Lucerne University of
Applied Sciences and Arts

HOCHSCHULE
LUZERN

Informatik

Bachelor-Diplomarbeit Informatik

KONSOLIDIERUNG UND ERWEITERUNG VON FORSCHUNGSPROTOTYPEN FÜR EIN WISSENSMANAGEMENT-TOOL

Julian Bigler und Robin Bürgi

Januar 2019

Hochschule Luzern - Informatik, Rotkreuz

Betreuender Dozent:
Prof. Dr. sc. inf. Michael Kaufmann

Experte:
Dipl. Inf. Ing. HTL Dominique Portmann

Lucerne University of
Applied Sciences and Arts

HOCHSCHULE
LUZERN

Informatik

Bachelorarbeit an der Hochschule Luzern – Informatik

Titel: Konsolidierung und Erweiterung von Forschungsprototypen für ein Wissensmanagement-Tool

Diplomandin / Diplomand: Julian Bigler
Diplomandin / Diplomand: Robin Bürgi

Studiengang: BSc Informatik

Abschlussjahr: 2019

Betreuende/r Dozentin / Dozent: Prof. Dr. sc. inf. Michael Kaufmann

Expertin / Experte: Dipl. Inf. Ing. HTL Dominique Portmann

Codierung / Klassifizierung der Arbeit:

☐ **A: Einsicht (Normalfall)**

☐ **B: Rücksprache** (Dauer: Jahr / Jahre)

☐ **C: Sperre** (Dauer: Jahr / Jahre)

Abgabe der Arbeit auf der Portfolio Datenbank:

Bestätigungsvisum durch betreuenden Dozent/Dozentin
Ich bestätige hiermit, dass der Student/die Studentin, die Bachelorarbeit korrekt auf der Portfolio Datenbank abgelegt hat. Die Verantwortlichkeit sowie die Berechtigungen hat er/sie abgegeben, so dass er/sie keine Änderungen mehr vornehmen kann oder weitere Dateien hochladen kann.

Rotkreuz, den _____ Visum: _____

Web-Abstract
Das Web-Abstract wurde überprüft und entspricht den verlangten Vorgaben. Hiermit wird es freigegeben.

Rotkreuz, den _____ Visum: _____

Eingangsvisum (durch das Sekretariat auszufüllen):

Rotkreuz, den _____ Visum: _____

Hinweis: Die Bachelorarbeit wurde von keinem Dozierenden nachbearbeitet. Veröffentlichungen (auch auszugsweise) sind ohne das Einverständnis der Studiengangleitung der Hochschule Luzern – Informatik nicht erlaubt.

Verdankung
Wir möchten uns an dieser Stelle bei all denjenigen bedanken, die uns während der Erstellung dieser Bachelorarbeit unterstützt haben.
Zuerst gebührt unser Dank Herrn Prof. Michael Kaufmann, der unsere Bachelorarbeit betreut hat. Für alle Anregungen, die neuen Blickwinkel und die konstruktive Kritik bei der Erstellung dieser Arbeit möchten wir uns herzlich bedanken.
Ebenfalls möchten wir uns bei der Autorenschaft der vorhergehenden Projekte im Bereich Knowledge Management bedanken, insbesondere bei Patrick Siegfried und Andreas Waldis für die aufschlussreichen Gespräche.
Ein besonderer Dank geht an unseren Kommilitonen Dorus Janssens für die Debatten zu unserem Anwendungsfallszenario und technischen Inputs.
Schlussendlich möchten wir uns bei unseren Freundinnen Rahel Geyer und Kathleen Moser für den starken emotionalen Rückhalt und die unfreiwilligen Lektoratstätigkeiten bedanken.

Julian Bigler und Robin Bürgi,
Rotkreuz, 18.01.2019

Selbstständigkeit
Hiermit erklären wir, dass wir die Bachelorarbeit "Konsolidierung und Erweiterung von Forschungsprototypen für ein Wissensmanagement-Tool" selbstständig angefertigt und keine anderen als die angegebenen Hilfsmittel verwendet haben.

Redlichkeit
Sämtliche verwendeten Textausschnitte, Zitate oder Inhalte anderer Verfasser werden in Übereinstimmung mit Art. 25, Abs. 2 URG ausdrücklich als solche gekennzeichnet.

Ort / Datum, Unterschrift _____

Ort / Datum, Unterschrift _____

Copyright © 2018 Hochschule Luzern – Informatik

Alle Rechte vorbehalten. Kein Teil dieser Arbeit darf ohne die schriftliche Genehmigung der Studiengangleitung der Hochschule Luzern – Informatik in irgendeiner Form reproduziert oder in eine von Maschinen verwendete Sprache übertragen werden.

Abstract

English

Extracting knowledge from large, inhomogeneous amouts of data is not an easy task. The Data Intelligence Research Team at Lucerne University of Applied Sciences and Arts (HSLU) conducts research in the field of knowledge management and has implemented software prototypes in this area. These tools should help users to structure and retrieve information. The aim of this bachelor thesis is to consolidate these prototypes into an open source software project. A usage scenario has been developed and requirements based on it have been defined. These requirements were implemented iteratively and incrementally using the SoDa project methodology. The key features that were identified and implemented are full text search, automatic linking of content and multi-user capabilities. The focus of the existing prototypes was on personal knowledge management, this shifted to a collaborative tool for teams. The open source web application was developed with React, Django, Textblob, Neo4j and PostgreSQL and can be retrieved at `https://github.com/grafit-io/grafit`.

Deutsch

Wissen aus grossen, inhomogenen Datenmengen zu entnehmen ist nicht einfach. Das Data-Intelligence-Forschungsteam der Hochschule Luzern (HSLU) forscht im Bereich des Wissensmanagements und hat dafür Softwareprototypen umgesetzt. Diese Tools sollen Nutzenden helfen, unstrukturierte Informationen zu gliedern und wieder abzurufen. Das Ziel dieser Bachelorarbeit ist es, eine Konsolidierung dieser Prototypen zu einem Open-Source-Softwareprojekt vorzunehmen. Ein Anwendungsszenario wurde entwickelt und darauf abgestützte Anforderungen wurden definiert. Diese wurden iterativ-inkrementell innerhalb der Projektmethodik SoDa umgesetzt. Als zentrale Funktionen wurden die Volltextsuche, automatische Verknüpfungen von Inhalten und die Multi-User-Fähigkeit identifiziert und umgesetzt. Bei den bestehenden Prototypen lag der Fokus auf dem persönlichen Wissensmanagement, dieser verschob sich hin zu einem kollaborativen Tool. Die Open-Source-Webapplikation wurde mit React, Django, Textblob, Neo4j, PostgreSQL u. a. entwickelt und ist unter `https://github.com/grafit-io/grafit` abrufbar.

Inhaltsverzeichnis

Abstract **5**

Inhaltsverzeichnis **5**

1 Einleitung **8**

1.1 Ausgangslage 8

1.2 Ziele 8

1.3 Anwendungsszenario 9

1.4 Vorgehensmodell 9

1.5 Randbedingungen 10

1.6 Scope 11

2 Stand der Technik **12**

2.1 Knowledge Management 12

2.2 Natural Language Processing 12

2.2.1 Tokens/Tokenization 12

2.2.2 Stemming 13

2.2.3 Stopwords 13

2.2.4 n-Gram 13

2.2.5 Concepts 13

2.2.6 Part-of-Speech Tagging 14

2.2.7 TF-IDF 14

2.3 Lokahi 15

2.4 IKC 17

3 Lösungsdesign **19**

3.1 Anwendungsszenario 19

3.2 Epics 21

3.3 Integration der Prototypen 22
3.4 Architektur- und Technologieentscheide 25
3.4.1 Client-Server 25
3.4.2 Frontend Framework 25
3.4.3 Webframework und Programmiersprache 25
3.4.4 Datenbank 26
3.4.5 Search Index 27
3.4.6 Graphdatenbank 27
3.4.7 Cloud-Integration 28
3.5 User-Interface-Design 30

4 Implementierung **31**

4.1 Systemübersicht 31
4.1.1 REST-Schnittstelle 32
4.1.2 Frontend 32
4.1.3 Backend 35
4.2 Persistenzlösung 36
4.3 Knowledge-Graph 39
4.3.1 Integration Keyphrase Extraction 39
4.3.2 Generierung Knowledge-Graph 47
4.4 Search 49
4.5 Integration Cloud Service 50
4.6 Build Pipeline 51
4.7 Qualitätssicherung 53
4.7.1 Unit-Tests 53
4.7.2 Integrationstests 53
4.7.3 Testabdeckung 54
4.7.4 Lizenzmanagement und Compliance 54

4.8 Verteilung 56

5 Validierung **58**

6 Ausblick und Schlusswort **61**

Literaturverzeichnis **63**

Abbildungsverzeichnis **65**

Listingsverzeichnis **66**

Tabellenverzeichnis **67**

Glossar **68**

Akronyme **69**

A Projektmanagement **71**

A.1 Rahmenplan 71

A.2 Meilensteine und Lieferobjekt 72

A.2.1 Meilenstein 1: Grobkonzept 72

A.2.2 Meilenstein 2: Zwischenpräsentation 72

A.2.3 Meilenstein 3: Übergabe Source Code 72

A.2.4 Meilenstein 4: Abgabe Dokumentation 72

A.3 Resultate 73

A.4 Risikomanagement 74

B Product Backlog **76**

C Code Snippets **101**

C.1 Beispiel Integrationstest 101

C.2 Article Model 102

C.3 Docker Compose Files 103

D Arbeitsjournal **106**

1 Einleitung

In dieser Arbeit wird der Prozess beschrieben, wie die Forschungsprototypen ***Intuitive Knowledge Connectivity (IKC)*** und ***Lokahi*** zu einem stabilen Open-Source-Projekt ***Grafit***[1] integriert werden. Beide Prototypen befassen sich mit der Generierung und Pflege von Informationen in Form eines Knowledge-Graphs. Ein Wissensnetzwerk besteht aus einer Menge von Inhalten, die untereinander beliebige Beziehungen haben und dadurch ein Netz formen.

1.1 Ausgangslage

Das Data-Intelligence-Forschungsteam der Hochschule Luzern (HSLU) hat über die letzten Jahre mehrere Softwareprototypen im Bereich des Knowledge-Managements entwickelt. Dabei wurde mit Verfahren wie Concept Extraction aus dem Natural Language Processing (NLP), Graph-Exploring, Peer-to-Peer-Netzwerken (P2P) und Machine Learning gearbeitet. Das Ziel war die Entwicklung einer Personal-Knowledge-Management-Lösung, die Menschen hilft, ihre persönlichen Daten zu speichern, zu verwalten und wiederzufinden. Die Prototypen stellen keine funktionierende Consumer-Software dar. Eine solche würde eine Grundlage bieten, um Entwicklungen zu demonstrieren und weitere Konzepte und Algorithmen zu testen.

1.2 Ziele

Der Auftraggeber, das Data-Intelligence-Forschungsteam, wünscht eine stabil funktionierende Consumer-Software. In dieser Arbeit werden daher aufeinander aufbauende Ziele verfolgt, um die Forschungsprototypen zu professionalisieren:

- Es wird ein stabiles Open-Source-Softwareprojekt initialisiert.
- Der Stand der Prototypen und Konzepte wird analysiert.
- Aus diesen Erkenntnissen wird ein Anwendungsszenario erarbeitet.
- Code und Konzepte aus bestehenden Prototypen werden zu einer nach bewährten Engineering-Methoden aufgebauten Software integriert.

[1] https://github.com/grafit-io/grafit/

1.3 Anwendungsszenario

Das Sammeln und die Weitergabe von Wissen ist in unserer Wissensgesellschaft unabdingbar. Für Firmen kann diese Fähigkeit ein entscheidender Wirtschaftsfaktor sein. So schreibt Klaus North, dass der Aufbau und Austausch von Wissen die Handlungsfähigkeit von Personen, Teams und Organisationen überhaupt erst sicherstellt (Ackermann, 2018, p. 1). Steigende Fluktuation und Trends wie Homeworking stören den klassischen direkten Wissensfluss innerhalb des Unternehmens. Diese Problemstellung wird in diesem Projekt angegangen.
Teams sollen dabei unterstützt werden, Informationen in einer verwertbaren und durchsuchbaren Form vorliegen zu haben. Dabei werden verschiedene Datenquellen indexiert und dadurch wird über föderierte Datenquellen eine Volltextsuche ermöglicht. Aus diesem Datenkorpus werden Konzepte extrahiert und ein Wissensnetzwerk wird generiert. Damit können semantische Aspekte in die Suche integriert werden. Daten können aufgrund der extrahierten Konzepte gesucht und gefiltert werden. Durch die Möglichkeit von Concept Surfing sollen noch unbekannte Informationen erschliessbar sein.
Der Vorteil dieser Lösung ist, dass es eine zentrale Anlaufstelle für die Datenerkundung gibt, die Daten aber immer noch in der besten geeigneten Form gehalten werden können.
Alle Informationen werden in sogenannten Workspaces gespeichert. Ein Workspace bezeichnet eine logische Entität, an die Datenquellen und User-Berechtigungen gekoppelt sind. Dadurch wird eine Zugriffskontrolle ermöglicht. Wie dieses Anwendungsszenario zu Stande kam, wird in Kapitel 3.1 erarbeitet.

1.4 Vorgehensmodell

Bei dieser Arbeit handelt es sich primär um ein klassisches Software-Engineering-Projekt. Es wurde beschlossen, das Vorgehens- und Projektmodell Software Development agile (SoDa) einzusetzen. Die Entwickler des Modells, Hoffstetter und Jud, beschreiben die Vorteile dieses hybriden Systems wie folgt:

> Dieses Modell trägt einerseits dem Bedürfnis von Auftraggebern und übergeordneter Organisation nach geordneter Projektdurchführung mit standardisierten Phasen und Meilensteinen Rechnung. Andererseits wird sichergestellt, dass die Software iterativ-inkrementell entwickelt wird und

> sich agil an den sich möglicherweise ändernden Bedürfnissen der Auftraggeber und an den Möglichkeiten des Entwicklungsteams orientiert (Hofstetter & Jud, 2013, p. 142).

Details zum Projektvorgehen, zum Beispiel der Rahmenplan, befinden sich im Anhang in Kapitel A. Die Entwicklung wird wie von SoDa vorgegeben agil nach Scrum umgesetzt. Dazu werden aus dem Anwendungsszenario vier Epics abgeleitet. Die Epics werden in Kapitel 3.2 beschrieben und dann in User Stories zerlegt. Zusammen mit weiteren Anforderungen, die bspw. aus Randbedingungen entstehen, bilden diese den Product Backlog. Der Product Backlog ist die Grundlage für das iterativ-inkrementelle Vorgehen.

1.5 Randbedingungen

Die folgenden Randbedingungen wurden aus der Aufgabenstellung abgeleitet:

- Open Source
  Das Projekt wird unter einer Open-Source-Lizenz veröffentlicht. Der Code und alle notwendigen Umsysteme wie Build Scripts sind frei zugänglich. Auch das Projektmanagement soll so weit wie möglich öffentlich umgesetzt werden. Der Product Backlog wird deshalb mit dem Issue-System von GitHub umgesetzt. Der Backlog wird in englischer Sprache verfasst, um mögliche sprachliche Hürden zu reduzieren.

- Self Hosted
  Die Software ist unabhängig lauffähig. Das gilt auch für allfällige zentrale Backend-Komponenten. Die Systemvoraussetzungen sind vernünftig. Die Software muss auf einem Server mit 4 GB Ram und zwei Prozessorkernen lauffähig sein. Dies ist vergleichbar mit einer Amazon EC2 t2.medium Instanz (Stand Okt. 2018).
  Eine einfache und plattformunabhängige Installation soll durch die Bereitstellung eines Docker Images ermöglicht werden.

- Keine Lizenzkosten
  Keines der verwendeten Softwarepakete darf mit der gewählten Open-Source-Lizenz in Konflikt stehen und es dürfen keine Lizenzkosten anfallen.

- Clean Code
  Die Software soll nach bewährten Software-Engineering-Prinzipien aufgebaut werden. Die umgesetzte Software soll einfach erweiterbar sein, speziell die Cloud-Integration und Concept-Extraction.

- Testing
  Es wird eine Test Suite aufgebaut, die bei jeder Änderung automatisch gegen den Source Code ausgeführt wird. Die Testabdeckung wird laufend aufgezeichnet und kann ausgewiesen werden.

1.6 Scope

Das Ziel der Arbeit ist nicht, die Keyword Extraction zu verbessern oder daran weiterzuforschen. Aus den bestehenden Algorithmen wird eine passende Lösung ausgewählt. Alle Natural Language Processing Aspekte werden nur für die englische Sprache optimiert. Bei der Concept Extraction und der Suche wird keine Mehrsprachigkeit unterstützt.
Es wird kein Software as a Service Produkt (SaaS) entwickelt. Jegliche potenzielle Vermarktung unter einem SaaS-Modell ist nicht Teil dieser Arbeit.

2 Stand der Technik

In diesem Kapitel soll der aktuelle Stand der Forschung im Bereich des Knowledge-Managements und den damit im Zusammenhang stehenden Disziplinen dargelegt werden. Dadurch werden die grundlegenden Begrifflichkeiten erklärt. Zusätzlich werden die bestehenden Resultate der Forschungstätigkeiten der HSLU in diesem Bereich vorgestellt.

2.1 Knowledge Management

Knowledge Management oder auf Deutsch *Wissensmanagement* ist eine disziplinenübergreifende Methode, organisatorische Ziele mithilfe von Wissen zu verfolgen. Im Fokus stehen dabei Prozesse, die das Akquirieren, Kreieren oder Teilen von Wissen ermöglichen oder vereinfachen (*Introduction to Knowledge Management*, 2007).

Eine einheitliche Definition von Wissensmanagement gibt es nicht. Jedoch wird im Paper *Defining knowledge management: Toward an applied compendium* von John Girard (2015, p. 14) Knowledge-Management folgendermassen definiert:

> Knowledge Management is the process of creating, sharing, using and managing the knowledge and information of an organization.

2.2 Natural Language Processing

Natural Language Processing (NLP) ist eine Teildisziplin der Informatik, die sich mit dem maschinellen Verständnis der menschlichen Sprache auseinandersetzt. Dieses Kapitel dient als Einführung in die Grundlagen von NLP.

2.2.1 Tokens/Tokenization

Um ein Stück Text analysieren zu können, muss dieses zuerst in einzelne Teile zerlegt werden. Hierfür wird sogenannte Tokenization eingesetzt. Ein Tokenizer gliedert eine Zeichenkette in Sätze, Satzzeichen oder Wörter auf. Dies kann anhand von Satzzeichen, Leerzeichen und der Grammatik des Satzes geschehen.

2.2.2 Stemming

Julie Beth Lovins, Erfinderin und Entwicklerin des ersten Stemming-Algorithmus, beschreibt Stemming als Prozedur, die alle Wörter mit dem gleichen Wortstamm zu einer gemeinsamen Form transformiert. Das geschieht normalerweise durch das Entfernen aller abgeleiteten Suffixe und Flexionssuffixe (Lovins, 1968, p. 1).

2.2.3 Stopwords

Als Stopwords werden die Wörter bezeichnet, die in einer Sprache sehr oft vorkommen und somit für eine algorithmische Verarbeitung oft nicht relevant sind. Beispiele dafür sind im Englischen *the* und *and*. Stopwords werden meist aus einem Dokument entfernt, bevor eine Klassifizierung vorgenommen wird (Rajaraman & Ullman, 2011, p. 8).

2.2.4 n-Gram

Ein n-Gram (auch k-Gram) ist eine geordnete Reihenfolge von n aufeinanderfolgenden Fragmenten in einem Text. Die Fragmente können zum Beispiel Wörter oder auch einzelne Buchstaben sein. Der Text *In a galaxy far away* hat als Beispiel drei 3-Grams, wenn nach Wörtern aufgeteilt wird: *In a galaxy*, *a galaxy far* und *galaxy far away* (Parameswaran, Garcia-Molina & Rajaraman, 2010, p. 2).

2.2.5 Concepts

Concepts werden nach Parameswaran, Garcia-Molina & Rajaraman (2010, p. 2) wie folgt definiert:

> We define a concept [...] to be a k-gram that represents a real or imaginary entity, event or idea that many users may be interested in (i.e., is popular) [...]

Als **Concept Mining** oder **Concept Extraction** wird der Prozess der Extraktion oben beschriebener Concepts aus einem Text mittels NLP-Methodiken genannt. Konzepte können beispielsweise mit einem Keyphrase-Extraction-Algorithmus bestimmt werden. Mit einem solchen Algorithmus können die wichtigsten Wörter oder

Wortkombinationen eines Textes bestimmt werden. Eine hierfür oft verwendete statistische Methode ist der TF-IDF-Algorithmus welcher in Kapitel 2.2.7 beschrieben ist.

2.2.6 Part-of-Speech Tagging

The Stanford Natural Language Processing Group (2004) beschreibt Part-of-Speech Tagging (PoS Tagging) als den Vorgang, jedes Wort innerhalb eines Textes der jeweiligen Gruppe zuzuordnen (z. B. Nomen, Verb, Satzzeichen usw.).

Folgendes Beispiel zeigt, wie in Python ein PoS Tagging vorgenommen werden kann. Wie in Zeile 5 ersichtlich, ist das Ergebnis des PoS Taggings, dass jedem Wort eine Gruppe zugeordnet wurde.

```
1 >>> from nltk import tag
2 >>> tokens = ['May', 'the', 'force', 'be', 'with', 'you', '.']
3 >>> tags = tag.pos_tag(tokens)
4 >>> tags
5 [('May', 'NNP'), ('the', 'DT'), ('force', 'NN'), ('be', 'VB'),
  ↪ ('with', 'IN'), ('you', 'PRP'), ('.', '.')]
```

Listing 1: PoS Tagging in Python

Eine Liste aller möglichen PoS Tags kann auf der Webseite der University of Pennsylvania abgerufen werden.[2]

2.2.7 TF-IDF

Term Frequency – Inverse Document Frequency (TF-IDF) ist ein Algorithmus zur Bestimmung der Wichtigkeit eines Wortes in einem Text. Dafür werden die Term Frequency (TF) und die Inverse Document Frequency (IDF) verwendet.

