikation selbst hinter einem Webserver laufen kann. Unter Zuhilfenahme eines Load Balancers könnten später entweder lokal oder über ein Netzwerk verteilt mehrere Instanzen parallel laufen.
  MindMeld bietet ab Werk nicht nur Support für Mehrsprachigkeit für die meisten dem Englischen ähnlichen Sprachen an (was unter anderem Deutsch beinhaltet), sondern offeriert sogar ein Lokalisierungsfeature, das Datenformate (wie z.B. Uhrzeit, Datum oder sogar bestimmte Feiertage) automatisch in das Format setzen kann, das sich der User gewohnt ist.
- **Rasa** [11]: Rasa ist ein Open-Source Chatbot-Framework, welches als Startup lanciert wurde. Im Gegensatz zu MindMeld findet das Abwickeln von User-Interaktion weniger programmatisch statt, vielmehr setzt Rasa auf einen deklarativen Ansatz. Dabei wird der Ablauf von verschiedenen Konversationspfaden in YAML-Dateien definiert statt direkt in Python implementiert. Damit geht einher, dass der Programmierer weniger granulare Kontrolle über den Programmablauf und die Details der Ausführung hat. Es ist zwar dennoch möglich, als Teil einer Konversation eigenen Code auszuführen, aber nur über Umwege.
  Einer der Punkte, die bei Rasa hervorstechen, ist das Conversation-Driven Development (CDD), bei dem tatsächliche Daten aus mit Nutzern geführten Konversationen wieder in die Entwicklung einfliessen. Dies ermöglicht ein optimales Anpassen des Bots an die in der Realität beobachteten Gegebenheiten und Arten der Konversationsführung.
  Rasa wird direkt mit einem HTTP-API ausgeliefert, über welches der Chatbot

erreicht werden kann. Darin lässt sich u.a. die Anzahl Arbeiter-Threads einstellen, und wie bei MindMeld ist Load Balancing eine Option. Damit steht Rasa MindMeld bezüglich der Skalierbarkeit ebenbürtig gegenüber.
Die Dokumentation von Rasa konnte allerdings trotz der Existenz eines Online-Sandkastens, der erste Schritte mit Rasa erlaubt, nicht wirklich überzeugen. Dies machte den Einstieg in das Framework und die Evaluation etwas aufwändiger.
Rasa bietet die Option an, Chatbots in jeder beliebigen Sprache zu trainieren, vorausgesetzt, es existieren genügend Trainingsdaten dafür. Dazu werden einige vortrainierte Modelle für gängige Sprachen angeboten. Support für Lokalisierung ist allerdings ab Werk nicht vorhanden.

- **OpenDialog** [12]: OpenDialog ist ein lizenzfreies Open-Source-Framework, um Conversational Applications zu bauen und zu managen. Die Konversationen werden dabei über die sogenannte Conversation Description Language, ein YAML-basiertes Format, erstellt und als Graphen gespeichert. Diese werden danach traversiert, um den Konversationsfluss zu tracken. Die Modellierung der Interaktion von Chatbot und Nutzer wird dabei als Multi-Agent-System angesehen, wobei sowohl der Bot als auch der Mensch einen Agenten darstellen.
  Die Plattform bietet Support für diverse Frontends an und erlaubt es, mittels einem eigenen Markup-Dialekt von XML ein einheitliches Format für Nachrichten festzulegen, das dann vom jeweiligen Frontend konkret dargestellt wird.
  OpenDialog befindet sich zum Zeitpunkt der Abgabe dieser Arbeit (Januar 2021) noch im Aufbau, weswegen die Dokumentation sehr lückenhaft und unvollständig ist. Zu den versprochenen Features gehört aber Mehrsprachigkeit und gute Skalierbarkeit. Trotzdem schlug sich diese Tatsache erheblich auf die Bewertung nieder.

Evaluation

	MindMeld	Rasa	OpenDialog
Dokumentation	4	3	2
Erweiterbarkeit	5	4	3
Skalierbarkeit	4	4	3
Mehrsprachigkeit	5	4	2
Total	**18**	**15**	**10**

Entscheidung

Basierend auf der obenstehenden Evaluationstabelle fiel die Entscheidung auf MindMeld. Damit ging einher, dass das Chatbot-Backend in Python geschrieben werden musste, da MindMeld darin implementiert ist.

Um durch voneinander abweichende Programmiersprachen entstehenden Overhead wie beispielsweise das Einrichten von separaten Testumgebungen zwischen den Modulen zu vermeiden, wurde definiert, dass Python die primäre Programmiersprache für das gesamte Projekt ist und wo immer möglich verwendet wird. Einzige Ausnahme hierzu ist das Moodle-Frontend, da Moodle auf PHP basiert und Plugins sich dementsprechend an diese Sprache halten müssen.

4.2.3 Zusätzliche Libraries und Frameworks

Docker

Um ein einfaches Deployment in vielen Umgebungen zu garantieren, wurde wo möglich das Virtualisierungstool **Docker** [13] als Deployment-Option bestimmt. Daraus entsteht der massive Vorteil, dass der Code nicht mehr an die physischen Gegebenheiten der Deployment-Umgebung gebunden ist. Darüber hinaus wird der Installationsprozess

vereinfacht, da alles über das Pullen eines Docker-Images und das Ausführen desselben erledigt werden kann.

Einzige Ausnahme hierbei ist das CLI-Frontend des Librarians, da es nicht als ein zentral erreichbarer Service implementiert wurde und direkte Nutzerinteraktion erfordert. In diesem Modul wird stattdessen mit **pip** gearbeitet (siehe Kapitel 4.2.10).

Flask

Während der Elaborationsphase stellte sich heraus, dass ein oder sogar mehrere Web-Server zum Scope des Nabu-Projektes gehören würden, um als Schnittstelle zwischen Chatbot und Studierenden oder zwischen den einzelnen Modulen der Applikation dienen zu können. Dazu wurden die beiden gängigen Python-Open-Source-Frameworks Django [14] und Flask [15] in Betracht gezogen.

Bei Django handelt es sich um ein Full-Stack-Framework, das dem Developer direkt ab Werk Zugriff auf mächtige Tools wie Authentifizierung, OR-Mapping und ein Administrations-Panel gewährt. Die Designphilosophie lautet «batteries included», was darauf hindeuten soll, dass der Programmierer selbst sich nicht mit der Implementation von weitverbreiteten Features befassen muss, sondern dass diese direkt zur Verfügung stehen.

Flask, welches auf der WSGI-Library Wekzeug [16] basiert, nennt sich selbst ein «Microframework» und verfolgt einen minimalistischeren Ansatz, der von Beginn an nur das Nötigste wie beispielsweise Routing und Error-Handling bereitstellt. Für die meisten gängigen Features, die Django direkt anbietet, existieren verschiedene Erweiterungspakete, damit nicht von Grund auf alles selbst implementiert werden muss, aber der Programmierer trotzdem die Wahl hat, wie er seine Features anbieten will. Flask selbst ist aber bewusst leichtgewichtig und flexibel gehalten.

Darin begründet sich auch die Entscheidung, für Python-Webserver in diesem Projekt **Flask** einzusetzen. Da der Grossteil der von Django direkt angebotenen Features im Rahmen des Nabu-Entwicklungsprozesses nicht notwendig ist, wäre Django überwiegend Ballast und der Nutzen, der aus der zusätzlichen Grösse und Komplexität erwachsen würde, eher gering. Durch die Verwendung von Flask werden die für das Nabu-Projekt wichtigen, grundlegenden Features wie Routing und Verarbeitung von POST-Requests verfügbar, während die Erweiterbarkeit trotzdem gewährleistet bleibt, sollte sich ein Bedarf nach zusätzlichen Features einstellen.

Waitress

Bei WSGI [17] (Web Server Gateway Interface, ausgesprochen «Whiskey») handelt es sich um einen python-spezifischen Calling-Standard zwischen Webserver und dahinter arbeitendem Webframework gemäss PEP 3333, der auch von Flask implementiert wird.

Flask wird mit einem eigenen WSGI-Server ausgeliefert, allerdings ist dieser gemäss Dokumentation nicht zum Deployment in eine Produktionsumgebung, sondern nur für Test- und Debugzwecke geeignet [18], da einzelne Requests zwar kein Problem darstellen, aber die Skalierbarkeit für viele parallele Anfragen nicht ausreicht. Eine entsprechende Meldung wird auch beim Starten des Servers direkt auf der Konsole ausgegeben.

Aus diesem Grund entstand die Notwendigkeit, der Flask-Applikation einen produktionstauglichen WSGI-Server voranzustellen. Die primären Kriterien, die dieser Server erfüllen musste, waren leichte Handhabbarkeit, die Möglichkeit, direkt aus Python-Code aufgerufen zu werden (zur Vereinfachung des Docker-Deployments) und Leichtgewichtigkeit.

Evaluiert wurden die WSGI-Server Gunicorn [19], uWSGI [20], TwistedWeb [21] und Waitress [22]. Kurz nach Evaluationsbeginn fielen Gunicorn und uWSGI als Option weg, da beide nicht ohne Weiteres aus Python-Code aufgerufen werden können; sie brauchen gemäss Dokumentation einen Kommandozeilenaufruf mit vorgängiger Konfiguration. Ausserdem handelt es sich insbesondere bei uWSGI um ein ausgedehntes Framework, das viel mehr Features anbietet als nur einen Server und somit der Anforderung

nach Leichtgewichtigkeit widerspricht. Zwischen den verbleibenden Kandidaten TwistedWeb und Waitress fiel die Wahl schlussendlich auf Waitress. Beide Kandidaten verfügen über die Fähigkeit, problemlos direkt aus Python aufgerufen zu werden. Waitress ist aber wesentlich leichter, da es sich bei TwistedWeb wiederum um ein grösseres Framework handelt. Zusätzlich könnte Waitress nicht einfacher zu verwenden sein – es handelt sich um einen einzeiligen Aufruf in Python. Somit wurde **Waitress** als WSGI-Produktionsserver für das Nabu-Projekt gewählt.

Click

Das Librarian-Frontend des Nabu-Projektes, das im Rahmen dieses Projektes implementiert wird, muss über die Kommandozeile beziehungsweise das Terminal ansprechbar sein. Zur Implementation dieses Interfaces wurde die Python-Library **Click** [23] verwendet.

Click steht für «Command Line Interface Creation Kit». Dementsprechend stellt die Library einen Wrapper um die optparse-Methode von Python dar und dient zur Vereinfachung der Implementierung eines CLI. Sie offeriert Support für POSIX-Kommandozeilenkonventionen und bietet eine einfache Verwaltung für obligatorische und optionale Argumente sowie Defaultwerte, falls ein Argument nicht übergeben wurde.

Click wurde aus verschiedenen Gründen gewählt: Erstens wurde es gemeinsam mit dem Flask-Microframework erstellt, welches bereits in diesem Projekt verwendet wird. Zweitens stützt sich MindMeld selbst ebenfalls auf Click, um ein Kommandozeilen-Interface anzubieten. Drittens handelt es sich um eine leichtgewichtige, einfach zu verwendende Library, womit Entwicklungszeit eingespart werden konnte.

NLTK

NLTK [24] [25] ist das Natural Language Toolkit, eine Plattform für natürliche Sprachverarbeitung in Python. Es wird im CLI-Frontend für den Librarian zum Tokenisieren der Inputdokumente eingesetzt und wurde primär aufgrund der Tatsache ausgewählt, dass es gut dokumentiert und einfach einzusetzen ist. Darüber hinaus ist es als Open-Source-Software verfügbar.

PyMuPDF

Zum Parsen von PDF-Dateien wurden die beiden Bibliotheken PDFMiner [26] und PyMuPDF [27] evaluiert. Beide Libraries eignen sich zum Auslesen von PDF-Inhalten sowie -Metadaten.

Allerdings bietet PyMuPDF einige Vorteile: So kann die natürliche Leseordnung in den ausgewerteten Daten wiederhergestellt werden, zusammenhängender Text wird besser erkannt und die Dokumentation ist insgesamt gründlicher. Aus diesen Gründen fiel die Wahl auf **PyMuPDF**.

pandas

Zur Analyse von CSV-Dateien wurde die Programmbibliothek **pandas** [28] ausgewählt. Hauptgründe für diese Wahl sind, dass pandas in die Liste der Python-3-Erklärung aufgenommen ist und somit aktiv unterstützt wird, Open Source ist, es sehr schnell Daten analysieren kann und eine einfache Konvertierung vom CSV- zum JSON-Format bereitstellt. Im Unterschied zur ebenfalls evaluierten Python-CSV-Bibliothek ist Pandas zwar nicht sehr leichtgewichtig, bietet aber eine einfache Verknüpfung der Daten mit einer Graphik-Bibliothek. Dieser Punkt wäre vor allem relevant geworden, wenn noch genug Zeit für ein Librarian-GUI zur Verfügung gewesen wäre. Es handelt sich dabei also um eine Form des Future-Proofing.

PonyORM

Um die Daten in der Datenbank des Librarian-Backends zu speichern, wird ein ORM benötigt. Da in der Flask-API vor allem CRUD-Operationen ausgeführt werden müssen, würde ein eigens dafür erstellter Datenbank-Layer zu viel Overhead darstellen. Evaluiert

wurden deshalb die drei Python-ORM-Tools PonyORM [29], Django ORM [14] und SQLAlchemy [30].

Django ORM gehört zum Django-Framework selbst, womit die Integration in die Flask API erschwert sein könnte. Darüber hinaus müsste das Einbinden des gesamten Django-Stacks in Kauf genommen werden, was viel unnötigen Ballast bedeute.

Verglichen mit PonyORM (3.6MB) ist SQLAlchemy dreimal grösser (9.4MB). Zusätzlich hat auch SQLAlchemy viele zusätzliche Funktionen, welche vom Nabu-Projekt nicht ausgenutzt werden könnten.

Nicht zuletzt bietet PonyORM eine direkte Flask-Integration [31]. Diese ermöglicht es, das ORM-Tool mit wenigen Zeilen Code direkt in die Applikation zu integrieren und erleichtert somit das Setup erheblich. Aus diesen Gründen fiel die Entscheidung auf **PonyORM**.

MariaDB

Um die Daten zentral im Librarian-Backend abzuspeichern wurden MariaDB [32] und PostgreSQL [33] evaluiert. Da die beiden Datenbanken im direkten Vergleich kaum Vor- und Nachteile für diesen konkreten Anwendungsfall zeigten, fiel die Wahl auf **MariaDB**. Dies beruht darauf, dass bereits bei Moodle eine MariaDB-Datenbank benutzt wird und somit eine konsistente Datenbank-Architektur im ganzen Projekt entstanden ist. Insbesondere muss kein zusätzliches Docker-Image für den Moodle-Client heruntergeladen werden, da der Container von Moodle dasselbe Image verwenden kann.

4.2.4 Architektur

Überblick

Die Architektur von Nabu ist prinzipiell in Frontend und Backend in vertikaler Richtung sowie Librarian und Nabu-Chatbot in horizontaler Richtung gegliedert.

Das Frontend besteht dabei aus all jenen Komponenten, mit denen ein User direkt interagieren kann. Dabei handelt es sich also um all jene Komponenten, die eine grafische Benutzeroberfläche oder eine Kommandozeile anbieten und zur Erfüllung ihrer Aufgaben mit dem Backend kommunizieren. Zum Backend gehören hingegen diejenigen Komponenten, die notwendig sind, damit der Chatbot seine Funktionalität erfüllen kann, wenn vom Frontend eine Anfrage ausgeht.

In horizontaler Richtung bezeichnet der Librarian alle Komponenten, welche für die Datenerfassung und -haltung zuständig sind. Der Nabu-Chatbot hingegen umfasst die Komponenten, die für das Führen einer nutzbringenden Konversation verantwortlich sind und die Abfrage der vom Librarian erfassten Daten ermöglichen.

Dem untenstehenden Architekturdiagramm ist eine grafische Repräsentation der Komponentenaufteilung zu entnehmen.

Architekturdiagramm

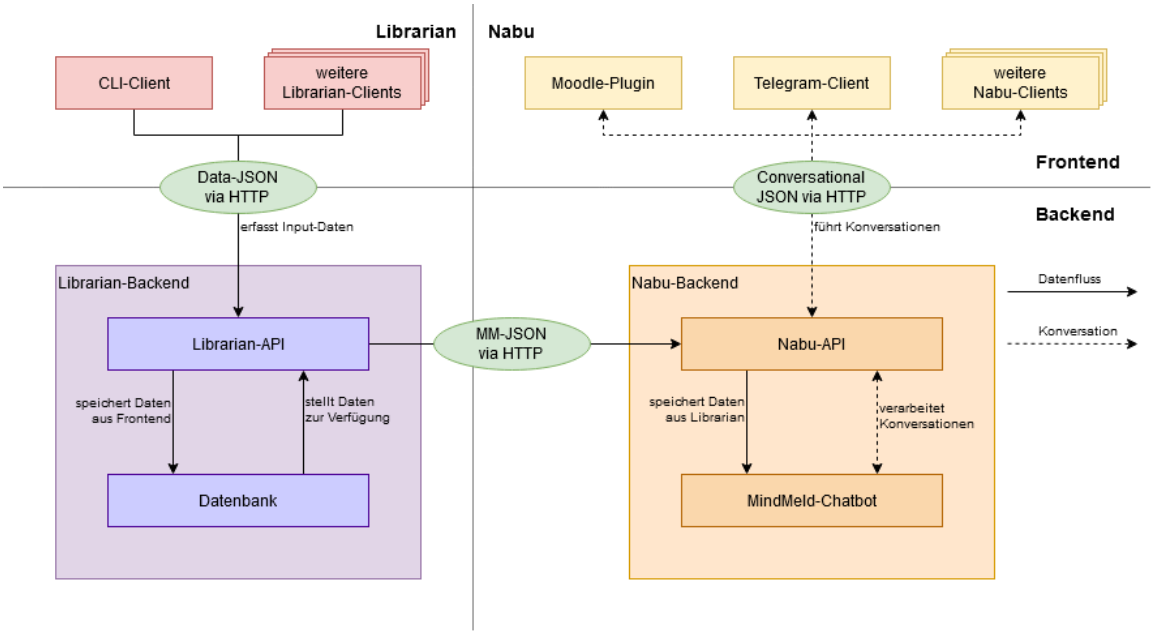


Abbildung 3: Architekturdiagramm

Frontend

Zum Frontend gehören alle Komponenten, die dem User ermöglichen, mit der Applikation als Ganzes zu interagieren. Dazu zählen:

- **Die Chat-Clients**: Insbesondere der Moodle-Client sowie der Telegram-Client, die im Rahmen dieses Projektes bereits entwickelt wurden. Diese Clients bieten dem User die Möglichkeit, Chat-Nachrichten an den Chatbot zu senden und eine Antwort auf Fragen zu erhalten. Zu diesem Zweck kommunizieren sie über eine konversationelle API mit dem MindMeld-Server, der im Backend läuft. Sie sind primär für Studierende gedacht, die über den Chatbot Informationen beziehen wollen.
- **Der Librarian-Client:** Dieser CLI-Client ermöglich es, Daten zum späteren Abruf über die konversationelle Schnittstelle zu erfassen, lokal in ein mit dem Librarian-Server kompatibles JSON-Format zu parsen und auf den Librarian-Server hochzuladen. In erster Linie richtet er sich an Dozierende, die ihre Lehrmittel und Daten über den Chatbot verfügbar machen wollen.

Backend

Zum Backend gehören die Services, die vorhanden sein müssen, um den Chatbot-Clients ihre Abfragen zu ermöglichen. Dies umfasst:

- **Das Nabu-Backend**: Dieses besteht aus einem API-Server und MindMeld selbst. Der API-Server ist für die Verarbeitung von Anfragen aus dem Frontend und das Halten von Sessions zuständig und stützt sich für das Parsen der natürlichen Spracheingabe auf die MindMeld-NLU-Pipeline, um seine Antworten zu generieren. Dabei bezieht er die notwendigen Informationen aus der MindMeld-Knowledge-Base. Darüber hinaus ist der API-Server auch für den Erhalt der vom Librarian stammenden Daten und das Hinterlegen in der Knowledge-Base verantwortlich.
- **Das Librarian-Backend**: Dieses Backend ist für die Datenhaltung zuständig. Über den Librarian-Client im Frontend ist es möglich Daten zu Learning Units zu erfassen. Diese werden in einer Datenbank im Librarian-Backend hinterlegt, aus welcher die für die MindMeld-Knowledge-Base notwendigen JSON-Formate generiert und dorthin hochgeladen werden können.

4.2.5 MindMeld-Chatbot

Konzepte

Der MindMeld-Chatbot ist der Kernbestandteil des Nabu-Backends und übernimmt das eigentliche Führen von Konversationen mit Nutzern über die konversationelle Schnittstelle sowie die Datenhaltung der abfragbaren Informationen in seiner Wissensbasis.

MindMeld lässt dem Programmierer sehr freie Hand, was die Implementation von Konversationslogik und den Aufbau der zu verwendeten Datenstrukturen anbelangt. Dennoch gibt es einige Grundkonzepte und Richtlinien, an die das Framework gebunden ist. In diesem Kapitel soll zunächst ein Überblick verschafft und danach das Design des Nabu-Chatbots unter Zuhilfenahme dieser Konzepte erläutert werden.

MindMeld operiert nach dem folgenden Schema zur Bearbeitung von Anfragen in natürlicher Sprache:

1. Eine Anfrage, auch Request genannt, geht bei MindMeld ein.
2. Der Request durchläuft die NLP-Pipeline und es werden Informationen zur späteren Verarbeitung extrahiert (darunter z.B. Domain, Intent, Entities; zu Details siehe unten).
3. Der Request sowie die zugehörigen Informationen erreicht die programmatisch definierte Konversationslogik.
4. Je nach Art des Requests werden gegebenenfalls Daten zur Antwort aus der Knowledge Base geladen.
5. Die Antwort wird zusammengestellt und an den Anfragesteller zurückgegeben.

Datenhaltung

Die Datenhaltung von MindMeld erfolgt im «data»-Ordner eines MindMeld-Moduls. Dort liegen verschiedene JSON-Dateien, die diejenigen Daten enthalten, die zur Build-Time von Elasticsearch indexiert und abfragbar gemacht werden. Der Inhalt der JSON-Dateien ist dabei komplett dem Programmierer überlassen.

Domains

Domains sind MindMelds Art, eingehende Anfragen grobgranular zu kategorisieren Sie entsprechen ungefähr der Vorstellung von Namespaces aus dem Software Engineering und unterteilen die Fähigkeiten des Chatbots in konkrete Gebiete. Beispiele für mögliche Domains wären «Wetter», «Haushalt» oder «Bestellungen». Es ist theoretisch möglich, beliebig viele Domains auf einem Chatbot zu definieren.

Die Definition von Domains erfolgt über den «domains»-Ordner im MindMeld-Modul, in dem jeder vorhandene Unterordner einer eigenen Domain entspricht.

Intents

Intents stehen, wie der Name schon andeutet, für die Absicht des Nutzers, die er mit einer Anfrage verfolgt. Sie sind jeweils einer bestimmten Domain zugehörig und helfen dabei, verschiedenen Abfragen eine gewisse Struktur zu verleihen und Abfragen mit dem gleichen Ziel auf die gleiche Art abhandeln zu können.

Der Intent einer Anfrage wird über die sogenannten Intent Classifier aufgelöst, die auf Machine Learning basieren und zur Build-Time des Chatbots mit Trainingsdaten versorgt werden müssen. Dies erlaubt MindMeld einen Abgleich einer Anfrage mit den vorhandenen Trainings-Abfragen, bei denen der Intent bereits bekannt ist, und somit eine Zuordnung der Anfrage zu einem Intent.

Beispiele für Intents wären beispielsweise «Wetter erfahren» auf der Domain «Wetter» oder «Ofen anschalten» auf der Domain «Haushalt». Anzumerken ist, dass Intents selbst sich nie auf konkrete Objekte beziehen (also nicht «Wetter in Rapperswil erfahren»), da diese Rolle den Entities zukommt (siehe unten).

Intents werden ebenfalls im «domains»-Ordner des MindMeld-Modules definiert. Jeder Unterordner einer spezifischen Domain entspricht dabei einem Intent auf dieser Domain und enthält eine «train.txt»-Textdatei, welche die Trainingsdaten für den jeweiligen

Intent beinhaltet. Zum Beispiel bestehen die Trainingsdaten für den Intent «Greet» (zur Begrüssung des Users) aus Phrasen wie «Hello» oder «Good day».

Entities

Entities sind Typdefinitionen, die MindMeld hinzuzieht, um Intents auf konkrete Umstände anzuwenden und den Intent als Ganzes zu verstehen und verarbeiten zu können.

Entities werden über die sogenannten Entity Recognizers verarbeitet, von denen pro Domain und Intent einer existiert und trainiert wird. Ihre Aufgabe ist es, die konkreten Bezugsobjekte eines Intents zu extrahieren und verfügbar zu machen.

Der Programmierer kann eigene Entities definieren und diese mit konkreten Werten versorgen. So wäre «Rapperswil» im Beispiel «Wetter in Rapperswil erfahren» eine «Ort»-Entity auf dem «Wetter erfahren»-Intent.

Alle Entities werden im «entities»-Ordner des MindMeld-Modules definiert, wobei jeder Unterordner für einen neuen Entity-Typ steht. In diesen Unterordnern enthalten sind zwei Dateien: Einerseits die «mapping.json»-Datei, die angibt, wie die erkannten Entitäten mit denjenigen in den Daten-JSONs korrespondieren, und andererseits eine «gazetteer.txt»-Datei, in der Beispiele für Entitäten zu Trainingszwecken zur Verfügung gestellt werden.

Dialogue Flow

Ein Dialogue Flow, oft nur Flow genannt, ist ein Weg, einen mehrteiligen, strukturierten Konversationsablauf zu modellieren. Dabei dient ein bestimmter Intent als Einstiegspunkt zum Flow. Nach dessen Aufruf werden die möglichen Intents, die als nächstes aufgerufen werden können, auf diejenigen begrenzt, die ebenfalls an diesem Flow registriert wurden. Intents können auch eine andere Behandlung erfahren, wenn sie in einem Dialogue Flow aufgerufen werden – gewisse Intents machen sogar überhaupt keinen Sinn, wenn sie nicht Teil eines Flows sind.

Ein Beispiel für die Anwendung von Flows ist die elegante Behandlung von unbekannten Einträgen: Sobald eine Abfrage gemacht wird, wird in einen Flow eingetreten. Ist die Abfrage erfolgreich, wird die Antwort zurückgeschickt und der Flow gleich wieder beendet. Ist sie erfolglos, wird der User aufgefordert, die Anfrage zu präzisieren, und solange im Flow verweilt, bis entweder die Antwort gefunden ist oder klar wird, dass eine Beantwortung nicht möglich ist. In beiden Fällen wird der Flow dann verlassen.

Nabu-Überblick

Mit der Verwendung von MindMeld als NLP-Framework geht einher, dass sich dieses Projekt an die oben erklärten Konzepte bindet. Untenstehendes Diagramm verschafft einen Überblick über die Umsetzung und stellt die acht definierten Intents verteilt auf die drei Domains dar.

Ein <u>unterstrichener</u> Name bedeutet dabei, dass der jeweilige Intent zum Einstieg in einen Dialogue Flow dient. Dieser Dialogue Flow wird durch Pfeile dargestellt, die vom jeweiligen Intent ausgehen und diejenigen Intents markieren, die durch den Flow erreichbar sind. Ein *kursiv* gesetzter Name bedeutet, dass der jeweilige Intent ausschliesslich als Teil eines Flows verfügbar ist.

Nabu-Diagramm

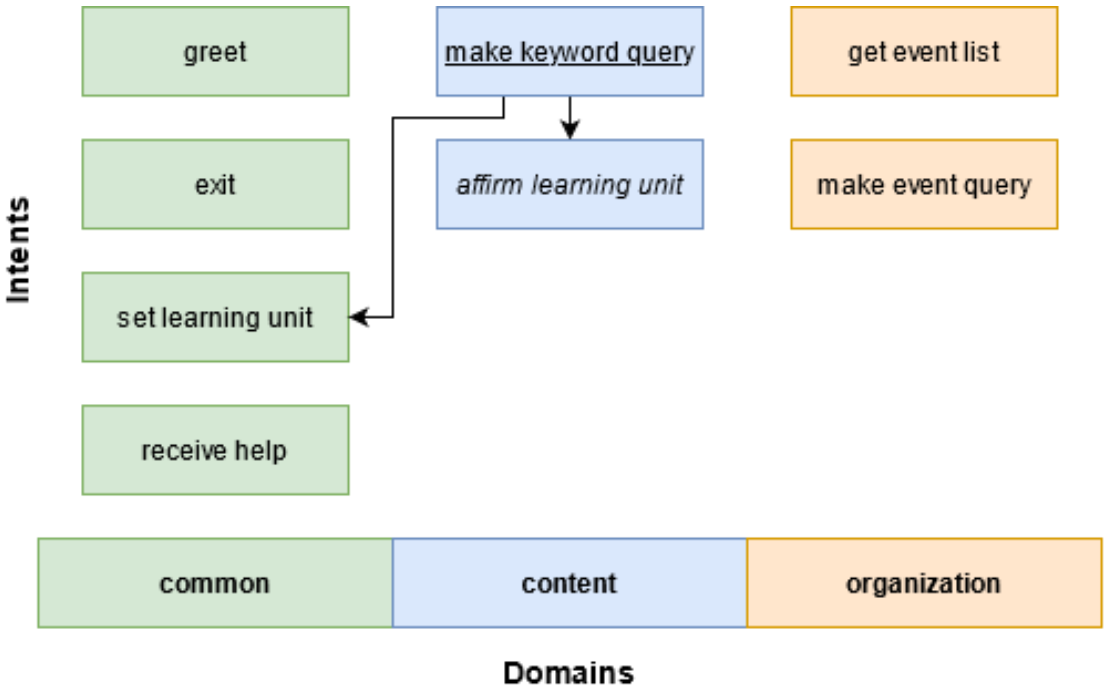


Abbildung 4: Nabu-MindMeld-Übersicht

Nabu-Domains

Zur generellen Einteilung der Fachgebiete des Chatbots wurden drei Domains definiert, namentlich:

- **Common**: Richtet sich nach keinem spezifischen Use Case, sondern ist für die Abhandlung von themenübergreifenden oder zu generellen Abfragen verantwortlich. Darunter fällt beispielsweise das Begrüssen oder Verabschieden.
- **Content**: Richtet sich nach Use Case 1: Fachliche Abfrage tätigen und befasst sich dementsprechend mit der Verarbeitung von Anfragen, die sich auf fachliche oder inhaltliche Informationen beziehen. Dies umfasst primär das Abfragen von Keywords und das Lenken des Users in die richtige Richtung, falls der erste Versuch nicht erfolgreich war.
- **Organization**: Richtet sich nach Use Case 2: Organisatorische Abfrage tätigen und hat somit das Beantworten von Fragen zu Organisation und Administration zur Aufgabe. Dazu gehört primär das Abfragen von anstehenden Events und Details zu diesen Events.

Nabu-Intents

Auf den Domains wurden die folgenden Intents definiert:

- **Common**:
  - Greet: Dient zur Begrüssung des Users. Dieser Intent ist zwar zur Abarbeitung der Use Cases nicht zwingend nötig, bietet aber eine natürlichere Chat-Erfahrung.
  - Exit: Verabschiedet den Nutzer nach Beendigung einer Abfrage. Ist gemäss Use Cases ebenfalls nicht unbedingt vonnöten.
  - Set Learning Unit: Legt die momentane Learning Unit fest. Ist Teil der Common-Domain, da dieser Intent sowohl zur Abfrage von Keywords als auch Events gebraucht wird.
  - Receive Help: Dient dazu, dem Nutzer in einer etwas längeren Nachricht einen Überblick über die Fähigkeiten von Nabu zu verschaffen.
- **Content**:
  - Make Keyword Query: Erlaubt das Abfragen eines Keywords. Dient als Einstiegspunkt in einen Flow, der das Festlegen der Learning Unit

erlaubt, falls sie noch nicht gesetzt ist. Dabei wird eine Annahme basierend auf dem gesuchten Keyword getroffen.
    - Affirm Learning Unit: Dient dazu, den Vorschlag von Nabu für unklare Learning Units anzunehmen und ist somit nur als Teil von Dialogue Flows verfügbar. Das Ablehnen und Anbringen eines Gegenvorschlags wird von Set Learning Unit gehandhabt.
- **Organization**:
    - Get Event List: Zeigt eine Liste (ohne Details) aller Events zu einer Learning Unit an Dient dazu, dem User auf einen Blick alle wichtigen Informationen zu Terminen anzuzeigen.
    - Make Event Query: Erlaubt das Abfragen eines Events mit dessen Namen und zeigt Details dazu an.

Nabu-Entities

Es wurden drei Typen von Entities definiert, die während den Requests auftreten können und somit von MindMeld erkannt werden:

- **Learning Unit Name**: Der Name einer Learning Unit. Dient zum Setzen oder Bestätigen einer solchen im Rahmen einer Keyword- oder Eventabfrage.
- **Keyword Name**: Der Name eines Keywords, nach dem gesucht werden soll. Wird vor allem für den «Make Keyword Query»-Intent gebraucht.
- **Event Name**: Der Name eines Events, beispielsweise «Testat 1 – Datenbank aufsetzen». Diese Entity dient dazu, einzelne Testate für den «Make Event Query»-Intent abfragbar zu machen.

4.2.6 Nabu-Clients

Überblick

Nach der Anforderungsanalyse wurde in Absprache mit dem Betreuer entschieden, dass im Rahmen dieses Projekts zwei verschiedene Nabu-Clients erstellt werden. Dabei handelt es sich um ein Plugin für die Moodle-Hochschulplattform, da diese von der OST Rapperswil-Jona verwendet wird, und um einen Bot für die Messenger-Applikation Telegram. Letztere wird an der OST Rapperswil-Jona erfahrungsgemäss von vielen Studierenden verwendet und ermöglicht es, den Zugriff auf den Nabu-Chatbot auch auf Mobilgeräten problemlos anzubieten.

Eine Client-Implementation für Microsoft Teams, wie sie ursprünglich erwogen wurde, wurde während der Elaborationsphase verworfen, da sie keinen nennenswerten Mehrwert gegenüber Moodle bieten würde.

Moodle

Moodle ist eine Open-Source Lernplattform, welche an der OST für die Kurs- und Modulverwaltung benutzt wird. Die Lernplattform beinhaltet bereits ein Nachrichtensystem, durch welches die Benutzer miteinander kommunizieren können.

Um Nachrichten abzufangen und weiterzuverarbeiten stellt Moodle die sogenannten Messaging Consumers [34] bereit. Einer dieser Messaging Consumer wird standardmässig von Moodle dazu benutzt, um dem Benutzer eine E-Mail zu schicken, der ihn über eine Nachricht informiert.

Diese Technik wird benutzt, um die Fragen eines Benutzers an unser Nabu-Backend weiterzuleiten.

Telegram

Telegram ist ein kostenloser Instant-Messaging Dienst, der sowohl auf Smartphones wie auch auf Desktop-Geräten benutzt werden kann. Der Dienst bietet die Möglichkeit zur Konfiguration und Implementation eigener Bots. So wird das Nabu-Backend auch Studenten und Schülern ausserhalb der Moodle Plattform zur Verfügung gestellt.

4.2.7 Librarian-CLI-Client

Librarian-CLI

Der Librarian-CLI-Client enthält die Logik, die für das Parsen von Dateien und Extrahieren der Daten notwendig ist. Dieser Client ist, wie schon am Namen ersichtlich, ein Command Line Interface. Damit kann der Benutzer PDF-Dateien (Use Case 1) und CSV-Dateien (Use Case 2) analysieren lassen und weiter an die Librarian-DB im Backend senden.

Driver

Für einen möglichst einfachen weiteren Ausbau wurde hier darauf geachtet, dass ein sogenannter Driver jeweils für ein Dateiformat zuständig ist. Für das Projekt wurde dementsprechend ein CSV-Driver und PDF-Driver implementiert. Jeder Driver kreiert beim Parsen ein JSON, welches danach via HTTP an das Librarian-Backend versendet wird.

Ein grösserer Unterschied zwischen der Analyse von organisatorischen und fachlichen Dateien ist, dass der implementierte CSV-Driver eine einzige CSV-Datei entgegennimmt und die Datei einen spezifischen Aufbau enthalten muss. Beim PDF-Driver ist es wiederum egal, ob nur eine einzige PDF-Datei oder ein ganzes Verzeichnis mit PDF-Dateien mitgegeben wird. Trivial ist dabei auch der Aufbau der PDF-Datei.

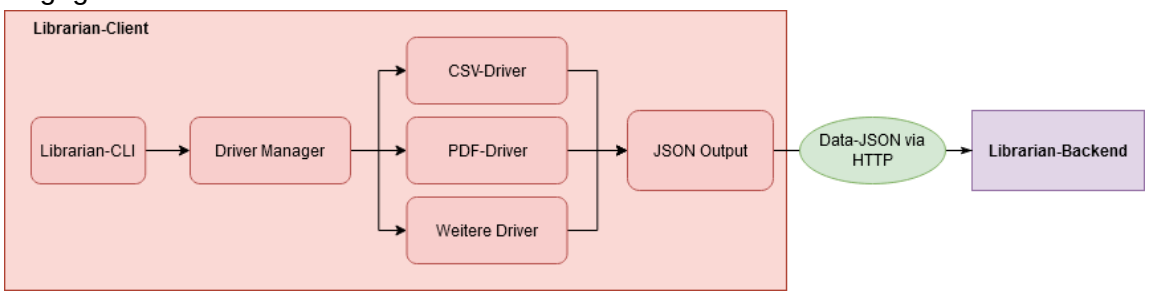


Abbildung 5: Librarian-Client-Datenfluss

4.2.8 Librarian-Backend

Überblick

Wie man im Architekturdiagramm sieht, besteht das Librarian-Backend aus zwei Teilen: Der Flask-Schnittstelle und der Datenbank. Die Flask-Schnittstelle stellt die verschiedenen Routen, welche von der Librarian-CLI für das Abspeichern der gesendeten Daten benutzt werden, bereit.

Datenbank

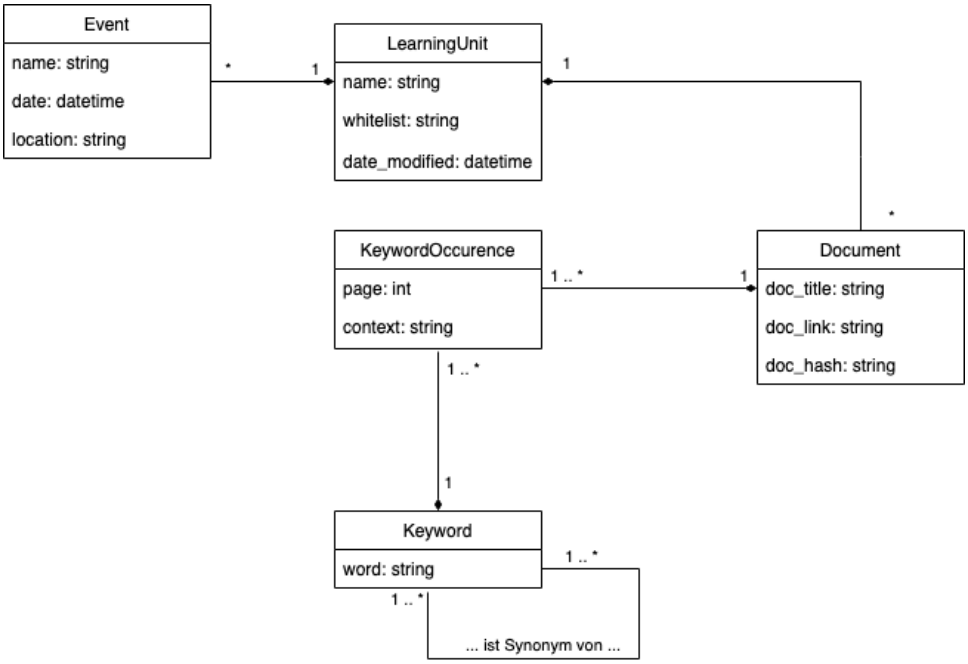


Abbildung 6: Entity-Relationship-Diagramm

Wie im obenstehenden Diagramm ersichtlich, werden die Daten in der Datenbank durch fünf Klassen modelliert: Event, LearningUnit, Document, Keyword und KeywordOccurrence. Durch die Abhängigkeit aller Klassen (direkt oder indirekt) zur Learning Unit wird sichergestellt, dass alle verbundenen Daten beim Löschvorgang einer Learning Unit ebenfalls nicht mehr vorhanden sind.

Eine Learning Unit kann mit mehreren Events sowie Dokumenten direkt verbunden sein, die Dokumente und Events sind jedoch jeweils nur einer Learning Unit zuordenbar.

Ein Dokument besteht aus einer oder mehreren KeywordOccurences. Die KeywordOccurence wird einerseits als Verbindung zwischen dem eigentlichen Keyword und als Speicherort für die page und den context benutzt. Die Keywords werden separat abgespeichert und können auch synonym zueinander sein. Im Dokument wird der Titel des Dokuments sowie ein automatisch generierter Hash und ein fakultativer Hyperlink zum Dokument hinterlegt.

4.2.9 Schnittstellen

Überblick

Im Zuge des Nabu-Projekts wurden drei Schnittstellen identifiziert, welche die einzelnen Komponenten miteinander verbinden. Dabei handelt es sich um die folgenden:

- **Datenerfassungsschnittstelle zwischen Librarian-Client und Librarian-Backend**: Die Schnittstelle dient zum Hochladen der im Client erfassten Daten in die serverseitige Datenbank.
- **Datentransferschnittstelle zwischen Librarian-Backend und Nabu-Backend**: Über diese Schnittstelle können die in der Librarian-Datenbank liegenden Daten in die Wissensbasis des Chatbots geladen werden.
- **Abfrageschnittstelle zwischen Nabu-Client und Nabu-Backend**: Über diese Schnittstelle werden die tatsächlichen Konversationen geführt, es werden also Dialogdaten ausgetauscht sowie Sessionsinformationen geführt.

Übertragungstechnologie

Die Wahl der Übertragungstechnologie geschah prinzipiell nach zwei Kriterien: Einerseits musste die Technologie aus Gründen der Leichtgewichtigkeit für alle Schnittstellen taugen, da ansonsten pro Schnittstelle zusätzliche Libraries hinzukommen würden. Das

Kriterium sah also eine einheitliche Lösung für alle Schnittstellen vor. Andererseits musste die Schnittstellentechnologie weitverbreitet und einfach anwendbar sein, insbesondere im Hinblick auf die Chatbot-Clients, die in Zukunft noch erstellt werden könnten.

Als Datenübertragungsmethode für alle Schnittstellen wurde anhand dieser Kriterien HTTP-Requests, insbesondere POST-Requests gewählt. Diese Wahl begründet sich darin, dass praktisch ausschliesslich JSON-Daten übertragen werden müssen, wozu HTTP direkt ein Feld anbietet. Ausserdem existiert HTTP in praktisch allen Sprachen und wird auf den meisten Plattformen unterstützt, egal, ob es sich um Web-, Desktop- oder Mobileumgebungen handelt. Dies garantiert, dass allfällige zukünftige Nabu-Clients praktisch überall implementierbar sind.

4.2.10 Deployment

Deploybare Komponenten

Das MindMeld-Projekt umfasst 5 eigenständig deploybare Komponenten:

- Das Nabu-Backend
- Das Librarian-Backend
- Den Telegram-Client
- Den Moodle-Client
- Das Librarian-CLI-Frontend

Aus diesem Grund wurde von Anfang an Wert auf eine Lösung gelegt, die das Deployen und Konfigurieren aller Komponenten unabhängig voneinander erlaubt und sich dazu in möglichst vielen Umgebungen einsetzen lässt, unabhängig von Betriebssystem oder Hardware. Darüber hinaus soll der Aufwand für den Nutzer möglichst klein gehalten werden.

Docker

Da die Applikation sich sauber in verschiedene Komponenten aufteilen lässt, bietet sich Docker [13] als Deployment-Option hervorragend an. Dabei existiert für die folgenden Komponenten jeweils ein Docker-Image:

- Das Nabu-Backend: Dieses Docker-Image erstellt eine leere, aber einsatzbereite Instanz des Nabu-Backends und alle nötigen Zusatzkomponenten wie MindMeld, dessen numerischen Parser und Elasticsearch.
- Das Librarian-Backend: Durch dieses Docker-Image wird eine Librarian-Backendinstanz zur Verfügung gestellt, welche zusätzlich einen separaten Datenbank-Container benötigt, um seine Daten zu hinterlegen.
- Der Telegram-Client: Dieses Docker-Image stellt eine Instanz des Telegram-Bots bereit, welche mit einer Nabu-Backendinstanz verknüpft werden kann und somit den Zugriff darauf über die Telegram-Messenger-App erlaubt.
- Der Moodle-Client: Dieses Image ist insofern speziell, als dass es nicht nur das Moodle-Plugin, sondern direkt einen ganzen Moodle-Stack mitsamt dem Plugin erstellt. Dies ist nur dann eine Option, wenn ein leerer Moodle-Stack deployt werden soll und wurde während dem Projekt vor allem zu Testzwecken verwendet. Zur Integration in eine bestehende Moodle-Instanz steht alternativ das Moodle-Plugin allein zur Verfügung.

Alle Docker-Images stehen für das Standalone-Deployment zur Verfügung. Dies erlaubt eine Verteilung aller notwendigen Komponenten über verschiedene physische Standorte, falls dies erwünscht ist. Soll die ganze Umgebung gleichzeitig am selben Ort kreiert werden, existiert dafür ein Docker-Compose-File.

pip-Install

Die einzige Ausnahme zum Docker-Deployment-Konzept bildet das Librarian-CLI-Frontend. Grund dafür ist, dass Docker sich primär zum Deployen von Services oder

Datenbanken eignet. Das CLI-Frontend soll aber jedem Dozierenden, der seine Daten hochladen möchte, auf der eigenen Maschine zur Verfügung stehen.

Aus diesem Grund wurde eine Installation über den Python-Packagemanager pip [35] und dessen Python Package Index (PyPI) [36] als Deployment-Konzept gewählt. Dieser erlaubt eine einfache Installation von online verfügbaren Packages und das anschliessende Ausführen über die Kommandozeile. Dies ist wünschenswert, da alle Nutzer diesen Befehl problemlos auf ihrer eigenen Maschine ausführen können, ohne eine Docker-Umgebung einrichten zu müssen.

Anzumerken ist, dass das CLI-Frontend bis zum Projektende nicht in den Python Package Index hochgeladen wurde und somit noch nicht direkt via install-Befehl verfügbar ist. Allerdings sind alle notwendigen Schritte, um einen solchen Upload zu ermöglichen, bereits getan (siehe Kapitel 4.4.3). Als Vorstufe zum Upload steht ein lokaler pip-install aus den Projektdateien zur Verfügung.

Deployment-Diagramm

Dem untenstehenden Deployment-Diagramm ist eine Übersicht über das gesamte Deploymentkonzept des Nabu-Projektes zu entnehmen. Die Anordnung entspricht dabei derjenigen, die während der Construction-Phase verwendet wurde. Allerdings ist zu beachten, dass alle Docker-Container nicht zwingend in der gleichen Docker-Instanz auf der gleichen physischen Maschine laufen müssen, sondern auch auf verschiedenen Servern lauffähig sind. Bei dieser Art von Deployment würde die Verbindung zwischen den Komponenten über das Netzwerk statt lokal innerhalb der Docker-Umgebung erfolgen.

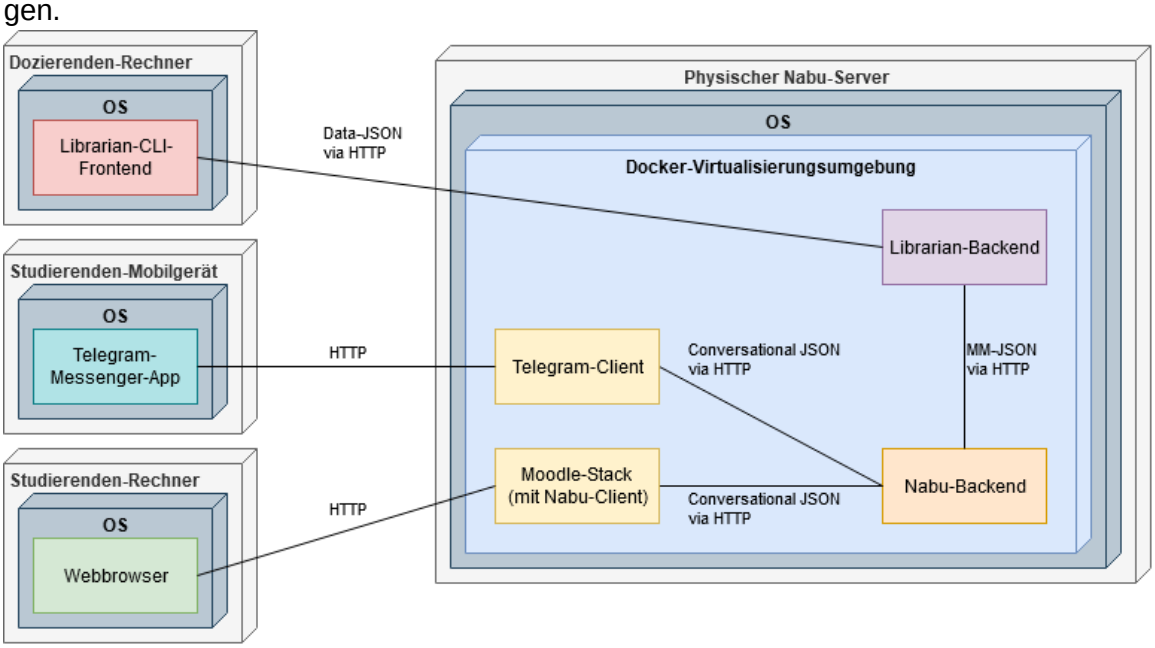


Abbildung 7: Nabu-Deploymentdiagramm

4.3 Implementation und Testing

4.3.1 Nabu-Backend

Überblick

Das Nabu-Backend umfasst den im Kapitel 4.2.5 umrissenen MindMeld-Chatbot sowie Server- und Persistenzkomponenten und macht es möglich, Abfragen über das Internet zu stellen und Daten in die Wissensbasis des Chatbots hochzuladen und dort zu speichern.

Package-Diagramm

Dem untenstehenden Package-Diagramm ist die grobe Struktur des Nabu-Backends zu entnehmen. Dabei ist zu beachten, dass aus Gründen der Lesbarkeit nicht alle Persister-, Mapper- und Answerer-Subklassen innerhalb der jeweiligen Packages angegeben wurden.

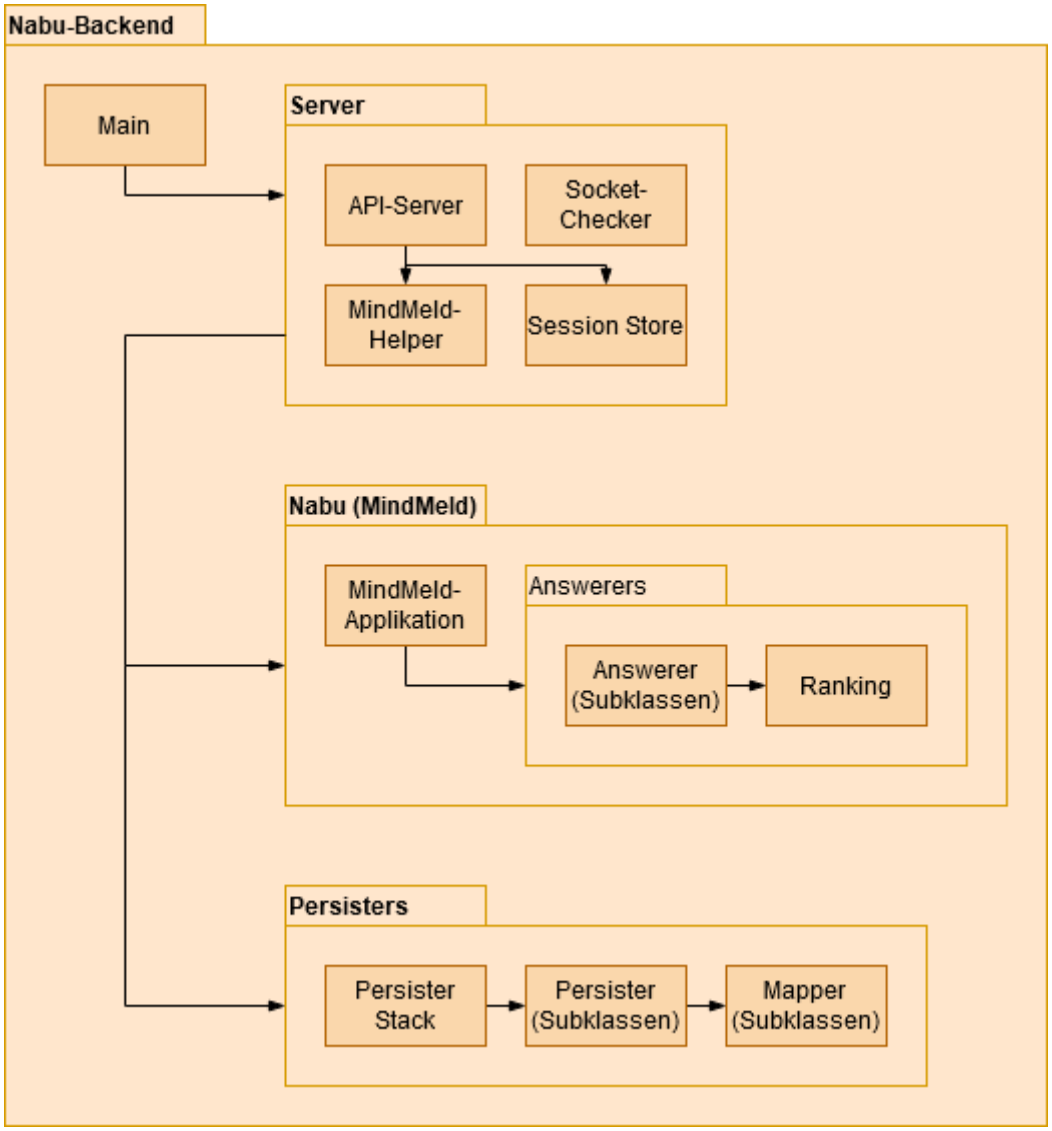


Abbildung 8: Nabu-Backend-Packagediagramm

Nabu-Backend-Package

An der Spitze der Hierarchie im Nabu-Backend steht das Nabu-Backend-Package, welches als auszuführendes Package für die gesamte Applikation dient. Seine Hauptaufgabe ist das Orchestrieren des Zusammenspiels zwischen den untergeordneten

Packages und Modulen (Server, Persisters und Nabu, siehe unten) und das Anbieten eines ausführbaren Main-Scripts (__main__.py).