Die einfachste Möglichkeit, um die Wichtigkeit eines Wortes in einem Text zu messen, ist die Vorkommenshäufigkeit, also die Term Frequency, zu messen.

Die Überlegung, dass die am häufigsten vorkommenden Begriffe in einem Text auch

[2] https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html

die Wichtigsten sind, stellt sich jedoch als falsch heraus. Die am häufigsten vorkommenden Begriffe sind meist Stopwords und diese haben kaum Bedeutung. Deshalb wird nebst der Term Frequency auch die Inverse Document Frequency verwendet, um die Relevanz eines Begriffes zu berechnen (Rajaraman & Ullman, 2011).

IDF misst die allgemeine Relevanz eines Begriffes in Relation zu allen Dokumenten. Die Überlegung hierbei ist, dass Begriffe, die in einem Text oft vorkommen, also einen hohen TF Wert haben, aber generell gesehen eher selten in einem Text vorkommen und somit einen hohen IDF Wert haben, essenzielle Begriffe dieses Textes sind.

2.3 Lokahi

Das HSLU-Forschungsprojekt Lokahi befasst sich mit der automatisierten Generierung von Wissensnetzwerken und wie diese durchsucht werden können. Mithilfe des Prototyps können Texte automatisch getaggt, durchsucht und in einem Graphen visualisiert werden. Die Visualisierung hilft dabei, verwandte Begriffe und Keyphrases zu erforschen und sich durch das Wissensnetzwerk zu navigieren.

Die Keyphrases der Texte werden mit einem TF-IDF-Algorithmus extrahiert. Anhand dieser Keyphrases werden anschliessend die Beziehungen der Texte und Konzepte ermittelt. Die berechneten Beziehungen werden anschliessend visualisiert.

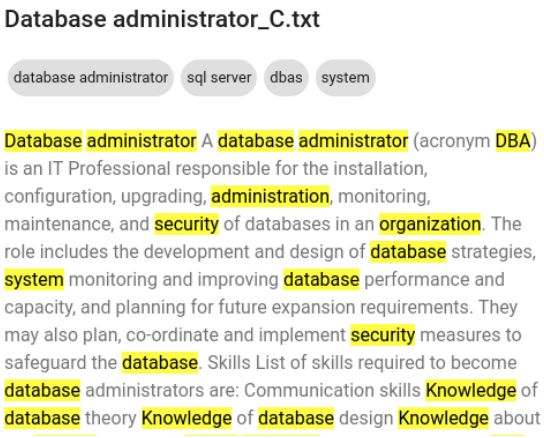


Abbildung 1: Lokahi – Keyphrase Extraction

Auf der Benutzeroberfläche kann nach einem Konzept gesucht werden. Verwandte Konzepte und der Suchbegriff werden anschliessend als Graphknoten dargestellt und entsprechend verbunden. Dies erlaubt den Benutzer_Innen, verwandte Konzepte zu

erforschen. Dieser Prozess wird **Concept Surfing** genannt.

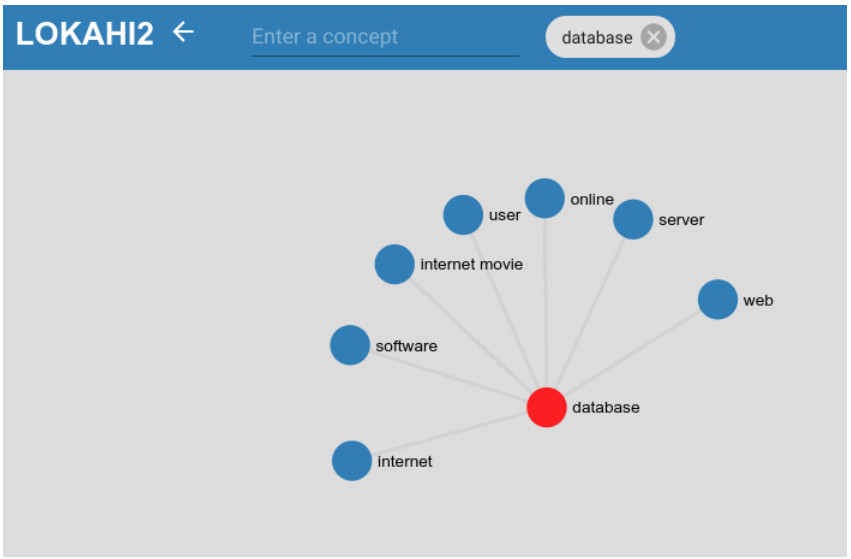


Abbildung 2: Lokahi – Graphvisualisierung

2.4 IKC

Intuitive Knowledge Connectivity (IKC) ist ein HSLU-Forschungsprojekt, für das bereits mehrere wissenschaftliche und Entwicklungsarbeiten getätigt wurden. Hauptfokus des Projekts ist es, Privatpersonen beim Umgang von Big Data im persönlichen Kontext zu helfen und Wissenskontexte zu erstellen.

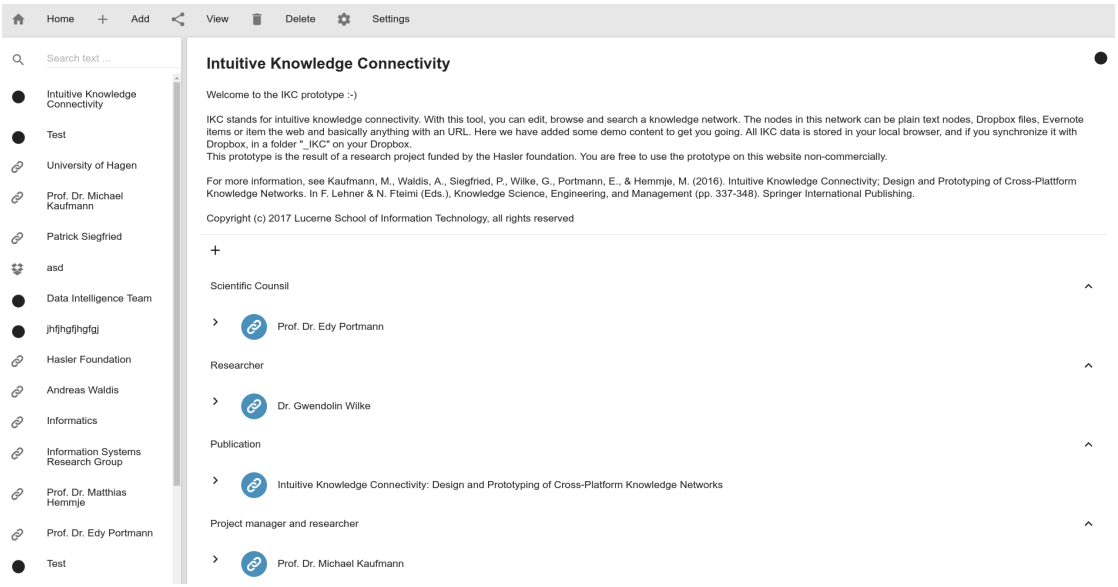


Abbildung 3: IKC – User Interface

Hauptfunktion von IKC ist es, Daten aus verschiedenen Cloud-Plattformen in einem unabhängigen Wissensnetzwerk zu integrieren. Die importierten Knoten, Objekte aus der Cloud-Plattform, können mit anderen Knoten der IKC-Plattform verknüpft werden. Es wurden Prototypen für die Anbieter Dropbox und Evernote umgesetzt.

Wie Lokahi bietet auch IKC eine Graphvisualisierung des Wissensnetzwerks an. So sollen die Daten für die Benutzer_Innen möglichst verständlich dargestellt werden, sodass im besten Fall neue Zusammenhänge erkannt und diese im Netzwerk verknüpft werden können. Das Erstellen von neuen Knoten und das Verknüpfen von bereits bestehenden Knoten kann direkt in der Graphvisualisierung getätigt werden.

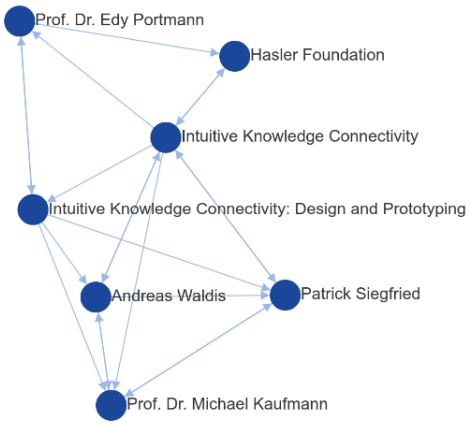


Abbildung 4: IKC – Graphvisualisierung

Als Persistenzlösung wird bei der IKC-Applikation Dropbox eingesetzt. Dies bedeutet, dass Anwendende einen Dropbox-Account hinterlegen müssen, wo deren Einstellungen und Daten als JSON serialisiert in einem Ordner namens *_IKC* gespeichert werden.

3 Lösungsdesign

Der Fokus der vorliegenden Arbeit liegt darauf, den bestehenden Stand der Forschung zu konsolidieren – konkret, die beiden Prototypen IKC und Lokahi sinnvoll zu vereinigen. Alle bestehenden Konzepte und Features müssen auf ihren Nutzen untersucht werden. Schlussendlich kann nur eine Teilmenge der Features übernommen werden. Um die Auswahl des Feature Sets auf eine klare Grundlage zu stellen, wird das Anwendungsszenario revidiert. So wird der Praxisbezug gewährleistet. Dabei stützen wir uns primär auf die Bachelorthesis von Kevin Stadelmann mit dem Titel *Entwicklung eines Business Modells für ein University Spin-off: wirtschaftliche und organisatorische Aspekte des Transfers von der Forschung in die Praxis* und die Vertiefungsarbeit von Matthias Eiholzer mit dem Titel *Search as a central aspect of personal data management*. Beide beschäftigen sich mit der Weiterentwicklung des IKC-Prototyps. Aus dem so entstehenden Anwendungsszenario werden schliesslich die Anforderungen abgeleitet.

Anhand dieser Anforderungen kann dargelegt werden, welche bestehenden Lösungen direkt übernommen werden können, was konzeptionell übernommen werden kann, was verworfen wird und welche Features neu umgesetzt werden müssen. Dieses Kapitel wird mit den Technologieentscheiden abgerundet.

3.1 Anwendungsszenario

Kevin Stadelmann hat in seiner Bachelor-Diplomarbeit die Voraussetzungen für eine Transformation des IKC-Prototyps in ein wirtschaftliches Geschäftsmodell untersucht. Er stützte sich dabei auf die Lean-Startup-Methodik. Dabei führte er Befragungen von gesamthaft 22 Fachpersonen im Bereich des Knowledge-Managements, Privatpersonen und Personen aus der Wirtschaft durch (Stadelmann, 2017, p. I). Dabei ist Stadelmann zu der Erkenntnis gekommen, dass Knowledge-Management-Lösungen im Bereich Business-to-Consumer (B2C) einen schweren Stand haben. Viele Nutzende sind nicht technikaffin genug, um ein System einzusetzen und bereits bei einem Ökosystem eines grösseren Cloud-Storage-Anbieters in einem Lock-In. Auch die oft geringen Datenmengen führen nicht zu einer Überforderung und die aktuelle Situation wird nicht als Problem wahrgenommen. Differenzierter ist die Situation im Business-to-Business-Bereich (B2B). In grösseren und über längere Zeit gewachsenen Organisationen bilden sich oft inhomogene Informationsinfrastrukturen. Gerade bei verteilten Informationsquellen wurde ein Defizit an Suchmöglichkeiten festgestellt.

Damit diese Daten gegliedert werden können, wurde ein automatisches Tagging als wirksame Lösung identifiziert, um ein kollaboratives Wissensnetzwerk aufzubauen.

Die These, dass die Suche im Bereich des Informationsmanagements eine zentrale Rolle spielt, wird von Eiholzer unterstützt. Durch eine Fallstudie und Nutzerbefragungen fand er heraus, dass die Suche für den IKC-Prototyp zentral ist (Eiholzer, 2017, p. 1). Er schlägt vor, dass auch Objekte aus externen Quellen in den Suchresultaten erscheinen (Eiholzer, 2017, p. 3). Auch Eiholzer identifizierte beim IKC-Prototyp einen Mangel an Möglichkeiten zur kollaborativen Pflege des Wissensnetzwerks (Eiholzer, 2017, p. 11).

Für die Konsolidierung der Prototypen werden die aus diesen beiden Arbeiten gewonnenen Erkenntnisse berücksichtigt und es entstand die Vision, kleinen Teams dabei zu helfen, Wissen zu strukturieren, zu finden und intern weiterzugeben. Die folgenden Punkte werden als entscheidend erachtet:

- Mehrbenutzerfähigkeit
  Die Mehrbenutzerfähigkeit mit integrierter Zugriffsbeschränkung ist zentral, um in einem wirtschaftlichen, kollaborativen Umfeld bestehen zu können.

- Suche über föderierte Datenquellen
  Wie Eiholzer in seiner Arbeit herausgefunden hat, ist eine Suchfunktion zentral. Viele Firmen haben eine Multi-Cloud-Umgebung und eine föderierte Suche kann einen Mehrwert bieten.

- Automatisches Tagging
  Artikel müssen automatisch in den Knowledge-Graph aufgenommen und verlinkt werden. Der Arbeitsaufwand für den User muss minimiert werden.

- Pflege des Wissensnetzwerks
  Der User muss trotz der Automatismen manuell Verbindungen hinzufügen und entfernen können. Nur so kann die Qualität und der Nutzen des Knowledge-Graphs langfristig garantiert werden.

- Cloud-Integrationen
  Es müssen auch externe Datenquellen in das System integriert werden können. Erst die Kombination von manuell erfassten Daten und bestehenden Daten schafft einen Mehrwert.

- Bearbeitbarer Inhalt
  Der User muss die einzelnen Knoten bearbeiten können. Die Artikel müssen direkt im System erfasst werden können.

3.2 Epics

Um diese Anforderungen zu strukturieren, werden Epics formuliert:

1. Knowledge Retrieval
   Die benutzende Person will Informationen aus dem System erhalten. Eine Volltextsuche über alle Nodes soll angeboten werden. Dabei können auch angrenzende Nodes erkundet werden.

2. Knowledge Supply
   Die benutzende Person will Informationen über Grafit zur Verfügung stellen. Fremdsysteme können integriert werden, Nodes und ihre Verbindungen können direkt in Grafit eingepflegt werden.

3. Workspace Management
   Die benutzende Person will die Workspaces verwalten. Pro Workspace können Einstellungen und Berechtigungen festgelegt werden.

4. Authentifizierung
   Die benutzende Person kann sich bei Grafit registrieren und wiederkehrend authentifizieren.

Die Applikation bringt erst Nutzen, wenn diese vier Epics umgesetzt sind. Diese werden weiter in User Stories zerlegt. Eine User Story soll so gestaltet werden, dass sie innerhalb eines Sprints umgesetzt werden kann und einen Mehrwert bringt.

Aus den Randbedingungen (Kapitel 1.5) wurden weitere Anforderungen abgeleitet. Zusammen mit den User Stories ergibt das eine Liste von funktionalen und nichtfunktionalen Anforderungen.

Diese Liste bildet den Product Backlog, der mithilfe von GitHub Issues geführt wird. Die Epics wurden auf GitHub als Labels definiert, um den Product Backlog zu strukturieren. Die exportierten Issues befinden sich in Anhang B.

3.3 Integration der Prototypen

Die folgenden Funktionen wurden als Hauptmerkmale von Lokahi und IKC identifiziert. Ob und wie diese integriert werden können, lässt sich primär vom gewählten Anwendungsszenario herleiten. Die Features wurden untersucht und in der folgenden Auflistung wird definiert, was übernommen werden kann, was angepasst werden muss und was verworfen wird.

- **TF-IDF für Auto Tagging**
  Andreas Waldis und Patrick Siegfried haben in ihrer Bachelorarbeit *Automatisierte Generierung von plattformübergreifenden Wissensnetzwerken mit Metadaten und Volltextindexierung* eine Erweiterung von IKC zum Taggen von Nodes im Knowledge-Graph geplant und umgesetzt. Die Implementierung ist auf der aktuellen IKC-Testinstanz nicht mehr lauffähig. Um das automatische Tagging umzusetzen, haben sie mehrere Varianten von TF-IDF Algorithmen getestet. Sie verwendeten eine Variation, die die Resultate mit der Dokumentlänge normalisierte (siehe Gleichung 1) (Waldis & Siegfried, 2017, p. 44).

$$tfidf = \underbrace{\frac{1}{length}}_{norm} * \underbrace{termDocFreq}_{TF} * \underbrace{\left(1 + \log\left(\frac{numDocs}{1 + termFreq}\right)\right)}_{IDF} \quad (1)$$

  Dieser Algorithmus wurde mit TypeScript implementiert. Wie in Kapitel 3.4.3 beschrieben, wird das Backend mit Python entwickelt. Der Algorithmus wird dementsprechend portiert.

- **Graph-View**
  Sowohl Lokahi als auch IKC bieten die Möglichkeit, das Knowledge Network als Graph darstellen zu lassen. Bei Lokahi ist dies die zentrale Darstellungsform. Gesprächspartner_Innen von Stadelmann bezeichneten den IKC-Graph-View auch als unübersichtlich und überladen (Stadelmann, 2017, p. 30). Der Graph-View hat deswegen eine niedrige Priorität und wird vorerst nicht integriert.

- **Concept Surfing**
  Die Funktionalität, neue, benachbarte Nodes zu erkunden, ist in beiden Prototypen implementiert. In Lokahi wird diese Funktionalität über die Graphdarstellung umgesetzt. So kann direkt im Graph ein Node angewählt werden, der dann in das Zentrum rückt und seine benachbarten Nodes werden angezeigt.

Bei IKC wird bei der Detailansicht von jedem Node eine Liste von dessen benachbarten Nodes angezeigt. Diese ist als Baumstruktur erweiterbar und als React-Komponente implementiert. Da der Graph-View als ein nicht zentrales Feature identifiziert wurde, wird das Concept Surfing mit der Baumstruktur aus IKC umgesetzt. Die React-Komponente kann somit aus IKC übernommen werden.

- **Node Editing**
  Beim IKC können die Texte von Nodes direkt editiert werden. Diese Funktionalität soll auch in Grafit übernommen werden.

- **Edges**
  IKC arbeitet mit bidirektionalen Kanten, dies wird auch bei Grafit so umgesetzt. Die Kanten können mit einem Label versehen werden. In der Detailansicht werden die Kanten nach diesen Labels gruppiert. Jedoch werden im IKC bei der Detailansicht eines Nodes die Incoming Edges in einer separaten Liste dargestellt. Das ist bei Grafit nicht der Fall.

- **IPFS**
  Eine Weiterentwicklung des IKC-Prototyps wurde in Richtung einer Peer-to-Peer-Applikation (P2P) gemacht. Hierfür wurde das InterPlanetary File System (IPFS)[3] als Protokoll eingesetzt.

  IPFS bietet den Grund-Layer, auf dem P2P-Applikationen aufgebaut werden können. Daten können mit IPFS global per Hash adressiert und verfügbar gemacht werden. Anschliessend können andere IPFS Nodes die Daten, ähnlich wie BitTorrent, von den Nodes herunterladen, welche die Daten schon besitzen. Allerdings bietet IPFS keine Funktionen auf einer höheren Abstraktionsebene, die von der Persistenzlösung einer komplexeren Applikation erwartet werden. OrbitDB[4] versucht dieses Defizit zu beheben, indem es eine NoSQL-Datenbank anbietet, die IPFS als Storage Layer verwendet. OrbitDB ist jedoch noch nicht stabil und robust genug für einen produktiven Einsatz. Funktionen wie ein dynamisches Berechtigungssystem, die in einer Multi-User-Umgebung zentral sind, sind damit zurzeit nicht umsetzbar.

  Peer-to-Peer-Applikationen bringen zudem diverse andere Schwierigkeiten mit

[3] https://ipfs.io/
[4] https://github.com/orbitdb/orbit-db

sich. Probleme bilden vor allem Verfügbarkeit und Privacy, da die Daten ungeschützt zugänglich sind.

Unter Berücksichtigung des Anwendungsszenarios wurde aus oben genannten Gründen beschlossen, nicht auf einen P2P-Ansatz zu setzen.

- **Search**
  Das IKC hat eine Volltext-Suchfunktion über alle Nodes. Dabei befinden sich aber alle durchsuchten Daten lokal im Browser des Clients und werden mithilfe der JavaScript Library elasticlunr durchsucht. Da durch das Anwendungsszenario die Multi-User-Fähigkeit als zentraler Aspekt identifiziert wurde, können nicht alle Daten an den Client übertragen werden. Die Suche wird aus Sicht des Users identisch zum IKC-Prototyp umgesetzt. Technisch muss die Lösung aber komplett überholt und serverseitig mit Python implementiert werden. Die Usability der Suche soll weiter verbessert werden, indem bei den Resultaten ein Preview-Text mit hervorgehobenen Suchbegriffen angezeigt wird.

- **Node types**
  Die IKC-Applikation unterscheidet zwischen verschiedenen Node-Typen innerhalb des Knowledge-Netzwerks. So gibt es spezielle URL Nodes oder Evernote Nodes, die unterschiedlich erstellt werden. Neu werden alle Nodes identisch behandelt, mit dem Ziel, das UI einfacher zu gestalten.

- **Cloud-Integration**
  IKC hat zwei Cloud-Integrationen zu Evernote und Dropbox. Beide sind wegen Änderungen der API nicht mehr lauffähig. Sie werden nicht repariert und ins Grafit-Projekt übernommen. Stattdessen wird eine neue API-agnostische Integrationsschnittstelle aufgebaut, siehe Kapitel 3.4.7·

3.4 Architektur- und Technologieentscheide

Basierend auf den Entscheidungen, wie die Prototypen integriert werden sollen (Kapitel 3.3), werden nun die passenden Architekturen und Technologien für die Umsetzung ausgewählt. Bei den Entscheidungen wird darauf geachtet, eine bestmögliche Abdeckung des Anwendungsszenarios zu ermöglichen. Zudem soll auf etablierte und bewährte Technologien und Softwareentwicklungspraktiken gesetzt werden. So kann eine stabile Software entwickelt werden. Die Wahl von populären Technologien fördert zudem eine mögliche Weiterentwicklung des Open-Source-Projekts durch Drittpersonen.

3.4.1 Client-Server

Wie in Kapitel 3.3 dargelegt wurde, ist ein P2P Ansatz nicht zu verfolgen. Stattdessen wurde entschlossen, eine klassische Client-Server-Architektur zu adoptieren.

3.4.2 Frontend Framework

Das Frontend aller aktuellen Prototypen von IKC oder Lokahi wurden mit React[5] entwickelt. React ist eine JavaScript-Bibliothek, die es ermöglicht, komponentenbasierte und interaktive Benutzeroberflächen zu entwickeln. React wird als stabil angesehen und hauptsächlich von Facebook unterhalten und entwickelt.
Es wurde beschlossen, das Frontend mit React umzusetzen, da die aktuellen Prototypen auch mit dieser Bibliothek implementiert wurden. So können Komponenten der UI einfach übernommen werden.

3.4.3 Webframework und Programmiersprache

Wie in Kapitel 3.4.1 entschieden wurde, soll eine Client-Server-Architektur umgesetzt werden. Da der IKC-Prototyp nur für den Single-User-Betrieb im Browser ausgelegt ist, wurde dort die gesamte Business-Logik auf dem Client mittels JavaScript implementiert. Neu soll nun ein entsprechendes Backend erstellt werden, das viele dieser Funktionen übernimmt. Deshalb wird hier entschieden, welches Webframework und welche Programmiersprache für die Umsetzung des Backends eingesetzt werden.

[5] https://reactjs.org/

Python ist eine simple aber mächtige Programmiersprache. Sie ist weit verbreitet und bietet exzellente Funktionalitäten für Natural Language Processing und linguistische Datenprozessierung (Bird et al., 2009, p.xiii).
Es existieren viele Python Frameworks, die speziell für Machine Learning und NLP ausgelegt sind. Das bekannteste darunter ist das Natural Language Toolkit (NLTK)[6].

Python eignet sich auch sehr gut für die Entwicklung von Webapplikationen und es existieren diverse Webframeworks. Django ist ohne Zweifel das bekannteste Python-Webframework und eines der bekanntesten Webframeworks überhaupt (de la Guardia, 2016, p. 31). Das selbsternannte *Webframework für Perfektionisten mit Deadlines* ist als Full-Stack-Framework ausgelegt. Das bedeutet, dass essenzielle Applikationsteile wie Authentication und Datenbank-Migrationen vom Framework selbst unterstützt werden. Django ist dogmatisch und gibt eine gewisse Applikationsstruktur vor, die einzuhalten ist u. a. das Model-View-Template-Pattern.

Wie in Kapitel 3.4.2 entschieden wurde, soll das Frontend mit React umgesetzt werden. Somit wird auf die Django-interne Template Engine verzichtet und stattdessen eine REST-Schnittstelle entwickelt werden. Hierzu wird das Django REST Framework (DRF) verwendet. DRF vereinfacht das Erstellen einer REST-API erheblich. So werden die Endpunkte der Schnittstelle als *Browsable API* im Webbrowser bedienbar gemacht, was den Entwicklungsprozess unterstützt. Zudem ist eine Authentifizierung mit JSON-Web-Tokens integriert.

Somit wurde entschieden, Python in Kombination mit Django und dem Django REST Framework für die Entwicklung des Backends einzusetzen.

3.4.4 Datenbank

Wie in Kapitel 2.4 beschrieben, wird im aktuellen IKC-Prototyp der State des Benutzers als JSON serialisiert und als Flat File in die entsprechende Dropbox abgespeichert. In einem Single-User-Kontext ist dies eine simple und funktionale Persistenzlösung. Für das Erreichen des Anwendungsszenarios wurde dies aber als unzureichend bewertet, daher soll eine relationale Datenbank eingesetzt werden.

[6] http://www.nltk.org/

Ein Relational Database Management System (RDBMS) bietet folgende Vorteile (Meier & Kaufmann, 2016, p. 9-11):

- **Konsistenz**: Das RDBMS stellt Hilfsmittel zur Gewährleistung der Datenintegrität zur Verfügung.
- **Multi-User**: Das Datenbanksystem unterstützt den Mehrbenutzerbetrieb. User können gleichzeitig dieselbe Datenbank abfragen oder bearbeiten.
- **Berechtigungen**: Ein RDBMS unterstützt Berechtigungen, die vor unbefugtem Zugriff schützen.

Diese Vorteile sollen genutzt werden und es wird beschlossen, Django in Kombination mit PostgreSQL (Postgres) einzusetzen. PostgreSQL ist ein Open Source RDBMS, das in der Django Community etabliert ist und offiziell unterstützt wird.[7]

3.4.5 Search Index

Um eine Suchfunktionalität über föderierte Datenquellen umzusetzen, bieten sich zwei Strategien an. Die Suchanfrage kann auf die Informationsquellen weitergeleitet werden oder die Quellen werden zentral indexiert. Anders formuliert, die Daten können bei der Indexierung oder zum Zeitpunkt der Suchanfrage kombiniert werden. Bei Grafit bestehen die externen Quellen aus URLs, daher ist eine Weiterleitung der Suchanfrage an die Informationsquelle nicht sinnvoll und ein zentraler Index wird aufgebaut. PostgreSQL bietet verschiedene Möglichkeiten, um anspruchsvolle Suchfunktionen umzusetzen, bspw. Trigram Matching, Stemming und gewichtete Indexe. Auf weitere Technologien wie Elastic Search kann verzichtet werden.

3.4.6 Graphdatenbank

Die Datenstruktur von IKC und Lokahi ist graphartig und wird in den Prototypen dem Benutzer sogar in einem Graphen visualisiert. Diese Daten können auch in einer relationalen Datenbank gespeichert werden, es bietet sich jedoch an, auch Graphdatenbanken in Betracht zu ziehen. Vorteil einer Graphdatenbank ist das intuitive Datenmodell, das spezifisch für das Speichern und Managen von vernetzten Daten

[7] https://docs.djangoproject.com/en/2.1/ref/databases/

vorgesehen ist. Es entspricht somit besser dem mentalen Modell. Zudem ist das Traversieren von Daten, die in einer Graphstruktur vorliegen, mit einer Graphdatenbank einfacher und schneller (Redmond et al., 2012, p. 282). Für die Interaktion mit einer Graphdatenbank kann eine spezifisch für diesen Anwendungsfall konzipierte, deklarative Abfragesprache wie Cypher[8] eingesetzt werden.

Nachteil einer Graphdatenbank ist, dass sich diese nur schwer skalieren lässt. Bei Neo4j zum Beispiel kann aus mehreren Neo4j Nodes ein Cluster gebildet werden. Die Daten werden zwischen den Nodes synchronisiert, was durch die Verteilung der Daten anschliessend eine hohe Verfügbarkeit garantiert. Da jedoch jeder Node alle Daten besitzen muss, also kein sharding möglich ist, kann auch im Cluster-Betrieb keine horizontale Skalierung erzielt werden (Redmond et al., 2012).

Als Graphdatenbank kommt vor allem Neo4j infrage. Neo4j ist die bekannteste und am weitesten verbreitete Graphdatenbank. Sie basiert auf dem Property-Graph-Modell. Ein Property-Graph hat nach Robinson et al. (2015) folgende Charakteristiken:

- Besteht aus Knoten (Nodes) und Beziehungen (Relationships)
- Knoten und Beziehungen können Eigenschaften haben (key-value pairs)
- Beziehungen sind gerichtet und haben immer einen Anfangs- und einen Endknoten

Das Property-Graph-Modell stimmt somit mit der Datenstruktur überein, die persistiert werden soll. Somit wird entschieden, Neo4j für das Persistieren des Wissensnetzes zu verwenden. Um das Problem der fehlenden horizontalen Skalierung zu entschärfen, werden nur die Relationen und Relationsattribute (TF-IDF Scores etc.) der Texte auf dem Graph gespeichert, aber nicht die Textdaten selbst. So entsteht nur eine geringe Datenmenge, die auf der Graphdatenbank gespeichert werden muss.

3.4.7 Cloud-Integration

Ein wichtiger Bestandteil des Anwendungsszenarios sind die Anbindungen an verschiedene Cloud-Dienste, um externe Informationsquellen in das Wissensnetzwerk

[8] https://www.opencypher.org/

aufzunehmen. Beim IKC-Prototyp wurde eine Schnittstelle zu Dropbox und Evernote angeboten. Ein Ziel dieser Arbeit ist, diese zu übernehmen und eine weitere zu implementieren. Bei genauerem Untersuchen der Anbindungen des IKC-Prototyps fiel auf, dass mit dem heutigen Stand der API die beiden Anbindungen nicht mehr lauffähig sind.

Anstelle eines Umbaus der Schnittstellen, sodass sie die neuen APIs unterstützen, bietet sich ein API-agnostischer Ansatz an. Um die Langlebigkeit und Stabilität der Schnittstelle zu gewährleisten, werden nicht datenquellenspezifische APIs angesprochen, sondern HTTP, das Standardprotokoll des Internets, verwendet.

Im Titel und Text eines Artikels können Links eingefügt werden. Die Applikation besucht anschliessend diese URLs und lädt den Inhalt herunter. Diese Inhalte werden gespeichert und fliessen in die Suche und Key-Word-Generierung ein. Soll z. B. ein Wikipedia-Artikel in das Netz integriert werden, kann ein Node mit dem Link angelegt werden und der Wikipedia-Artikel ist auffindbar und wird richtig vertagt. Es ist geplant, dass neben HTML- auch PDF- und DOCX-Dateien unterstützt werden.

Die Nachteile sind, dass immer nur ein Snapshot der Daten gemacht wird und es keine Möglichkeit gibt, dass die Datenquelle eine Aktualisierung triggern kann. Das System basiert somit auf einem Polling- und nicht auf einem Push-Mechanismus. So können Inhalte auch nicht automatisch integriert werden und als Informationsquellen kommen nur System in Frage, die von Grafit per HTTP erreichbar sind. Ein Vorteil ist, dass keine Probleme mit Berechtigungen auftreten, da diese nicht von Grafit verwaltet werden, sondern an die Datenquelle übertragen werden.

3.5 User-Interface-Design

Das User Interface wird wie auch beim IKC-Prototyp responsiv aufgebaut, um die Nutzung über mobile Geräte zu ermöglichen. Einen zentralen Teil der Interaktion mit Grafit ist die Suchfunktion. Daher wird das Suchfeld zentral platziert. Dazu wurden vor der Entwicklung UI Mockups, u. a. Abbildung 5, erstellt.

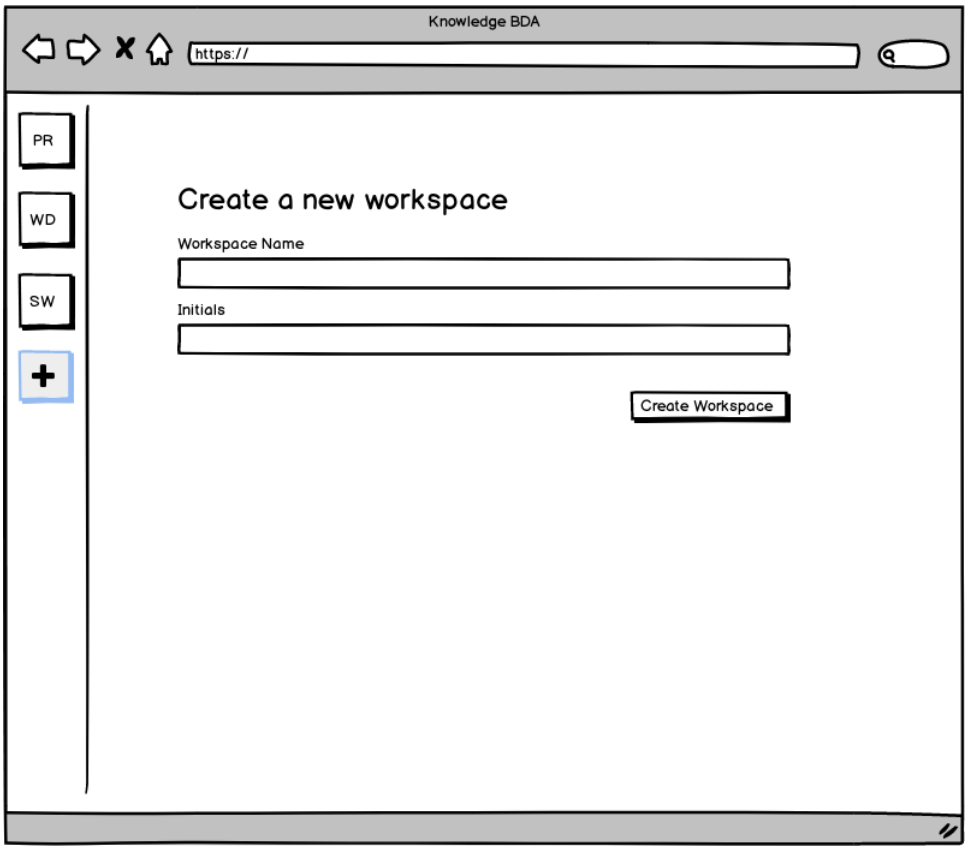


Abbildung 5: Grafit – Mockup User Interface (UI)

Die Workspace-Leiste auf der linken Seite wurde von Slack[9] inspiriert. Bei Geräten mit kleineren Bildschirmen (< 768px Breite) wird die Workspace-Leiste standardmässig ausgeblendet und kann durch einen Toggle-Knopf eingeblendet werden.

[9] https://slack.com

4 Implementierung

Dieses Kapitel beschreibt den Aufbau und die Funktionalität des entwickelten Produkts. In Kapitel 4.1 werden der generelle Aufbau der Applikation, das Front- und Backend sowie die REST-Schnittstelle dargelegt. Die nachfolgenden Kapitel behandeln anschliessend wichtige Komponenten, Module oder Aspekte der Umsetzung.

4.1 Systemübersicht

Wie in Abbildung 6 ersichtlich, ist die Applikation in ein React Frontend und ein Django Backend aufgeteilt. Diese beiden Bereiche kommunizieren über eine REST-Schnittstelle. Die Abbildung beinhaltet nur die wichtigsten Komponenten und Module der Implementation.
In den nachfolgenden Kapiteln wird weiter auf die Umsetzung und den Aufbau der einzelnen Teile eingegangen.

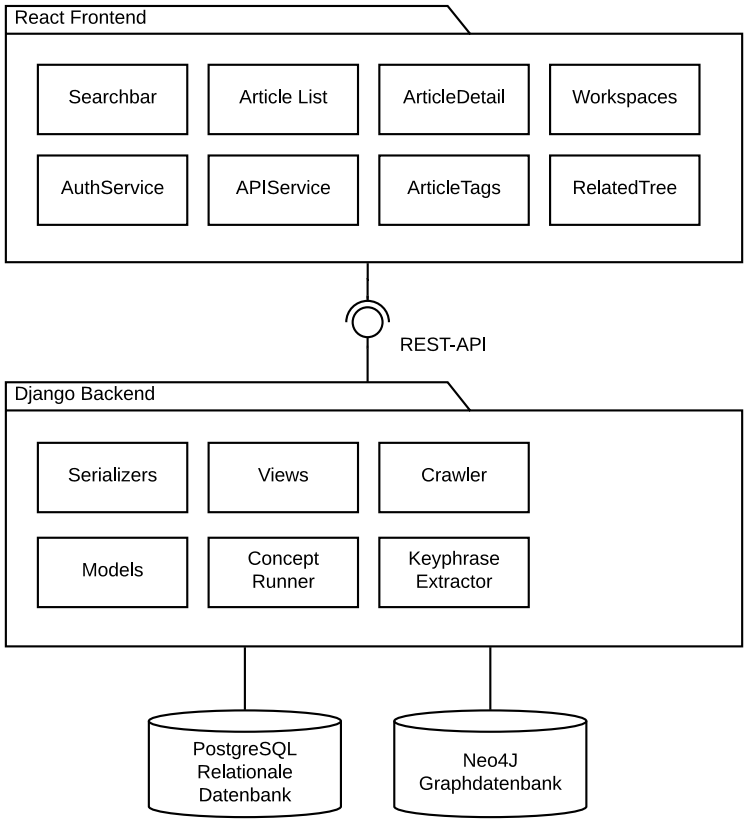


Abbildung 6: Systemübersicht

4.1.1 REST-Schnittstelle

Die zentrale Schnittstelle ist eine REST-API, die vom Django Backend angeboten wird und vom React Frontend konsumiert wird. Diese Abstrahierung ermöglicht, dass anderweitige Frontends für Grafit entwickelt werden können, ohne dass das Backend angepasst wird. So könnten zum Beispiel eine native mobile App oder eine Browser Extension, die mit dem Backend kommuniziert, entwickelt werden.

Um die Sicherheit der Schnittstelle zu garantieren, wurde eine Authentifizierung mit JSON-Web-Tokens (JWT) eingesetzt. Diese wird wie in Kapitel 3.4.3 beschrieben bereits durch das Django REST Framework (DRF) mitgeliefert und musste nicht selbst von Grund auf implementiert werden. Die Authentifizierung wurde so eingesetzt, dass alle Zugriffe auf die REST API (abgesehen vom Zugriff auf den Login-Endpunkt) authentifiziert erfolgen müssen. Die JWT laufen nach sieben Tagen ab und die Benutzer_In muss sich neu einloggen.

4.1.2 Frontend

Das React Frontend ist in React-Komponenten und Services aufgeteilt. Die React-Komponenten sind für die Darstellung und Funktionalität eines für den Benutzer sichtbaren Teils des Frontends verantwortlich.
Die Services sind Funktionalitäten, die im Hintergrund laufen und beispielsweise die Interaktion mit der REST-Schnittstelle abstrahieren oder die Authentifizierung des Benutzers abwickeln.

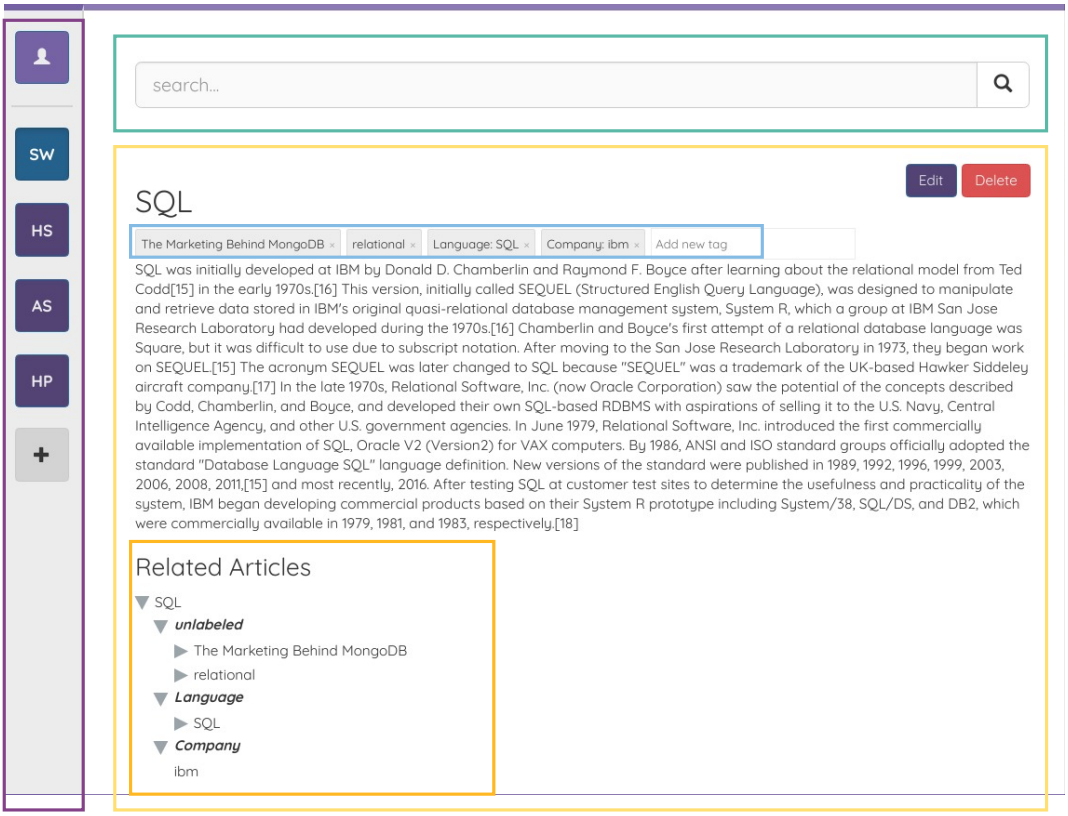


Abbildung 7: Übersicht Frontend

Die wichtigsten React-Komponenten sind:

- **ArticleList**
  Die ArticleList-Komponente stellt die Artikel in einem Workspaces in Listenform dar. Dabei werden eine verkürzte Version des Textes, die Überschrift und die wichtigsten verlinkten Artikel als Tags dargestellt. Um auch bei vielen Artikeln in einem Workspace eine hohe Performance zu wahren, wurde ein Paging implementiert, das nur die 25 zuletzt bearbeiteten Artikel von der REST-Schnittstelle abruft. Scrollt der Benutzer bis an das Ende der Liste, können mit einem Klick auf *Load More* die nächsten 25 Artikel geladen werden.