In diesem Main-Script werden die nötigen Konfigurationsdaten zur Ausführung des Nabu-Backends zur Verfügung gestellt. Dazu gehören die Pfade zu den Daten-Files, die für MindMeld und die Persister wichtig sind und die Ports, die Elasticsearch und der MindMeld-Parser benötigen.

Bei der Ausführung erstellt das Package eine Flask-Instanz und verknüpft diese mit der Nabu-MindMeld-Applikation, um letztere abfragbar zu machen und das Hochladen von für die MindMeld-Wissensbasis bestimmten Daten zu ermöglichen. Dabei werden zwei Endpunkte angeboten:

- **/**: Der /-Endpunkt (Top-Level-Endpunkt, kein Pfad) ist für das Führen von Konversationen zuständig. An diesen Endpunkt kann Konversations-JSON geschick werden, bestehend aus Sessions-ID und Nachricht, und der Server wird die Anfrage entsprechend beantworten.
- **/librarian**: An den /librarian-Endpunkt kann MindMeld-JSON aus dem Librarian-Backend geschickt werden, welches dann vom Server validiert und in der Wissensbasis abgespeichert wird. Somit wird es für die Nabu-Frontends abfragbar.

Zu Details bezüglich der verwendeten JSON-Formate und der Kommunikation zwischen den Komponenten siehe Kapitel 4.3.6.

Server-Package

Das Server-Package stellt sicher, dass die Mindmeld-Applikation über das Internet abfragbar ist. Dazu nutzt es einen mit Flask implementierten API-Server. Dieser ist keine eigene Klasse, sondern via Factory-Pattern [37] über die Methode create_api_server erstellbar. Die API kann danach über einen WSGI-Server angeboten werden, in diesem Projekt wird hierzu Waitress verwendet (siehe Kapitel 4.2.3).

Um diesen Webserver mit einer MindMeld-Instanz zu verknüpfen, existiert der MindMeldHelper. Es handelt sich dabei um eine Klasse, welche mit Informationen über die MindMeld-Applikation versorgt wird (Name, Pfad und eine Referenz auf die Applikation) und die folgenden wichtigsten Methoden anbietet:

- **build_and_load_app**: Stellt sicher, dass alle Persistenzfiles vorhanden sind, lädt alle registrierten Wissensdatenbank-Dateien in den Elasticsearch-Index und trainiert die NLP-Pipeline mit den vorhandenen Trainingsdaten. Initialisiert alle Komponenten, damit Anfragen möglich werden.
- **register_knowledge_base**: Registriert eine Datei, die während des Ladevorgangs in den Index geladen und somit abfragbar gemacht wird. Muss nach dem Erstellen, aber vor dem Initialisieren der MindMeld-App geschehen.
- **register_persister_stack**: Registriert einen Persister-Stack, der beim Eingehen von neuen JSON-Daten aus dem Librarian aufgerufen wird und für das Persistieren dieser Daten verantwortlich ist. Muss ebenfalls zwischen Erstellung und Initialisierung geschehen.
- **get_answer**: Führt im Namen einer Session aus dem Session Store eine Anfrage an die MindMeld-Applikation aus und gibt die Antwort zurück. Die Session wird, wenn bereits vorhanden, vom Anfragensteller mitgeschickt und ansonsten neu erstellt und mit der Antwort zurückgegeben. Ist erst nach der Initialisierung möglich.

Darüber hinaus stellt das Server-Modul den Session Store bereit. Dieser ist dazu gedacht, in Verbindung mit dem API-Server verwendet zu werden. Seine Aufgabe ist es, verschiedenen Nutzern über das Assoziieren mit einer einzigartigen ID den parallelen Zugriff auf den Chatbot zu ermöglichen. Es handelt sich im Grunde genommen um einen Key-Value-Store, der IDs auf Sessions abbildet. In einer solchen Session werden Informationen wie letzter Aufruf, momentaner Zustand der Konversation und aktuelles Dialog-Frame gehalten. Beim erstmaligen Anschreiben des Chatbots erhält ein Nabu-

Client dabei eine Session-ID, mit der er sich ab diesem Punkt identifizieren kann, um die Kohärenz seiner Konversation zu gewährleisten. Der Session Store stellt die folgenden Funktionen zur Verfügung:

- **get_or_generate_session**: Wird diese Methode mit Sessions-Identifier aufgerufen, so wird die damit verknüpfte Session aus dem Store zurückgegeben. Wird keiner mitgegeben oder existiert der übergebene Identifier nicht, wird eine neue Session erstellt, gespeichert und der damit verknüpfte Identifier zurückgegeben, der fortan zu verwenden ist.
- **start_session_cleanup, stop_session_cleanup**: Startet beziehungsweise stoppt den Reinigungsprozess, der in vorgegebenen Intervallen (standardmässig alle 60 Sekunden) untersucht, ob Sessions gespeichert sind, die ihre Inaktivitätsdauer (standardmässig 500 Sekunden) überschritten haben. Werden solche Sessions gefunden, werden sie gelöscht, um Speicherplatz freizugeben. Für den Nutzer äussert sich das darin, dass aus seiner Sicht eine neue Konversation begonnen wird. Implementiert wurde der Cleanup-Prozess über einen separaten Thread. Wichtig hierbei ist das externe Stoppen über die dafür vorgesehene Methode, z.B. zum Programmende, da der Session Store dies selbst nicht kann. Der Thread prüft alle drei Sekunden, ob er zwischenzeitlich den Befehl zum Abbruch erhalten hat und beendet sich dann selbst.

Als letztes existiert im Server-Modul die Funktion **check_port**. Diese dient zur Überprüfung, ob ein bestimmter Port auf der lokalen Maschine offen ist und wird dazu benötigt, das Starten des Servers nur dann zuzulassen, wenn der numerische MindMeld-Parser und die Elasticsearch-Instanz bereit sind.

Nabu-Package

Das Nabu-Package enthält die eigentliche MindMeld-Applikation, die den Nabu-Chatbot erstellt. Da es dabei um eine Applikation handelt, die sich auch standalone ausführen liesse, wurden die in MindMeld gängigen Konventionen so weit als möglich eingehalten. Das bedeutet, dass das Nabu-Package architektonisch dem Template entspricht, das über das MindMeld-Kommandozeilentool generiert werden kann (mit wenigen Additionen, siehe «Answerer-Submodul» unten).

MindMeld stützt sich beim Ausführen auf die __init__.py-Datei. Darin werden die Konversationsabläufe definiert, die später vom Chatbot verwendet werden sollen. Zudem existiert eine __main__.py-Datei, die beim Standalone-Ausführen des MindMeld-Packages das Kommandozeilen-Interface aufruft (was für dieses Projekt nur zu Testzwecken von Belang war) und eine config.py-Datei, in welcher die Konfigurationen für die MindMeld-Komponenten gespeichert werden.

Zusätzlich liegen im MindMeld-Package die gesammelten Wissensdatenbank-Daten, auf die sich der Chatbot berufen kann, um Anfragen zu bearbeiten. Diese werden zur Build-Zeit von Elasticsearch indexiert und zum Abruf bereitgestellt. Genaueres dazu findet sich in Kapitel 4.2.5.

Answerers-Subpackage

Das Nabu-Package enthält darüber hinaus das Answerers-Subpackage. Dieses ist nicht Teil der ursprünglichen MindMeld-Architekturstruktur, sondern wurde im Rahmen des Nabu-Projektes angelegt.

Aufgabe des Answerer-Subpackages ist es, die konkrete Antwortlogik für eingehende Requests an den Chatbot zur Verfügung zu stellen. Dazu existiert eine abstrakte Answerer-Klasse, von der pro MindMeld-Domain eine Subklasse abgeleitet werden soll. Sie enthält die Logik, die allen Domains gemeinsam ist, wie z.B. das Extrahieren von Entitäten aus Requests oder das Senden von Abfragen an die Wissensdatenbank.

Die von der Answerer-Klasse abgeleiteten Klassen haben die Aufgabe, für alle Intents und Flows auf der jeweiligen Domain eine Methode zur Verfügung zu stellen, die eine Antwort generieren kann. Sie befassen sich also mit der konkreten Gesprächslogik. Im

Rahmen dieses Projektes wurden drei solche Subklassen implementiert, die sich jeweils mit den Domains Common, Content und Organization befassen.

Hauptgrund für die Implementation dieser Logik als eigenes Package, im Gegensatz zur normalen MindMeld-Methode im __init__.py-File war in erster Linie, dass ansonsten dieses File mehrere Hundert Zeilen gross und unübersichtlich würde, was der Wartbarkeit des Codes nicht zuträglich ist. Durch das Auslagern in eigene Klassen wird die Antwortlogik überschaubarer und die Aufgabenteilung besser eingehalten.

Das Answerers-Package enthält darüber hinaus den Ranking-Algorithmus, der verwendet wird, um die aus der Wissensbasis abgeholten Resultate vor dem Anzeigen in eine absteigende Rangordnung zu bringen. Dieser einfache Algorithmus ist in der Datei ranking.py implementiert. Er operiert in zwei Stufen:

- Einerseits bringt er die Occurrences des Keywords in den einzelnen Dokumenten in eine Rangordnung. Dazu verwendet er die folgende Formel, die das relative Positionsgewicht und die Kontextlänge in Wörtern mit einbezieht:

$$rating = \frac{context\ length}{average\ context\ length} * relative\ position\ weight$$

  Dabei entspricht das relative Positionsgewicht einem Wert, der wie folgt errechnet wird:

$$weight\ diff = lowest\ weight - highest\ weight$$

$$rel.pos.weight = \frac{weight\ diff * (page - first\ page)}{(last\ page - first\ page)} + highest\ weight$$

  Die Überlegung dahinter ist, dass ein Keyword beim ersten Auftreten mit höherer Wahrscheinlichkeit detailliert erklärt wird als bei späteren Vorkommen, was zur Anzeige in den Suchresultaten wünschenswert ist. Es handelt sich also um eine Art lineare Interpolation, die einer Occurrence basierend auf einem festgelegten Tiefst- und Höchstgewicht (standardmässig 1.0 und 1.5) sowie der ersten und letzten Seite, auf denen das Keyword im Dokument vorkommt, ein bestimmtes Gewicht zuteilt. Dieses Gewicht fällt höher aus, je früher die Occurrence in ihrem Dokument auftritt.
  Somit setzt sich die Gesamtheit des Ratings für Occurrences aus der Position im Dokument sowie der Kontextlänge verglichen mit der durchschnittlichen Kontextlänge zusammen. Occurrences mit detailliertem Kontext, die vergleichsweise früh im Dokument auftreten, werden dadurch zuoberst eingestuft.
- Andererseits führt er ein Ranking aller Dokumente durch, in denen das gesuchte Keyword vorkommt. Der hierzu verwendete Rating-Wert berechnet sich wie folgt:

$$doc\ rating = \frac{occurrences\ in\ document}{average\ occurrences\ per\ document} * average\ occurrence\ rating$$

  Er bezieht also mit ein, wie gut das Dokument verglichen mit allen anderen Dokumenten bezüglich der Auftretenshäufigkeit des Keywords abschneidet und wie gut die Occurrences im Dokument in Schritt 1 im Durchschnitt bewertet wurden. Dies führt dazu, dass Dokumente, in denen gut bewertete Occurrences oft vorkommen, an der Spitze der Rangliste zu liegen kommen, während Dokumente, in denen das Keyword weniger oft oder ohne viel Kontext auftritt, schlechter eingeordnet werden.

Persisters-Package Das Persisters-Package ist für das Persistieren von Daten verantwortlich, die in die MindMeld-Knowledge-Base geschrieben werden sollen. Dazu gehören, wie in Kapitel 4.2.5 bereits angeschnitten drei Typen:

- Daten-JSONs, die in die von MindMeld abfragbare Elasticsearch-Instanz aufgenommen werden.
- Mapping-JSONs, die MindMeld beim Erkennen der Entitäten in eingehenden Anfragen helfen und in einer Whitelist Synonyme definieren.
- Ein Gazetteer in einem Text-File. Dabei handelt es sich um eine textuelle Liste aller bekannten Entitäten sowie deren Synonyme, die MindMeld dabei hilft, Entitäten präziser aufzulösen.

Der grundlegende Ablauf eines Persistiervorganges sieht demnach wie folgt aus:

1. Aus den neu zu schreibenden Daten, die vom Librarian erhalten wurden, wird ermittelt, ob überhaupt ein Persistiervorgang nötig ist. Der Ablauf fährt nur fort, falls dem so ist.
2. Es wird geprüft, ob die fragliche Output-Datei existiert. Falls nein, wird eine neue Struktur angelegt, die später abgespeichert wird.
3. Es wird geprüft, ob die Datenstruktur valide ist. Ist dies nicht der Fall, wird wiederum eine neue Struktur angelegt, als ob die Datei nicht existieren würde.
4. Die erhaltenen Daten werden überprüft und mit den bereits vorhandenen Daten aus der geladenen Struktur vereint, woraus die neue, zu speichernde Datenstruktur resultiert.
5. Die neu erstellte Struktur wird validiert und in die Daten-JSON-Datei zurückgeschrieben, um sie persistent zu machen und Elasticsearch Zugriff zu gewähren.
6. Falls nötig, wird für MindMeld ein Mapping bereitgestellt, welches die eben erfassten Daten besser als Entität abfragbar macht.
7. Falls nötig, wird ein MindMeld-Gazetteer erstellt und abgespeichert.
8. Beim nächsten Ladevorgang der MindMeld-App werden die Daten für den Chatbot verfügbar sein.

Zur Umsetzung dieses Vorgangs existieren in erster Linie zwei abstrakte Klassen:

- Die Persister-Klasse, welche die grundlegende Input-Output-Logik zum Verarbeiten von neuen Daten aus der Librarian-Schnittstelle definiert. Zu dieser Logik gehört das Prüfen, ob überhaupt eine Output-JSON-Datei im entsprechenden Ordner existiert, das Auslesen dieser Datei und das Schreiben der über die Serverschnittstelle vom Librarian erhaltenen Daten in die Datei. Dies sind alles Vorgänge, die unabhängig vom eigentlichen Inhalt der Daten sind und immer gleich ausgeführt werden können.
- Die Mapper-Klasse, welche eine Funktion analog des Persisters einnimmt, aber für Mappings und Gazetteers statt für Wissensdatenbank-JSON-Dateien.

Grund für die Unterteilung dieser Logik in zwei Klassen ist die Tatsache, dass das Manipulieren einer einzelnen JSON-Datei unter Umständen mit Veränderungen in mehr als einem Mapping bzw. Gazetteer einhergehen kann, je nach Aufbau der Daten. Das Aufteilen der Logik in mehrere Klassen erlaubt es, eine solche Situation über das Registrieren von mehreren Mappern an einem Persister abzuhandeln.

Damit domainspezifische Daten verarbeitet werden können, muss eine Subklasse der oben angesprochenen abstrakten Persister-Klasse erstellt werden. Diese stellt die notwendige, inhaltsabhängige Logik zum Validieren der existierenden Struktur und zur Vereinigung der neuen Daten mit den alten durch das Überschreiben von vier Methoden zur Verfügung:

- Die Methode **should_persist** gibt an, ob überhaupt eine Persistierung notwendig ist. Falls ein bestimmter Persister durch die fehlende Revelanz der eingehenden Daten nicht gebraucht wird, spart man in diesem Fall Rechenzeit. Ein Beispiel für diesen Fall wäre, dass ein Update eingegangen ist, das nur Events,

nicht aber Keywords modifiziert. Das Laden der JSON-Datei mit den Keywords kann dann übersprungen werden.

- Die Methode **merge_data** definiert, wie die geladenen Daten aus dem JSON-Datenfile mit den neu erhaltenen Daten vereinigt werden müssen. Sie soll ausserdem prüfen, ob die neuen Daten valide sind.
- Die Methode **validate_structure** gibt an, ob die aus der Datei geladene Struktur valide ist (sie könnte z.B. zwischenzeitlich gelöscht oder manuell bearbeitet worden sein).
- Die Methode **generate_default_structure** stellt eine Default-Struktur zur Verfügung, falls in den eingehenden Daten ein Fehler vorliegt. Diese Default-Struktur darf allerdings nicht leer sein, da in MindMeld ansonsten beim Laden ein Fehler auftritt. Deshalb sollte sie einen Dummy-Eintrag enthalten.

Dabei zu beachten ist, dass die Implementation dieser Methoden idempotent sein muss und die ursprüngliche Datenstruktur nicht verändern darf, damit sie im Stack funktioniert (siehe unten). Die Persister-Klasse stellt darüber hinaus die Methode **ensure_existence** zur Verfügung, die garantiert, dass die zur Persistierung benötigten Dateien schreib- und ladebereit existieren.

Um Mappings und Gazetteer-Einträge zu erstellen, existiert die abstrakte Mapper-Klasse. Deren Vererbungsmechanismus funktioniert konzeptuell analog zur Persister-Klasse. Alle konkreten Mapper sind Subklassen der Mapper-Klasse und optionaler Bestandteil einer Persister-Instanz. Mapper können separat mit eigenem Pfad für Mappings und den Gazetteer erstellt werden. Die Logik in der Mapper-Klasse ist wesentlich simpler als diejenige des Persisters: Mappings und der Gazetteer werden bei jedem Update neu erzeugt und müssen nicht mit existierenden Daten abgeglichen werden. In Mapper-Subklassen muss dementsprechend auch nur die Methode **create_mapping_data** überschrieben werden. Diese beschreibt, wie das Datenmapping erfolgen soll.

Zur praktischen Anwendung der Persister- sowie Mapper-Subklassen existiert der sogenannte Persister-Stack, ebenfalls ein Bestandteil des Persister-Submodules. Er stellt die folgenden Funktionen bereit:

- **persist_json**: Ruft in einer Variante des Chain-of-Responsibility-Patterns [37] alle registrierten Persister auf. Die übergebene JSON-Struktur wird durch jeden Persister hindurchgereicht, der dann denjenigen Teil der Daten abspeichert, für den er verantwortlich ist.
- **register_persister**: Registriert einen neuen Persister am Stack.
- **ensure_existence**: Ruft auf jedem Persister im Stack wiederum ensure_existence auf, um sicherzustellen, dass alle Dateien bereit sind.

Dieser Aufbau sorgt für eine saubere Aufteilung der Verantwortlichkeiten zwischen den Persistern und für eine einfache Erweiterbarkeit.

Im Rahmen dieses Projektes wurden drei konkrete Persister-Subklassen sowie drei Mapper-Klassen erstellt. Sie befassen sich mit Learning Units, Keywords und Events, wobei je eine Subklasse für die Daten und eine für die Mappings, die erstellt werden müssen, existiert. Die folgenden Klassendiagramme verschaffen einen Überblick über diese Klassen.

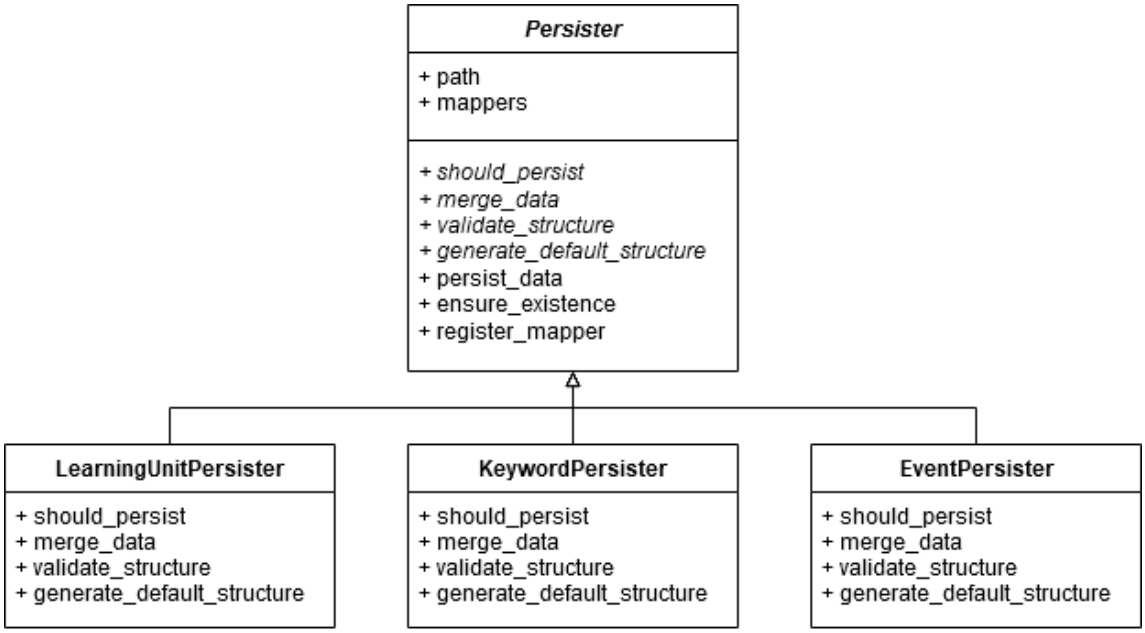


Abbildung 9: Persister-Klassendiagramm

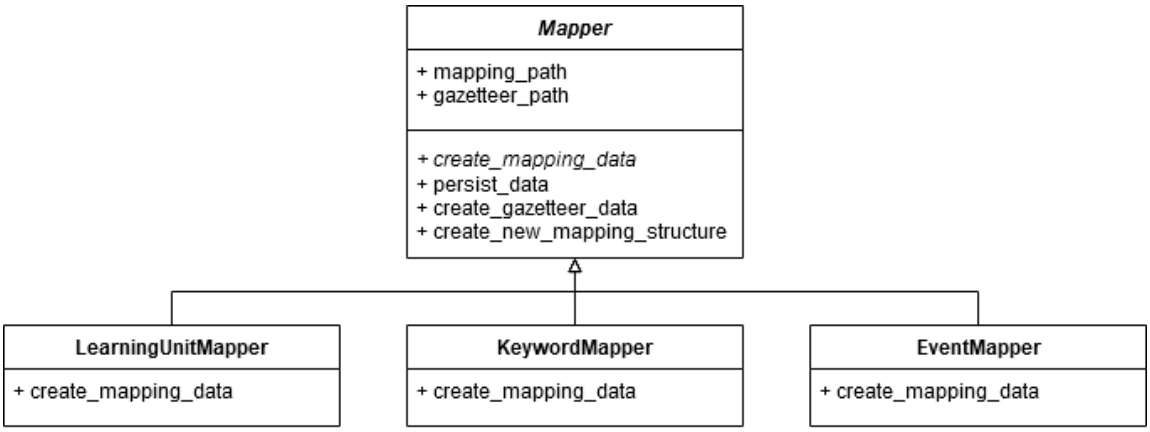


Abbildung 10: Mapper-Klassendiagramm

Sollte ein späterer Nutzer mehr Daten via Librarian-Schnittstelle übermitteln wollen, so müsste jeweils eine neue Persister-Subklasse mit ihren eigenen Mappern erstellt und am Stack registriert werden.

4.3.2 Librarian-CLI-Client

Librarian-CLI

Dank diversen in Click definierten Optionen werden hier die nötigen und zusätzlich möglichen Parameter gesetzt. Für die Analyse der PDF-Dateien wurde PyMuPDF verwendet und für die CSV-Dateien die Bibliothek Pandas. Spezifischeres zu diesen Bibliotheken ist im Kapitel 4.2.3 auffindbar.

Im Package librarian_cli ist die sogenannte librarian-Methode zu finden. Diese leistet dank den mitgegebenen Parametern Vorarbeit und leitet den jeweils spezifischen Arbeitsfluss für den kommenden Entscheidungsverlauf an den Drivermanager weiter. Nachdem determiniert wurde ob es ein organisatorischer oder fachlicher Datenfluss ist und die Datei(en) geparsed wurde(n), werden Daten im JSON-Format zurückgegeben. Diese Daten werden via HTTP über die Librarian-API an die Librarian-DB gesendet.

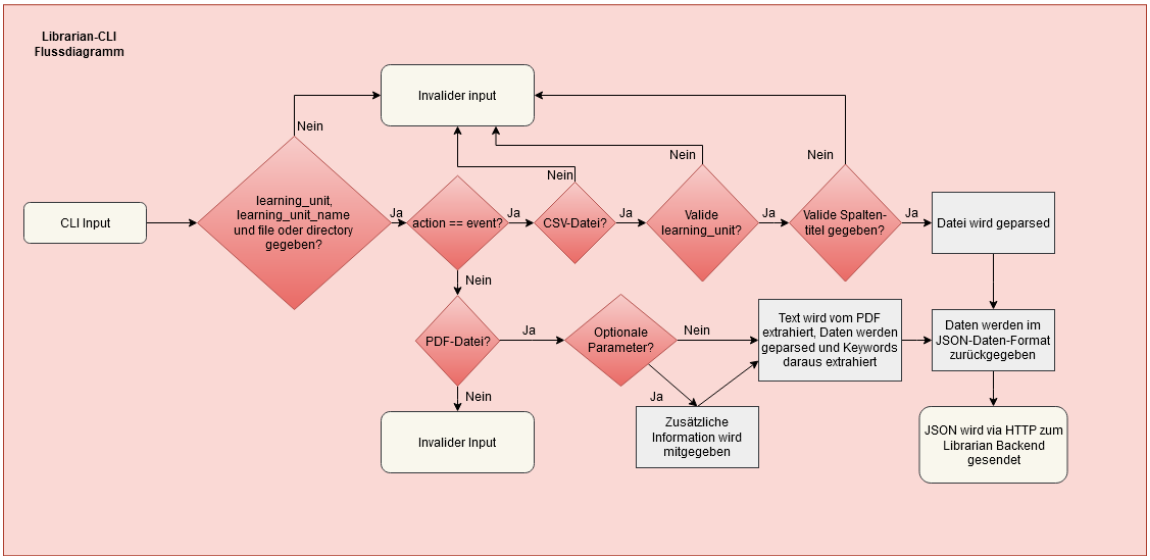


Abbildung 11: CLI-Programmablauflogik zur Datenerfassung

Click in Librarian

Zur Erfassung von Dokumenten im Librarian sind die folgenden Kommandozeilenparameter zwingend notwendig:

- **--learning_unit**: Die abgekürzte Version der Learning Unit, die zur eindeutigen Identifikation verwendet wird. Entspricht der ID, die später in der Datenbank hinterlegt wird.
- **--learning_unit_name**: Der vollständige Anzeigename der Learning Unit.
- **--file oder --directory**: Entweder muss ein Pfad oder Verzeichnis mitgegeben werden. Es ist darauf zu achten, dass bei einer organisatorischen Analyse --file genommen wird.
- **--action**: Definiert, ob fachliche oder organisatorische Daten mitgegeben werden. Hierbei wurde ‘keywords’ für die fachliche und ‘event’ für die organisatorische Analyse definiert. Wird standardmässig auf ‘keywords’ gesetzt.