- **Searchbar**
  Diese Komponente ist für die Suche zuständig und in Abbildung 7 in grün eingezeichnet. Mit dieser Komponente kann Workspace-übergreifend nach Artikeln gesucht werden. Die Suchtreffer werden in einem Overlay eingeblendet. Mehr zur Implementation der Suche ist in Kapitel 4.4 ersichtlich.

- **Workspace**
  Die Workspace-Komponente, in Abbildung 7 in violett eingezeichnet, zeigt dem Benutzer eine seitliche Leiste an, auf der der Workspace ausgewählt werden kann. Der aktuelle Workspace wird zudem farblich hervorgehoben.

- **ArticleDetail**
  Diese Komponente (in Abbildung 7 in Gelb umrandet) ist für die Detailansicht eines einzelnen Artikels zuständig. Sie zeigt den gesamten Text des Artikels an und bietet die Optionen, den Artikel zu bearbeiten oder zu löschen. Zudem verwendet sie die beiden Komponenten *RelatedTree* und *RelatedTags*.

- **RelatedTags**
  Die RelatedTags-Komponente erlaubt es dem User, alle verbundenen Artikel als Tags einzusehen (in Abbildung 7 in blau markiert). Neue Verbindungen zu einem Artikel können erstellt werden, indem in der Eingabemaske *Add new Tag* nach diesem gesucht wird. Es erscheint ein Overlay mit Autovervollständigung. Falls noch kein Artikel mit diesem Namen existiert, kann dieser direkt erstellt werden. Zudem kann den Verbindungen ein Label gegeben werden, um diese zu benennen.

- **RelatedTree**
  Diese Komponente wurde aus dem IKC-Prototyp übernommen und zeigt die mit diesem Artikel verbundenen Elemente in einer aufklappbaren Baumstruktur an (ersichtlich in Abbildung 7 in orange). Die Verbindungen werden im Baum nach ihrem Label gruppiert.

Zudem wurden zwei Services implementiert:

- **APIService**
  Der APIService abstrahiert die Interaktion mit der REST API und bietet anderen Services oder Komponenten Funktionalitäten wie *getArticles*, *updateArticle* und *getSearchResults*.

- **AuthService**
  Dieser Service wickelt die Authentifizierung des Benutzers mittels JSON-Web-Tokens ab. Das Token wird anschliessend in den LocalStorage des Browsers gespeichert. Über den AuthService können andere Komponenten und Services den Login-Status überprüfen.

4.1.3 Backend

Das Backend ist in verschiedene Module und Komponenten aufgeteilt. Die wichtigsten Teile sind folgende:

- **Models**
  Das Model-Modul beinhaltet die Datenklassen, die für die Interaktion mit den Datenbanken verwendet werden. Der genaue Mechanismus und eine Beispielklasse sind in Kapitel 4.2 ersichtlich.

- **Views**
  Mit Django Views werden die einzelnen REST-Endpunkte implementiert.

- **Serializers**
  Die Serializers definieren, wie die Daten aus den Model-Klassen transformiert werden, damit sie anschliessend von einer View verwendet werden können.

- **Keyphrase Extractor/Concept Runner**
  Der Keyphrase-Extraction-Algorithmus und das Concept-Runner-Modul sind in Kapitel 4.4 dokumentiert.

- **Crawler**
  Der Crawler ist Teil der Cloud-Integration und wird in Kapitel 4.5 abgehandelt.

- **Backend**
  Im Modul *Backend* werden die URLs der Applikation gesetzt und andere Konfigurationen, wie Datenbankverbindungen, definiert.

- **Migrations**
  In diesem Modul sind alle Datenbankmigrationen hinterlegt. Mithilfe der Migrations können Änderungen an der Datenbank reproduzierbar im Code abgelegt werden. Wie diese erstellt werden, wird in Kapitel 4.2 beschrieben.

4.2 Persistenzlösung

Wie in Kapitel 3.4.4 beschrieben, wurde beschlossen, von einer JSON-Datei als Persistenzlösung wegzukommen und PostgreSQL einzusetzen. Das Datenschema wurde mit einem *Code-First*-Ansatz entwickelt. Zuerst wurden Model-Klassen in Python implementiert. Ein Beispiel eines solchen Models ist in Listing 2 zu sehen.

```
1  class Workspace(models.Model):
2      name = models.CharField(max_length=250)
3      initials = models.CharField(max_length=2)
4      users = models.ManyToManyField(User)
```

Listing 2: Django Workspace Model

Anschliessend wurden, mithilfe des makemigration-Befehls (Listing 3) die entsprechenden SQL-Statements zu den Models generiert.

```
1  docker-compose -f docker-compose.dev.yml run --rm backend
   ↪  ./manage.py makemigrations
```

Listing 3: Django Admin – Generieren der SQL-Statements

Die Interaktion mit der Datenbank erfolgt hauptsächlich über den Django-ORM, spezielle Funktionen wie die Volltextsuche (Kapitle 4.4) wurden jedoch direkt in SQL umgesetzt.

Das entwickelte Datenbankschema ist in Abbildung 8 ersichtlich.

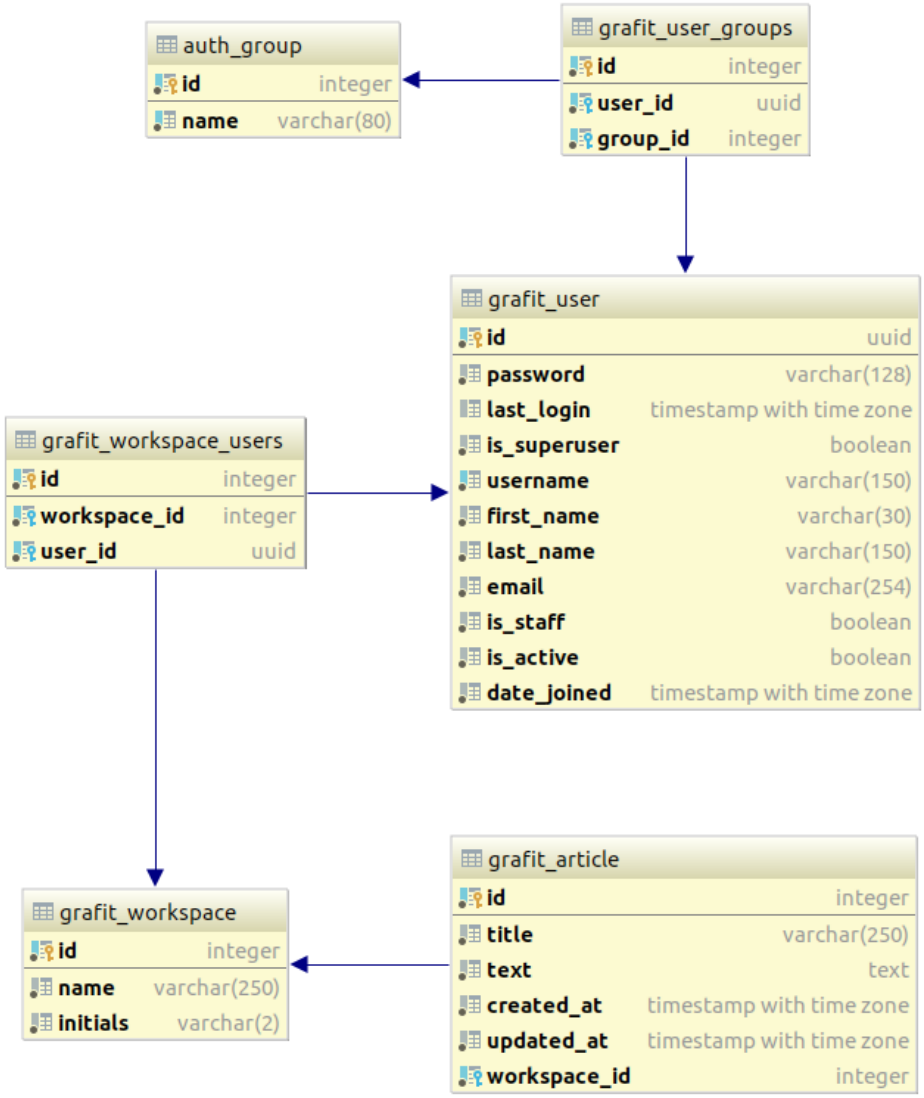


Abbildung 8: Datenbankschema – PostgreSQL

Wie in Kapitel 3.4.6 beschlossen, wird neben der PostgreSQL-Datenbank Neo4j als Graphdatenbank eingesetzt. Um die Entwicklung für beide Datenquellen konsistent zu halten, wird auch bei der Graphdatenbank mit einem *Code-First*-Ansatz gearbeitet. Zu diesem Zweck wird Neomodel[10] als Object-Graph-Mapper (OGM) eingesetzt. So konnten auch hier die Models zuerst in Python implementiert werden. Wichtig ist hierbei, dass nicht nur die Knoten (Listing 4), sondern auch die Kanten (Listing

[10]https://neomodel.readthedocs.io/en/latest/index.html

5) modelliert werden müssen.

```
1  class GraphArticle(StructuredNode):
2      uid = UniqueIdProperty()
3      name = StringProperty()
4      related = Relationship('GraphArticle', 'RELATED',
       ↪  model=ArticleRel)
```

Listing 4: Neomodel GraphArticle Model

```
1  class ArticleRel(StructuredRel):
2      created_at = DateTimeProperty(default=lambda: datetime.now())
3      tf_idf = FloatProperty()
4      hidden = BooleanProperty(default=False)
5      label = StringProperty()
```

Listing 5: Neomodel ArticleRel Beziehung

Wie in Code Listing 5 in Zeile zwei ersichtlich, können auch dynamische Defaultwerte mithilfe einer Lambda-Funktion gesetzt werden. In diesem Fall wird das created_at-Attribut auf den Aktuellen Timestamp gesetzt.

Das auf der Graphdatenbank persistierte Datenmodell (Abbildung 9) ist bewusst simpel gehalten. Es gibt lediglich die Entitätsmenge *GraphArticle*, die über die gerichtete Beziehungsmenge *ArticleRel* mit sich selbst verbunden werden kann.

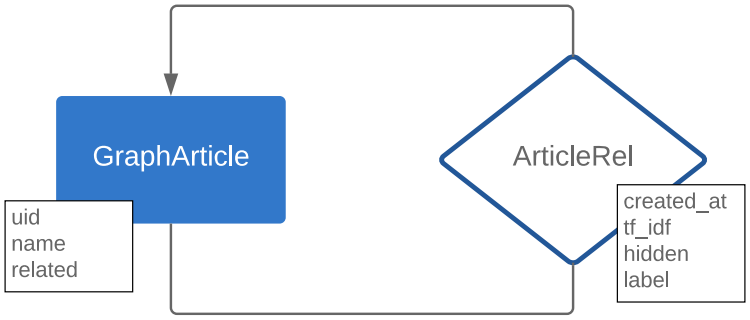


Abbildung 9: Graphmodell – Neo4j

Durch den Einsatz von Django-ORM und Neomodel können die beiden Datenquel-

len kombiniert werden. Dies wurde beim Article Model (ersichtlich in Anhang C.2) über ein Property-Attribut bewerkstelligt. Das Property-Attribut kann gleich wie die restlichen Attribute abgefragt werden, die Daten kommen in diesem Fall jedoch nicht aus der PostgreSQL-Datenbank, sondern aus Neo4j.

4.3 Knowledge-Graph

Die Generierung des Knowledge-Graphen besteht aus zwei Teilen. Einerseits ist es nötig, eine Keyphrase Extraction basierend auf den einzelnen Textinhalten durchzuführen. Hierfür wird der in Kapitel 2.2.7 vorgestellte TF-IDF-Algorithmus übernommen. Anschliessend wird basierend auf diesen Daten ein Knlowledge-Netz erstellt, das danach auf der Neo4j-Datenbank persistiert wird. Das Zusammenspiel dieser beiden Komponenten, *Concept Runner* und *Keyphrase Extractor*, und den beiden Datenbanken wird in Abbildung 10 visualisiert.

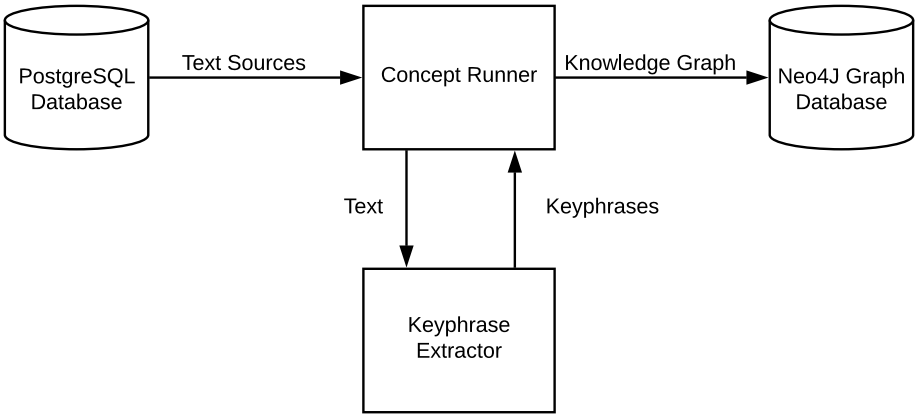


Abbildung 10: Datenfluss der Generierung des Knowledge Graphs

In den folgenden Unterkapiteln wird erläutert, wie die Schlüsselfunktionen der einzelnen Komponenten integriert und umgesetzt wurden.

4.3.1 Integration Keyphrase Extraction

Wie in Kapitel 3.3 beschrieben, wird der Keyphrase-Extraction-Algorithmus aus der Bachelorarbeit *Automatisierte Generierung von plattformübergreifenden Wissensnetzwerken mit Metadaten und Volltextindexierung* von TypeScript nach Python portiert.

Die Keyphrase Extraction besteht aus verschiedenen Schritten, die in Abbildung 11 visualisiert sind und in den folgenden Abschnitten weiter erläutert werden.

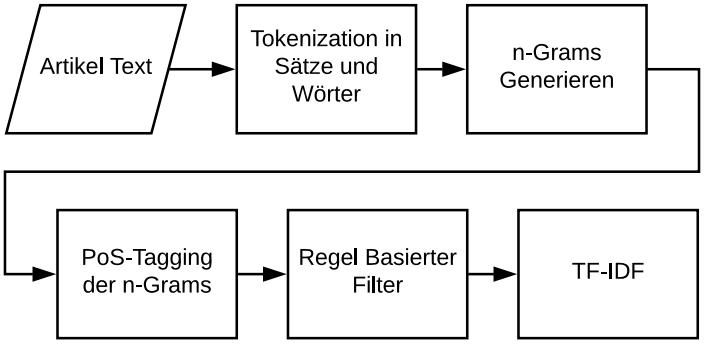


Abbildung 11: Keyphrase Extraction Pipeline

Tokenization

Als erster Schritt wird der Text in Sätze und Wörter zerstückelt. Dieser Prozess wird Tokenization genannt und ist in Kapitel 2.2.1 beschrieben.
In der ursprünglichen Implementation mit TypeScript wurde dies mithilfe der split-Funktion und einem Regex bewerkstelligt. Um die Implementation in Python etwas robuster und erweiterbar zu gestalten, wurden die beiden Softwarebibliotheken Natural Language Toolkit (NLTK) und Textblob[11] eingesetzt. Textblob ist eine Wrapper-Bibliothek, die den Umgang mit NLTK vereinfacht.

Eine Tokenization nach Sätzen und anschliessend nach Wörtern kann mithilfe dieser Bibliotheken folgendermassen umgesetzt werden:

```
1  tb = TextBlob(text)
2  sentences = tb.sentences
3  for sentence in sentences:
4      words = sentence.words
5      # handle words
```

Listing 6: Tokenization mit Textblob

[11] https://github.com/sloria/TextBlob

Generierung n-Grams

Wie es auch im IKC-Prototyp der Fall war, sollen n-Grams mit einem n von eins bis drei generiert werden. Die n-Grams werden basierend auf den Sätzen erstellt, die mit dem Tokenizer extrahiert wurden. So werden keine n-Grams erstellt, die über eine Satzgrenze hinausgehen.
Das Generieren der n-Grams wurde mit der Textblob-ngrams-Funktion, einer Flatten-Funktion und einer Python-Listen-Abstraktion umgesetzt.

```
1 def flatten(l): return [item for sublist in l for item in sublist]
2
3 ngrams = flatten([sentence.ngrams(n=1) + sentence.ngrams(n=2) +
  ↪  sentence.ngrams(n=3) for sentence in blob.sentences])
```

Listing 7: Generierung n-Grams

Filter mit PoS Tagging

Nach dem Generieren der n-Grams müssen diese gefiltert werden. Dies wird mithilfe der in Kapitel 2.2.6 beschriebenen Methode des Part-of-Speech Tagging (PoS Tagging) bewerkstelligt.

Der hierbei verwendete Tagger ist der NLTK Treebank Tagger, der mit den Daten des *Penn Treebank Project*[12] trainiert wurde. Eine Treebank ist ein grosser Korpus von Text, der auf die syntaktische Struktur untersucht und annotiert wurde.

Die n-Grams werden anhand von drei Regeln gefiltert, die aus der Bachelorarbeit *Automatisierte Generierung von plattformübergreifenden Wissensnetzwerken mit Metadaten und Volltextindexierung* übernommen werden.

[12] https://web.archive.org/web/19970614160127/http://www.cis.upenn.edu/ treebank/

Regel Nr.	**Beschreibung**
R1	Jede Keyphrase muss ein Nomen oder ein fremdes Wort beinhalten.
R2	Die Keyphrase muss entweder mit einem Nomen, einem Adjektiv oder einem fremden Wort beginnen.
R3	Die Keyphrase muss entweder mit einem Nomen, einem Adjektiv oder einem fremden Wort enden.

Tabelle 1: PoS n-Gram Filterregeln

Im nachfolgenden Code (Listing 8) ist die Implementation der Filter ersichtlich. Auf Zeile 9 werden die PoS Tags des n-Grams ermittelt. Anschliessend werden in Zeile 12 die Regel R1 und in Zeile 17 die Regeln R2 und R3 geprüft. Nur wenn alle Regeln eingehalten sind, wird das n-Gram zur ngramsFiltered-Liste hinzugefügt und weiterverarbeitet.

```
nouns = ['NN', 'NNS', 'NNP', 'NNPS', 'FW']
nounOrAdj = ['JJ', 'JJR', 'JJS', 'NN', 'NNP', 'NNPS', 'NNS', 'FW']
ngramsFiltered = []

for ngram in ngrams:
    ngramText = ' '.join(ngram)

    # filter based on pos tagging
    pos_tags = [result[1] for result in tb(ngramText).tags]

    # check if ngram contains a noun or unknown word
    if any(elem in nouns for elem in pos_tags):

        # a valid keyphrase has to start with a noun, adjective or
        # unknown word and also has to end with a noun, adjective
        # or unknown word
        if pos_tags[0] in nounOrAdj and pos_tags[-1] in nounOrAdj:
            ngramsFiltered.append(ngramText)

logger.info(f"filtered {len(ngrams)} ngrams down to
↪ {len(ngramsFiltered)}")
```

Listing 8: PoS-Tag-Filter

TF-IDF

Die eigentliche Extraktion der Keyphrases wird mit dem in Kapitel 3.3 vorgestellten TF-IDF-Algorithmus (Formel 1) umgesetzt. Hierfür müssen zuerst die für den TF-IDF-Algorithmus relevanten Metriken berechnet werden. Die Metriken wurden aus der Bachelorarbeit *Automatisierte Generierung von plattformübergreifenden Wissensnetzwerken mit Metadaten und Volltextindexierung* übernommen.

Metrik	**Beschreibung**
numDocs	Anzahl Dokumente im Korpus
termFreq	Häufigkeit einer Keyphrase innerhalb des Korpus
length	Anzahl Wörter des zu analysierenden Dokuments
termDocFreq	Häufigkeit der Keyphrase innerhalb des Dokuments

Tabelle 2: Metriken für TF-IDF

Der TF-IDF-Algorithmus (Formel 1) kann in mehrere Teile zerlegt werden. Diese sind unterhalb ersichtlich, zusammen mit der von uns entwickelten Python-Implementation.

$$tf = termDocFreq \tag{2}$$

```
1  def tf(self, word, blob):
2      return blob.words.count(word)
```

Listing 9: Berechnung Term Frequency (TF)

$$idf = 1 + \log\left(\frac{numDocs}{1 + termFreq}\right) \tag{3}$$

```
1  def idf(self, word, corpus):
2      return math.log(len(corpus) / (1 + self.n_containing(word,
       ↪  corpus)))
3
4  def n_containing(self, word, corpus):
5      return sum(1 for blob in corpus if word in blob.words)
```

Listing 10: Berechnung Inverse Document Frequency (IDF)

$$lengthNorm = \frac{1}{length} \tag{4}$$

```
def norm(self, blob):
    return 1 / len(blob.words)
```

Listing 11: Berechnung lengthNorm

Schlussendlich wird aus den einzelnen Teilen der TF-IDF Score der Keyphrase berechnet.

$$tfidf = tf * idf * lengthNorm \tag{5}$$

```
def tfidf(self, word, blob, corpus):
    return self.tf(word, blob) * self.idf(word, corpus) *
    ↪ self.norm(blob)
```

Listing 12: Berechnung TF-IDF

Der TF-IDF Score wird für alle Keyphrases berechnet. Anschliessend werden mithilfe eines Schwellenwerts die höchsten n Keyphrases zurückgegeben. Die Konzeptextraktionskomponente akzeptiert hierfür einen Parameter, über den sich dieses n konfigurieren lässt.

Strategy Pattern

Um die Wiederverwendbarkeit der Konzeptextraktion zu erhöhen, wurde diese in einem eigenen Python-Modul umgesetzt. Die Extraktionslogik wurde anschliessend mithilfe eines Strategy-Patterns (ersichtlich in Abbildung 12) implementiert. So kann zu einem späteren Zeitpunkt mit minimalen Anpassungen am Rest des Systems eine andere Extraktionsstrategie entwickelt werden.

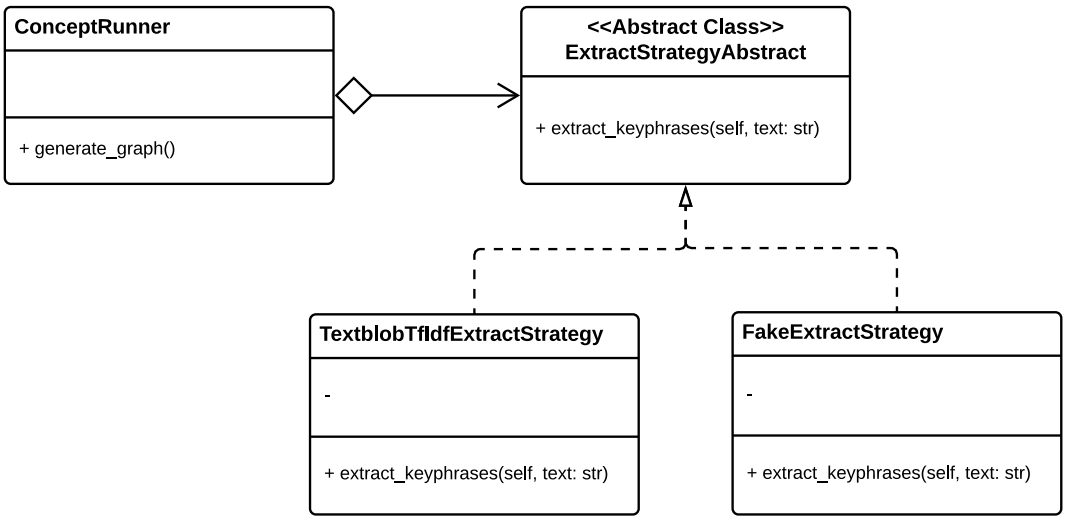


Abbildung 12: Concept Extraction – Strategy Pattern

Die abstrakte Klasse wurde folgendermassen umgesetzt:

```
1  class ExtractStrategyAbstract(object):
2      """Abstract strategy class for the extract method."""
3
4      __metaclass__ = abc.ABCMeta  # define as absctract class
5
6      @abc.abstractmethod
7      def extract_keyphrases(self, text: str) -> List[Dict]:
8          """ Required Method.
9              Expects a text and extracts keyphrases from this text.
10             Returns a list of dicts which contain the extracted
11             keyphrases.
12         """
```

Listing 13: Abstrakte Klasse – ExtractStrategy

Durch die Abstrahierung der Konzeptextraktion in ein eigenes Python-Modul kann diese sehr einfach verwendet werden:

```
1 tfidf_extractor = TextblobTfIdfExtractStrategy()
2 keyphrases = tfidf_extractor.extract_keyphrases(article.text)
```

Listing 14: Anwendung implementierter Keyphrase Extraction

4.3.2 Generierung Knowledge-Graph

Die extrahierten Keyphrases werden im IKC-Prototyp zum Taggen der Artikel verwendet. Diese Funktionalität wird erweitert, um automatisch ein Wissensnetzwerk zu generieren.
Wie in Abbildung 10 ersichtlich, verwendet die Komponente *Concept Runner* zur Erstellung des Wissensnetzwerks die Keyphrases, die mit der in Kapitel 4.3.1 beschriebenen Methode extrahiert wurden. Das Netzwerk wird anschliessend auf der Neo4j-Datenbank persistiert.
Die grundsätzliche Idee ist, dass ein neu erstellter oder editierter Artikel analysiert wird und die entsprechenden Keyphrases extrahiert werden. Anschliessend werden die Artikel, die die extrahierte Keyphrase als Titel haben, mit dem analysierten Artikel auf dem Wissensnetzwerk verbunden. Falls noch kein Artikel mit diesem Titel existiert, wird er automatisch erstellt.
Eine Annahme hierfür ist, dass alle Artikel in einem Workspace unterschiedliche Titel haben. Der Titel eines Artikels muss also pro Workspace einzigartig sein.
Auf der Verbindung zwischen dem analysierten Artikel und den neu verbundenen Artikeln wird der TF-IDF Score gespeichert. Ein Ausschnitt des generierten Graphen, die auf der Neo4j-Datenbank persistiert wird, ist in Abbildung 13 ersichtlich.
Der Code dieser Funktionalität kann auf GitHub abgerufen werden.[13]

[13] https://github.com/grafit-io/grafit/blob/2b920c570a80f221b9b0e2386f628b5a980e8df1/backend/grafit/concept_runner.py#L11

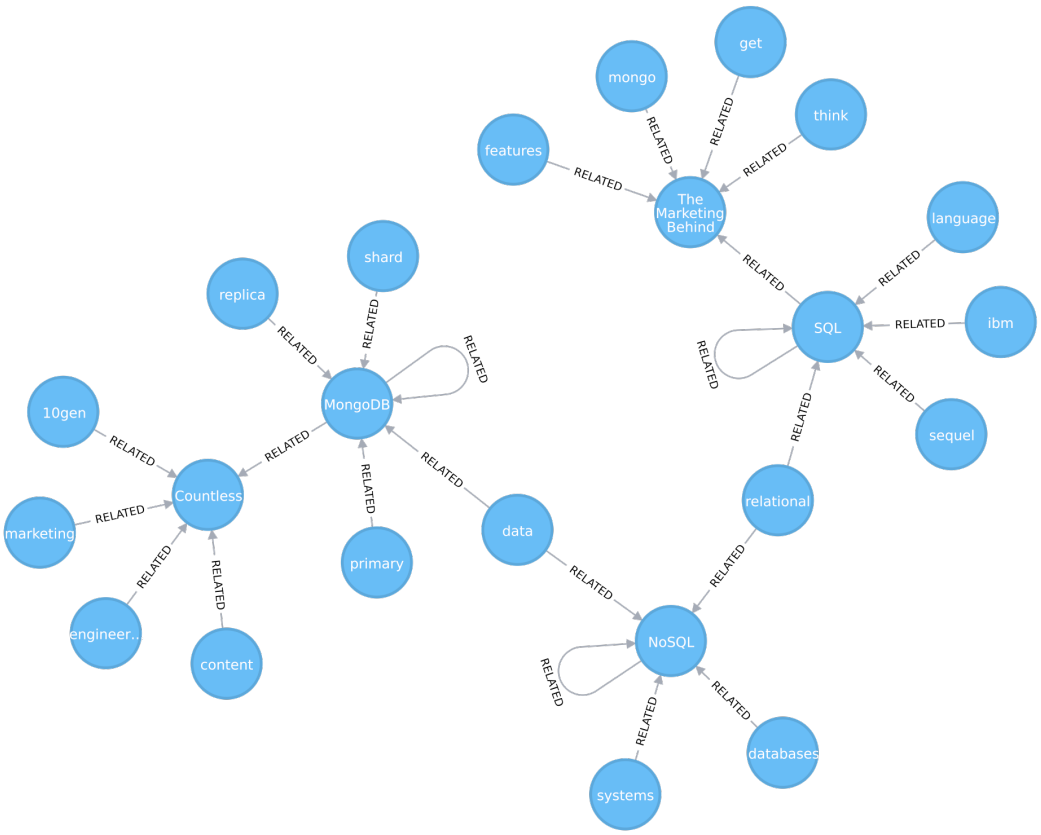


Abbildung 13: Generiertes Wissensnetzwerk in Neo4j

Anschliessend kann der Benutzer selbst neue Verbindungen im Wissensnetzwerk erstellen oder bestehende mit einem Label versehen. Die Verbindungen werden anschliessend nach ihrem Label gruppiert. Diese Funktionalität wurde aus dem IKC-Prototyp übernommen. Wie die Benutzerschnittstelle zu dieser Funktionalität aufgebaut wurde, ist im Benutzerhandbuch[14] ersichtlich.

[14]https://grafit.readthedocs.io/en/latest/userdoc/createarticle/#create-a-new-connection-between-articles

4.4 Search

Um eine Volltextsuche über alle Datenquellen anzubieten, wird ein zentraler Suchindex angelegt. Für jeden Node wird darin ein gewichteter Vektor abgespeichert. Der Vektor beinhaltet den Text des Titels, des Inhalts und den Webcontent. Um dem Titel eine höhere Relevanz beim Berechnen der Suchresultate zu geben, wird der Titel mit einem Faktor von 1, der Inhalt mit einem Faktor von 0.4 und der Webcontent mit einem Faktor von 0.1 gewichtet (Redmond et al., 2012, p. 44). Beim Erstellen des Vektors werden Stopwords der englischen Sprache entfernt, um den Signal to Noise Ratio der Daten zu erhöhen und somit den Index zu verkleinern. PostgreSQL 10.5 unterstützt neben Englisch noch vierzehn weitere Sprachen, u. a. Deutsch. Eine mögliche Erweiterung wäre, die Sprache des Textes zu erkennen und bei der Aufnahme in den Index sprachspezifisch die Stop-Words zu entfernen. Sprache ist oft mehrdeutig und exaktes Matching erzeugt viele False Negatives. Deshalb wird zusätzlich ein Stemming der Wörter durchgeführt. Welche Auswirkungen das Stemming auf eine Textfolge hat, ist im Listing 15 ersichtlich.

```
1 SELECT to_tsvector('Create a search index with PostgresSQL');
2 'creat':1 'index':4 'postgressql':6 'search':3
```

Listing 15: Stemming mit to_tsvector

Für jeden Node wird neben dem Vektor auch die ID und der Titel in einer Materialized View gespeichert. Um ein schnelles Durchsuchen der Vektoren zu ermöglichen, wurde darauf ein GIN-Index berechnet (Redmond et al., 2012, p. 47-48).

PostgreSQL bietet die Möglichkeit, Views nebenläufig zu aktualisieren. Das verhindert einen Lock während des Aktualisierungsprozesses mit dem Nachteil, dass inkonsistente Lesezugriffe auftreten können (Group, 2018, p. 1646). In diesem Fall zeigt sich das jedoch als unproblematisch.

Der Suchindex kann mit dem Befehl to_tsquery() durchsucht werden. Alle Wörter der Suchanfrage werden mit einer logischen OR-Verknüpfung verbunden.

Um auch Suchanfragen mit Rechtschreibfehlern korrekt zu beantworten, wird der Suchterm weiterverarbeitet, sofern keine Resultate mit einem direkten Match gefunden werden, siehe Listing 16.

```
results = Search.getSearchResult(searchTerm)
if(len(results) < 1):
    searchWords = searchTerm.split("|")
    for word in searchWords:
        similarWord = Search.findSimilarWord(word)
        if similarWord is not None:
            searchTerm += '|'
            searchTerm += similarWord
    results = Search.getSearchResult(searchTerm)
return results
```

Listing 16: Suche nach ähnlichen Wörtern

Dabei wird auf eine Lexemtabelle zugegriffen. Wie der Suchindex wurde auch diese Tabelle als Materialized View mit nebenläufiger Aktualisierung implementiert. Diese beinhaltet den Stamm aller vorhandenen Wörter. Um ähnliche Wörter zu finden, wird die Distanz zu einem Wort berechnet. Dabei werden beide Wörter in Trigrame zerlegt, alle mehrfach vorkommenden Trigrame werden entfernt. Für die Trigrame wird der Wert $MatchingTrigram/TotalDistinctTrigram$ berechnet. Dabei wird die Ähnlichkeit in einem Wertebereich von $[0, 1]$ berechnet. Es werden nur der beste Match und Resultate mit einem Wert von mindestens 0.5 weiterverarbeitet. Dieser Threshold wurde durch Trial-and-Error bestimmt. Die so gefundenen ähnlichen Wörter werden an den Search-Term angefügt und erscheinen dementsprechend in den Resultaten.

4.5 Integration Cloud Service

Die Cloud-Integration ermöglicht es, extern Informationsquellen in das Wissensnetzwerk aufzunehmen. Wird ein Artikel erstellt oder bearbeitet, werden Titel und Text nach URLs durchsucht. Alle gefundenen URLs werden gecrawlt. Dazu wird die Python-HTTP-Bibliothek requests[15] verwendet. Sie vereinfacht die Implementierung von Interaktionen über das HTTP-Protokoll. Es gibt bspw. Helferfunktionen für HTTP Response Codes und eine simple Lösung, um Timeouts zu setzen.

[15] http://python-requests.org/

Die HTTP Responses werden auf das Header-Attribut *Content-Type* untersucht. Es wurde eine Implementation für den MIME Type text/html entwickelt. Mit dem Python Packet html2text[16] kann HTML zu Markdown konvertiert werden. Der so extrahierte Text wird dem Artikel als web_content hinterlegt und fliesst u. a. in den Search-Index und das Tagging der Artikel ein.

Es wäre möglich, Parser für weitere MIME Types zu implementieren. Um den Inhalt aus PDF oder Microsoft Word-Dateien (.docx) zu extrahieren, gibt es Bibliotheken wie Textract[17]. Weitere Parser können dank einer Implementation mit einem Strategy Pattern einfach eingebaut werden.

4.6 Build Pipeline

Autor Martin Fowler beschreibt das Konzept einer Continous Integration Pipeline in Kombination mit automatisierten Tests als die beiden Kernaspekte, die agile Softwareentwicklung ermöglichen (Fowler, 2000).

Laut Agile Manifest hat eine frühe Continuous Delivery der Software die höchste Priorität (Beck et al., 2001).

Continuous Integration und Continuous Delivery sind ein Kern von agiler Softwareentwicklung. Die Codequalität kann laufend überprüft werden und es ist möglich, die Software immer in einem lieferbaren Stand zu haben. Diese Pipeline muss automatisiert bei jeder Änderung am Code durchlaufen.

Es wurde beschlossen, für dieses Projekt einen Build Server des Saas-Dienstleisters Travis CI[18] einzusetzen. Die Build-Infrastruktur kann für Open-Source-Projekte kostenlos genutzt werden und hat sich für auf GitHub gehostete Projekte als Industriestandard durchgesetzt.

Mithilfe der Konfigurationsdatei .travis.yml[19] können die einzelnen Schritte des Builds definiert werden, die in Abbildung 14 veranschaulicht werden.

[16] https://github.com/Alir3z4/html2text/
[17] https://textract.readthedocs.io/
[18] https://travis-ci.com/
[19] https://github.com/grafit-io/grafit/blob/master/.travis.yml

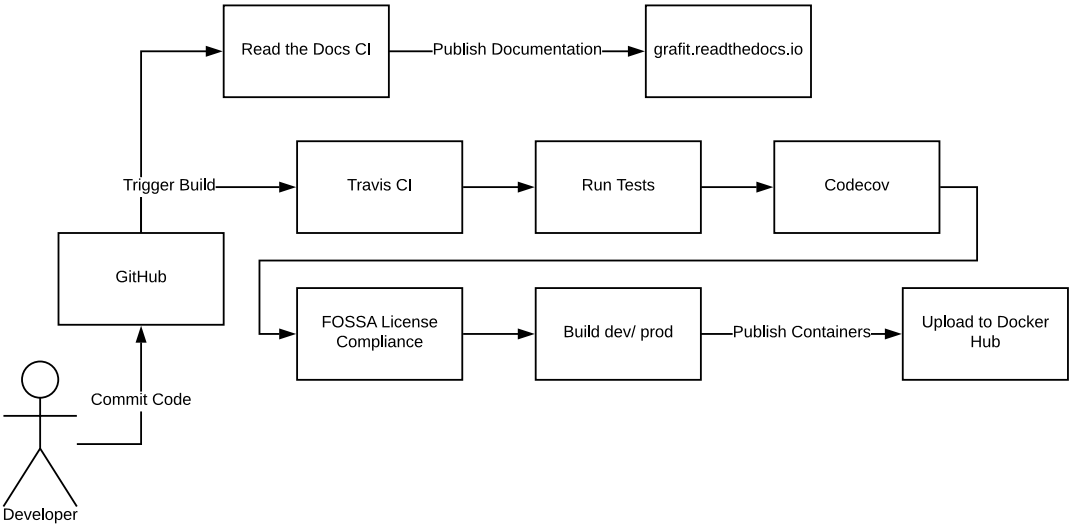


Abbildung 14: Build Pipeline

Der Travis Buildserver führt als erstes alle Tests des Backends und des Frontends aus. Danach wird gemessen, wie gross die momentane Testabdeckung ist. Mehr dazu ist in Kapitel 4.7.3 zu finden. Nachdem alle Tests erfolgreich durchgelaufen sind, wird überprüft, ob die Lizenzbestimmungen aller verwendeten Codefragmente eingehalten sind. Auf das Lizenzmanagement wird in Kapitel 4.7.4 weiter eingegangen. Zuletzt werden die Entwicklungs- und Produktivcontainer erstellt und auf Docker Hub[20] publiziert.

Wie in Abbildung 14 ersichtlich, wird bei einem Commit nicht nur der Build auf dem Travis Buildserver angestossen, sondern auch der Continuous-Integration-Server von Read the Docs[21]. Read the Docs ist eine Dokumentationsplattform, auf der die Benutzer- und Entwicklerdokumentation von Markdown zu einer HTML-Webseite konvertiert und anschliessend gehostet werden.

[20] https://hub.docker.com/u/grafitio/
[21] https://readthedocs.io

4.7 Qualitätssicherung

Wie in den Randbedingungen des Projekts (Kapitel 1.5) festgelegt, soll die Applikation qualitativ und erweiterbar aufgebaut sein. Um die Qualität des Projekts laufend zu prüfen, wurden diverse Methoden zur Qualitätssicherung eingesetzt. Die Key Features wurden mit Unit- und Integrationstests verifiziert. In der Open Source Community etablierte Technologien wurden eingesetzt, um die Testabdeckung und Lizenz-Compliance zu garantieren.

4.7.1 Unit-Tests

Unit-Tests verifizieren die Korrektheit von einzelnen Funktionen eines Programms. Dabei wird darauf geachtet, möglichst keine Abhängigkeiten von anderen Programmteilen oder Ressourcen zu haben, um nur eine einzelne Funktion testen zu können.
Im Backend von Grafit wurden gezielt Unit-Tests eingesetzt, um die Funktionen einzelner Komponenten zu testen. Folgender Test überprüft beispielsweise, ob die Keyword Extraction die korrekte Anzahl Rückgabewerte liefert.

```
1  def test_tfidf_strategy_nr_keywords(self):
2      keyword_extractor =
       ↪  TextblobTfIdfExtractStrategy(loadFromDB=False)
3      keywords = keyword_extractor.extract_keyphrases(self.text,
       ↪  top_n_words=10)
4      assert len(keywords) == 10
```

Listing 17: Unit-Test – Keyword Extraction

4.7.2 Integrationstests

Im Gegensatz zu Unit-Tests, die einen kleineren Scope besitzen, wird bei einem Integrationstest eine ganze Funktion der Business-Logik verifiziert. Dabei wird das Zusammenspiel verschiedener Komponenten des Systems getestet. Integrationstests sind oft von externen Ressourcen wie Datenbanken abhängig. Es ist auch möglich, dass solche Tests über die API der Applikation erfolgen und somit alle Komponenten, von API bis Persistenzlayer, testen.
Ein Beispiel eines solchen Integrationstests ist in Anhang C.1 ersichtlich. Dieser Test

verifiziert, dass ein User nicht auf Daten Zugriff hat, die sich in einem fremden Workspace befinden.

4.7.3 Testabdeckung

Bei jedem Commit in das GitHub Repository wird ein Build ausgeführt, der unter anderem auch die Testabdeckung misst. Mit dem Tool Codecov[22] wird die Testabdeckung anschliessend analysiert. Sollte diese unter einen gewissen Schwellwert fallen, wird eine entsprechende Warnung angezeigt.

Eine hohe Testabdeckung garantiert keinesfalls, dass das Programm keine Fehler mehr beinhaltet. Sie kann jedoch als Metrik für die Gesundheit einer Codebasis verwendet werden.

4.7.4 Lizenzmanagement und Compliance

Eines der grössten Probleme in der heutigen Softwareentwicklung, die stark auf diversen Open Source Frameworks und Libraries aufbaut, ist, die Lizenzen der einzelnen Komponenten unter Kontrolle zu behalten. Hierzu wird FOSSA[23] verwendet. FOSSA ist ein Tool, das es ermöglicht, Lizenzaudits und Schwachstellen-Scans auf Basis des Projekt-Source-Codes durchzuführen. Nach jedem Build wird ein Bericht (Abbildung 15) erstellt, der eine Warnung anzeigt, falls ein Stück Code verwendet wird, das sich nicht mit der Lizenz des Projekts vereinbaren lässt. So kann garantiert werden, dass keine Lizenzverletzungen entstehen und die Software für Firmen attraktiver ist, da sie sicher sein können, dass keine rechtlichen Probleme bezüglich Lizenzierung der Software auf sie zukommen.

[22] https://codecov.io/
[23] https://fossa.com/

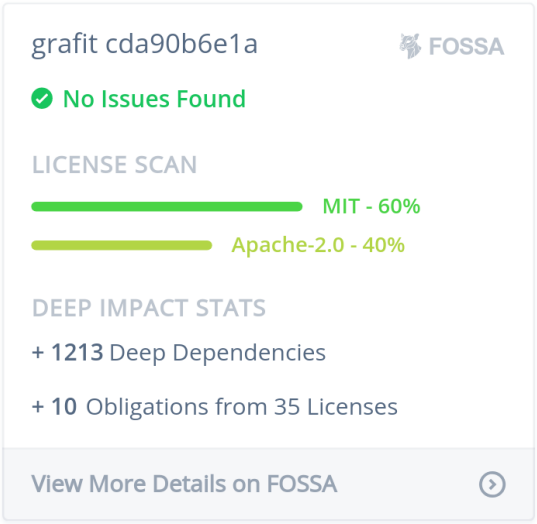


Abbildung 15: FOSSA – Übersicht verwendeter Lizenzen

4.8 Verteilung

Die Verteilung der Software erfolgt in Form von Dockercontainern, die über Docker Hub unter `https://hub.docker.com/u/grafitio` verfügbar gemacht werden.

Ein komplettes Deployment der Applikation beinhaltet fünf Container, die in Abbildung 16 ersichtlich sind.