Darüber hinaus sind die folgenden, optionalen Parameter verfügbar:

- **--learning_unit_whitelist**: Hiermit kann der Benutzer alternative Namen der Learning Unit definieren, was die Qualität der Abfrage über den Nabu-Chatbot erheblich verbessern kann. So wären «DB1» oder «DBS1» beispielsweise gute Synonyme für die Learning Unit «Datenbanken 1».
- **--link**: Hiermit kann eine URL zum Dokument mitgegeben werden, um einen schnelleren Zugriff darauf zu ermöglichen. Wird es in Kombination mit dem Parameter --directory verwendet, erhalten alle Dateien den gleichen Link.
- **--language**: Diese Option tangiert das Stretch Goal der Mehrsprachigkeit. Hiermit wird definiert, in was für einer Sprache die Datei verfasst ist, damit die Tokenization möglichst effizient funktioniert.

Driver-Package

Hier sind der Drivermanager, CSV-Driver und PDF-Driver zu finden. Der Drivermanager entscheidet basierend auf den erhaltenen Arbeitsfluss und der mitgegeben Dateienendung, wie die Analyse fortschreiten soll.

In der Klasse PdfDriver ist die Parser-Logik für PDF-Dokumente zu finden. Hier sind die folgenden drei Methoden zu finden, welche einen Kernpunkt der Arbeit darstellen und zur Vorbereitung von Abfragen gemäss Use Case 1 unerlässlich sind.

- **tokenizer**: Durch Benutzung von NLTK, dem Python Natural Language Toolkit, wird hier der mitgegebene Kontext analysiert und jeweils alle Nomen extrahiert

und als Liste zurückgegeben. Die Analyse des Tokenizers beruht jeweils auf der mitgegebenen Sprache.

- **get_context**: Die PDF-Dokumente werden mit der Bibliothek PyMuPDF analysiert, hierbei werden jedoch eine Unmenge an Daten, die für das Projekt irrelevant sind, zurückgegeben. Deswegen wird in dieser Methode der für das Projekt relevante Kontext extrahiert.
- **get_filtered_content**: Verknüpfungsort der diversen analysierten Daten und den Metadaten, z.B. den Hash oder den Titel des Dokuments. Gibt schlussendlich die für das Projekt relevanten Daten im JSON-Format zurück.

In der Klasse CsvDriver wird die CSV-Datei analysiert. Hier wird in der Methode **csv_to_dictionary** zuerst überprüft, ob die Datei valid ist. Heisst, die folgenden Spaltentitel müssen vorhanden sein: learningunit, eventname, eventdate, eventtime, location. Wenn valid, werden die Daten im Dictionary-Format zurückgegeben.

Damit invalide Learning Units beziehungsweise ungültige Zeilen nicht unnötigerweise an das Librarian-Backend gesendet werden, wird in der Methode **get_filled_learningunit_object** mit den im Librarian-Backend vorhandenen Learning Units filtriert, bevor die Daten im JSON-Format für den weiteren Verlauf zurückgegeben werden.

setup.py

Damit das CLI-Tool nicht immer direkt im Root von librarian_cli aufgerufen werden muss, wurde die Möglichkeit für einen lokalen pip install hinzugefügt. Hierbei muss der User einmal im Verzeichnis librarian_cli «pip install .» ausführen und kann danach das Command Line Interface global ausführen [38].

4.3.3 Librarian-Backend

Datenbank

Um die Daten an einem zentralen Ort abzuspeichern wurde eine MySQL-Datenbank aufgesetzt. Das untenstehende Diagramm zeigt die Datenbankumsetzung von Abbildung 6: Entity-Relationship-Diagramm.

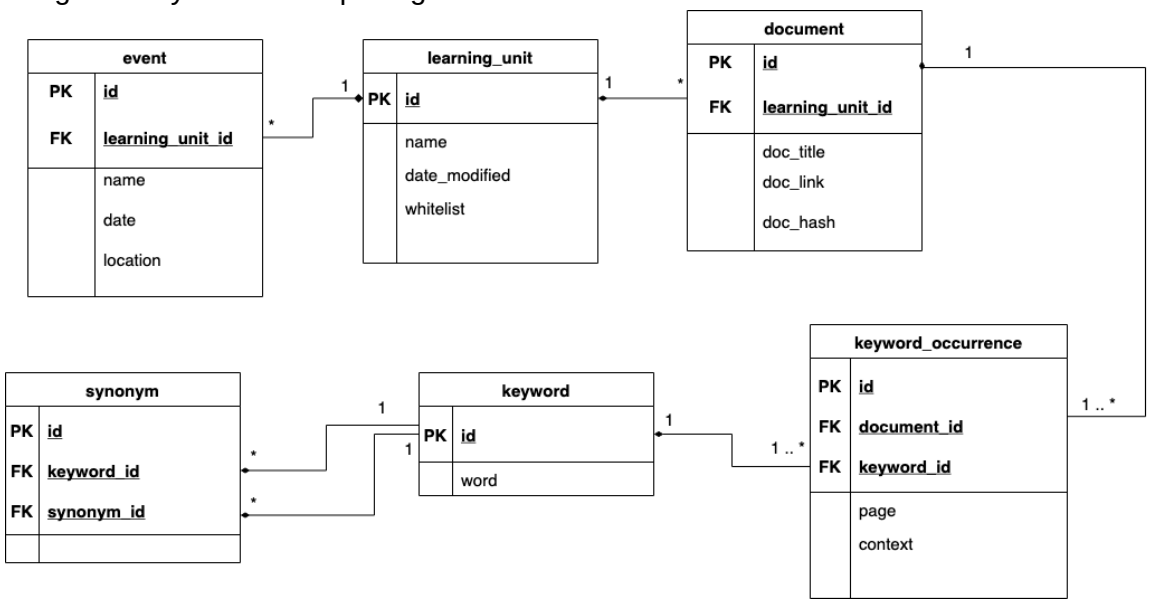


Abbildung 12: Datenbankdiagramm

In der Datenbank werden sechs Tabellen abgebildet – eine davon ist eine Zwischentabelle für die Synonyme der Keywords (**synonym**).

In der **event**-Tabelle werden der *name*, das *date* sowie die *location* des Events abgespeichert. Zum Endzeitpunkt des Projektes ist es noch nicht möglich eine genauere Beschreibung (*description*) eines Events abzuspeichern. Ein **event** ist immer von einem Learning Unit abhängig und wird über dessen Primary Keys verbunden.

In der **learning_unit** werden die *id*, der *name*, die *whitelist* sowie ein *last_modified* Datum abgespeichert. Dieses wird jeweils beim Speichern der Events und Keywords automatisch aktualisiert.

Alle Keywords werden aus Dokumenten ausgelesen, welche mit dem Learning Unit verbunden sind. Hier gehört ein *document_title* sowie ein *document_link* dazu, wobei der *document_link* fakultativ ist. Ein **keyword** an sich ist über die sogenannte **keyword_occurrence** mit dem **document** und der **learning_unit** verbunden. In dieser Tabelle wird der PK des Keywords als FK mitsamt der Seite des Dokuments und dem Kontext abgespeichert.

Alle Tabellen sind entweder direkt oder indirekt vom Learning Unit abhängig – wird dieses gelöscht, werden ebenfalls alle dazugehörigen Daten aus der Datenbank entfernt.

Librarian Schnittstelle

Die Librarian-Backend-API wird von zwei Komponenten gebraucht: Das Librarian-CLI nutzt die API, um die empfangenen Daten vom Nutzer in die Datenbank abzuspeichern. Bevor die abgespeicherten Daten im Nabu-Backend abgespeichert werden können, müssen diese in ein vorgegebenes JSON-Format generiert werden.

Die Librarian-Backend-API besteht aus den folgenden Methoden:

- **get_learning_unit_id**: Dient dazu, vom Librarian-CLI zu überprüfen, ob eine mitgegebene Learning Unit bereits in der Datenbank vorhanden und gültig ist. Falls sie noch nicht abgespeichert ist und die nötigen Daten wie Name und Whitelist mitgegeben wurden, wird sie in der Datenbank abgespeichert.
  In beiden Fällen wird die Learning-Unit-ID (zum Beispiel «DB1») zurückgegeben. Falls keine Learning Unit mitgegeben wird, wird eine Fehlermeldung mit der Aufforderung, eine Learning Unit mitzugeben, zurückgegeben.
- **get_learning_unit_ids**: Dieser Methode kann man einen «Last Modified»-Parameter mitgeben. Dieser wird benutzt um zum Beispiel alle Learning Units, die in den letzten 24 Stunden modifiziert wurden, zurückzugeben. Falls der Parameter nicht gesetzt ist, wird eine Liste von allen vorhandenen Learning Units zurückgegeben.
- **save_keywords**: Die Keywords, welche von der Librarian CLI geschickt werden, werden pro Dokument verarbeitet. Mit jedem Dokument, das eingelesen wurde, wird ein Hash mitgeschickt, mit welchem die Librarian API überprüfen kann, ob es Änderungen in dem jeweilligen Dokument gab. Falls es keine Änderungen gab, wird das Dokument übersprungen. Im gegensätzlichen Fall wird anhand vom Dateinamen überprüft, ob das Dokument bereits vorhanden ist, ansonsten wird ein neuer Datensatz erstellt. Ist das Dokument bereits vorhanden, werden alle alten Keywords und deren Occurrences des veralteten Dokumenten-Datensatzes durch die neuen ersetzt, um den neusten Stand im Dokument widerzuspiegeln. Zusätzlich wird am Schluss das date_modified der betroffenen Learning Unit auf den aktuellen Zeitpunkt gesetzt.
- **save_events**: Die Events von der Librarian-CLI werden ebenfalls in einem JSON mitgeschickt, in welchem sie nach Learning Units unterteilt sind. Da es zu dem Zeitpunkt des abgeschlossenen Projektes keine Möglichkeit gibt um herauszufinden, ob sich ein Event nur verändert hat oder komplett neu ist, werden alle

Events in der jeweiligen Learning Unit gelöscht und neu abgespeichert. Auch hier wird am Schluss das date_modified der Learning Unit auf den aktuellen Zeitpunkt angepasst.

- **generate_mindmeld_json**: In dieser Funktion werden die Daten für das Nabu-Backend zusammengestellt. Dafür braucht man mindestens die Informationen vom jeweiligen Learning Unit (Name, ID, Whitelist), die dazugehörigen Keywords (inklusive Documents und Occurences) und Events. Daraus resultiert eine JSON-Struktur, welche vom Nabu-Backend korrekt entgegengenommen und verarbeitet werden kann (siehe Kapitel 4.3.6, MindMeld-JSON).

Wie auch die Nabu-Backend Schnittstelle wird die Librarian-Backend-Schnittstelle auf dem Server mit Waitress gestartet. Mit einer ENV-Umgebungsvariable kann man hier jedoch auch auf den Development-Modus von Flask umschalten, welcher mehr Infos bietet und Debugging erlaubt.

PonyORM und Datenbank

Wie in Kapitel 4.2.3 bereits erwähnt wurde sich beim ORM für PonyORM entschieden. Diesen ORM kann man für beide Ansätze gebrauchen: Code-First und Database-First. Bei diesem Projekt fiel die Entscheidung auf Code-First.

Mit PonyORM stellt jede Datenbankentität eine eigene Klasse dar. Sobald alle Klassen erstellt sind, wird eine Verbindung mit der Datenbank hergestellt und das Mapping erstellt.

Zusätzlich kann man an dieser Stelle mit einer einzigen Zeile die Datenbank unter dem Host ändern. Dies hat während der Implementation viel Zeit gespart, da in der Mitte der Construction-Phase festgelegt wurde, dass die Datenbank statt lokal gehostet einen zentralen Punkt auf dem Server darstellen sollte. Dies vereinfacht es, die Daten abzugleichen, falls mehrere Beteiligte an einer Learning Unit mitwirken. Deshalb wurde die lokale SQLite-Datenbank, die anfangs im Einsatz war, mit einer MariaDB-Datenbank auf dem Server ersetzt. Erheblicher Mehraufwand ist daraus nicht entstanden, da mit PonyORM die Datenbank einfach ausgetauscht werden konnte. Einzig die Datentypen konnten durch die grössere Vielfalt in der MariaDB verfeinert werden.

Da PonyORM Flask von sich aus unterstützt, war es auch hier möglich, mit einem Einzeiler das ORM-Tool zu instanziieren und die Entity-Klassen direkt benutzen.

MindMeld-Modul

Um das Nabu-Backend regelmässig mit den neusten Daten zu aktualisieren, wurde ein Skript geschrieben. Dieses wird über einen automatisierten Cronjob standardmässig alle 24 Stunden aufgerufen. Somit wird der Stand der Daten im Nabu-Backend einmal pro Tag aktualisiert. Sollte es notwendig sein, kann das Skript allerdings auch vom Server-Administrator manuell angestossen werden.

Das Skript besteht aus drei Requests: Zuerst werden alle Learning Units, die in den letzten 24 Stunden modifiziert wurden, als Liste angefordert. Danach wird das JSON über die Funktion generate_mindmeld_json generiert und im letzten Schritt an die Nabu-Backend Schnittstelle geschickt.

4.3.4 Moodle-Client

Überblick

Um den Moodle-Benutzern zu ermöglichen, mit dem Nabu-Backend zu kommunizieren, wurde ein User mit dem Usernamen nabu_bot erstellt. Durch einen Messaging Consumer werden die Nachrichten, welche diesem User zugesendet werden, abgefangen und an das Nabu-Backend weitergeleitet.

Für die Installation und Lauffähigkeit des Nabu-Messaging-Consumer-Plugins müssen einige Einstellungen sowie Berechtigungen gesetzt werden. Diese sind im Readme-File des Plugins einsehbar.

Messaging Consumer

Das Nabu-Messaging-Consumer-Plugin ist eine Subklasse der abstrakten Klasse message_output, die aus dem Moodle-Core stammt. Diese Klasse gibt folgende Methoden vor, die implementiert sein müssen:

- **send_message**: Diese Methode fängt die Nachricht ab, die verschickt wurde. In der Nabu Messaging Consumer Implementation wird hier überprüft, ob der Empfänger der Benutzer nabu_bot ist. Falls nein, wird der Rest des Codes nicht aufgerufen.
  Bei einer Nachricht an den Nabu-Bot wird die Nachricht inklusive dem eventuellen Session-Token an die API des Nabu-Backends gesendet. Falls noch keine Session gesetzt war, wird eine solche zusätzlich mit der Antwort des Nabu-Bots zurückgesendet. Die Session wird in einem Cookie abgespeichert.
  Die Antwort des Backends wird formatiert und durch eine interne Message-API-Funktion von Moodle selbst an den User zurückgeschickt.
- **config_form**: Diese Methode wäre dafür da, um ein Formular zu generieren, mit dem der Benutzer Einstellungen für das Plugin setzen könnte. In diesem Fall wird diese Methode nicht benötigt und ist deswegen leer.
- **process_form**: Falls Einstellungen in im Formular von config_form gesetzt worden wären, würden diese hier verarbeitet und in der Datenbank abgespeichert werden. Auch diese Methode wird nicht benötigt.
- **load_data**: Mit dieser Methode werden die vorher abgespeicherten User-Einstellungen wieder geladen und für das Plugin zugriffbar gemacht. Diese Methode wird ebenfalls nicht benötigt.
- **is_system_configured**: Damit der Nabu-Bot funktioniert, muss ein Benutzer mit dem korrekten Usernamen vorhanden sein. Mit dieser Methode wird überprüft, ob der Benutzer korrekt erstellt wurde und verfügbar ist.

Hosting

Moodle selbst wird in diesem Projekt zu Testzwecken durch einen bitnami-Docker-Container [39] gehostet. Das Messaging-Consumer Plugin wird nach der automatischen Installation von Moodle an den korrekten Ort in der Ordnerstruktur kopiert und danach manuell vom Moodle-Administrator über die Oberfläche installiert.

Performance

Zu Projektende werden die Nachrichten in den ersten paar Sekunden nach Öffnen der Konversation nicht in der chronologisch korrekten Reihenfolge angezeigt. Auch wird die Antwort des Nabu-Backends erst nach einigen Sekunden angezeigt.

Dies begründet sich darin, dass nicht bei jeder gesendeten Nachricht die Konversation aktualisiert wird. Die Nachrichten werden in fixen Intervallen über einen Ajax-Request aus dem PHP-Backend angefordert und aktualisiert. Diesen Vorgang kann man jedoch nicht direkt nach dem Empfang der neuen Nachricht vom PHP-Messaging-Consumer aus anstossen. Nach der Evaluation des Problems wurde aus Zeitgründen kein Lösungsansatz entwickelt.

4.3.5 Telegram-Client

Framework

Der Telegram-Bot basiert auf dem Python-Telegram-Bot Framework [40]. Die essentielle Kommunikation für den Bot wird durch das Framework abgehandelt, weswegen nur noch die für dem Nabu-Bot spezifischen Funktionen implementiert werden mussten.

Anzumerken ist, dass der Nabu-Telegram-Bot nicht auf den Mobilgeräten der Nutzer direkt läuft. Vielmehr läuft er als Docker-Container zusammen mit dem Rest der Nabu-Umgebung und erhält Telegram-Nachrichten der Nutzer, die er dann zur Verarbeitung ans Nabu-Backend überstellt. Er bildet also eine Art Brücke zwischen der eigentlichen Telegram-Messenger-Applikation und dem Nabu-Backend.

Botfather

Damit ein Telegram-Bot von aussen erreichbar ist, muss dieser durch den sogenannten Botfather [41] [42] generiert werden. Der Botfather schickt nach der Generierung ein Token zurück, das im Python-Code implementiert werden muss. Nur mit diesem Token ist man berechtigt, den Telegram-Bot zur Verfügung zu stellen. Dieses Token wird zusammen mit der API-URL des Nabu-Backends in einer .env Datei auf dem Server gespeichert. Der Telegram-Bot wird auf dem Server in einem Docker Container gestartet.

Methoden

Im Telegram-Bot wurden folgende Methoden implementiert:

- **start**: Jeder Telegram-Bot benötigt eine start-Methode. Diese wird automatisch aufgerufen, sobald ein Nutzer den fraglichen Bot in seiner Telegram-App als Kontakt hinzufügt und somit eine Konversation beginnen kann. In diesem Fall erscheint ein Begrüssungstext.
- **assemble_request_json**: Hiermit wird das JSON, welches den Text und die Session ID beinhaltet, zusammengestellt.
- **question**: Normalerweise wird mit einem Telegram-Bot nur über Commands wie /help oder /question kommuniziert. Da dieser Bot jedoch nur Fragen beantworten muss, wird die Methode “question” als Default Handler benutzt. Somit wird sie bei jeder vom Bot erhaltenen Nachricht aufgerufen. Die Methode sendet die Nachricht an die API der Nabu-Backends und wartet direkt die Antwort ab.
- **send_message**: Mit dieser Methode wird schlussendlich die Antwort vom Nabu-Backend an den Client zurückgesendet. Da eine Nachricht nur 4096 Bytes gross sein darf, werden zu lange Nachrichten vor dem Senden in mehrere Nachrichten unterteilt [43].

4.3.6 Schnittstellen

Überblick

Wie in Kapitel 4.2.9 beschrieben, wurden insgesamt drei Netzwerk-Schnittstellen analysiert, über welche JSON-Daten ausgetauscht werden. Hierbei muss vor allem zwischen dem Datenaustausch vom Librarian-CLI zum Librarian-Backend und vom Librarian-Backend zu MindMeld unterschieden werden. Nachfolgend sind die definierten Schemas gefüllt mit Beispiel-Daten gelistet:

Data-JSON

Data-JSONs werden zum Hochladen von Daten aus der Librarian-CLI in die Librarian-Backend-Datenbank gebraucht. Es existieren drei verschiedene Ableger dieses JSON-Typs.

Learning-Unit-Data

Um sich unnötigen Aufwand zu sparen, falls eine Learning Unit nicht valide ist, wird vor jedem Daten-Upload ein Request auf die Learning-Unit-Daten vollzogen. Eine Learning Unit ist dann valide, wenn die ID bereits vorhanden ist und es keine Überschneidungen mit anderen Learning Units gibt, oder wenn die ID noch nicht existiert und frisch erstellt werden kann. Das Learning-Unit-Data-JSON wird, um genau diese Bedingungen zu testen, von der CLI an das Librarian-Backend geschickt.

```
{
  "learning_unit_id":"DB1",
  "learning_unit_name":"Datenbanken 1",
  "learning_unit_whitelist":"DBS1, DB1"
}
```

Key	Beschreibung
"learning_unit_id":	Die ID der Learning Unit.
"learning_unit_name":	Ganzer Name der Learning Unit.
"learning_unit_whitelist":	Alternativen festlegen, die für die Ansprache dieser Learning Unit möglich sein sollen.

Document-Data

Document-Data-JSONs repräsentieren die geparsten fachlichen Dokumente und werden jeweils zur Bereicherung der Librarian-Datenbank an das Librarian-Backend geschickt.

```
{
  "learning_unit_id":"DB1",
  "documents":[
    {
      "doc_title":"Dbs1 - DB-Entwurf: Abbildung UML zu rel. Modell",
      "doc_link":"https://ost.edu.ch/link_zu_file",
      "doc_hash":"f2604b81c010373d3bc8ec002360ae5716969c420a1ddbbca1c8072682",
      "pagecount":2,
      "pages":[
        {
          "number":1,
          "keywords":[
            {
              "word":[
                "Modell",
                "UML",
                "rel",
                "Abbildung"
              ],
              "context":"1 Dbs1 - DB-Entwurf: Abbildung UML zu rel. Modell"
            },
            {
              "word":[
                "Datenbanksysteme"
              ],
              "context":"Datenbanksysteme 1"
            }
          ]
        },
        {
          "number":2,
          "keywords":[
```

```
            {
              "word":[
                "Worum",
                "geht"
              ],
              "context":"Worum geht es"
            }
          ]
        }
      ]
    }
  ]
}
```

Key	Beschreibung
“learning_unit_id”:	Die ID der Learning Unit.
“documents”:[...]	Beinhaltet die extrahierte Information von mindestens einem PDF-Dokument.
“doc_title”:	Der Titel des Dokuments. Erhalten durch den Dateinamen der jeweiligen Datei, der aus dem Pfad extrahiert wird.
“doc_link”:	Wird gesetzt, wenn die Click-Option --link mitgegeben wurde, ansonsten leerer String.
“doc_hash”:	Die durch das Parsen resultierenden Daten werden mit SHA256 gehashed. Dies dient zur Überprüfung von möglichen Änderungen in der Datei. Dabei wird im Librarian-Backend in der Datenbank der Hash und der Titel des Dokuments mit dem erhaltenen Hash und Titel von den JSON-Daten verglichen.
“pagecount”:	Die Anzahl Seiten des geparsten Dokuments.
“pages”:[...]	Beinhaltet die extrahierte Information zu jeder Seite.
“number”:	Die Seite des Dokuments.
“keywords”:[...]	Beinhaltet die schlussendlich für das Projekt wichtige Keywords.
“word”:[...]	Enthält extrahierte Nomen des geparsten Kontextes.
“context”:	Enthält eine extrahierte Textzeile.

Event-Data

Das Event-Data-JSON beinhaltet die Event-Daten, die durch das Parsen von organisatorischen Dokumenten entstehen. Diese werden ebenfalls zur Bereicherung an die Librarian-Datenbank im Backend gesendet.

```
{
  "learningunit":{
    "DB1":[
      {
        "name":"Testat 1",
        "date":"20.11.2020",
        "time":"12:00",
        "location":"Aula"
      }
    ],
    "BuRe1":[
      {
        "name":"Testat 1",
        "date":"15.11.2020",
        "time":"09:30",
        "location":"2.012"
      },
      {
        "name":"Testat 2",
        "date":"25.12.2020",
        "time":"00:00",
        "location":""
      }
    ]
  }
}
```

Key	Beschreibung
“learningunit”:	Enthält die Daten, die durch das geparste CSV-Dokument erhalten wurden.
“DB1 und BuRe1”:[…]	Hier wird der Key direkt mit der Learning Unit ID gesetzt.
“name”:	Enthält alle zu dieser Learning Unit gehörenden Events.
“date”:	Titel des organisatorischen Events.
“time”:	Wenn gegeben, Ort der Durchführung des Events.
"location”:	Ort des Events.

MindMeld-JSON

Das MindMeld-JSON repräsentiert das Schema von Librarian-DB zu Nabu-Backend, also diejenigen Daten, welche in der MindMeld-Wissensbasis hinterlegt werden sollen. Es wird an den /librarian-Endpunkt des Nabu-Backend-Servers geschickt und enthält jeweils genau eine komplette Learning Unit. Der Erhalt eines solchen JSON-Datenstruktur veranlasst das Backend, die gesamte Learning Unit durch die neuen Informationen zu ersetzen. Ein Update widerspiegelt also immer alle zwischenzeitlich geschehenen Änderungen.

```
{
  "learning_unit":{
    "id":"DB1",
    "name":"Datenbanken 1",
    "learning_unit_whitelist":[
      "DBS1",
      "DB1"
    ],
    "events":[
      {
        "date":"Fri, 20 Nov 2020 12:00:00 GMT",
        "event_location":"Raum 2.017",
        "id":"3",
        "name":"Testat 1 - Datenbank aufsetzen"
      }
    ],
    "keywords":[
      {
        "word":"Transaktion",
        "id":"1",
        "doc_occurrences":[
          {
            "doc_title":"Woche 5: Transaktionen und Rollbacks",
            "doc_link":"https://ost.edu.ch/db1/woche_5.pdf",
            "occurrences":[
              {
                "page":"3",
                "context":"geht es vor allem um Transaktionen, die man"
              },
              {
                "page":"5",
                "context":"Datenbank mittels Transaktionen auf einen Rutsch"
              }
            ]
          }
        ]
      },
      {
        "word":"Rollback",
        "id":"2",
        "doc_occurrences":[
          {
            "doc_title":"Woche 5: Transaktionen und Rollbacks",
            "doc_link":"https://ost.edu.ch/db1/woche_5.pdf",
            "occurrences":[
              {
                "page":"5",
                "context":"mit dem Rollback wieder rückgängig gemacht"
              }
```

```
            ]
          }
        ]
      }
    ],
    "keyword_synonyms":[
      {
        "keyword":"Transaktion",
        "synonyms":[
          "Transaction"
        ]
      }
    ]
  }
}
```

Key	Beschreibung
“learning_unit”:	Enthält die gesamte relevante Information von einer Learning Unit.
“id”:	Die Learning Unit ID.
“name”:	Ganzer Name der Learning Unit.
“learning_unit_whitelist”:[...]	Enthält Alternativen, die für die Anspra-che dieser Learning Unit möglich sind.
“events”:[...]	Enthält alle zu dieser Learning Unit gehö-renden Events.
“date”:	Fälligkeit oder Datum des Events.
“event_location”:	Wenn gegeben, Ort der Durchführung des Events.
“id”: in “events”:[…]	Die DB ID des Events.
“name”:	Name des Events.
“keywords”:[...]	Enthält die gesamte fachliche Information zu dieser Learning Unit.
“word”:	Das jeweils betrachtete Keyword.
“id”: in “keywords”:[…]	Die DB ID des Keywords.
“doc_occurrences”:[…]	Alle Vorkommnisse des jeweils betrachte-ten Keywords.
“doc_title”:	Der Name des Dokuments.
“doc_link”:	Wenn gegeben, wird hier die URL ge-setzt.
“occurrences”:[…]	Enthält alle Vorkommnisse aus denen das Keyword extrahiert wurde .
“page”:	Die Seitenzahl der Dokumentseite auf der dieses Keyword gefunden worden ist.
“context”:	Die Zeile in der das Keyword gefunden worden ist.

Conversational JSON

Das konversationelle JSON wird zwischen dem Nabu-Backend und den Clients, die es ansprechen, ausgetauscht. Es enthält die Session-ID, die vom Client gehalten wird sowie den Inhalt der Nachricht, die ausgetauscht werden soll.

Die Session-ID wird, falls die Nachricht von Client zu Nabu-Backend geht, dazu verwendet, den richtigen Gesprächskontext zu laden. Geht die Nachricht vom Backend an den Client, so merkt sich der Client die erhaltene ID und nutzt sie für die weitere Kommunikation.

```
{
  "nabu-session": "5aba6969-ac07-2403-ae86-1337c117d00d",
  "text": "I forgot what to hand in for DB1!"
}
```

Key	Beschreibung
“nabu-session”:	Enthält den vom Session Store des Servers generierten Schlüssel zur Identifikation des Clients.
“text”:	Enthält den Text-Inhalt der Nachricht.

4.3.7 Automatische Testverfahren

Unit Tests

Für die Komponenten des Nabu-Projekts, die eine gewisse Komplexität haben, nicht nur Requests ausführen und nicht Teil des MindMeld-Moduls sind (siehe Kapitel 4.3.8, Tests des Chatbots), wurden spätestens ab der Construction-Phase zwecks Qualitätssicherung und Code-Korrektheit Unit Tests implementiert, wie in Kapitel 4.5.10 definiert. Die so getesteten Module umfassen den Nabu-API-Server und das Librarian-CLI Tool.

Die Tests erfolgten jeweils in zwei Stufen: Zunächst lokal während der Entwicklung und danach auf dem Continuous-Integration-Tool als Teil des Commit-Workflows. Damit wurde sichergestellt, dass der Code auf verschiedenen Systemen korrekt läuft.

Framework

Zur Implementierung dieser Tests wurde das Pytest-Framework [44] verwendet, da es sich durch guten Support, weite Verbreitung und gute Skalierbarkeit auszeichnet und sich darüber hinaus nahtlos mit den zur Entwicklung des Projekts verwendeten JetBrains-Entwicklungsumgebungen integrieren lässt.

Coverage-Statistik

Die untenstehende Statistik zeigt die Test-Coverage am Ende des Projektes auf.

Die im Projektplan definierte Package-Coverage von 60% (siehe Kapitel 4.5.10) wurde auf den als wichtig erachteten Subkomponenten im Nabu-Backend sowie im Librarian-CLI-Client erreicht.

Komponente	Package	File	Coverage [%]
Nabu-Backend	**persisters**		**71**
		eventmapper.py	100
		eventpersister.py	92
		keywordmapper.py	100
		keywordpersister.py	100
		learningunitmapper.py	100
		learningunitpersister.py	91

		mapper.py	66
		persister	40
		persisterstack.py	37
	server		**63**
		sessionstore.py	63
Librarian-CLI	**-**		**61**
		librarian.py	54
	driver		**66**
		csvdriver.py	40
		drivermanager.py	82
		pdfdriver.py	74

Tabelle 2: Test-Coverage

Für das Librarian-Backend, den Moodle-Client und den Telegram-Bot wurden keine Tests erstellt. Dies begründet sich vor allem darin, dass für all diese Komponenten ein massiver Mocking-Overhead entstanden wäre. Im Falle des Moodle-Clients betrifft dies vor allem die Messaging Outputs, die schwer zu testen sind. Im Falle des Telegram-Bots handelt es sich um das gesamte Nachrichtenverarbeitungs-API von Telegram, und im Falle des Librarian-Backends um die Datenbank. Zum Erstellen von Mocks und zum Testen dieser Komponenten reichte die Zeit in diesem Projekt nicht aus.

4.3.8 Manuelle Testverfahren

Test des Chatbots

Zu Beginn der Construction-Phase wurde die Entscheidung gefällt, dass für das dem Chatbot zugrunde liegende MindMeld-Modul keine automatischen Tests implementiert werden. Diese Entscheidung begründet sich darin, dass es umständlich gewesen wäre, jede Änderung in den Trainingsdaten des Chatbots sowie dem konkreten Ablauf der Konversation in den Unit Tests zu reflektieren und ein solcher Versuch mehr Aufwand als Nutzen bereitet hätte. Die Interaktion mit dem Chatbot ist insofern organisch, als dass die genaue Abhandlung einer Konversation letztendlich nicht dem Programmierer, sondern den trainierten Resolvern, der Wissensdatenbank und der NLP-Pipeline obliegt. Zusammen mit der Option, mehrere Antwortphrasen für den gleichen Intent zu definieren, bedeutet dies, dass die Antwort auf dieselbe Anfrage, wenn sie mehrmals gestellt wird, verschieden ausfallen kann. Aus diesem Grund macht es wenig Sinn Unit-Tests für diesen Teil des Projekts zu schreiben.

Um den Chatbot trotzdem testen zu können, wurde nach grösseren Änderungen an der Konversationslogik oder den Trainingsdaten und -parametern das Telegram-Frontend verwendet, um Konversationen gemäss Use Cases 1 und 2 komplett durchzuspielen. Darüber hinaus wurden die nicht in den Use Cases erfassten, aber trotzdem implementierten Konversationsbausteine (wie beispielsweise das Grüssen des Nutzers) auf die gleiche Weise getestet. Damit liess sich sicherstellen, dass keine Änderungen eingebaut worden waren, die sich negativ auf die Qualität der Interaktion niederschlugen. Gleichzeitig war bewiesen, dass die Clients auch nach dem Update noch wie geplant mit dem Chatbot funktionierten.

Usability Tests

In der zweiten Hälfte des Projekts wurden zwecks Analyse der Praktikabilität des Nabu-Chatbots Usability Tests mit sechs Personen aus dem Hochschulumfeld, meist selbst Studierende der OST Rapperswil-Jona, durchgeführt. Diese Usability-Tests richten sich spezifisch nach Use Cases 1 und 2 und zielen auf die Beschaffung von Informationen

zu inhaltlichen und organisatorischen Daten über die Nabu-Konversationsschnittstelle ab.

Im Kontext eines Chatbots kommt den Usability Tests eine weitere, besondere Bedeutung zu – sie eignen sich nämlich perfekt, um weitere Trainingsdaten für MindMeld zu erlangen. Stellt sich also heraus, dass User Abfragen in einer unvorhergesehenen Art und Weise tätigen, so lassen sich diese Daten zur Erweiterung des Trainings nutzen. Deshalb werden die Usability-Test-Abfragen serverseitig geloggt und im Nachhinein analysiert und die Trainingsdaten entsprechend ergänzt.

Zur Durchführung der Usability Tests wurde pro Test ein Protokollrohling erstellt, der die Umstände der Durchführung definiert und beschreibt. Ein solcher Rohling weist die folgende Struktur auf:

- Protokollarische Informationen zum Test, wozu Datum, Name und Funktion bzw. Rolle des Probanden (z.B. «Studierender») gehören, vor dem Test auszufüllen.
- Die Vorbereitung des Tests inklusive der Definition der Erkenntnisziele und Erfolgskriterien.
- Die Ausganglage des Tests, also eine Garantie, dass alle Probanden dieselbe Situation antreffen und veränderte anfängliche Umstände kein Faktor sind, der das Resultat beeinflussen kann.
- Die Aufgabenstellung des Tests, verpackt in ein Rollenspiel-Szenario, das dem Probanden zur Durchführung des Tests vorgelegt wird.
- Raum für die Erfassung der Testresultate, nach dem Test auszufüllen.

Aus diesen Rohlingen wurden danach die tatsächlichen Testberichte erstellt und per Laufnummer eindeutig gekennzeichnet.

Usability-Testresultate

Bei der Auswertung der sechs durchgeführten Usability Tests wurden die folgenden Testresultate erzielt:

- **Use Case 1**: Vier von sechs Probandinnen und Probanden erreichten das Ziel, mehr Informationen über Transaktionen zu erhalten, was einer Erfolgsrate von 66.6% entspricht. Gründe des Scheiterns waren vor allem, dass versucht wurde, die Informationen auf Deutsch abzufragen, ein inkorrektes Kürzel für die Learning Unit verwendet wurde oder die Abfragen in einer Weise formuliert waren, mit denen MindMeld noch nicht umgehen konnte.
- **Use Case 2**: Fünf von sechs Probandinnen und Probanden erreichten das Ziel, zunächst einen Überblick über alle Events zu erhalten und dann die Detailinformationen zu einem dazu abzurufen. Dabei handelt es sich um eine Erfolgsrate von 83.3%. Anzumerken ist, dass alle Teilnehmenden die Übersicht über alle Ereignisse angezeigt bekommen konnten, jemand scheiterte aber am Aufrufen der Detailansicht des einzelnen Events. Grund für die höhere Erfolgsrate ist vermutlich, dass die Mechanik mit der Auswahl der Learning Unit vor der Abfrage an dieser Stelle bereits aus dem UC1-Test vertraut war und es somit nur noch ein kleiner Schritt zur Eventliste war.

Von den Probanden bemängelt wurde insbesondere, dass die Antworten im Fehlerfall und die Hilfestellungen nicht sonderlich aussagekräftig seien, und dass Abfragen auf Deutsch nicht funktionieren. Letzteres wurde in den Stretch Goals als potentielles Ziel definiert, aber zum Zeitpunkt der Usability Tests nicht implementiert und konnte somit auch nicht getestet werden.

Gelobt wurde die Idee, über einen Chatbot den Studierenden solche Abfragen zu ermöglichen, die Geschwindigkeit, mit der das möglich ist, und die Flexibilität des Ansatzes. Das informelle Feedback abseits der Usability Tests fiel insgesamt ebenfalls recht positiv aus, und viele der befragten Studierenden könnten sich durchaus vorstellen, einen solchen Chatbot regelmässig zu nutzen, vorausgesetzt, die Mängel würden

behoben und die Fähigkeiten noch etwas erweitert. Hier wurde auch die Möglichkeit erwähnt, über den Chatbot das Menü in der Kantine abfragen zu können.

Verbesserungen

Aus den Use-Case-Testresultaten wurden zwei direkt implementierbare Konsequenzen gezogen.

Wie vor den Tests bereits erwartet, war die Menge der Trainingsdaten zu klein, was sich negativ auf die Nutzererfahrung auswirkte. Klar formulierte Absichten konnten von Nabu nicht erkannt werden, da sie noch nicht eintrainiert worden waren. Zur Behebung dieses Problems wurden einerseits die eingegangenen Abfragen direkt, andererseits darauf basierende abgewandelte Formulierungen zu den Trainingsdaten hinzugefügt. Daraus ergab sich in nachfolgenden manuellen Tests eine unmittelbar feststellbare Verbesserung bei der Erkennung der User-Intents.

Ausserdem wurde die Hilfestellung, die Nabu geben kann, als unzureichend bewertet und noch einmal überarbeitet, um Abfragebeispiele und mehr Details zur Verfügung zu stellen. Darüber hinaus wurden die Hilfestellungen, die beispielsweise bei fehlender Learning Unit angezeigt werden, noch einmal überarbeitet und umformuliert.

Die Implementation von zusätzlich vorgeschlagenen Features, wie beispielsweise das oben erwähnte Kantinen-Menü, war nicht mehr Teil dieses Projektes, wurde aber auf die Liste der möglichen Erweiterungen aufgenommen. Für mehr Details siehe Kapitel 4.4.2.

4.4 Resultate und Weiterentwicklung

4.4.1 Resultate

Zusammenfassung

Das zu Beginn der Arbeit definierte Minimum Viable Produkt wurde grösstenteils erreicht. Im Rahmen des Nabu-Projekts wurde eine komplette, funktionstüchtige Pipeline erstellt, die es einem Dozierenden erlaubt, Vorlesungsunterlagen und Event-Daten zu erfassen und in die Wissensbasis eines MindMeld-Chatbots einfliessen zu lassen. Dort können sie danach über zwei Frontend-Clients von Studierenden abgefragt werden.

Resultierend aus den Schwierigkeiten, die sich im Umgang mit der für das Projektteam neuen Programmiersprache Python sowie der MindMeld-Technologie ergaben, mussten allerdings einige Abstriche gemacht werden. Priorisiert wurden, wie im Projektplan definiert, die grundlegende Funktionstauglichkeit der Pipeline und die Qualität des Codes über das Einbauen von neuen Features.

Nichtsdestotrotz wäre die entstandene Applikation nach einigen Anpassungen bereit zum Einsatz in einer Produktionsumgebung an der OST Rapperswil-Jona oder einer anderen schulischen Institution. Diese Anpassungen und Ergänzungen sind weniger funktionaler, sondern vielmehr organisatorischer und praktischer Natur (siehe nächstes Kapitel).

Erreichte Features

Zu den erreichten Features des MVP gehören die folgenden:

- Es existiert ein Chatbot-Backend, genannt Nabu, welches auf dem MindMeld-Framework basiert und über ein Netzwerk angesprochen werden kann. Auf diesem Weg können Anfragen zu vorgängig erfassten Learning Units gemacht werden, die entweder fachlicher («wo finde ich mehr Informationen zu diesem Thema?») oder organisatorischer («welche Termine muss ich wahrnehmen?») Natur sind, gemäss den definierten Use Cases 1 und 2.
- Es existiert ein Backend-Modul, der sogenannte Librarian, welcher die erwähnten fachlichen und organisatorischen Informationen entgegennehmen und in einer Datenbank hinterlegen kann. Von dort können die Informationen dann in die eigentliche Wissensbasis des MindMeld-Chatbots übertragen werden und sind dadurch abfragbar.
- Es existiert ein kommandozeilenbasiertes Librarian-Frontend, welches es erlaubt, Daten für den Librarian zu erfassen und hochzuladen. Somit können von Dozierenden ihre eigenen Vorlesungen und Termine im Chatbot verfügbar gemacht werden.
- Es existieren zwei Frontends, eines für die Hochschul-Plattform Moodle und eines für die Messenger-Applikation Telegram, mit welchen mit dem Nabu-Chatbot Konversationen geführt werden können. Damit ist es Studierenden möglich, auf die hochgeladenen Daten und Inhalte zuzugreifen.

Somit ist die zu Beginn des Projektes definierte Pipeline von Datenerfassung zu Datenabfrage funktionell abgeschlossen. Die Stretch Goals konnten allerdings auf Grund des Zeitmangels nicht mehr in Angriff genommen werden und verbleiben deshalb unbearbeitet.

Moodle-Beispiel

Der untenstehende Screenshot zeigt eine beispielhafte Interaktion mit dem Nabu-Chatbot über das Moodle-Chatinterface. Gezeigt ist eine fachliche Abfrage, die Use Case 1 entspricht.

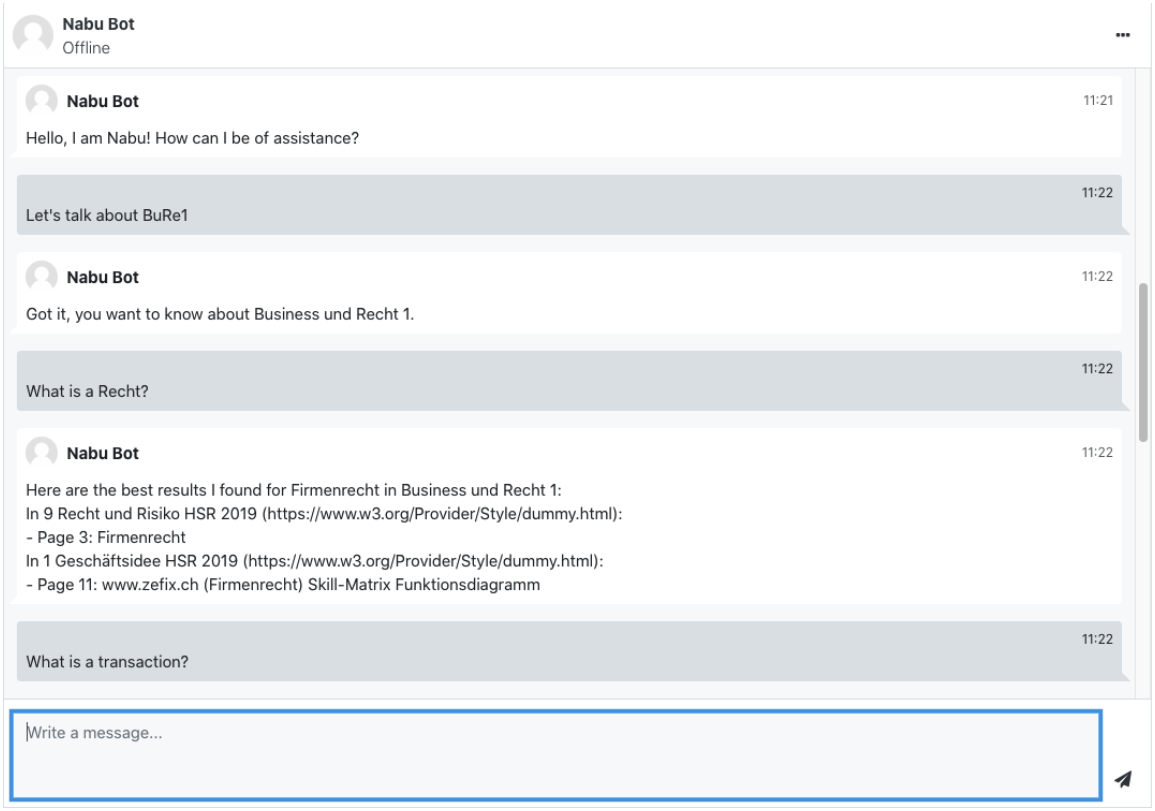


Abbildung 13: Moodle-Beispielkonversation

Telegram-Beispiele Die zwei untenstehenden Screenshots zeigen zwei beispielhafte Konversationsabläufe mit Nabu über den Telegram-Client. Dabei entspricht die linke Konversation Use Case 1 (fachliche Abfrage), die rechte Use Case 2 (organisatorische Abfrage).

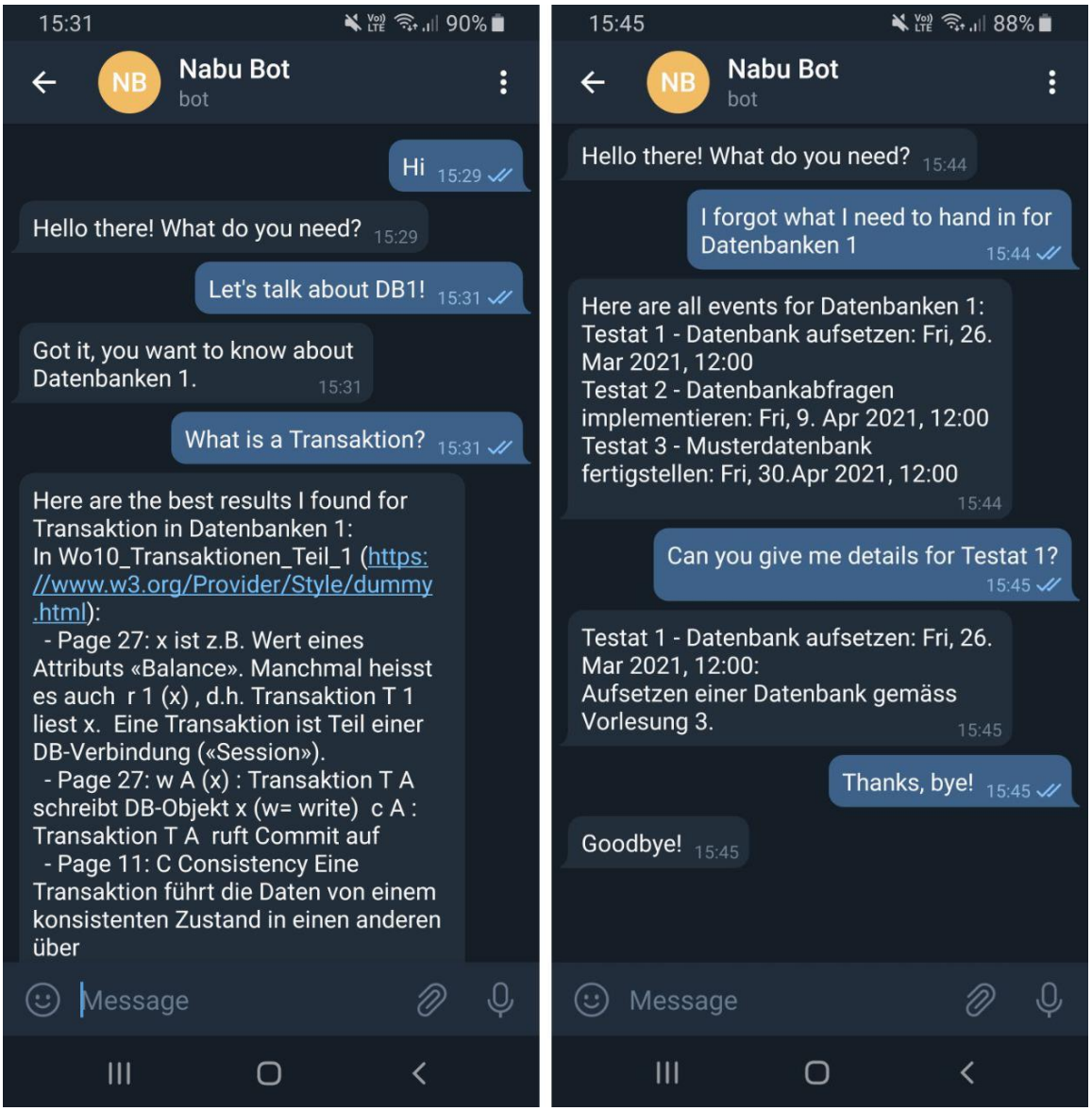


Abbildung 14: Telegram-Beispielkonversationen

Probleme Die grössten Schwierigkeiten, die sich während der Planung und Entwicklung des Nabu-Projektes einstellten, hingen mit Python als Programmiersprache und MindMeld als Chatbot-Framework der Wahl zusammen. Da bei keinem der Autoren dieser Arbeit vorgängig Erfahrungen im Umgang mit diesen Technologien bestand, stellte die Anfangsphase des Projektes ein grundlegendes Einarbeiten dar. Dazu gehörte insbesondere das Zusammenspiel der verwendeten Bibliotheken mit der Testing-Infrastruktur und der Continuous-Integration-Umgebung, das bis spät im Projekt regelmässig Probleme verursachte.

Gegen Schluss der Construction Phase ergaben sich auch ein paar Probleme mit dem setup.py-File, welches ein lokales Installieren des Librarian-CLI-Tools über «pip install» ermöglichen sollte. Bedingt durch die Breite des Projekts war es darüber hinaus nötig, sich mit vielen sekundären Libraries und Tools wie PyMuPDF oder Flask zu familiarisieren (siehe Kapitel 4.2.3), was ebenfalls Zeit in Anspruch nahm.

Daraus entstand ein erheblicher Mehraufwand, insbesondere in technologiespezifischer Recherche und Debugging. Vor allem bei der externen Interaktion mit der MindMeld-Applikation, also der Schnittstelle zwischen API-Server und Chatbot, musste viel Zeit aufgewendet werden, um erfolgreiche Abfragen durchführen zu können. Trotz vorgängiger Einkalkulierung im Risikomanagement (siehe Kapitel 4.5.9) war dieser Mehraufwand mit dem Eingehen von gewissen Kompromissen bei der Implementierung verbunden. Funktionelle Einschränkungen liegen zwar keine vor, da die gesamte Pipeline funktionstauglich ist, allerdings mussten gewisse Einbussen bei nichtfunktionellen Anforderungen in Kauf genommen werden.

Priorisierung

Zu den wichtigsten nicht implementierten Anforderungen gehören die folgenden zwei Punkte, die im nächsten Kapitel wesentlich detaillierter behandelt werden:

- HTTPS: Geplant war das Übertragen aller Daten in verschlüsselter Form. Die Implementation von HTTPS wurde allerdings zeitbedingt nicht mehr vollzogen.
- Upload-Autorisierung: Im Moment wird keinerlei Autorisierungsabfrage gemacht, was eine massive Sicherheitslücke darstellt.

Soll eine Bewegung in Richtung Produktionsdeployment erfolgen, so sind dies die mit Abstand wichtigsten zwei Punkte und müssen zwingend zuerst implementiert werden, da sie in einer Live-Umgebung eine massive Sicherheitslücke darstellen.

4.4.2 Möglichkeiten der Weiterentwicklung

Trainingsdaten, Interaktionsqualität

Dem Chatbot fehlt eine gewisse Finesse, was die Qualität und Quantität der MindMeld-Trainingsdaten anbelangt. Sämtliche Trainingsdaten, die im Rahmen des Nabu-Projekts erhoben wurden, stammen entweder direkt von den Autoren oder von den im Rahmen der Usability-Tests befragten Probanden.

Dies ist eine ausserordentlich kleine Stichprobengrösse, und es ist sehr wahrscheinlich, dass in einer Produktionsumgebung Intents in einer Art und Weise formuliert würden, an die bisher nicht gedacht wurde. Darunter würde die Qualität der Intent-Erkennung leiden – unter Umständen bis hin zum Nichterreichen der Use-Case-Ziele.

Aus diesem Grund ist es ratsam, die Trainingsdaten zu erweitern, um mehr Intent-Formulierungen abdecken und die Erfolgsrate von User-Interaktionen dementsprechend erhöhen zu können. Insbesondere auf dem Intent «make keyword query» besteht Verbesserungsbedarf.

HTTPS

Die Daten, die zwischen Frontend und Backend sowie zwischen Librarian und Nabu-Chatbot übertragen werden, werden über normales HTTP übertragen. In den NFRs (siehe Kapitel 4.1.2) wurde aber definiert, dass die Übermittlung der Daten zwischen den verschiedenen Komponenten des Projektes auf verschlüsseltem Weg via HTTPS erfolgt. Zur Umsetzung dieser Anforderung sowie zur Erstellung der dazu notwendigen Zertifikate fehlte zum Ende der Construction-Phase allerdings die Zeit.

Upload-Autorisierung und Authentifizierung

Im Moment wird weder im Librarian-Backend noch im Nabu-Chatbot überprüft, ob die Autoren der hochgeladenen Daten die dazu notwendige Berechtigung haben. Alle Uploads auf die jeweilige API werden fraglos verarbeitet. In einer Produktionsumgebung ist dies ein massives Sicherheitsleck, da es beliebigen Drittpersonen erlauben würde, eigene Uploads in die Nabu-Wissensdatenbank zu laden. Deshalb muss eine solche Autorisierungs-Prüfung zwingend nachgerüstet werden, bevor die Applikation tatsächlich live genutzt werden kann.