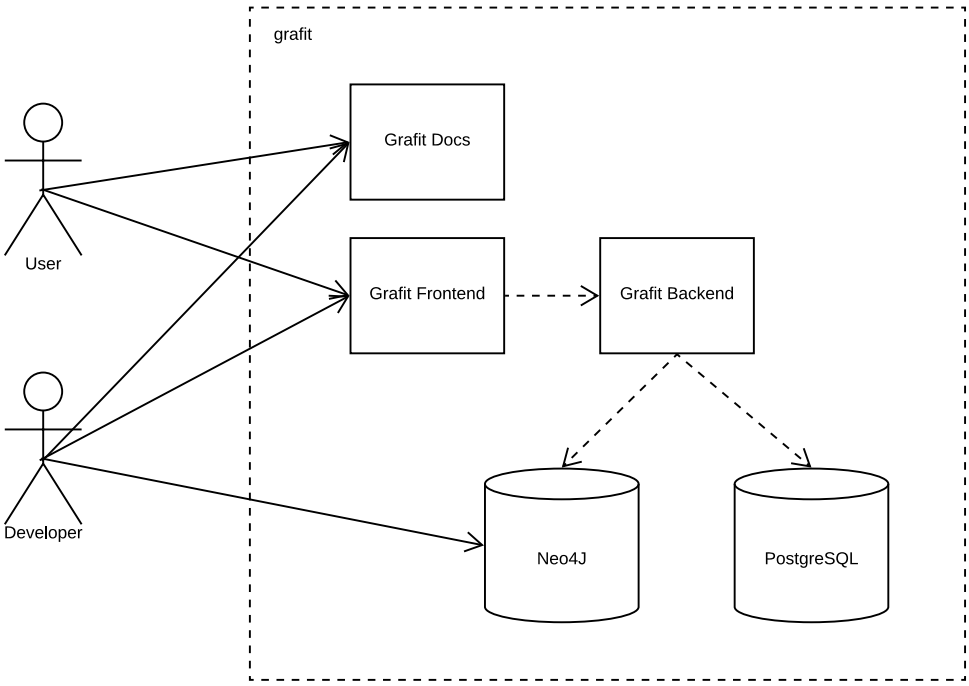


Abbildung 16: Docker Container

Die Orchestrierung der Container wird über das docker-compose[24] Tool gelöst. Es existieren zwei docker-compose Konfigurationsdateien, eine für den produktiven Einsatz und eine für die lokale Entwicklung. Die produktive Konfigurationsdatei, ersichtlich im Anhang in Kapitel C.3, ist restriktiver ausgelegt und erlaubt dem Benutzer auf die Dokumentation und das Frontend zuzugreifen. Welche Voraussetzungen eine Installation der Software hat und mit welchen Befehlen ein Deployment vorgenommen werden kann, wurde in der Entwicklerdokumentation unter `https://grafit.readthedocs.io/en/latest/#deployment` beschrieben.

Das bei der lokalen Entwicklung eingesetzte docker-compose.dev.yml, ist im Anhang

[24] https://docs.docker.com/compose/

unter Kapitel C.3 ersichtlich. Diese Konfiguration erlaubt es dem Entwickler, auf das Neo4j-Webinterface zuzugreifen. Dieses unterstützt beim Erarbeiten neuer Queries und ermögliche eine Visualisierung des Wissensnetzes.

Nebst der Webapplikation existiert auch eine Benutzer- und Entwicklerdokumentation, die über die Dokumentationsplattform Read the Docs unter `https://grafit.readthedocs.io` verteilt wird. Wie in Abbildung 16 ersichtlich, beinhaltet ein komplettes Deployment der Applikation einen Container namens Grafit Docs. Dieser hosted dieselbe Dokumentation zusätzlich lokal.

5 Validierung

In diesem Kapitel wird der Stand der Arbeit mit den Zielen und Randbedingungen abgeglichen.

Zuerst wurde das Ziel **der Stand der Prototypen und Konzepte wird analysiert** in Angriff genommen. Die bestehenden Programme wurden auf konzeptuelle und tatsächliche Funktionalität untersucht und die Arbeiten zum IKC-Prototyp studiert. Weiter wurden Gespräche mit den Entwicklern geführt, um einen möglichst vollständigen Überblick zu gewinnen.

Das zweite Ziel, dass **aus diesen Erkenntnissen ein Anwendungsszenario erarbeitet wird**, wurde erfüllt, indem aus den gewonnenen Erkenntnissen und Gesprächen mit dem Auftraggeber eine Vision geformt wurde. Durch das agile, iterative Vorgehen fanden regelmässig Sprint-Reviews statt, bei denen der Fortschritt validiert und die Entwicklung des Szenarios mit dem Auftraggeber abgesprochen wurden. Das Anwendungsszenario hat sich genau wie die Software inkrementell weiterentwickelt und wurde in Kapitel 3.1 dokumentiert. Die bestehenden Prototypen waren auf einen Single-User-Betrieb fokussiert. Durch das Anwendungsszenario war klar, dass diese Änderung umgesetzt wurde muss, obwohl die Aufgabenstellung diese nicht explizit forderte.

In allen Bereichen, d.h. für alle Epics, wurden die substanziellen Bestandteile implementiert. Dabei konnten in mehreren Bereichen Teile der bestehenden Prototypen direkt oder zumindest konzeptionell übernommen werden. Die bei den Prototypen fehlende Multi-User-Fähigkeit hat bspw. bei der Suche und der Graphgenerierung dazu geführt, dass Etliches neu entwickelt werden musste. Als Erweiterung wurde die Multi-User-Fähigkeit, die Knowledge-Graph-Generierung, eine komplett überarbeitete Cloud-Integration und ein Ausbau der Suchfunktionalität umgesetzt. Das Ziel **Code und Konzepte aus bestehenden Prototypen werden zu einer nach bewährten Engineering-Methoden aufgebauten Software integriert** konnte somit auch umgesetzt werden. Um die Qualität des Engineerings zu gewährleisten, wurde mit Entwurfsmustern, verbreiteten und bewährten Technologien, einer kompletten Dockerisierung der Applikation und mit einer automatisierten Test Suite gearbeitet. Aus den integrierten und neu entwickelten Funktionen entstand eine stabile und einheitliche Codebasis, die unter der MIT Open-Source-Lizenz auf Github

verfügbar ist.[25]

Zum Erfüllen des Ziels, **ein stabiles Open-Source-Software-Projekt zu initialisieren**, gehört aber nicht nur die Offenlegung des Quellcodes und eine passende Lizenz. Dazu wurde eine klassische Open Source Build Pipeline aufgebaut, bei der der Code getestet, auf Lizenzeinhaltung überprüft, in einen Docker Container integriert und auf Docker Hub publiziert wird. Die Software kann aus dem Docker Hub heruntergeladen oder direkt auf grafit.io verwendet werden. Grafit kann per Docker Image selbst gehostet werden und die in den Randbedingungen geforderten maximalen Systemvoraussetzungen werden nicht überschritten.

Um ein möglichst zugängliches Open-Source-Projekt zu lancieren, wurden die Empfehlungen der GitHub Open-Source-Checklist umgesetzt und eine Entwicklerdokumentation bereitgestellt.

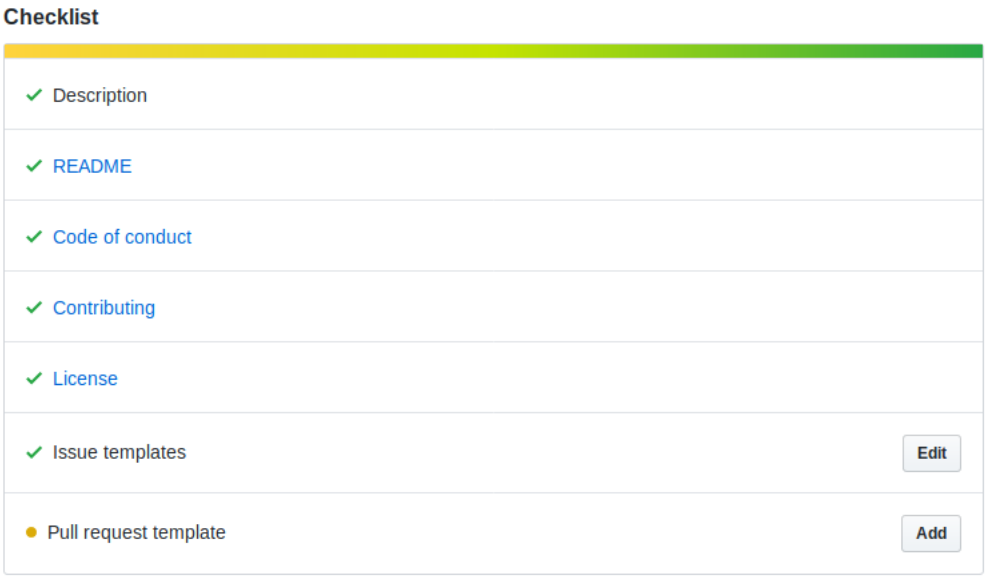


Abbildung 17: GitHub Open-Source-Checklist von Grafit

Zusammenfassend kann gesagt werden, dass die Zielsetzung des Projekts erfüllt wurde und die Randbedingungen nicht verletzt wurden. Um den Stand des Produkts bei einer möglichen Kundschaft zu testen, wurden zwei informelle Gespräche mit Personen aus unterschiedlichen potenziellen Nutzungsgruppen geführt. Felix Imobersteg, Partner der whatwedo GmbH, einer Softwareentwicklungsfirma in Bern, fand Grafit

[25] https://github.com/grafit-io/grafit

sei eine potenzielle Lösung für sein KMU. Er wünschte sich mehr Editiermöglichkeiten für den Text der Artikel und sieht eine Gefahr bei den Berechtigungen, da diese implizit über die Workspaces vergeben werden. Für whatwedo wäre es spannend, GitLab zu integrieren, um Commits und Issues daraus zu durchsuchen.

Die zweite potenzielle Nutzerin, eine Autorin, fand die automatische Verlinkung super. Aktuell pflegt sie selbst ein Netz auf Papier, um den Überblick über die Charaktere und deren Beziehungen nicht zu verlieren. Für sie wäre es wichtig, dass es möglich ist, auch Bilder und Skizzen in das Netz aufzunehmen.

Diese zwei Statements lassen darauf schliessen, dass durchaus ein Bedarf an Knowledge-Management-Tools vorhanden ist, der mit Grafit befriedigt werden kann.

6 Ausblick und Schlusswort

Die Prototypen wurden konsolidiert und es wurden Erweiterungen vorgenommen. Es wurde auf Mehrbenutzerfähigkeit umgestellt und der neue Ansatz der Cloud-Anbindung ist innovativ. Daneben haben wir uns in die Thematik des NLP eingearbeitet und einen offenen Build-Prozess aufgebaut. Es ist ein vielseitiges Projekt entstanden, das wir aktiv prägen konnten.

Wir sind agil vorgegangen und haben den ursprünglichen Plan regelmässig angepasst, da wir uns darauf fokussiert haben, ein nützliches Projekt umzusetzen. Das stellte während des gesamten Projekts durchaus einen Zielkonflikt mit der Aufgabenstellung dar. So wurde der Mehrbenutzerfähigkeit viel Aufmerksamkeit geschenkt, obwohl diese nicht explizit unsere Aufgabe war.

Vor Weiterentwicklungen würden wir empfehlen, systematische User-Tests durchzuführen, um neben der Software auch das Anwendungsszenario zu validieren, da dieses primär auf den anderen Arbeiten und dem Gespräch mit unserem Auftraggeber beruht.

Dank dem Feedback der potenziellen Kundschaft und den Gesprächen mit dem Auftraggeber sind die folgenden möglichen Ausbauschritte entstanden:

- Browser Extension, um externe Quellen einfach ins Netz aufnehmen zu können.
- Ein Share Button, um Nodes zwischen Workspaces zu teilen.
- Die Möglichkeit, Einladungen für Workspaces zu versenden und zu akzeptieren.
- Bei der Cloud-Integration weitere Datentypen (.docx, .pdf) crawlen.
- Unterstützung für Markdown im Artikeltext.
- Das Onboarding verbessern, indem ein Default Workspace mit einigen Tutorial Nodes erstellt wird.

Eine Kommerzialisierung wäre möglich, indem Grafit als SaaS angeboten wird. Durch die Art und Weise, wie die Software aufgebaut ist, sollte das grundsätzlich umsetzbar sein.

Am Ende von jedem Sprint gab es einen stabilen Stand des Produkts. Seit Ende des zweiten von sechs Sprints konnte mit Grafit auf `https://grafit.io/` interagiert werden. Das war ein enormer Vorteil und half, konkret weiterzuentwickeln.

Damit Grafit weiterlebt, war es wichtig, das Projekt zugänglich zu gestalten. So haben wir bei der Entwicklungsdokumentation spezielle Anstrengungen betrieben und hoffen, dass jemand davon profitiert.

Literaturverzeichnis

Ackermann, B. (2018). Digitalisierung und wissenstransfer: praxiserprobte online-materialien für agile organisationen. In *Erfolgreicher wissenstransfer in agilen organisationen* (S. 201–218). Springer.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., ... Sutherland, J. (2001). *Principles behind the Agile Manifesto.* Zugriff am 2018-12-12 auf `https://agilemanifesto.org/principles.html`

Bird, S., Klein, E. & Loper, E. (2009). *Natural language processing with python - analyzing text with the natural language toolkit* (First Edition Aufl.). Sebastopol: O'Reilly Media.

de la Guardia, C. (2016). *Python web frameworks* (First Edition Aufl.). Sebastopol: O'Reilly Media.

Eiholzer, M. (2017). *Search as a central aspect of personal data management* (Vertiefungsarbeit Master). Hochschule Luzern (HSLU).

Fowler, M. (2000, 7). *Is Design Dead?* Zugriff am 2018-12-12 auf `https://martinfowler.com/articles/designDead.html`

Group, T. P. G. D. (2018). *Postgresql 10.6 Documentation.* Zugriff am 2019-01-17 auf `https://www.postgresql.org/files/documentation/pdf/10/postgresql-10-A4.pdf`

Hofstetter, J. & Jud, M. (2013). Was fehlt Scrum? - ein beispielhafter Lösungsansatz aus der Hochschulpraxis. In (Bd. P-224, S. 133–145). Gesellschaft fur Informatik (GI).

Introduction to Knowledge Management. (2007). Zugriff am 2018-12-28 auf `https://web.archive.org/web/20070319233812/http://www.unc.edu/~sunnyliu/inls258/Introduction_to_Knowledge_Management.html`

John Girard, J. G. (2015). *Defining knowledge management: Toward an applied compendium.* Zugriff am 2018-12-28 auf `http://www.iiakm.org/ojakm/articles/2015/volume3_1/OJAKM_Volume3_1pp1-20.pdf`

Lovins, J. B. (1968). *Development of a Stemming Algorithm.* Zugriff am 2018-12-30 auf `https://pdfs.semanticscholar.org/6b38/53f08c482fe1bfbe39d656d50a8c73976f3c.pdf`

Meier, A. & Kaufmann, M. (2016). *Sql- & nosql-datenbanken.* Berlin: Springer Berlin Springer Vieweg.

Parameswaran, A., Garcia-Molina, H. & Rajaraman, A. (2010). *Towards The Web Of Concepts: Extracting Concepts from Large Datasets.* Zugriff am 2018-12-28 auf `http://ilpubs.stanford.edu:8090/917/1/conceptMining-Techrep.pdf`

Rajaraman, A. & Ullman, J. (2011). *"Data Mining". Mining of Massive Datasets.* Zugriff am 2018-12-28 auf `http://i.stanford.edu/~ullman/mmds/ch1.pdf`

Redmond, E., Wilson, J. R. & Klicman, P. (2012). *Sieben Wochen, sieben Datenbanken: moderne Datenbanken und die NoSQL-Bewegung.* Beijing: O'Reilly. (OCLC: 819510566)

Robinson, I., Webber, J. & Eifrem, E. (2015). *Graph databases: New opportunities for connected data* (2nd Aufl.). O'Reilly Media, Inc.

Stadelmann, K. (2017). *Entwicklung eines Business Modells für ein University Spin-off: wirtschaftliche und organisatorische Aspekte des Transfers von der Forschung in die Praxis* (Bachelor's Thesis). Hochschule Luzern (HSLU).

The Stanford Natural Language Processing Group. (2004). Zugriff am 2018-12-30 auf `https://nlp.stanford.edu/software/tagger.shtml`

Waldis, A. & Siegfried, P. (2017). *Automatisierte Generierung von plattformübergreifenden Wissensnetzwerken mit Metadaten und Volltextindexierung* (Bachelor's Thesis). Hochschule Luzern (HSLU).

Abbildungsverzeichnis

1 Lokahi – Keyphrase Extraction 15
2 Lokahi – Graphvisualisierung 16
3 IKC – User Interface 17
4 IKC – Graphvisualisierung 18
5 Grafit – Mockup User Interface (UI) 30
6 Systemübersicht 31
7 Übersicht Frontend 33
8 Datenbankschema – PostgreSQL 37
9 Graphmodell – Neo4j 38
10 Datenfluss der Generierung des Knowledge Graphs 39
11 Keyphrase Extraction Pipeline 40
12 Concept Extraction – Strategy Pattern 46
13 Generiertes Wissensnetzwerk in Neo4j 48
14 Build Pipeline 52
15 FOSSA – Übersicht verwendeter Lizenzen 55
16 Docker Container 56
17 GitHub Open-Source-Checklist von Grafit 59

Listingsverzeichnis

1 PoS Tagging in Python 14
2 Django Workspace Model 36
3 Django Admin – Generieren der SQL-Statements 36
4 Neomodel GraphArticle Model 38
5 Neomodel ArticleRel Beziehung 38
6 Tokenization mit Textblob 40
7 Generierung n-Grams 41
8 PoS-Tag-Filter 43
9 Berechnung Term Frequency (TF) 44
10 Berechnung Inverse Document Frequency (IDF) 44
11 Berechnung lengthNorm 45
12 Berechnung TF-IDF 45
13 Abstrakte Klasse – ExtractStrategy 46
14 Anwendung implementierter Keyphrase Extraction 47
15 Stemming mit to_tsvector 49
16 Suche nach ähnlichen Wörtern 50
17 Unit-Test – Keyword Extraction 53

Tabellenverzeichnis

1 PoS n-Gram Filterregeln 42

2 Metriken für TF-IDF 44

Glossar

API bedeutet Application Programming Interface und ist eine Schnittstelle eines Softwaresystems, welche zur Kommunikation mit anderen Programmen verwendet wird. 24, 53

Cluster oder auch Rechnerverbund genannt, sind mehrere Computer, die zusammen vernetzt sind und als gemeinsames System funktionieren. 28

Docker Hub ist ein Docker Image Verzeichnisdienst. 56

Dropbox ist ein Cloud Storage Dienst für Firmen und Privatpersonen. 17, 18, 26

Evernote ist eine Cloudbasierte Notizapplikation, die das Speichern und Synchronisieren von Text, Sprach, Bild und Video-Dateien erlaubt. 17, 24

GitHub ist ein SaaS Dienstleister, der eine auf dem Versionsverwaltungssystem Git aufbauende kollaborative Code Verwaltung anbietet. 21, 47

grafit ist das dieser Bachelorarbeit entwickelte Open-Source Knowledge-Management-Tool. 8, 21, 23, 24, 27, 29, 30, 32, 53, 59–62, 65

horizontale Skalierung ist das Hinzufügen von Rechnerknoten zu einem verteilten System. 28

JSON bedeutet JavaScript Object Notation und ist ein weit verbreitetes Datenformat, welches sich durch seine einfach lesbare Textform auszeichnet. 18

lokahi ist ein Forschungsprojekt der HSLU, das sich mit der Generierung von Wissensnetzwerken befasst. 15, 17, 19, 22, 25, 27

NoSQL ist eine Klasse von Datenbanken, die ihre Daten nicht relational verwalten. 23

polling bedeutet das immer wiederkehrende Abfragen eines Status. 29

PostgreSQL ist ein objektrelationales, ACID-konformes, offenes RDBMS. 27, 36, 37, 39, 49, 65

push -Modell ist ein vom Sender gesteuerter Informationsfluss. 29

Regex wird auch regulärer Ausdruck genannt und ist eine Zeichenkette, die zur Filterung und zur Erkennung von Mustern in einem Text dient. 40

REST steht für Representational State Transfer und ist ein Architekturstil der speziell für Maschine-zu-Maschine-Kommunikation über HTTP ausgelegt ist. 26

sharding wird das horizontale partitionieren einer Datenbank auf mehrere Rechner genannt. 28

Webframework ist eine Bibliothek oder Software, welche für die Entwicklung von dynamischen Webseiten und Webservices ausgelegt ist. 25, 26

Akronyme

B2B Business-to-Business. 19

B2C Business-to-Consumer. 19

DRF Django REST Framework. 26, 32

HSLU Hochschule Luzern. 4, 8, 12, 15, 17

IDF Inverse Document Frequency. 14, 15, 44, 66

IKC Intuitive Knowledge Connectivity. 8, 17–19, 22–27, 29, 30, 34, 41, 47, 48, 58

IPFS InterPlanetary File System. 23

JWT JSON-Web-Tokens. 26, 32, 34

NLP Natural Language Processing. 8, 11–13, 26, 61

NLTK Natural Language Toolkit. 26, 40, 41

OGM Object-Graph-Mapper. 37

ORM Object-Relational Mapping. 36

P2P Peer-to-Peer. 8, 23–25

PoS Tagging Part-of-Speech Tagging. 14, 41, 66

Postgres PostgreSQL. 27

RDBMS Relational Database Management System. 27, 68

SaaS Software as a Service. 11, 61, 68

SoDa Software Development agile. 9, 10

TF Term Frequency. 14, 15, 44, 66

TF-IDF Term Frequency – Inverse Document Frequency. 14, 15, 28

UI User Interface. 17, 24, 25, 30, 65

A Projektmanagement

A.1 Rahmenplan

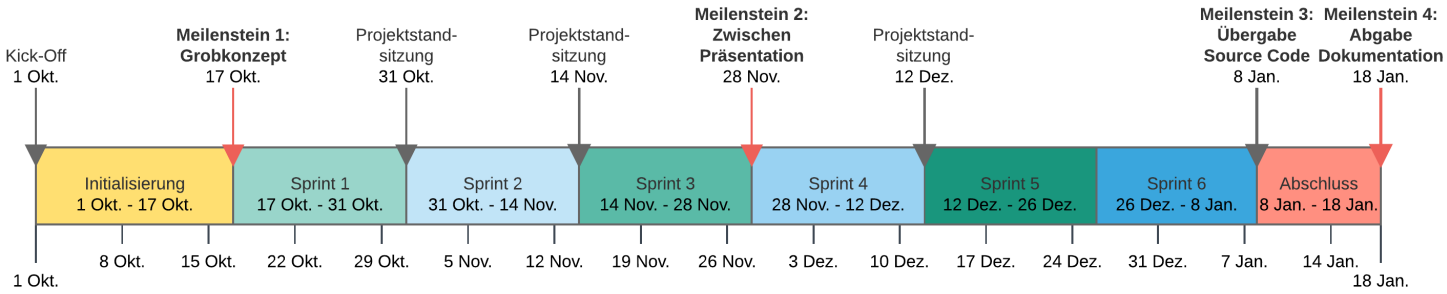


Abbildung 18: Rahmenplan BDA

A.2 Meilensteine und Lieferobjekt

Im Rahmen dieses Projekts sind sechs Sprints nach Scrum eingeplant. Bei Ende eines Sprints wird eine lauffähige und getestete Version der Applikation bereitgestellt. Diese kann vom Versionkontrollsystem bezogen werden. Zudem wird jeweils der nächste Sprint geplant und der Backlog aktualisiert. Daneben gibt es Meilensteine, die eine Abgabe von weiteren Resultaten verlangen.