Im Librarian-Frontend kommt zur reinen Autorisierung unter Umständen noch eine Authentifizierung mit einem Rollen-Management hinzu. Damit würde es möglich, individuellen Nutzern individuelle Bearbeitungsrechte zu geben.

Abfrage-Autorisierung

Im Moment ist es allen, die Zugriff auf ein entsprechendes Frontend haben, problemlos möglich, Abfragen über alle erfassten Learning Units zu tätigen. Gemäss Betreuer stellt dies für den unmittelbaren Einsatz an der OST Rapperswil-Jona auch kein Problem dar.

Sollte allerdings ein Einsatz in einer Umgebung in Betracht gezogen werden, in der sensible Daten in der Wissensbasis hinterlegt werden, so würde sich die Implementation eines Autorisierungsmoduls für Anfragen über das konversationelle Interface anbieten. Damit würde die Abfrage von Daten beispielsweise nur noch für diejenigen Studierenden möglich, die sich im betreffenden Kurs eingeschrieben haben.

Dieses Feature wurde bereits als Stretch Goal erfasst, aber nicht umgesetzt.

Globaler pip-install

Per Stand vom Projektende ist der CLI-Client für den Librarian zwar lokal per pip-install verfügbar, allerdings nicht auf den Python Package Index (PyPI) [36] hochgeladen. Um den Dozierenden eine angenehmere Installationserfahrung zu bieten, wäre dies vor dem Deployment in einer Live-Umgebung empfehlenswert. Damit würde die Verteilung des Moduls über File-Transfer oder USB-Stick obsolet werden.

Weitere Abfragen und Aktionen

Bis anhin kann Nabu nur Abfragen auf vorher erfasste Unterrichts- und Termindaten bearbeiten. Dies liesse sich allerdings auch auf Live-Daten erweitern, die auf dem Web verfügbar sind, wie beispielsweise das tägliche Angebot in einer Mensa oder Cafeteria.

Durch seine Eigenschaft als konversationelles Interface eignet sich der Nabu-Chatbot darüber hinaus nicht nur zum Abfragen von Daten (wozu theoretisch eine konventionelle Suchmaschine ebenfalls reichen würde), sondern auch zum Erteilen von Aufträgen und zum Durchführen von Aktionen. Ein Beispiel aus dem Hochschulkontext wäre das An- und Abmelden von Modulen und Prüfungen.

Hier besteht die grösste Möglichkeit für Ergänzungen, zumal im Rahmen dieses Projektes nichts Derartiges eingeplant worden war, da sich unsere Ziele auf das Fundament des Chatbots, die Clients und die fachlichen und organisatorischen Abfragen beschränkten. Aus der Aufnahme weiterer Abfragen und Aktionen könnte allerdings ein enormer Mehrwert für die Studierenden entstehen, da viele verschiedene Informationsquellen hinter einer einzigen, einfach abfragbaren Quelle vereinigt würden. Hier kann das Potential der Interaktivität in beide Richtungen (Informationen erhalten, Anweisungen erteilen) ausgeschöpft werden.

Damit würde Nabu von einem reinen Wissensdatenbank-Bot in einen generelleren Hochschul-Hilfsbot umgewandelt. Damit geht allerdings einher, dass an Generalität eingebüsst wird, da es sehr unwahrscheinlich ist, dass die verwendeten APIs einer Hochschule deckungsgleich mit denen einer zweiten Universität sind. Dieser Tradeoff müsste in Kauf genommen werden. Somit würde die Entwicklung dieser Features sehr wahrscheinlich bei der jeweiligen Institution liegen.

Grafisches Librarian-Frontend

Momentan ist das Librarian-Frontend, welches die Erfassung der Learning-Units und zugehörigen Daten sowie das Senden an die Librarian-Datenbank erlaubt, als Konsolenapplikation gehalten. Zumal dieses Projekt im Rahmen eines Informatik-Studienganges erstellt wurde, steht zu erwarten, dass die unmittelbar beteiligten Personen über das notwendige technische Geschick verfügen, um mit dieser Applikation umzugehen.

Es ist allerdings davon auszugehen, dass dem durchschnittlichen Nutzer der Umgang mit der Kommandozeile nicht sonderlich vertraut ist, insbesondere abseits des Kontexts eines Informatik-Studienganges. Aus diesem Grund wurde dieses Ziel auch im Rahmen der Stretch Goals zu Beginn des Projektes erfasst. Sollte also ein Deployment in einem

weniger technischen Kontext interessant werden, so würde es sich lohnen, Ressourcen in die Erstellung eines grafischen Clients zu erstellen, der auch ohne Grundwissen über Python und die Kommandozeile bedient werden kann.

Im Idealfall würde es sich bei der neu implementierten Frontend-Anwendung sogar um eine Webapplikation handeln, da dann das mühselige Verteilen und Installieren auf allen Maschinen der Dozierenden wegfallen würde. Ausserdem sind sich die meisten Nutzer das Umgehen mit Webapplikationen über den Browser aus dem Alltag bereits gewohnt, was die Lernschwelle zur Nutzung weiter senkt.

Automatische und manuelle Synonyme

Zum Projektabschluss werden weder im Librarian-Backend noch im Chatbot selbst Synonyme für die mitgesandten Daten abgespeichert. Dies könnte man für Englisch mit der implementierten NLTK-Bibliothek von Python automatisieren, für andere Sprachen müsste ein separater Container aufgesetzt werden [45].

Zusätzlich besteht hier die Herausforderung, dass für viele Fachbegriffe keine oder inkorrekte Synonyme im natürlichen Sprachgebrauch und der inbegriffenen NLTK-Bibliothek zur Verfügung stehen. Dies müsste man mit einer manuellen Synonym-Erfassung lösen.

Weitere Dateiformate

Momentan werden zwei Dateiformate von der Librarian-CLI unterstützt: .pdf und .csv. Um jedoch mehr Daten aus verschiedenen Quellen erfassen zu können, wäre es von Vorteil, mehr Dateiformate zu unterstützen. Am naheliegendsten wären .docx (Word-Dateien) oder .pptx (PowerPoint-Dateien), da diese oft für Vorlesungen oder Übungen benutzt werden.

4.4.3 Konkretes Vorgehen zur Weiterentwicklung

Trainingsdaten, Interaktionsqualität

Um die Erfassung einer ausreichenden Menge an Trainingsdaten zu gewährleisten, wäre es ratsam, vor dem Deployen in eine Produktionsumgebung eine zweite Serie an Usability Tests durchzuführen. Diese hätten das explizite Ziel, Trainingsdaten für MindMeld zu erfassen und würden eine grössere Menge an Probanden (40-60) beinhalten. Idealerweise würde man direkt auf einen Vorlesungs-Jahrgang in einem Modul, in dem Nabu später eingesetzt werden soll, zugehen und die Daten dort sammeln. Zur Auswertung der Daten kann das Logfile des Nabu-Servers verwendet werden, in dem alle eingehenden Anfragen abgelegt werden.

Die so erhobenen Daten müssten dann nach Intents sortiert und in die korrekten Trainingsfiles eingepflegt werden. Diese finden sich im MindMeld-Modul des Nabu-Servers in den Domain-Unterordnern. Mit diesem Schritt sollte sich erneut eine merkliche Verbesserung der Interaktionsqualität einstellen, wie dies bereits nach den Usability Tests ersichtlich war.

HTTPS

Das konkrete Implementieren von HTTPS umfasst die folgenden Schritte:

1. Das Anlegen eines gültigen Zertifikats, vorzugsweise bei einer anerkannten Certificate Authority und nicht self-signed.
2. Das Laden dieses Zertifikats auf den fraglichen Server. Hierzu würde es sich anbieten, das Dockerfile des Servers so zu bearbeiten, dass ein Laden als Argument über Docker möglich ist, z.B. durch das Hinzufügen eines Volumes für Zertifikate. Dies erlaubt allen individuellen Deployments, ohne grossen Mehraufwand ihre eigenen Zertifikate zu nutzen.
3. Das Einbinden des Zertifikats in den Waitress-WSGI-Server, um das Anbieten von verschlüsselter Kommunikation zu ermöglichen. Die genaue Durchführung dieses Schritts hängt von Schritt 2 ab.

Diese Schritte wären für die Librarian-Backendkomponente (zum Verschlüsseln der Uploads von Daten aus dem Frontend) sowie für die Nabu-Chatbotkomponente (zum Verschlüsseln der Transfers aus dem Librarian sowie der Kommunikation mit den Nabu-Frontends) nötig. Der hierfür nötige Aufwand würde vergleichsweise niedrig ausfallen.

Upload-Autorisierung und Authentifizierung

Es gibt im Nabu-Projekt zwei Punkte, an denen die Autorisierung beim Upload von Daten eine Rolle spielt:

Als Erstes soll sichergestellt werden, dass die Datenbank nur mit Daten geladen wird, die auch enthalten sein sollen, sprich: Allfällige Librarian-Frontends müssen sich autorisieren, um Daten in die DB laden zu dürfen. Um dies auf einfache Art und Weise zu verhindern, könnte man in der Datenbank das Setzen eines Passworts implementieren, welches dann nur denjenigen Dozierenden mitgeteilt wird, welche das Recht haben sollen, Inhalte hochzuladen. Fehlt das korrekte Passwort bei einem Upload, so werden die Daten nicht in die Datenbank persistiert, sondern verworfen. Mit diesem System wäre allerdings eine feingranulare Nutzerverwaltung nicht möglich.

Zwecks Authentifizierung von individuellen Nutzern und Rollen wäre auch die Implementation oder Adaption eines kompletten User-Management-Systems mit individuellen Accounts und Passwörtern für die Datenbank denkbar. Damit würde allerdings ein erheblicher Mehraufwand einhergehen. Idealerweise liesse sich dieses System als Single-Sign-On in die bereits bestehende Struktur der OST Rapperswil-Jona integrieren. Für andere Einsatzumgebungen würde sich dazu beispielsweise OAuth [46] in Verbindung mit OATH [47] oder OpenID [48] nutzen.

Andererseits muss das Librarian-Backend sich beim Nabu-Chatbot autorisieren, wenn es ein Update der Datenbank in die Wissensbasis laden will. Ansonsten könnte ein Drittprogramm oder eine Drittperson sich jeweils als Datenbank ausgeben und Daten in den Chatbot injizieren. Da es sich beim Upload von der Datenbank zur Wissensbasis um einen automatisierten Prozess ohne grosse Menscheneinwirkung handelt, ist das Eingeben eines Passworts keine Option. Deshalb wäre eine mögliche Lösung für dieses Problem, sowohl im Chatbot als auch in des Librarian-Backends ein geteiltes Geheimnis zu hinterlegen, anhand dessen die «Identität» des Librarian-Backends bestimmt werden kann. Dazu würde sich beispielsweise eine Docker-Umgebungsvariable eignen. Nur, wenn dieses Geheimnis korrekt mit einem Datenupload zusammen übermittelt wird, werden die Daten im Chatbot akzeptiert und in die Wissensbasis geladen.

Abfrage-Autorisierung

Idealerweise könnte bei der Implementierung des Abfrage-Autorisierungsmoduls mit bereits bestehenden Identifikationssystemen, beispielsweise einem Hochschulaccount, zusammengearbeitet werden. Es müsste dann ein Modul implementiert werden, welches die Autorisierung an diesem System erlaubt. Im Falle von Moodle besteht eine entsprechende Struktur bereits.

Darüber hinaus müsste im Chatbot-Backend ein Modul erstellt werden, welches die Berechtigungen der einzelnen Studierenden kennt und für jede Abfrage einen Abgleich machen kann, ob die fragliche Person berechtigt ist, Antworten aus der jeweiligen Learning Unit abzurufen.

Zum Ausweisen der eigenen Identität beim Backend könnte ein Token verwendet werden, welches jeweils mit den Anfragen mitgeschickt wird und das Feststellen der Identität des Anfragers zweifelsfrei erlaubt.

Globaler pip-install

Um den Librarian-CLI-Client auf PyPI hochladen zu können, müsste dort ein Account erstellt werden. Dann müssten gemäss dem Upload-Guide von PyPI [49] die entsprechenden Schritte mit Twine [50] unternommen werden, um einen Upload und eine Verteilung via PyPI und pip zu ermöglichen.

Weitere Abfragen und Aktionen

Um neue Abfragen und Aktionen zu implementieren, müsste zunächst analysiert werden, wie darauf zugegriffen werden kann beziehungsweise was für eine API angesprochen werden soll. Je nach Komplexität dieser API müsste dann die notwendige Logik direkt im entsprechenden Answerer integriert oder ein neues Modul erstellt werden, das diese Abfragenlogik implementiert. Aufgrund der Vielfalt und Vielseitigkeit der Möglichkeiten müsste diese Entscheidung auf Fallbasis getroffen werden.

Gehören diese Aktionen oder Abfragen zu einer neuen Domain, so müsste diese auf dem MindMeld-Chatbot erstellt und entsprechend mit Trainingsdaten versorgt werden. Dazu gehört unter Umständen auch das Erstellen von neuen Entities, Persistern und Mappern, falls Daten permanent in der Wissensbasis hinterlegt werden müssen.

Grafisches Librarian-Frontend

Untenstehend sind die ersten Haupt-Entwürfe einer möglichen Librarian-Frontend-Benutzeroberfläche. Das Frontend ist als Web-Applikation gehalten. Zu diesen Entwürfen kommen natürlich zum Beispiel Bestätigungsseiten, die jedoch nicht explizit entworfen wurden.

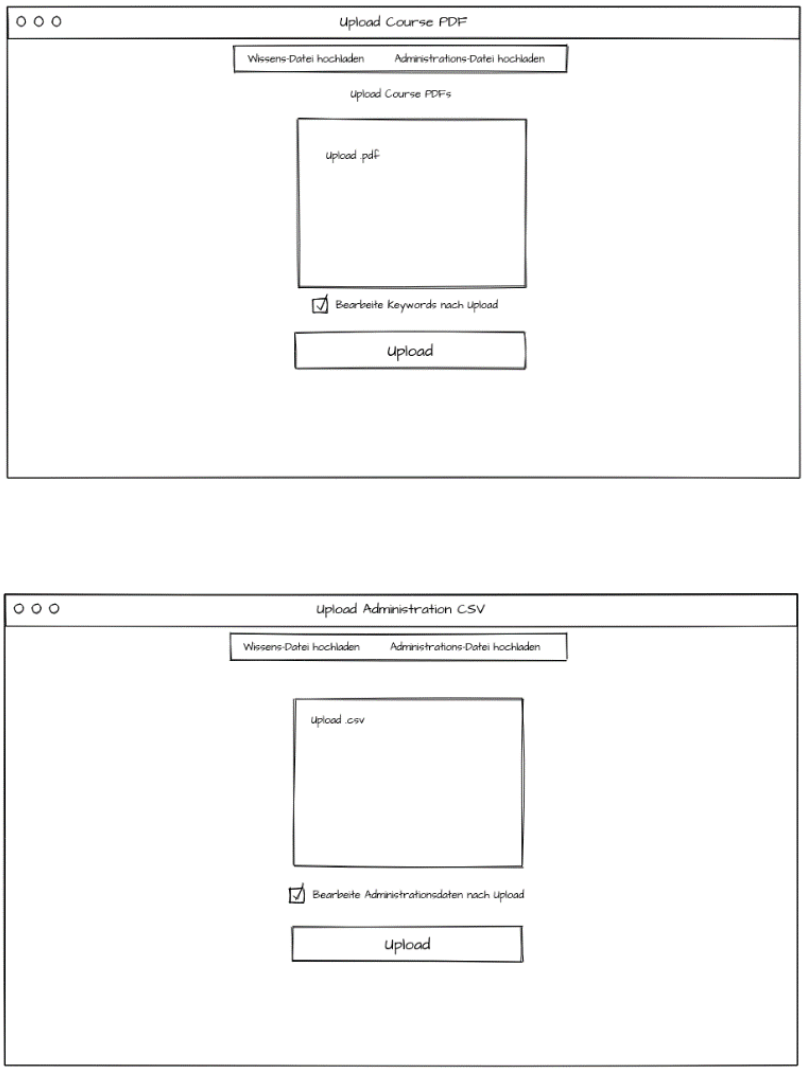


Abbildung 15: Grafisches Konzept Librarian-Upload

Obige Grafik zeigt den Entwurf der Upload-Seiten für die inhaltlichen und organisatorischen Daten. Es ist jeweils ein Upload-Feld dargestellt, in das man per Drag & Drop

oder per Mausklick eine Datei hochladen kann. Zusätzlich hat man die Option, nach dem Upload die Keywords oder die Events noch einmal zu bearbeiten.

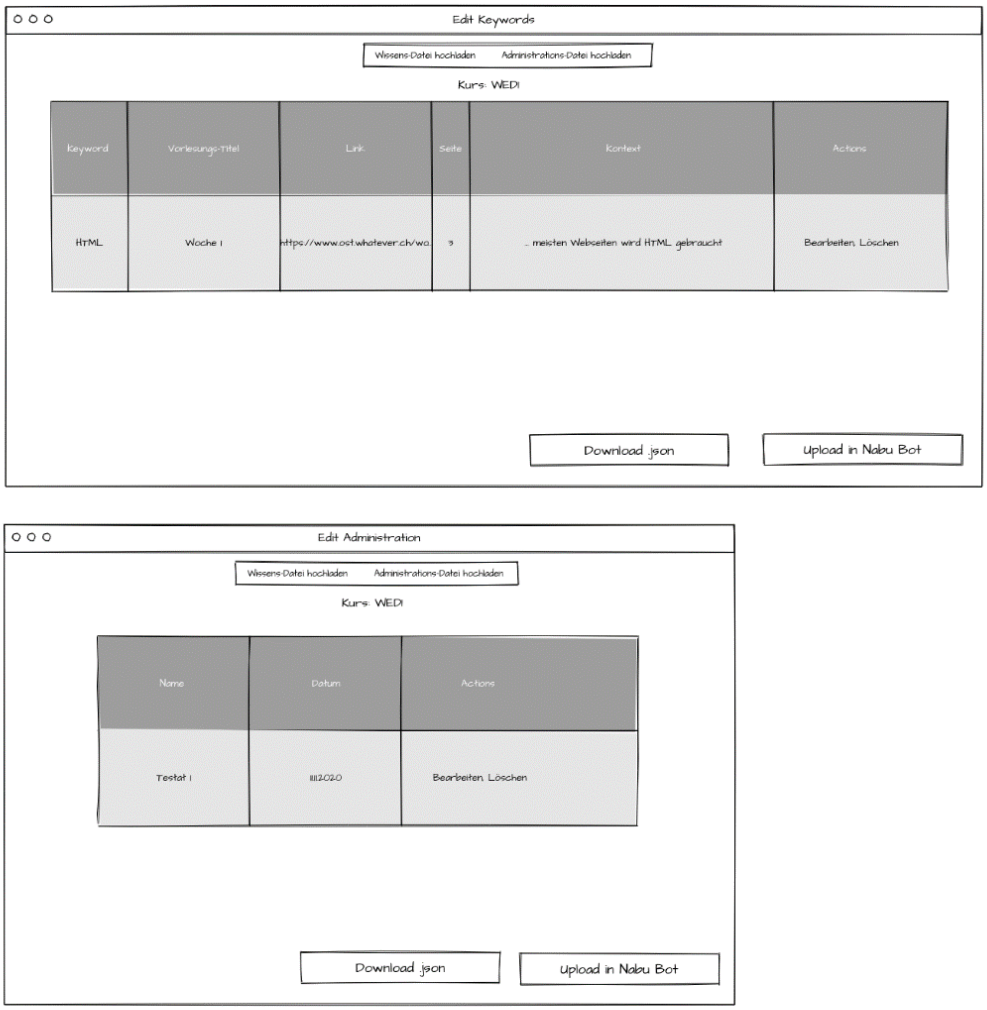


Abbildung 16: Grafisches Konzept Librarian-Edit

Die Daten werden in einer Tabelle dargestellt. Beim Klick auf Bearbeiten sollte sich ein Modal öffnen, in dem man die einzelnen Felder des Events oder des Keywords inklusive des Kontextes bearbeiten kann. Zusätzlich gäbe es die Option, einzelne Datenzeilen zu löschen. Für technisch versierte Benutzer wäre die Option vorhanden, die JSON-Datei herunterzuladen und direkt zu bearbeiten. Zusätzlich könnte eine Tabellenansicht für das manuelle Bearbeiten von Synonymen zur Verfügung gestellt werden.

Automatische und manuelle Synonyme

Wie bereits in Kapitel 4.4.2 erwähnt, ist es möglich automatische Synonyme generieren zu lassen. Diese sind jedoch oft nicht zielbringend, da zum Beispiel viele Fachbegriffe aus der Informatik nicht in einem normalen Wörterbuch vorhanden sind.

Idealerweise wäre es also möglich, eigene Synonyme direkt zu erfassen. Mit dem Librarian-CLI-Tool müsste man direkt das generierte JSON, welches normalerweise automatisch zum Librarian-Backend geschickt wird, bearbeiten und die Synonyme einfügen. Da dies auch für technisch versierte Benutzer eher mühselig ist, wird empfohlen dieses Feature erst zusammen mit dem grafischen Librarian-Frontend umzusetzen.

Weitere Dateiformate

Um ein neues Dateiformat zu unterstützen, muss als Erstes eine passende Verarbeitungs-Library evaluiert oder selbst geschrieben werden. Im zweiten Schritt wird ein Driver für das Librarian-Frontend geschrieben, welcher die ausgewählte Library anwendet und die Daten aus der Quelle extrahiert und gegebenenfalls weiterverarbeitet. Allenfalls müssen neue Click-Parameter für das CLI-Tool hinzugefügt werden. Ausserdem muss festgelegt werden, ob das neue Dateiformat für Daten von organisatorischer oder fachlicher Natur gilt.

4.5 Projektplanung (Soll)

Projektplan

Das hier geführte Kapitel «Projektplanung» ist deckungsgleich mit dem in der Elaborationsphase erstellten Projektplan, der während dem Projekt als separates Dokument gehalten wurde. Zu Projektende wurde er als Teil der Dokumentation in dieses Kapitel überführt.

4.5.1 Projektorganisation

Prozessmodell

Das für diese Arbeit verwendete Prozessmodell ist das in Daniel Kellers Vorlesung «Software Engineering 1» der HSR umrissene Scrum+-Framework [51]. Es handelt sich hierbei um eine Kombination aus reinem, agilem Scrum und dem Rational Unified Process (RUP).

Aus dem regulären Scrum übernommen wird dabei das Sprint-Konzept als abgesteckte Arbeitseinheit sowie der Product Backlog als Mittel, um den Stand des Projektes festzustellen sowie Aufgaben zu priorisieren.

Aus RUP stammt die Einteilung des Projektes in die vier Phasen Inception, Elaboration, Construction und Transition. Diese Kombination ermöglicht es, zum Grossteil agil zu arbeiten, aber trotzdem konkrete Meilensteine zu definieren und den Ablauf des Projektes besser zu lenken.

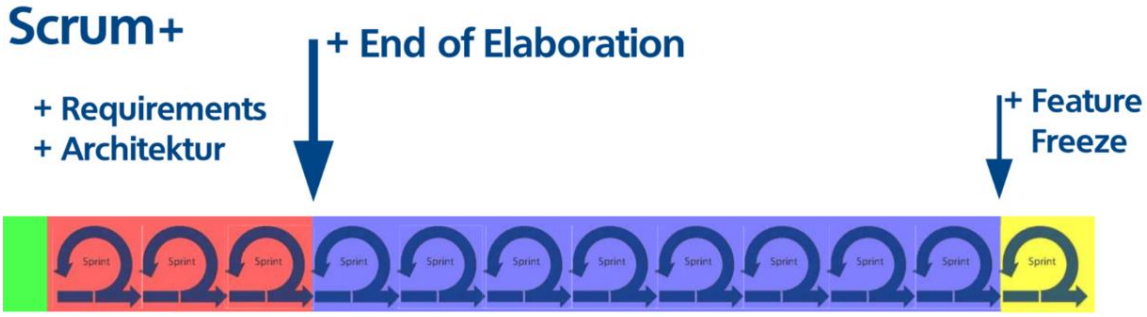


Abbildung 17: Scrum+-Prozessmodell [51]

Dokumentenhaltung

Sämtliche im Rahmen dieses Projektes entstehende Dokumente sind in einem geteilten Microsoft-OneDrive-Ordner für alle beteiligten Personen verfügbar. Die Versionierung der Dokumente erfolgt ebenfalls durch OneDrive, ebenso wie die kollaborative Bearbeitung, die zusätzlich von Microsoft Word sichergestellt wird. Dies garantiert, dass während der Bearbeitung stets alle Beteiligten auf dem neusten Stand sind.

Der Code der Software wird mittels dem Git-Versionskontrollsystem verwaltet und liegt auf einem privaten GitHub-Repository, auf welches wiederum alle beteiligten Personen Collaborator-Zugriff erhalten. In einem an dieses Repository angeschlossenen Wiki werden relevante Informationen zum Projekt (z.B. offene und geschlossene Fragen, Sitzungsprotokolle) gehalten.

Issue-Tracking

Mit dem GitHub-Repository verknüpft wird ein klassischer Issue-Tracker, der zur Nachverfolgung der Arbeitspakete des Projektes dient. Der Tracker wird in vier separate Boards unterteilt, eines pro Phase der Arbeit (Inception, Elaboration, Construction, Transition). Diese Boards sind wiederum in die vier Zustände «not started», «in progress», «pending review» und «done» unterteilt, nach denen die Arbeitspakete sortiert werden.

4.5.2 Involvierte Personen

Autoren

Das Autorenteam besteht aus drei Personen: Alexandra Diener und Julia Fritsche, die diese Arbeit als Studienarbeit schreiben, sowie Christoph Streiff, für den diese Arbeit eine Bachelorarbeit darstellt.

Betreuer/Auftraggeber

Betreuer der Arbeit ist Stefan Keller, Professor für Informatik und Institutsleiter IFS an der OST Rapperswil. Er steht den Autoren mit Rat und Tat zur Seite.

Bei ihm handelt es sich für die Zwecke des Projektmanagements auch um den Auftraggeber der Arbeit, zumal nicht mit externen Kontaktpersonen oder Unternehmen zusammengearbeitet wird. Dies bedeutet, dass er in Erarbeitung von Vision, Scope und Zielsetzung massgeblich involviert ist.

4.5.3 Team und Rollen

Teamaufbau

Grundsätzlich herrscht Gleichstellung zwischen allen Teammitgliedern und alle besitzen die gleiche Entscheidungsgewalt. Sollten sich allerdings Unstimmigkeiten einstellen, so ist es Verantwortung des Projektleiters, diese aufzulösen und den weiteren Verlauf des Streitpunktes festzulegen.

Die Vision des Projektes, grundlegende Anforderungen und Designentscheide werden kollektiv bestimmt und darüber mit dem Betreuer Rücksprache gehalten. Ansonsten gilt die untenstehende Rollenverteilung.