A.2.1 Meilenstein 1: Grobkonzept

Ein Rahmenplan mit allen Meilensteinen und den dazugehörigen Lieferobjekten wurde erstellt. Die zu erarbeitenden Resultate wurden analysiert, spezifiziert und festgehalten. Es wurde ein Anwendungsszenario für eine Softwarelösung identifiziert und angearbeitet. Der Scope des Projektes wurde abgeklärt und dokumentiert.

A.2.2 Meilenstein 2: Zwischenpräsentation

Eine Präsentation der bisherigen Arbeiten und dem weiteren Vorgehen wurde vorbereitet und gehalten.

A.2.3 Meilenstein 3: Übergabe Source Code

Der Source Code und alle dazugehörigen Umsysteme wie eine Build Pipeline wurden dem Auftraggeber übergeben. Allfällige Fragen und bestehende Probleme wurden kommuniziert.

A.2.4 Meilenstein 4: Abgabe Dokumentation

Alle im Kapitel A.3 beschriebenen Resultate wurden abgeliefert.

A.3 Resultate

Im Rahmen dieser Bachelor Diplomarbeit werden folgende Resultate erarbeitet:

- **Schlussbericht**
  Gemäss den Anforderungen der Hochschule Luzern Informatik. Beinhaltet u.a.
  - Projektplanung
  - Analyse zum Stand der Technik
  - Lösungsdesign
- **Source Code**
  Der Source Code wird auf GitHub bereitgestellt.
- **Kompilierte Docker Container**
  Die entwickelte Software kann als fertige Docker Container von Docker Hub bezogen werden.
- **Benutzerhandbuch**
  Das Benutzerhandbuch wird in einer für User nutzbaren Form angeboten. Beispielsweise auf einer Produktwebsite oder auf readthedocs.org.

A.4 Risikomanagement

Um auf mögliche Gefahren vorbereitet zu sein wurde eine Risikoanalyse in Tabelle 3 durchgeführt. Die Eintrittswahrscheinlichkeit wurde mit P (1-5) und der Schaden mit S (1-4) abgekürzt.

ID	**Risiko Beschreibung**	**P**	**S**	**Massnahmen**
R1	Kein laufendes Produkt am Ende des Projektes	2	4	Agiles Vorgehen mit Ziel auf MVP einsetzen. Einsatz von CI, Head des Master Branch sollte immer lauffähig sein
R2	Niedrige Code Qualität	1	3	Code Coverage in Build Prozess integrieren und überwachen. GitHub Issues und Tasks dürfen nicht direkt von doing auf done gesetzt werden, sondern müssen einen Review Prozess durchlaufen. Dabei wird auf Clean Code geachtet
R3	Kein nützliches Produkt/ schlechter Market-Fit	3	3	Stützen uns auf BDAs, Alle zwei Wochen Gespräch mit Auftraggeber, reden mit KMU
R4	Langfristiger Ausfall eines Teammitglieds	1	4	Vermeiden von Extremsport
R5	Abweichen vom Zeitplan	3	3	Agiles Vorgehen in Sprints und Visualisiertes Zeittracking
R6	Die Software ist nicht erweiterbar oder wartbar	2	4	Robuste CI Pipelines, Entwicklerdokumentation, Gutes GitHub Readme
R7	Verlust des Programmcodes	1	4	Der Code wird mit Git versioniert und auf GitHub gehostet

Tabelle 3: Risikoanalyse

Eintrittswahrscheinlichkeit / Schadensfaktor	**1 Unwahrscheinlich**	**2 Niedrig**	**3 Wahrscheinlich**	**4 Hoch**	**5 Sicher**
4-Katastrophal	R4, R7	R1, R6			
3-Kritisch	R2		R3, R5		
2-Mittlere Auswirkungen					
1-Geringe Auswirkung					

Tabelle 4: Risikomatrix

B Product Backlog

#1 Collect and Persist Basic Text Data

Description of problem
To build a concept extraction algorithm, we need some text data. The data should be in a script readable format.

Solution
A table/file with ten to twenty articles. Every article should contains 200 to 1000 words.

Additional context

- [x] What language should be supported?
- [x] Where should the data be stored? Database?

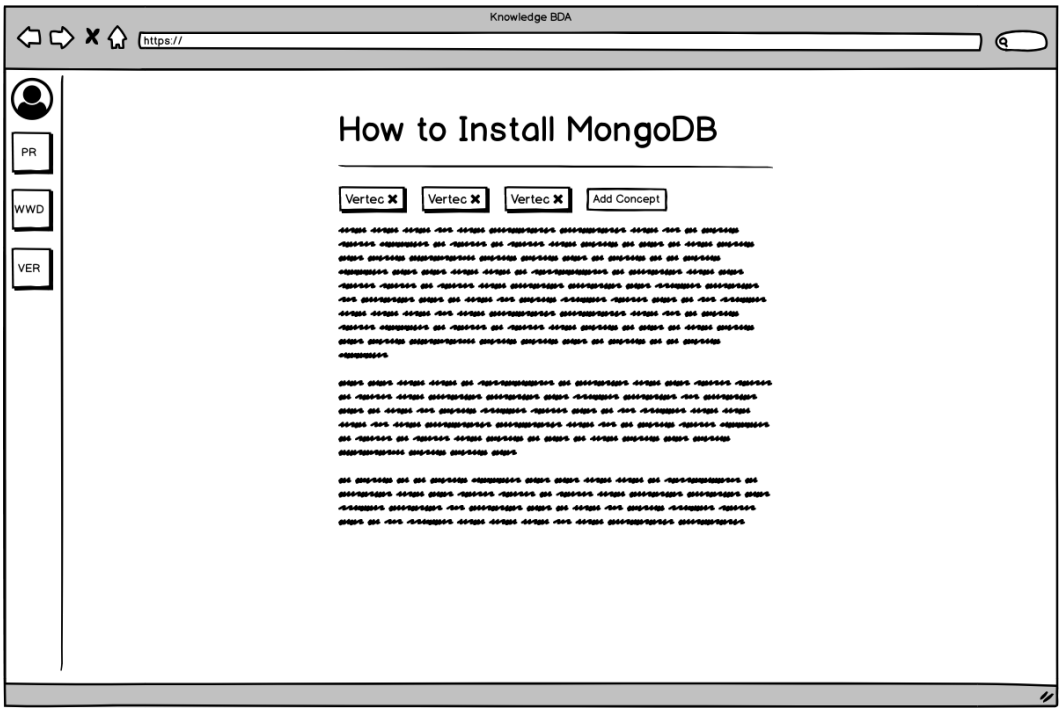
Criteria for acceptance

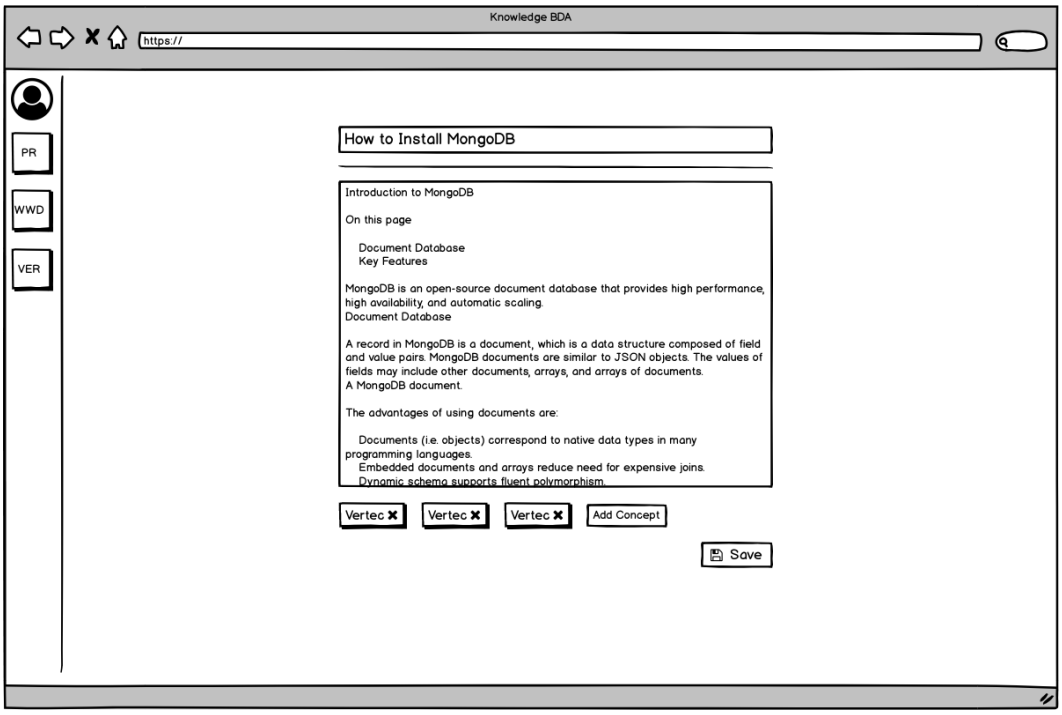
- [x] define the supported language
- [x] define data structure
- [x] collect data
- Don't Be Cute with Your Test Data

#2 Manage Articles

Description of problem
Article nodes are content nodes that are created by the user.

Solution
CRUD functionality for the article node is implemented for the front and backend. The articles are persisted into a datastore. As follows are the UI concepts:





Additional context

- [x] What datastore to use?

Criteria for acceptance

- [x] The user can create a new article node
- [x] The user can edit an existing article node
- [x] The user can delete an existing article node
- [x] Trigger the concept extractor when a node content has changed

#3 Import Text Files with File Uploader

Description of problem
A user wants to extract knowledge from existing text sources and import them into the application.

Solution
The user should be able to upload existing text files. The text will then be extracted and stored into the datastore. The newly created concept nodes should be linked to the related concepts.

Additional context

- [ ] Which file formats should be supported? Candidates are: txt, md, docx, pdf, html
- [ ] Should formatting be included in the analysis? e.g. what is the title of the document.

Criteria for acceptance

- [ ] The user can select one or multiple files to upload
- [ ] The text is extracted from the files and saved to the datastore
- [ ] Concepts are extracted and linked to the imported content nodes

#4 Manually connect Nodes

Description of problem
There needs to be a possibility of connecting a content-node to a concept when the desired connection isn't generated automatically.

Solution
The application allows to user to connect a content-node to a concept. There is a search bar that suggests existing concepts to the user. If the desired concept does not exist it should be created.

Criteria for acceptance

- [x] The user can search for an existing concept and connect it to a content-node
- [x] The user can create a new concept and connect it directly to a content-node

#5 Delete a Relation between Nodes

Description of problem
There should be a way for a user to delete connections between a content-node and a concept-node.

Solution
Provide the possibility for the user to delete existing connections between content and generated or manually added nodes. Add a small delete button is displayed on the concept.

Criteria for acceptance

- [x] The user can delete unwanted connections between nodes

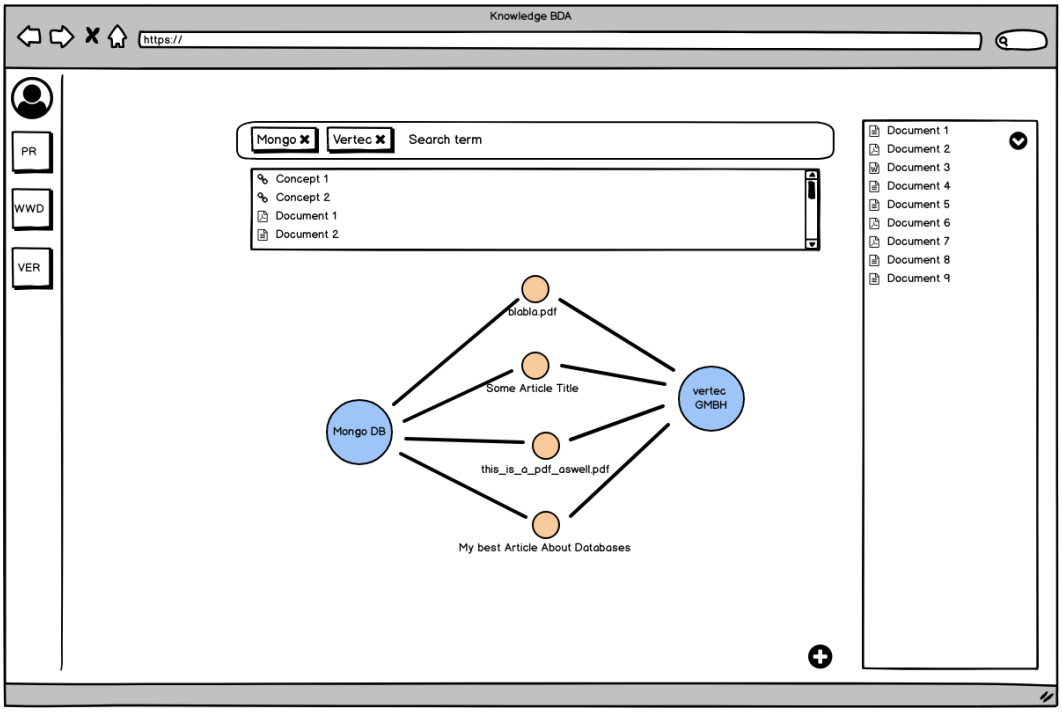
#6 Find Nodes by Searching for Concepts

Description of problem

The user wants to search for content-nodes that are related to one or more specific concepts.

Solution

A user can enter one or more concepts into a search bar. The top related nodes should be visualized in a graph (see picture below).



A list of all related documents is shown as well. The user can click on a content node to switch to a detail view and read the content of that node.

Additional context

- [ ] What graph visualisation library should be used?
- [ ] How are the top related content-nodes calculated?

Criteria for acceptance

- [ ] There is a search bar that lets you enter (and delete already entered) concepts
- [ ] All content-nodes that are connected to the selected concept*s are shown in a list
- [ ] The top related content-nodes are displayed in a graph

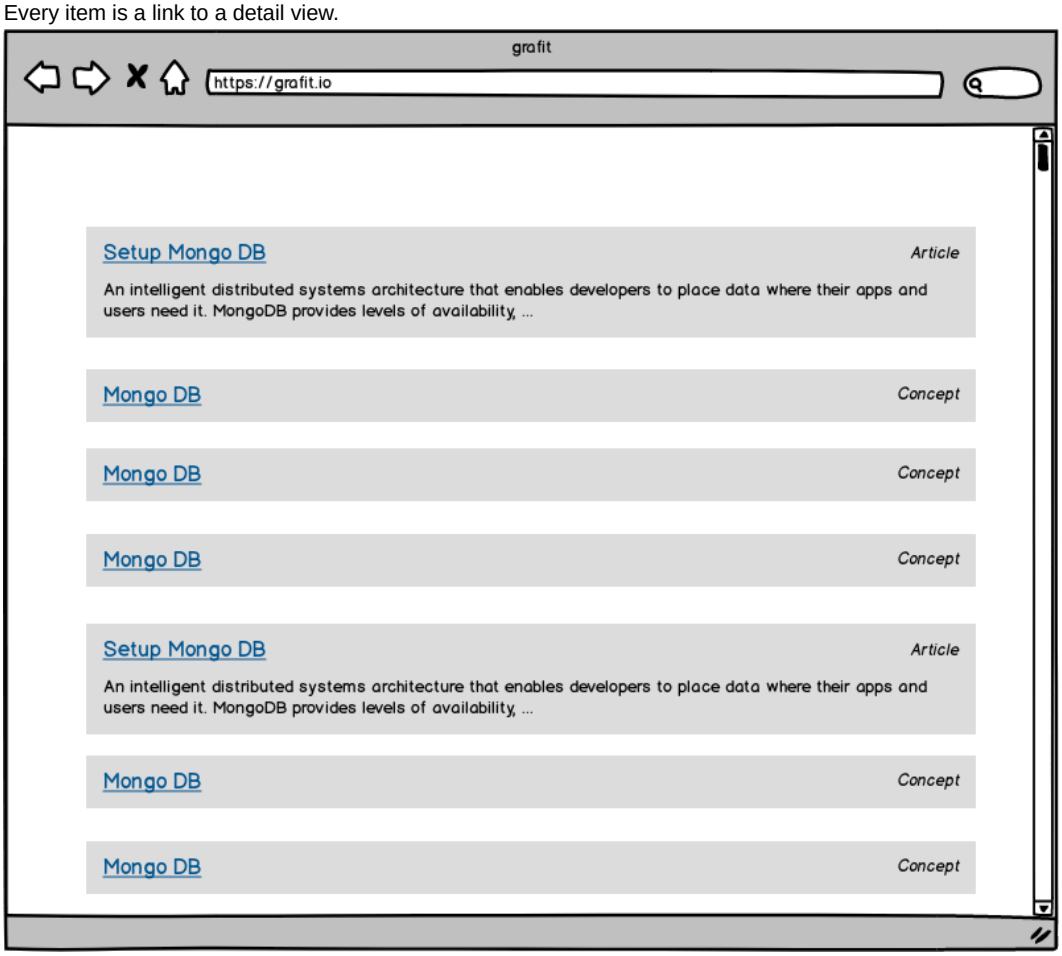
#7 Setup Frontend and add Article View

Description of problem

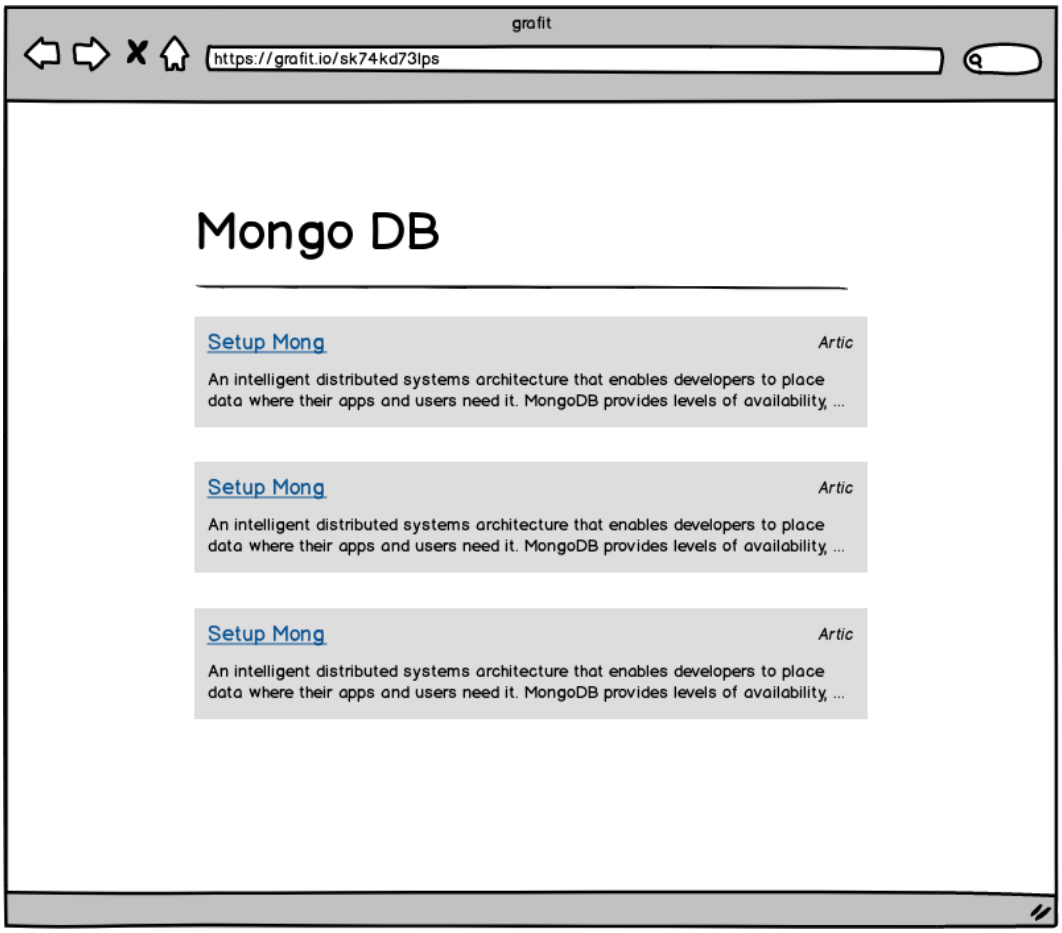
A user wants to be able to see all articles and the generated concepts.