Rollenverteilung

- **Christoph Streiff**:
  Nimmt die Rolle des Projektleiters ein und ist für Organisation und Überblick über das Projekt zuständig. Fungiert auch als primäre Kommunikationsschnittstelle zwischen Team und Betreuer und ist Ansprechsperson für Administratives. Als Entwickler ist er in erster Linie für das Backend des Chatbots selbst zuständig.
- **Alexandra Diener**:
  Verantwortlich für Continuous Integration und DevOps. Hauptaufgabe in der Entwickler-Rolle ist ebenfalls das Backend. Nimmt darüber hinaus die Rolle des Systemadministrators ein und ist damit verantwortlich für das Aufsetzen von Servern und CI-Tools.
- **Julia Fritsche**:
  Ist in erster Linie Entwickler und bedingt durch die Erfahrung in PHP primär für das Moodle-Frontend und die Benutzeroberfläche zuständig. Im Backend ist sie für die Architektur und Betreuung der API zuständig.

4.5.4 Besprechungen und Meetings

Teammeeting

Alle Projektmitarbeitenden treffen sich mindestens ein Mal pro Woche. Dieses Meeting findet üblicherweise am Mittwochmorgen statt und nimmt neben der Klärung von administrativen und fachlichen Fragen auch die Rolle des Scrum-Meetings ein (Sprintplanung/Review/Retrospektive).

Falls nötig, können nach Absprache weitere Meetings (z.B. zur Besprechung von konkreten Problemstellungen) stattfinden, die auch remote abgehalten werden können.

Betreuermeeting

Auf Wunsch des Betreuers findet ein Mal pro Woche, normalerweise am Mittwochnachmittag um 13:00, ein Meeting des Projektteams mit dem Betreuer statt, wo über Fortschritt, nächste Ziele und Probleme informiert wird.

Zum Zweck der Vorbereitung auf diese Sitzung erfolgt bis spätestens Montag ein Update über Stand der Dinge und neue Traktanden per Mail an den Betreuer.

4.5.5 Abgabetermine

Terminsetzung

In Absprache mit dem Betreuer wurde vereinbart, dass die Terminsetzung für Studienarbeiten für dieses Projekt keine Gültigkeit hat. Alle Autoren, explizit auch die, die eine Studienarbeit schreiben, richten sich nach den Abgabeterminen für Bachelorarbeiten.

Abstract-Abgabetermin

Die finale Version des Abstracts muss bis zum 04.01.2021 im dazu vorgesehenen Online-Tool erfasst werden.

Projekt-Abgabetermin

Die Abgabe des Projektes an den Betreuer und das Hochladen aller Dokumente in das Online-Tool erfolgt bis zum 08.01.2021 um 17:00 Uhr.

4.5.6 Zeitmanagement

Zeitraum

Die Unterrichtszeit des Herbstsemesters 2020/2021 dauert vom 14.09.2020 bis zum 18.12.2020, ist also 14 Wochen lang. Mit den hinzukommenden drei Wochen bis zum endgültigen Abgabetermin beträgt der Gesamtzeitraum der Arbeit 17 Wochen.

Zeitbudget

Bei der Zeitbudgetierung muss zwischen dem Budget für Studien- und Bachelorarbeiten unterschieden werden. Für abgeschlossene Studienarbeiten werden 8 ECTS-Punkte abgerechnet, während für Bachelorarbeiten 12 ECTS-Punkte gutgeschrieben werden.

Pro ETCS-Punkt kann mit einer Arbeitsmenge von 25-30 Stunden gerechnet werden. Dadurch entsteht das folgende Zeitbudget:

	Diener (SA)	Fritsche (SA)	Streiff (BA)	Total
ECTS	8	8	12	28
Stunden total	200 - 240	200 - 240	300 - 360	700 - 840
Stunden / Woche	12 - 14	12 - 14	18 - 21	42 – 49

Tabelle 3: Zeitbudget

Zeiterfassung

Für die Erfassung der am Projekt verrichteten Arbeitszeit wird die Webapplikation Clockify [52] verwendet, mit der alle Autoren bereits Erfahrung haben.

Es handelt sich dabei um ein gratis verfügbares, recht einfach gehaltenes Online-Tool zur Zeiterfassung, welches neben der individuellen Beschriftung eines Tasks auch die Zuweisung von Tags unterstützt, um Aktivitäten zu gruppieren. Für dieses Projekt werden dazu die folgenden sieben Tag-Kategorien definiert:

- **Teamsitzung**:
  Sitzungen, die ohne Beisein des Betreuers stattfinden, was auch Remote-Sitzungen beinhalten kann.
- **Betreuersitzung**:
  Sitzungen, die gemeinsam mit dem Betreuer stattfinden, üblicherweise ein Mal pro Woche und persönlich vor Ort.

- **Setup**: Vorbereitende und administrative Aufgaben wie z.B. das Einrichten eines Repositories, das Aufsetzen des Fileshares oder das Aufbauen der Toolchain.
- **Recherche**: Sammlung und Analyse von benötigten Daten, beispielsweise eine Evaluierung von zu verwendenden Technologien.
- **Dokumentation**: Aufbau und Erweiterung der Projektdokumentation, darunter fallen auch Projekplan, Zwischenpräsentation und ähnliche Dokumente.
- **Implementation**: Arbeit am Code der Projektsoftware, worin auch Refactorings und Bugfixes enthalten sind.
- **Testing**: Durchführen von automatischen oder manuellen Tests der Software, insbesondere auch Usability Tests.

Um runde Zahlen zu erhalten und somit die Abrechnung am Ende des Projekts zu vereinfachen, wird die Arbeitszeit in der Clockify-Erfassung jeweils auf die nächsten 15 Minuten gerundet.

4.5.7 Meilensteine

Zielsetzung

Die untenstehende Tabelle verschafft einen Überblick über die angestrebten Meilensteine des Projektes. Massgeblich für den Meilenstein ist jeweils die wöchentliche Sitzung am Mittwoch.

SW	KW	Meilenstein	Zielsetzung
1	38	**M0: Kickoff**	Das Projekt wird gestartet. Eine erste Sitzung mit dem Betreuer erfolgt, wo Ziele und Vision diskutiert werden.
2	39	**M1: End of Inception**	Die Inception-Phase ist beendet. Es besteht eine grobe Idee zu Scope und Zielsetzung des Projekts.
3	40	**M2: Projektplan**	Eine erste vollständige Version des Projektplans wird mit dem Betreuer besprochen.
7	44	**M3: End of Elaboration**	Die Elaboration-Phase ist beendet. Die Arbeitsumgebung ist eingerichtet, die Planung von Architektur und Dokumentation weitgehend abgeschlossen. Es besteht ein funktionsfähiger Proof of Concept.
10	47	**M4: Zwischenpräsentation**	Die Zwischenpräsentation Mitte Semester wird gehalten. Fortschritte in Projektdokumentation und Umsetzung werden dem Betreuer präsentiert.
12	49	**M5: Feature Freeze**	Es werden keine neuen Features mehr zum Produkt hinzugefügt. Arbeiten am Code beschränken sich ab diesem Punkt auf Refactorings und Bugfixes.
15	52	**M6: End of Construction**	Die Construction-Phase ist beendet. Ab diesem Punkt gilt ein Code Freeze. Die Arbeiten am Programm selbst sind abgeschlossen und es werden keine Veränderungen mehr vorgenommen.
17	2	**M7: Projektende**	Das Projekt ist abgeschlossen. Alle Abgaben sind gemäss Vorgaben der OST eingereicht.

Tabelle 4: Projektmeilensteine

4.5.8 Iterationen

Sprints

Im Sinne des verwendeten Scrum+-Modells wird der Projektablauf in Sprints aufgeteilt. Während der Evaluation-Phase haben diese eine Dauer von einer Woche, um hohe Flexibilität bei der Evaluation der zu verwendenden Technologien und dem Aufsetzen der Entwicklungsumgebung zu gewährleisten. Mit Beginn der Construction-Phase, wenn die grobe Richtung des Projektes klar ist, wird die Sprintdauer auf zwei Wochen erhöht, damit der organisatorische Zusatzaufwand verringert werden kann.

Gantt-Diagramm

Dem folgenden Gantt-Diagramm ist die Scrum+-gemässe Phasenaufteilung des Projekts mit Meilensteinen einerseits und die ungefähr geplante Einteilung der Sprint-Zeit andererseits zu entnehmen.

Abbildung 18: Gantt-Diagramm

4.5.9 Risikomanagement

Risikoanalyse

In der Elaborationsphase wurden die folgenden Risiken erarbeitet, wobei eine Unterteilung in projektspezifische und nicht projektspezifische Risiken gemacht wurden:

Nr.	Risiko	Beschreibung	Schaden [h]	Eintritts-W'keit	Gewichteter Schaden [h]
1	Neue Technologien (Python, PHP)	Bis jetzt hat nur eine Person aus dem Team Erfahrung PHP, auf Python kennen alle Teammitglieder nur die Grundlagen. Deshalb wird eine gewisse Einarbeitungszeit benötigt, bei der trotzdem ein Risiko besteht, sie zu überschreiten.	24	50%	12
2	Chatbot Framework + Machine Learning	Probleme bei der Verwendung und Weiterentwicklung des Chatbot Frameworks	50	50%	25

Tabelle 5: Projektspezifische Risiken

Nr.	Risiko	Beschreibung	Schaden [h]	Eintritts-W'keit	Gewichteter Schaden [h]
1	Hardware-Schaden	Mit unvorhersehbaren Hardware-Schäden muss gerechnet werden.	10	10%	1
2	Fehleinschätzungen des Arbeitsaufwandes von Arbeitspaketen	Der Arbeitsaufwand von Arbeitspaketen wird zu niedrig eingeschätzt, womit mehr Zeit benötigt wird	60	50%	30
3	schlechte Kommunikation	Durch eine schlechte Kommunikation (wenige Updates, Missverständnisse) kann es zu Verzögerungen oder Fehlimplementationen kommen	24	50%	12

4	Fehleinschätzung des Scopes	Wird der Scope zu weit gesteckt, geht der Fokus aufs Essentielle und damit wichtige Zeit verloren	10	30%	3
5	Architektur	Fehlerhafte Architekturentscheide in der Elaborationsphase	20	50%	10
6	Krankheit, Unfall	Besonders in der Corona-Zeit müssen Krankheiten als Risiko definiert werden.	240	5%	12

Tabelle 6: Nicht projektspezifische Risiken

Pufferzeit

Basierend auf dem totalen gewichteten Schaden von 105h wurden ca. 2 Wochen als Pufferzeit eingeplant.
Eine Woche planen wir vor der Zwischenpräsentation für allfällige Probleme mit dem Prototyp ein. Vor dem Code Freeze wird die zweite Woche eingeplant.

Massnahmen

Basierend auf den festgestellten Risiken wurden die folgenden vorbeugenden Massnahmen sowie Massnahmen zur Schadensbegrenzung im Fall eines Eintretens des Risikos definiert.

Risiko-nummer	Vorbeugende Massnahmen	Schadensbegrenzung nach Eintritt
1	Genug Zeit für die Einarbeitung einplanen.	Wissenslücken ausbessern.
2	In der Elaboration-Phase werden mehrere Frameworks evaluiert, um eine Vorauswahl zu erstellen. Der Entscheid für ein Framework fällt möglichst früh, um genügend Einarbeitungszeit zu haben. Auch die Architektur als Ganzes könnte von der Wahl des Chatbot-Frameworks beeinflusst sein.	Bessere Einarbeitung ins Framework und Recherche. Im Worst Case Wechsel des Chatbot-Frameworks.

Tabelle 7: Projektspezifische Massnahmen zur Risikominimierung

Risiko-nummer	Vorbeugende Massnahmen	Schadensbegrenzung nach Eintritt
1	Ersatzgeräte werden frühzeitig bereitgestellt.	Ersatzgerät wird direkt neu aufgesetzt.
2	Bei der Schätzung der Tickets sollte das Schreiben von Tests sowie die Komplexität des Tasks mitbeachtet und geschätzt werden.	Allfällige zusätzliche Features nicht umsetzen und den Fokus neu auslegen.

3	Es findet jeden Mittwochmorgen ein Meeting vor Ort statt. Den Rest der Woche wird nach Bedarf über Telegram oder Discord kommuniziert.	Allfällige weitere Statusmeetings und Klärungsrunden einleiten.
4	Es wird mit einer MVP Planung gearbeitet, die genügend Puffer offen lässt, falls es Verzögerungen gibt. Gleichzeitig wird genügend Arbeit vorbereitet, um im Idealfall noch weitere Features in der gleichen Qualität einbauen zu können.	Die angefangene Arbeit qualitativ fertigstellen und Features aus dem Scope entfernen.
5	Möglichst einfache Architekturentscheide fällen, um kein Overengineering zu provozieren. Die Architektur sollte einfach erweiterbar sein.	Architektur anpassen und gegenfertig vereinfachen.
6	Es werden die Hygienemassnahmen bezüglich Corona eingehalten und Masken in Meetingräumen getragen. Sobald sich jemand krank fühlt, bleibt derjenige zuhause.	Kurzfristig verschiebt man Tasks in die Zukunft oder delegiert sie an andere Teammitglieder. Bei einem langfristigen Ausfall wird nach Absprache mit dem Auftraggeber der Scope des Projektes angepasst.

Tabelle 8: Nicht projektspezifische Massnahmen zur Risikominimierung

Weiterhin werden folgende Massnahmen zur Risikovorbeugung und Schadensverminderung getroffen:

- Obligatorische Codereviews, Unit Tests sowie allgemeine Sensibilisierung auf gute Codequalität
- Pflege einer offenen Kommunikation und ehrlichen Fehlerkultur, um allfällige Risiken und Probleme möglichst früh zu erkennen.
- “Keep it simple”-Ansatz mit Modularität und simpler Architektur im Fokus.

4.5.10 Qualitätsmanagement

Review-Prinzip

Zur grundlegenden Qualitätssicherung wird in diesem Projekt das Vier-Augen-Prinzip verwendet. Insbesondere während der wöchentlichen Teamsitzungen am Mittwochmorgen erhalten alle Beteiligten die Gelegenheit, ihre geleistete Arbeit dem Team zu präsentieren und sie im Plenum zu besprechen. Falls nötig, kann ein solches Review auch remote stattfinden.

Ein Dokument gilt nicht als fertiggestellt, bis es nicht von mindestens einer Person neben dem Autor reviewt wurde. Selbiges gilt auch für verfassten Code (siehe «Code-Qualität»). Dies dient dazu, dass Dokumente stets einen Teamkonsens und nicht eine Einzelmeinung repräsentieren.

Coding Style Guidelines

Für dieses Projekt werden die folgenden Code-Style-Richtlinien eingesetzt:

- **Python**: PEP 8 Python Style Guide [53]
- **PHP**: PSR-12 Extended Coding Style Guide [54]
- **JavaScript**: Google JavaScript Style Guide [55]

Code-Qualität

Im Rahmen der Code-Qualitätssicherung werden von den Entwicklern regelmässig Code-Reviews durchgeführt, insbesondere bei Fertigstellung eines signifikanten Features. Im Idealfall geschehen diese Reviews zu zweit an einer Maschine, doch sollte dies nicht möglich sein, ist auch ein eigenständiges Review mit späterer Rückmeldung möglich. Zu einem solchen Code-Review gehört eine Prüfung des Codes auf Funktionstüchtigkeit und Erfüllung der Style- und Qualitätsrichtlinien sowie das Ausführen und Überprüfen der bestehenden Unit Tests.

Fehlgeschlagene Tests werden dabei niemals ignoriert, sondern das Problem schnellstmöglich lokalisiert und behoben, bevor weiter am Code gearbeitet wird. Die Qualität der bestehenden Codebase wird über neue Features priorisiert, um das Ansammeln von Technical Debt zu vermeiden.

Zur Gewährleistung von Sicherheit und Qualität kommen statische Code-Analysetools (z.B. SonarQube) zum Einsatz.

Im Ermessen der Entwickler kann für komplexe oder schwierig zu implementierende Features auf die Techniken des Test-Driven Development (TDD) oder des Pair Programming zurückgegriffen werden.

Pull Requests

Der Code liegt in einem GitHub-Repository, was bedeutet, dass verschiedene parallele Branches existieren, in denen neue Features implementiert oder Bugs gefixt werden.

Der Master wird als geschützter Branch eingerichtet, sodass jeglicher Code, der in den Master gemerged werden soll, zunächst ein Code-Review nach dem oben erklärten Prinzip bestehen muss. Dazu wird vom Autor des fraglichen Codes ein Pull Request eröffnet, der dann von einer anderen Person bearbeitet, reviewt und – im Akzeptanzfall – gemerged und geschlossen wird. Sollte der Code das Review nicht bestehen, so bleibt der Pull Request offen, bis die notwendigen Änderungen getätigt wurden.

Testing

Im Rahmen einer umfassenden Qualitätssicherung wird spätestens ab Beginn der Construction-Phase gründliches Testing durchgeführt. Jeder Developer ist selbst für die ausreichende Abdeckung seines eigenen Codes durch Unit-Tests verantwortlich. Zielwert ist hierbei eine Abdeckung von 60%, besser sind aber 80% oder höher. Dabei soll der gesunde Menschenverstand eingesetzt werden – es müssen keine Tests geschrieben werden, wo sie nicht sinnvoll sind. Für maschinell schwierig zu testende Features (wie z.B. eine Benutzeroberfläche) werden stattdessen manuelle Tests durchgeführt.

Wo möglich und sinnvoll, werden mit nicht am Projekt beteiligten Personen Usability Tests durchgeführt, um durch das Band eine gute Usability und User Experience zu gewährleisten. Zu diesem Zweck werden Usability-Testprotokollvorlagen erstellt und jeder Test individuell protokolliert.

4.5.11 Hardware

Entwicklungsgeräte

Als Hardware für das Projekt werden in erster Linie Geräte benötigt, auf denen entwickelt werden kann. Zu diesem Zweck bringen alle Projektmitarbeitenden private Hardware ein (z.B. einen Laptop), für deren Instandhaltung und Funktionstauglichkeit sie selbst verantwortlich sind. Spezialisierte Hardware ist nicht vonnöten.

Serverbeantragung

Sollte sich während der Entwicklung des Projektes herausstellen, dass ein Service über das Internet bereitgestellt werden muss, so kann dieser gegebenenfalls zunächst auf den privaten Geräten der Projektmitarbeitenden aufgesetzt und so evaluiert werden.

Sollte dann eine Entscheidung zur längerfristigen Bereitstellung gefällt werden, wird ein virtueller Server von der OST beantragt, auf welchem dieser Service verfügbar gemacht werden kann.

Unter diesen Punkt fallen auch entwicklungsrelevante Tools wie z.B. eine CI-Umgebung, nicht nur im Rahmen des Produkts selbst anfallende Services.

4.5.12 Software

Entwicklungsumgebung Zur Entwicklung der Software werden von allen Entwicklern in erster Linie die Entwicklungsumgebungen von JetBrains [56] (PyCharm für Python, PHPStorm/WebStorm für PHP und JavaScript) verwendet, da alle Beteiligten bereits damit gearbeitet haben und sich die Tools durch eine sehr kompatible Benutzeroberfläche sowie einfaches Setup auszeichnen.

Continuous Integration Als Continuous-Integration-Tool wurde Jenkins [57] bestimmt, eines der weitverbreitetsten CI-Tools überhaupt. Grund dafür ist die Tatsache, dass das Tool Open Source ist und guten Plugin-Support sowie statische Code-Analyse anbietet.

4.6 Projektmonitoring (Ist)

4.6.1 Arbeitsverteilung

Rollen

Die in Kapitel 4.5.3 definierte Rollenverteilung wurde zum Grossteil eingehalten:

- **Christoph Streiff** befasste sich neben diversen administrativen Aufgaben vor allem mit dem Implementieren des Nabu-Backends und dem MindMeld-Chatbot. Darüber hinaus erstellte er aufgrund seines grösseren Zeitbudgets einen Grossteil der Dokumentation, um den anderen Autorinnen mehr Arbeit am Code zu ermöglichen, und arbeitete an einigen kleinen Bugfixes und Refactorings im Telegram-Frontend.
- **Alexandra Diener** arbeitete primär als Systemadministratorin und setzte die gesamte CI-Toolchain sowie den Projektserver auf. Von ihr stammt auch das CLI-Frontend für den Librarian sowie die zugehörige Dokumentation, was die einzige Abweichung von der definierten Rollenverteilung darstellt.
- **Julia Fritsche** arbeitete primär an den Nabu-Frontends für Moodle und Telegram sowie dem Librarian-Backend und der dazugehörigen Datenbank inklusive der Dokumentation für diese Komponenten. Mit Alexandra zusammen entwarf sie die Schnittstellen zwischen den Librarian-Komponenten. Ausserdem half sie bei der Entwicklung der Librarian-CLI.

4.6.2 Soll-Ist-Zeitvergleich

Aufgewendete Zeit

Der folgenden Tabelle ist die totale aufgewendete Zeit für das Nabu-Projekt zu entnehmen. Die Sollzeit entspricht dabei derjenigen aus Tabelle 3.

	Zeit Soll [h]	Zeit Ist [h:m]
Julia Fritsche	200 - 240	216:00
Alexandra Diener	200 - 240	218:15
Christoph Streiff	300 - 360	331:30
Total	**700 - 840**	**765:45**

Tabelle 9: Ist-Zeitabrechnung

Wie ersichtlich wird, bewegt sich die tatsächlich aufgewendete Zeit überall um den Durchschnitt der Sollzeit. Damit wurde der in der Elaborationsphase (siehe Kapitel 4.5.6) erstellte Zeitplan gut eingehalten.

Zeiterfassung

Die folgenden Abbildungen zeigen, wie sich die erfassten Stunden pro Person über die Dauer des gesamten Projekts verteilen.

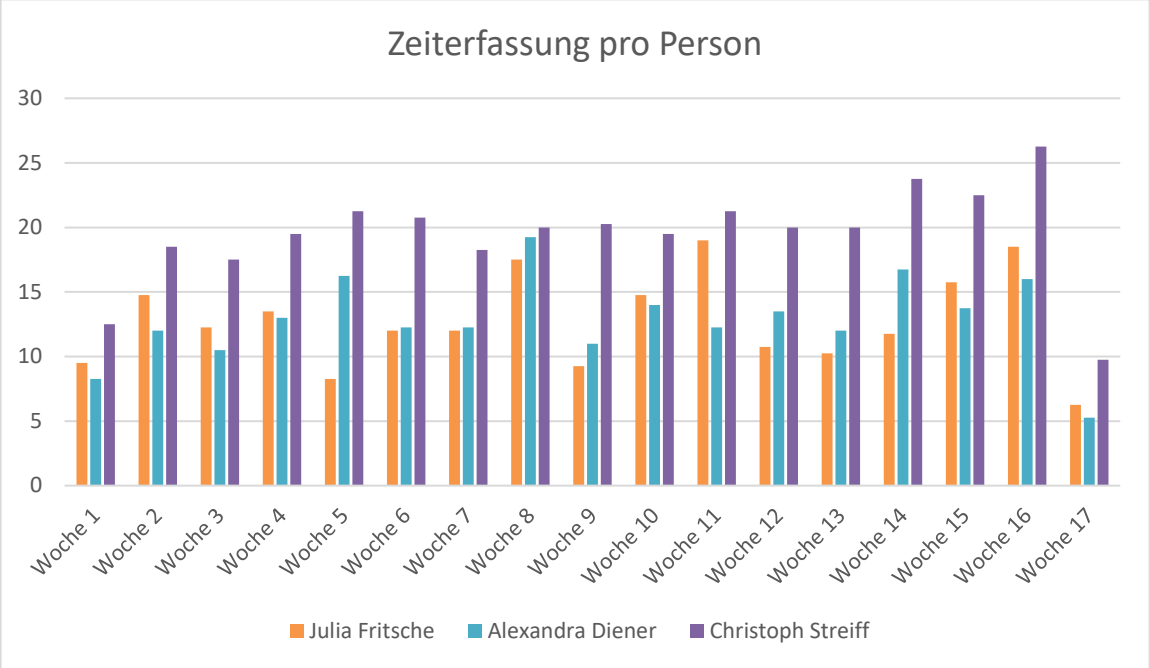


Abbildung 19: Balkendiagramm Zeiterfassung pro Person

Zeit pro Kategorie

Dem nachfolgenden Kreisdiagramm ist die Verteilung der total aufgewendeten Zeit auf die sieben definierten Kategorien abzulesen. Dabei ist anzumerken, dass unter der Kategorie Teamsitzung auch Pair Programming und gemeinsames Editieren der Dokumentation erfasst wurde. Damit lässt sich die scheinbar unverhältnismässige Grösse dieser Kategorie begründen.

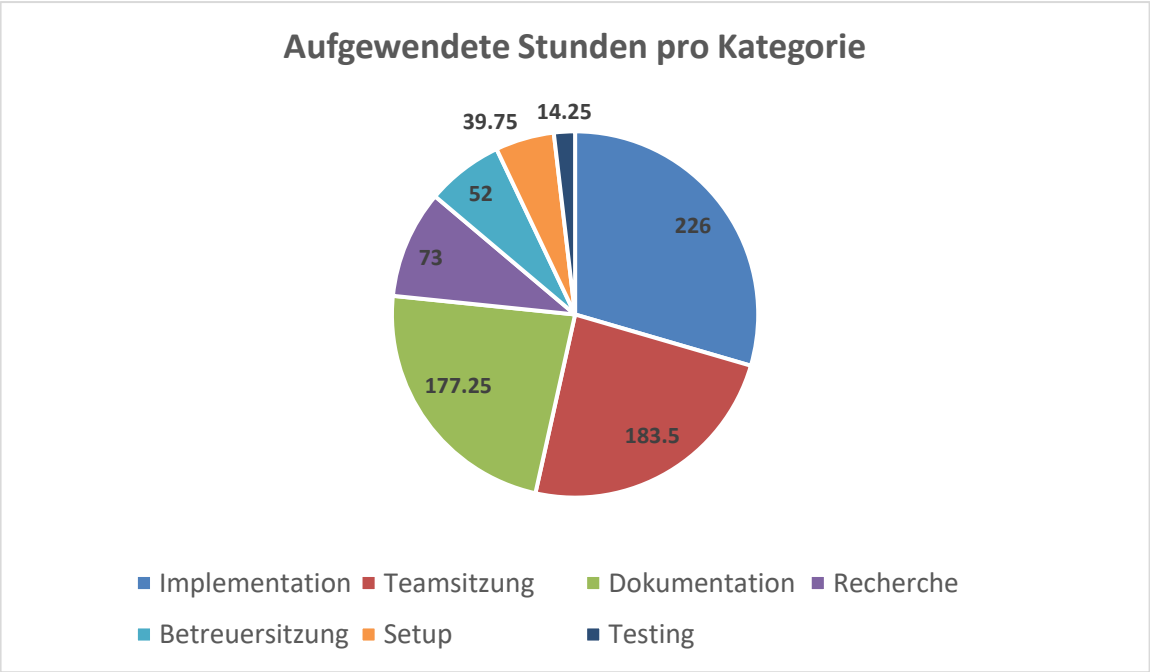


Abbildung 20: Kreisdiagramm Aufgewendete Stunden pro Kategorie

4.6.3 Meilenstein-Einhaltung

Auflistung

Die untenstehende Tabelle erlaubt einen Überblick über die im Projektplan in Kapitel 4.5.7 definierten Meilensteine sowie deren Einhaltung während dem Verlauf des Projektes.

SW	KW	Meilenstein	Eingehalten?
1	38	**M0: Kickoff**	Meilenstein wurde eingehalten.
2	39	**M1: End of Inception**	Meilenstein wurde eingehalten.
3	40	**M2: Projektplan**	Meilenstein wurde eingehalten.
7	44	**M3: End of Elaboration**	Meilenstein wurde teilweise eingehalten. Anpassungen an Toolchain, CI-Setup und Testumgebung waren teilweise auch später noch notwendig.
10	47	**M4: Zwischenpräsentation**	Meilenstein wurde eingehalten.
12	49	**M5: Feature Freeze**	Meilenstein wurde eingehalten.
15	52	**M6: End of Construction**	Meilenstein wurde eingehalten.
17	2	**M7: Projektende**	Meilenstein wurde eingehalten.

Tabelle 10: Meilenstein-Einhaltung

Auswertung

Es ist ersichtlich, dass 7 der 8 Meilensteine ohne Vorbehalte eingehalten werden konnten, bei M3: End of Elaboration liegt allerdings nur eine teilweise Einhaltung vor. Grund dafür waren Veränderungen im Code und der Struktur des Projekts und das Einbinden von neuen Libraries, was dazu führte, dass die Toolchain und die Continuous-Integration-Pipeline auch nach dem End-of-Elaboration-Meilenstein weiter angepasst werden mussten.

4.6.4 Risiken

Eingetretene Risiken

Pauschal lässt sich zum Eintreten der in Kapitel 4.5.9 definierten Risiken sagen, dass die nicht projektspezifischen Risiken nicht oder nur sehr geringfügig eingetreten sind.

Allenfalls Risiko Nummer 5 (Architektur) kostete während der Construction-Phase einige Entwicklungsstunden. Doch auch dabei handelte es sich nie um gravierende Fehlentscheide, die den Erfolg des Projektes massgeblich gefährdet hätten, sondern eher um Fleissarbeit.

Die projektspezifischen Risiken hingegen wurden während der Risikoanalyse tendenziell unterschätzt. Risiko 1 (Neue Technologien) schlug sich konkret in vielen Stunden des Refactorings nieder, da das saubere Umsetzen der Architektur in Python mangels Erfahrung eher schwierig war.

Risiko 2 (Chatbot Framework + Machine Learning) erwies sich als das folgenreichste der analysierten Risiken. Während sich das Aufsetzen des Proof-of-Concept-Chatbots als nicht allzu aufwändig herausstellte, stellte sich bei der Implementation des Nabu-

Chatbots ein massiver Mehraufwand ein, der insbesondere aus dem Finetuning der Trainingsdaten und der Chatbot-Konfiguration sowie der Interaktion zwischen Server-Schnittstelle und MindMeld herrührte. Mit ein Grund dafür war, dass die Dokumentation von MindMeld zu diesem letzten Thema eher spärlich ausfällt und somit viel Eigenerarbeitung notwendig war.

Tabellarischer Überblick

Der nachfolgenden Tabelle ist ein Überblick über die tatsächlich nennenswert eingetretenen Risiken und eine Gegenüberstellung des erwarteten und des eingetretenen Schadens zu entnehmen. Die Nummernpräfixe stehen für projektspezifische beziehungsweise nicht projektspezifische Risiken.

Nr.	Risiko	Erwarteter Schaden [h]	Eingetretener geschätzter Schaden [h]
PS1	Neue Technologien (Python, PHP)	12	35
PS2	Chatbot Framework + Machine Learning	25	50
NPS5	Architektur	10	20

Tabelle 11: Eingetretene Risiken

Konsequenzen

Als Konsequenzen der Unterschätzung der projektspezifischen Risiken mussten gewisse Abstriche bei den nichtfunktionalen Anforderungen gemacht werden. Zu Details siehe Kapitel 4.4.1.

4.6.5 Codestatistik

Erfassung

Die Codestatistik wurde direkt nach dem Code Freeze erstellt und reflektiert den finalen Stand der Code Base zum Ende der Construction-Phase.

Zeilen Code

Der untenstehenden Tabelle ist eine Analyse der Code Base zu entnehmen. Dabei wird sowohl die totale Anzahl Zeilen als auch die reine Codezeilenzahl ohne Leerzeilen und Kommentare aufgeführt.

Nicht in dieser Statistik eingerechnet sind Files, die keinen Code enthalten, aber trotzdem eine Rolle spielen, wie beispielsweise YAML-Konfigurationsfiles der CI-Pipeline oder die Trainings-Textfiles des MindMeld-Chatbots.

Komponente	Sprache	Zeilen total	Zeilen reiner Code
Librarian-CLI	Python	724	544
Librarian-Backend	Python	399	304
MindMeld-Backend	Python	2085	1651
Moodle-Client	PHP	216	73
Telegram-Client	Python	102	75
Total		**3526**	**2647**

Tabelle 12: Codezeilenstatistik

Kreisdiagramm

Das folgende Kreisdiagramm verschafft einen Überblick über die Verteilung des Codes auf die verschiedenen Komponenten des Projektes.

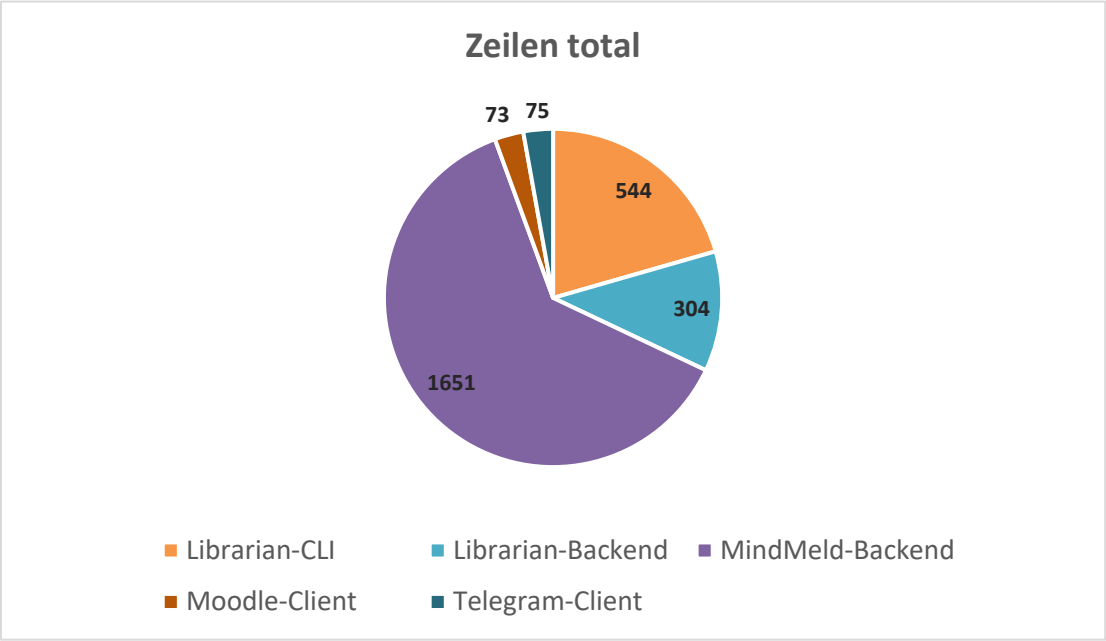


Abbildung 21: Kreisdiagramm Zeilen Code

5 Verzeichnisse

5.1 Glossar und Abkürzungsverzeichnis

Glossar

Das folgende Glossar dient zum Etablieren eines an die Problemstellung angepassten Wortschatzes, um die Kommunikation zu vereinfachen, Klarheit zu schaffen und Missverständnisse zu vermeiden.

Begriff	Erläuterung
Chat-Interface	Ein Chat-Interface ist eine Programmschnittstelle, die dem Nutzer eine textuelle Interaktion mit mindestens einem zweiten Nutzer oder einem Chatbot erlaubt. Im Rahmen dieser Arbeit wird mit Chat-Interface entweder das Moodle- oder das Telegram-Frontend bezeichnet.
Dialogue Flow	Ein Dialogue Flow ist ein Weg, um Konversationsabläufe mit mehreren Stufen in MindMeld zu modellieren. Dabei wird unter bestimmten Umständen in einen Flow eingetreten und darin verweilt, bis eine bestimmte Bedingung eingetreten ist. Diese Definition entspricht derjenigen aus der MindMeld-Dokumentation [58].
Dialogue Frame	Der Dialogue Frame oder Dialograhmen dient als kurzfristiger Speicher für den Kontext eines Dialogs. In ihm können Werte, die später wieder ausgelesen werden sollen, hinterlegt werden. Diese Definition entspricht derjenigen aus der MindMeld-Dokumentation [58].
Dialogue State	Der Dialogue State beschreibt, in welchem «Zustand» eine Konversation mit einem Chatbot gerade ist. Er verändert sich mit jeder neuen Nachricht. Diese Definition entspricht derjenigen aus der MindMeld-Dokumentation [58].
Domain	Im Sinne dieses Projekts ist die Domain oder Domäne ein klar abzugrenzender Wissensbereich mit eigenem Vokabular. Diese Definition entspricht derjenigen aus der MindMeld-Dokumentation [58].
Dozierende/r	Die Rolle des Dozierenden bezeichnet im Kontext dieser Arbeit den User, der für die Abfüllung und Instandhaltung der Wissensbasis des Chatbots mit fachlichen und organisatorischen Daten zuständig ist.
Driver	Driver sind ein Bestandteil der Librarian-CLI und dienen dazu, einen Weg anzubieten, verschiedene Filetypen zu laden und zu analysieren. Im Rahmen dieses Projekts wurden ein CSV-Driver und ein PDF-Driver implementiert.
Entity	Eine Entity ist ein fester Teil eines Requests eines Users, der notwendige Informationen zum Erfüllen eines Intents zur Verfügung stellt. Intents haben immer eine Bezeichnung. Ein Beispiel wäre «location» für den «Flug buchen»-Intent. Diese Definition entspricht derjenigen aus der MindMeld-Dokumentation [58].

Fachliche Daten	Fachliche Daten sind Daten, die den konkreten Inhalt einer Learning Unit betreffen. Es handelt sich dabei um den effektiven Lernstoff, der den Studierenden verfügbar gemacht werden soll. Ein Beispiel wäre eine Slideshow zu einer Vorlesung.
Gazetteer	Der Gazetteer ist ein Text-File, das MindMeld zur besseren Erkennung von Entities verwendet. Es handelt sich dabei um eine Liste von Entity-Namen sowie Synonymen. Pro Entity-Typ existiert ein Gazetteer-File.
Intent	Ein Intent ist ein Ausdruck dessen, was der User im Rahmen der Interaktion mit einem Chatbot erreichen möchte. Dabei handelt es sich um unmittelbare Ziele, nicht das Ziel der ganzen Konversation. Beispiele wären «Flug buchen» oder «Reservationslokal wählen». Diese Definition entspricht derjenigen aus der MindMeld-Dokumentation [58].
Knowledge Base	Die Knowledge Base, oder Wissensbasis, ist die Gesamtheit aller Daten, aus der ein Chatbot seine Antworten beziehen kann. In unserem Fall handelt es sich dabei um vom Librarian erfasste Daten zu Learning Units, die später abgefragt werden sollen.
Kontext	Kontext ist ein Begriff, der oft im Zusammenhang mit Keywords verwendet wird und die Stellung bezeichnet, die eine spezifische Instanz des Keywords innerhalb eines grösseren Textabschnitts einnimmt. Ein Beispiel: Ein möglicher Kontext für das Keyword «POST-Request» wäre beispielsweise: «…das Formular über einen POST-Request einzureichen…».
Learning Unit	Im Rahmen dieser Arbeit ist eine Learning Unit ein «Fach» im hochschultechnischen Sinne. Sie umfasst eine fachliche und eine fakultative organisatorische Seite und dient zur groben Unterteilung der abzufragenden Daten.
Librarian	Der Librarian besteht aus Frontend (Librarian-CLI) und Librarian-Backend und dient in erster Linie dazu, bereits vorhandene Daten (insbesondere fachliche) in ein Format zu bringen, das in der MindMeld-Wissensbasis hinterlegt und abgefragt werden kann. Er ist also ein Datenverwaltungstool.
Librarian-Backend	Das Librarian-Backend ist ein Baustein der Nabu-Umgebung. In ihm werden Daten zum späteren Upload ins Nabu-Backend gesammelt.
Librarian-CLI	Die Librarian-CLI ist eine Kommandozeilenapplikation, welche den Upload von fachlichen und/oder organisatorischen Daten ins Librarian-Backend ermöglicht.
Messaging Consumer	Messaging Consumers [34] sind Moodle-Komponenten, welche zum Abfangen und Verarbeiten von über Moodle verschickten Chatnachrichten dienen. Ein solcher Messaging Consumer wird im Rahmen des Nabu-Projekts für das Moodle-Frontend eingesetzt.
Minimum Viable Product	Das Minimum Viable Product (MVP) entspricht dem Produkt mit dem Mindestmass an Features, die nötig sind, um in einer Produktionsumgebung eingesetzt werden zu können.

Nabu-Backend	Das Nabu-Backend ist ein Baustein der Nabu-Umgebung und verantwortlich für den eigentlichen MindMeld-Chatbot sowie dessen Wissensbasis.
Organisatorische Daten	Organisatorische Daten sind Daten, die zwar eine Learning Unit betreffen, jedoch nicht oder nur indirekt mit dem Lernstoff zu tun haben. Beispiele für organisatorische Daten wären ein Abgabetermin für ein Testat oder das Datum einer Prüfung.
pip	Bei pip [35] handelt es sich um den Python-Packagemanager, der ein schnelles Installieren von zuvor registrierten Packages erlaubt.
Proof of Concept	Das Proof of Concept (PoC) ist ein erster Machbarkeitsbeweis, der dazu dient, zu zeigen, dass ausgewählte Technologien für einen bestimmten Zweck hinreichend sind.
Stretch Goal	Stretch Goals sind hoch gesteckte Ziele oder zusätzliche Features, die explizit nicht Teil des Minimum Viable Products sind und nur in Angriff genommen werden, wenn ein Projekt besser fortschreitet als erwartet.
Studierende/r	Die Rolle des Studierenden bezeichnet im Kontext dieser Arbeit den User, der den Chatbot nutzen möchte, um fachliche oder organisatorische Abfragen zu machen.
Trainingsdaten	Trainingsdaten werden von MindMeld vor allem zur Intent-Erkennung benötigt. Es handelt sich dabei um Textdaten, welche pro Intent hinterlegt werden müssen und eine Auswahl an möglichen Arten darstellen, diesen Intent zu formulieren.
Whitelist	Der Begriff der Whitelist wird in diesem Projekt vor allem im Zusammenhang mit Synonymen verwendet. Dabei ist festzuhalten, dass dieser Begriff nicht von den Autoren stammt, sondern von MindMeld eingeführt wurde. Dort steht er für alternative Namen von Entitäten, was im Falle dieses Projektes vor allem für Learning Units (DB1 für Datenbanken 1) und Keywords (die oben erwähnten Synonyme) wichtig ist.

Abkürzungen

Das Abkürzungsverzeichnis verschafft Überblick über die im Rahmen dieser Arbeit verwendeten Abkürzungen.

Abkürzung	Bedeutung
MVP	Minimum Viable Product
NLP	Natural Language Processing, also Verarbeitung von natürlicher Sprache.
PoC	Proof of Concept
SG	Stretch Goals

5.2 Quellenverzeichnis

[1] Business Insider Intelligence, “80% of businesses want chatbots by 2020,” Business Insider, 14 12 2016. [Online]. Available: https://www.businessinsider.com/80-of-businesses-want-chatbots-by-2020-2016-12. [Accessed 29 12 2020].

[2] Cisco Systems, “MindMeld Home Page,” 2019. [Online]. Available: https://www.mindmeld.com/. [Accessed 16 11 2020].

[3] A. Raghuvanshi, L. Carroll and K. Raghunathan, “Developing Production-Level Conversational Interfaces with Shallow Semantic Parsing,” in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 2018, pp. 157-162.

[4] The Moodle Project, “Moodle Home Page,” 2020. [Online]. Available: https://moodle.org/. [Accessed 01 01 2021].

[5] Telegram Messenger Inc., “Telegram Messenger Home Page,” 2020. [Online]. Available: https://telegram.org/. [Accessed 01 01 2021].

[6] ISO International Organization for Standardization, 2001. [Online]. Available: https://www.iso.org/standard/22749.html. [Accessed 05 11 2020].

[7] Amazon Web Services, Inc., “Amazon Lex Home Page,” 2020. [Online]. Available: https://aws.amazon.com/lex/. [Accessed 16 11 2020].

[8] Microsoft, “Azure Bot Service Home Page,” 2020. [Online]. Available: https://azure.microsoft.com/en-us/services/bot-service/. [Accessed 17 11 2020].

[9] Elasticsearch B.V., “Elasticsearch Home Page,” [Online]. Available: https://www.elastic.co/. [Accessed 16 11 2020].

[10] Cisco Systems, “Getting Started with MindMeld,” 2019. [Online]. Available: https://www.mindmeld.com/docs/userguide/getting_started.html. [Accessed 16 11 2020].

[11] Rasa Technologies, “Rasa Home Page,” 2020. [Online]. Available: https://rasa.com/. [Accessed 16 11 2020].

[12] GreenShoot Labs, “OpenDialog Home Page,” 2020. [Online]. Available: https://www.opendialog.ai/. [Accessed 21 11 2020].

[13] Docker Inc., “Docker Home Page,” 2020. [Online]. Available: https://www.docker.com/. [Accessed 20 12 2020].

[14] Django Software Foundation, “Django Home Page,” 2020. [Online]. Available: https://www.djangoproject.com/. [Accessed 03 11 2020].

[15] Pallets, “Flask Home Page,” 2010. [Online]. Available: https://flask.palletsprojects.com/en/1.1.x/. [Accessed 30 11 2020].

[16] Pallets, “Werkzeug,” 2020. [Online]. Available: https://palletsprojects.com/p/werkzeug/. [Accessed 30 11 2020].

[17] P. Eby, “PEP 3333 -- Python Web Server Gateway Interface,” Python Software Foundation, 26 09 2010. [Online]. Available: https://www.python.org/dev/peps/pep-3333/. [Accessed 30 11 2020].

[18] Pallets, “Flask Deployment Options,” 2010. [Online]. Available: https://flask.palletsprojects.com/en/1.1.x/deploying/. [Accessed 30 11 2020].

[19] B. Chesneau, “Gunicorn Home Page,” 2019. [Online]. Available: https://gunicorn.org/. [Accessed 30 11 2020].

[20] uWSGI, “uWSGI Home Page,” 2016. [Online]. Available: https://uwsgi-docs.readthedocs.io/en/latest/. [Accessed 11 30 2020].

[21] Twisted Matrix Labs, “Twisted Web Home Page,” 2020. [Online]. Available: https://twistedmatrix.com/trac/wiki/TwistedWeb. [Accessed 30 11 2020].

[22] Pylons Project, “Waitress Home Page,” 2020. [Online]. Available: https://docs.pylonsproject.org/projects/waitress/en/stable/. [Accessed 30 11 2020].

[23] Pallets, “Click Home Page,” 2014. [Online]. Available: https://click.palletsprojects.com/en/7.x/. [Accessed 28 12 2020].

[24] NLTK Project, “NLTK Home Page,” 2020. [Online]. Available: https://www.nltk.org/. [Accessed 03 01 2021].

[25] S. Bird, E. Loper and E. Klein, Natural Language Processing with Python, O'Reilly Media Inc., 2009.

[26] Y. Shinyama, “PDFMiner Documentation Home Page,” 2013. [Online]. Available: https://pdfminer-docs.readthedocs.io/pdfminer_index.html. [Accessed 30 12 2020].

[27] J. X. McKie, 2020. [Online]. Available: https://pymupdf.readthedocs.io/en/latest/index.html. [Accessed 30 12 2020].

[28] pandas development team, “pandas Home Page,” 2020. [Online]. Available: https://pandas.pydata.org/docs/index.html. [Accessed 30 12 2020].

[29] Pony ORM, LLC, “PonyORM Home Page,” 2020. [Online]. Available: https://ponyorm.org/. [Accessed 27 12 2020].

[30] M. Bayer, “SQLAlchemy Home Page,” 2020. [Online]. Available: https://www.sqlalchemy.org/. [Accessed 27 12 2020].

[31] Pony ORM, LLC, “Pony ORM Flask Integration,” 2020. [Online]. Available: https://docs.ponyorm.org/integration_with_flask.html. [Accessed 27 12 2020].

[32] MariaDB Foundation, “MariaDB Home Page,” 2020. [Online]. Available: https://mariadb.org/. [Accessed 30 12 2020].

[33] The PostgreSQL Global Development Group, “PostgreSQL Home Page,” 2020. [Online]. Available: https://www.postgresql.org. [Accessed 30 12 2020].

[34] The Moodle Project, “Moodle - Messaging Consumers,” 26 11 2012. [Online]. Available: https://docs.moodle.org/dev/Messaging_consumers. [Accessed 27 12 2020].

[35] Python Software Foundation, “Pip Home Page,” 2020. [Online]. Available: https://pypi.org/project/pip/. [Accessed 28 12 2020].

[36] Python Software Foundation, “PyPI - The Python Package Index,” 2020. [Online]. Available: https://pypi.org/. [Accessed 29 12 2020].

[37] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

[38] M. e. a. Sarahan, “Making a Python Package,” 2018. [Online]. Available: https://python-packaging-tutorial.readthedocs.io/en/latest/setup_py.html. [Accessed 30 12 2020].

[39] Bitnami, “Bitnami Docker Image for Moodle LMS,” 2020. [Online]. Available: https://hub.docker.com/r/bitnami/moodle/. [Accessed 30 12 2020].

[40] Python Telegram Bot, “Python Telegram Bot Github Page,” 2020. [Online]. Available: https://github.com/python-telegram-bot/python-telegram-bot. [Accessed 30 12 2020].

[41] Telegram Messenger Inc., “Botfather Home Page,” 2020. [Online]. Available: https://t.me/botfather. [Accessed 03 01 2021].

[42] Telegram Messenger Inc., “Bots: An introduction for developers,” 2020. [Online]. Available: https://core.telegram.org/bots#3-how-do-i-create-a-bot. [Accessed 06 01 2021].

[43] sprnza, “[FeatureRequest] Split too long messages,” 3 8 2017. [Online]. Available: https://github.com/python-telegram-bot/python-telegram-bot/issues/768. [Accessed 30 12 2020].

[44] H. Krekel, “Pytest Home Page,” 2020. [Online]. Available: https://docs.pytest.org/en/stable/. [Accessed 10 12 2020].

[45] W. Roberts, “pygermanet PyPI Page,” 2014. [Online]. Available: https://pypi.org/project/pygermanet/. [Accessed 30 12 2020].

[46] OAuth, “OAuth Home Page,” 2020. [Online]. Available: https://oauth.net/. [Accessed 12 30 2020].

[47] OATH Authentication, LLC, “OATH Authentication Home Page,” 2020. [Online]. Available: https://openauthentication.org/. [Accessed 30 12 2020].

[48] OpenID Foundation, “OpenID Home Page,” 2020. [Online]. Available: https://openid.net/. [Accessed 30 12 2020].

[49] Python Software Foundation, “Packaging Python Projects,” 2020. [Online]. Available: https://packaging.python.org/tutorials/packaging-projects/. [Accessed 30 12 2020].

[50] D. Stufft, “twine PyPI Page,” 2020. [Online]. Available: https://pypi.org/project/twine/. [Accessed 30 12 2020].

[51] D. Keller, Software Engineering 1, 2019.

[52] COING Inc., “Clockify Home Page,” 2020. [Online]. Available: www.clockify.me. [Accessed 03 01 2021].

[53] G. van Rossum, B. Warsaw and N. Coghlan, “PEP 8 -- Style Guide for Python Code,” 05 07 2001. [Online]. Available: www.python.org/dev/peps/pep-0008/. [Accessed 03 01 2021].

[54] K. Szanto, A. Lai, A. Makarov, M. Cullum and R. Deutz, “PSR-12: Extended Coding Style,” 2019. [Online]. Available: www.php-fig.org/psr/psr-12/. [Accessed 03 01 2021].

[55] Google, Inc., “Google JavaScript Style Guide,” [Online]. Available: google.github.io/styleguide/jsguide.html. [Accessed 03 01 2021].

[56] JetBrains s.r.o., “JetBrains: Essential tools for software developers and teams,” 2021. [Online]. Available: www.jetbrains.com. [Accessed 03 01 2021].

[57] Jenkins, “Jenkins Home Page,” 2021. [Online]. Available: https://www.jenkins.io/. [Accessed 03 01 2021].

[58] Cisco Systems, “MindMeld Key Concepts,” 2019. [Online]. Available: https://www.mindmeld.com/docs/intro/key_concepts.html. [Accessed 26 10 2020].

5.3 Abbildungen

5.4 Tabellen

6 Danksagung und Fazit

6.1 Danksagung

Unser Dank gilt unserem Betreuer, Professor Stefan F. Keller, für seinen Enthusiasmus und seine guten Ratschläge während der ganzen Nabu-Projektarbeit. Er war praktisch Tag und Nacht für uns verfügbar.

Wir danken Lukas Röllin für das Gegenlesen und das Feedback der Dokumentation sowie seine Unterstützung mit Python.

Dylan Baumgartner danken wir für die Hilfestellung mit Machine Learning und den Informationen zu der damit verbundenen Komplexität.

Für den Python-Workshop und das liebevoll erstellte Jupyter-Notebook danken wir Raphael Das Gupta.

Darüber hinaus danken wir unseren sechs Usability-Testern, namentlich:
Matteo Demasi, Mijo Lovric, David Kalchofner, Fabienne Lienhard, Jana Kravarik und Leandro Kuster.

6.2 Fazit

Beim Nabu-Projekt handelte es sich um eine sehr offene Arbeit. Das war anfänglich etwas einschüchternd, da das Ziel des Projektes nicht von Beginn weg klar war, sondern erst nach und nach erarbeitet wurde. Dazu kam, dass von den Autoren niemand Erfahrung mit Python oder sogar Chatbots und Machine Learning aufwies. Ausserdem war der Aufbau der CI von GitHub, definiert durch Workflows, ungewohnt und fühlte sich zum Teil labil an. Dies führte dazu, dass die Anfangsphase des Projektes vor allem aus Einarbeiten, Recherche, Lektüre und Experimentieren bestand.

Erschwert wurde dies noch durch die Situation mit COVID-19, die eine physische Präsenz an der OST Rapperswil-Jona für den Grossteil der Projektzeit verunmöglichte. Das bedeutete, dass Meetings und Absprachen vor allem remote abgehalten werden mussten.

Nichtsdestotrotz sind wir sehr zufrieden mit dem Resultat des Projektes. Fast alle gesteckten Ziele wurden trotz der oben erwähnten Hindernisse erreicht, da wir als hervorragend harmonierende Gruppe die meisten Schwierigkeiten meistern konnten. Wann immer ein Problem auftrat, war jemand zur Stelle, um es gemeinsam zu lösen. Die grösste Auseinandersetzung im Team war dementsprechend über das Setzen eines Kommas in der Dokumentation. Im Nachhinein müssen wir allerdings eingestehen, dass Webcams für die Remote-Meetings eventuell eine gute Idee gewesen wären.

Wir hoffen nun, zum Abschluss dieser Arbeit, dass Nabu schon bald nutzbringend an der OST eingesetzt werden kann und den Studierenden das Leben etwas erleichtert.