Solution

The front-end development has to be setup. A front-end framework has to be chosen. A basic front-end development workflow has to be documented. Add a view with a list containing all concepts and articles. Every item is a link to a detail view.



Add a detail view for each node displaying the node content and a link to all adjunct nodes. A concept node would look like this (has no content):



A article node like this:

Discussion

- ☑ Evaluate what front-end framework to use

Criteria for acceptance

- ☑ The front-end is able to display a list of all articles and concepts
- ☑ The front-end can display an article and a concept with all adjunct nodes.
- ☑ The data is fetched from the REST-API
- ☑ The build process of the front-end is integrated into the CI/ CD pipeline
- ☑ There are instructions on how to get started developing the front-end

#8 Setup Backend with Article Endpoint

Description of problem

There needs to be a back-end that lets you fetch a graph of all articles.

Solution

The back-end development has to be setup. The language and framework for the development of the back-

end have to be evaluated.

Discussion

- [x] What language/ framework should be used for back-end development

Criteria for acceptance

- [x] There is a REST-endpoint that lets you fetch all articles and their adjunct articles
- [x] The build process of the back-end is integrated into the CI/ CD pipeline
- [x] There are instructions on how to get started developing the back-end

#9 Basic Keyphrase Extraction

Description of problem
The application should be able to extract keyphrases from a text.

Solution
To be able to automatically connect concept and content nodes the application has to be able to extract keyphrases from a text. In the first iteration of this project a tf-idf algorithm has to be implemented for this purpose. The algorithm should be easily swappable for a different one (use something like the strategy pattern).

Additional context
A possible solution for the first version of keyphrase extraction is to use a library like textblob and build a basic tf-idf algorithm on top of it. https://stevenloria.com/tf-idf/ For now a existing text corpus from nltk can be used. https://www.nltk.org/book/ch02.html

Criteria for acceptance

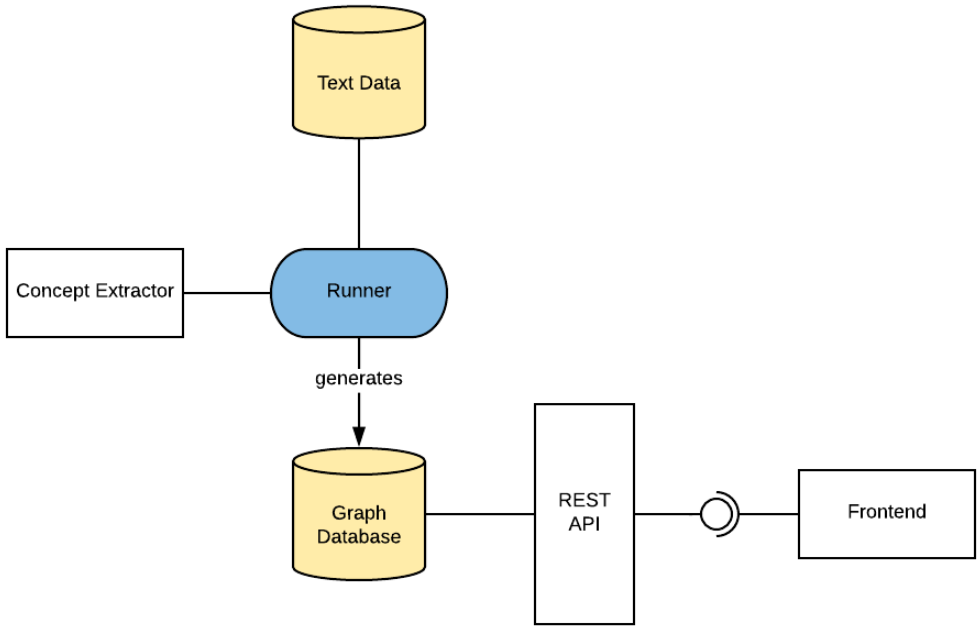
- [x] A basic implementation of the keyphrase extraction is working
- [x] The keyphrase extraction can be swapped easily for a different one

#10 Knowledge Graph Runner

Description of problem
The application needs to be able to generate a graph from text data.

Solution
The knowledge graph runner should be the link between the text data #1 and the keyphrase extractor #9. A graph has to be created that combines the generated concept nodes and text/ content nodes. The nodes should be automatically linked to the corresponding concepts

Additional context



Criteria for acceptance

- ☑ The knowledge graph runner uses the keyphrase extractor #9 and applies it on the text data #1
- ☑ Using the auto generated keyphrases a graph is created that connects content nodes with their concepts
- ☑ The graph is made available for the REST API (stored in a graph database)

#11 Setup CI/ CD Pipeline

Description of problem

To make development easier a CI/ CD pipeline has to be setup.

Solution

Different approaches for CI/ CD have to be evaluated (CircleCI, Travis etc.) and a pipeline has to be built.

Criteria for acceptance

- ☑ The CI/ CD pipeline is running
- ☑ Results are saved to registry (Docker Hub, PyPi)
- ☑ There is a badge in the project README.md that shows if the build is successful

#12 Setup Automated Testing

Description of problem
To improve code quality automated testing has to be setup and integrated into the CI process #11

Criteria for acceptance

- [x] Automated testing is setup and integrated into the CI process
- [x] There is a badge in the project README.md that shows the current code coverage

#13 Add License Scan Report and Status

Your FOSSA integration was successful! Attached in this PR is a badge and license report to track scan status in your README. Below are docs for integrating FOSSA license checks into your CI:

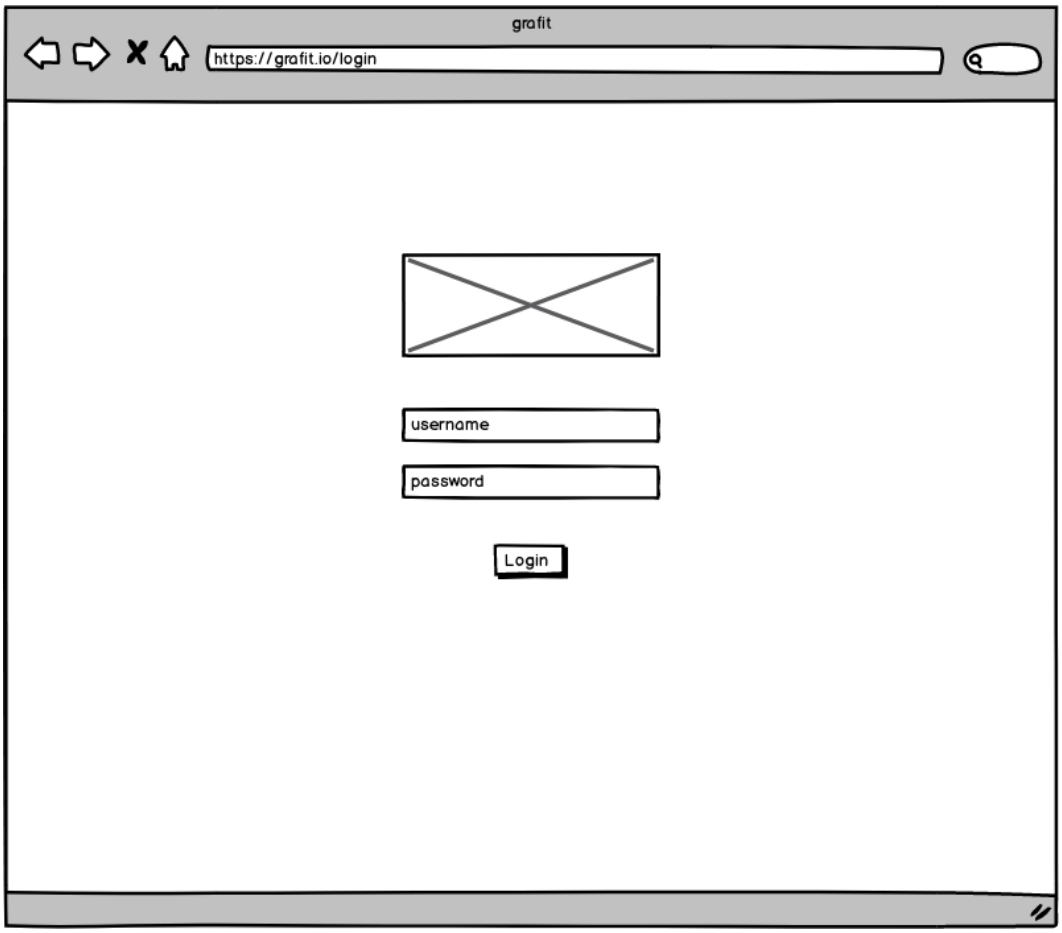
- CircleCI
- TravisCI
- Jenkins
- Other

#14 Login Form

Description of problem
Currently the API Endpoint for user authentication is available. There is an Endpoint at `/api-token-auth` where you can redeem a Token by providing a valid username and password. The token is necessary to authenticate any further requests on the API. As a user i want to login to authenticate myself on the service once.

Solution
Provide a login form where the user can authenticate himself. Store the token for a reasonable time frame to prevent the user from having to login again. If the token is not available, redirect the user to the login form.

Additional context



Criteria for acceptance

- [x] There is a login form.
- [x] Login data is send to the API Endpoint and the Token is stored on the client device.
- [x] The token expires.
- [x] If the login data is not correct, an error is promoted and the user has the change to try again.
- [x] If no Token is available, the user is redirected to the login page.

#15 Setup Production Build

Description of problem

There should be a current build of the software available online. The hosted solution should support HTTPS and update automatically if a new version of the software is pushed to Docker Hub.

Additional context

A server can be requested from enterpriselab Let's Encrypt for SSL certs Watchtower for automatic docker container updates

Criteria for acceptance

- [x] The production server exposes the frontend and API on port 80 and 443
- [x] The production server automatically updates to the newest build of the application

#16 Add Graph Database

Description of problem
Currently the data knowledge graph data is stored in a relational database. To improve performance and to better match our data structure, we need to change it to a graph database.
Solution
A clear and concise description of what you want to happen.
Additional context
Add any other context or screenshots about the feature request here.

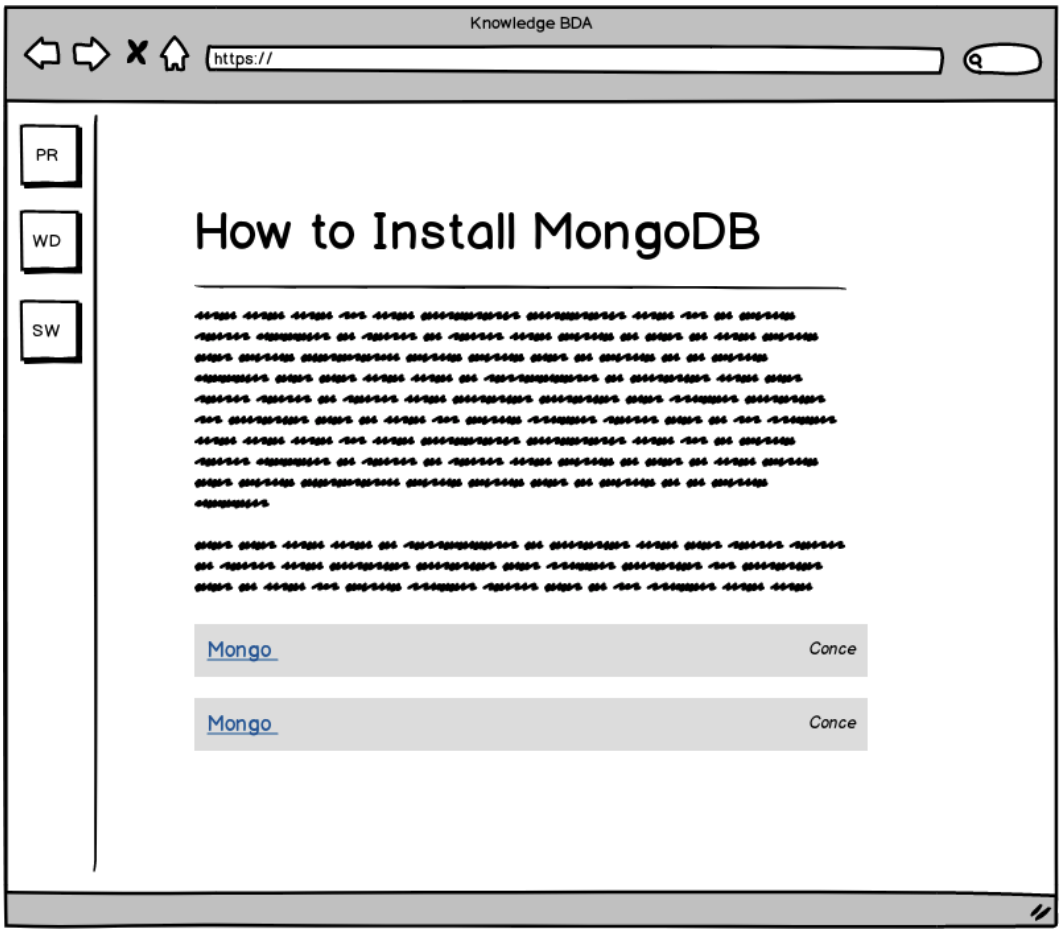
Criteria for acceptance

- [x] choose a graph database

#17 Workspace Sidebar

Description of problem
As a user it is possible to be part of multiple work-spaces. A workspace is a logical unit grouping users to a set of documents.
Solution
There is a sidebar with all workspaces of the current user. By clicking on a workspace it will become the current workspace. All `article lists` etc. are updated.

Additional context



Criteria for acceptance

- [x] a list with all workspaces of the current user is displayed
- [x] by clicking on a workspace, the workspace article list is shown
- [x] the workspace list is hidden on small devices The workspace list is only displayed if:
- [x] a user is authenticated
- [x] the user is part of more than one workspace

#18 Icon not showing on mobile

The icon is not loading when the website is saved to the homescreen.

#19 Default Workspace with Tutorial

Description of problem

A new user is not part of any workspace. This leads to a confusing user experience. To prevent a user from

having no workspace, it would be an improvement if the user is automatically part of some kind of default workspace after sign up.

Solution
The user gets assigned a tutorial workspace. In this workspace are articles with tutorials how to use grafit and explaining the different concepts. There is a new Workspace for every user.

Criteria for acceptance

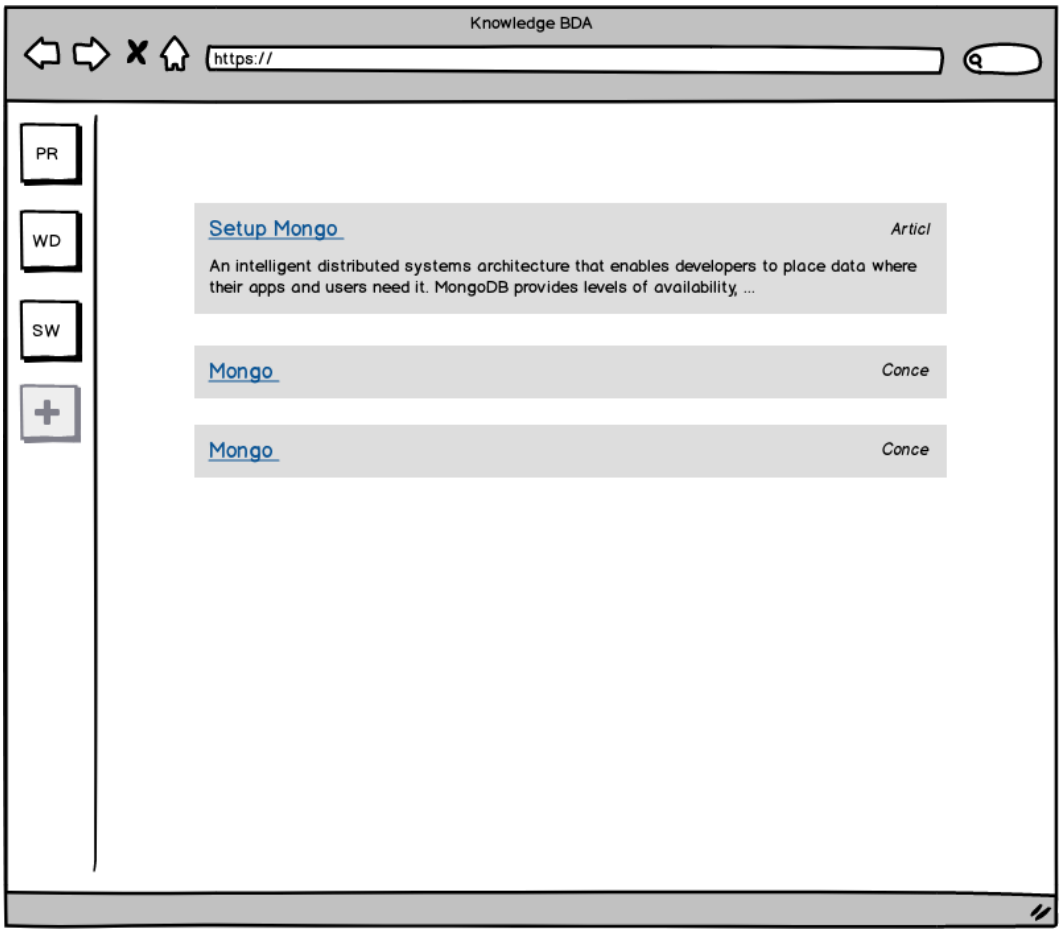
- ☐ After sign up, a user is automatically part of an workspace.
- ☐ The workspace contains articles explaining `Workspace`, `Auto Tagging` and `Node editing`.
- ☐ The user is alone in this workspace.

#20 Workspace Creation

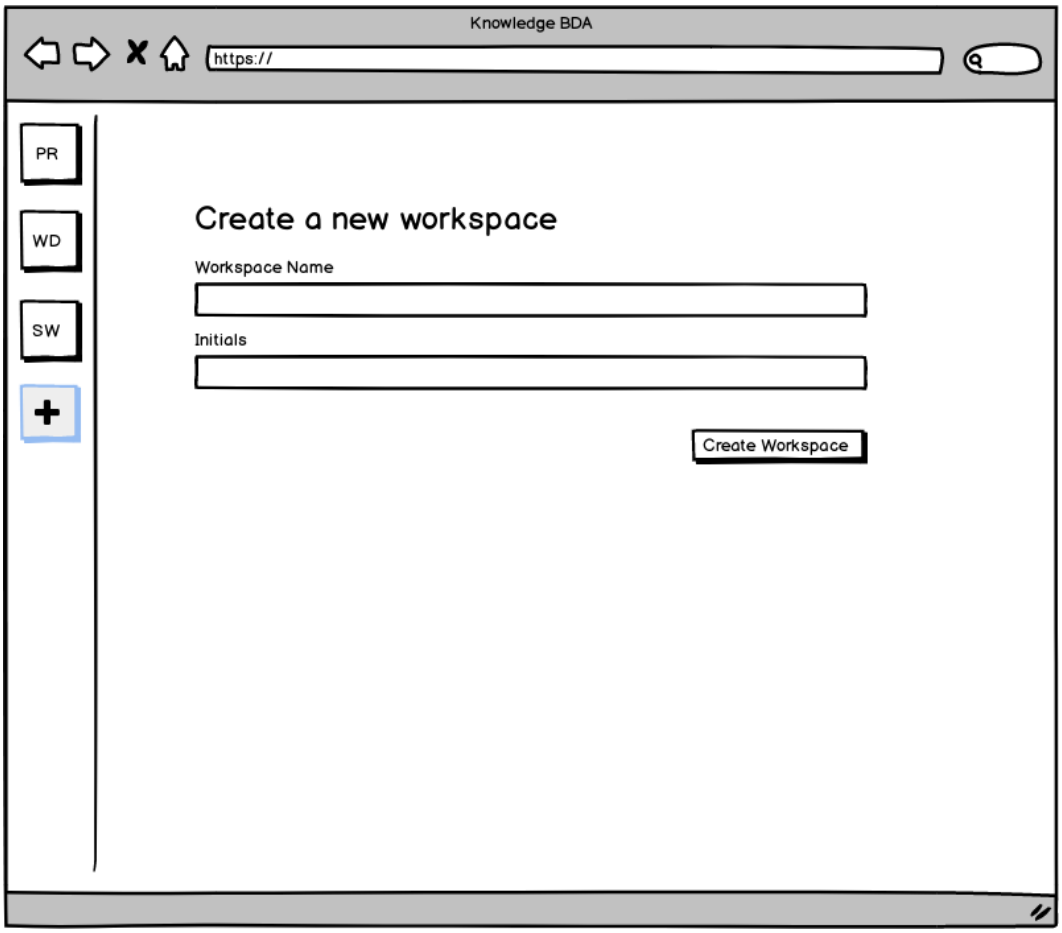
Description of problem
Currently there is no possibility for a user to create a new workspace.

Solution
Add a + Button to the workspace bar. By clicking a form is opened. The form contains a field for the name and a field for the Initials. After creation the user is added to this workspace. After adding the name in the form, a initial suggestion is automatically calculated for the user. If a single word is added in the name input field, the initials are the first two letters. If multiple words are added in the name input field, the primary of the first two words are combined. As soon as the user writes to the Initials field, no further initials suggestions are calculated to prevent from overwriting user content. Currently the workspace bar is hidden if the user is not part of any workspace. As the create button is part of this bar, it needs to be visible all the time.

Additional context
Create Workspace Button:



Create Workspace Form



Criteria for acceptance

- [x] A create button is in the workspace bar
- [x] The user can create a workspace by defining a name and initials
- [x] The name and the initials are validated directly in the form
- [x] As long as the user did not manually edit the initials field, the initials are calculated automatically
- [x] The creator is a member of the created workspace
- [x] The workspace bar is never hidden if a user is logged in

#21 Search

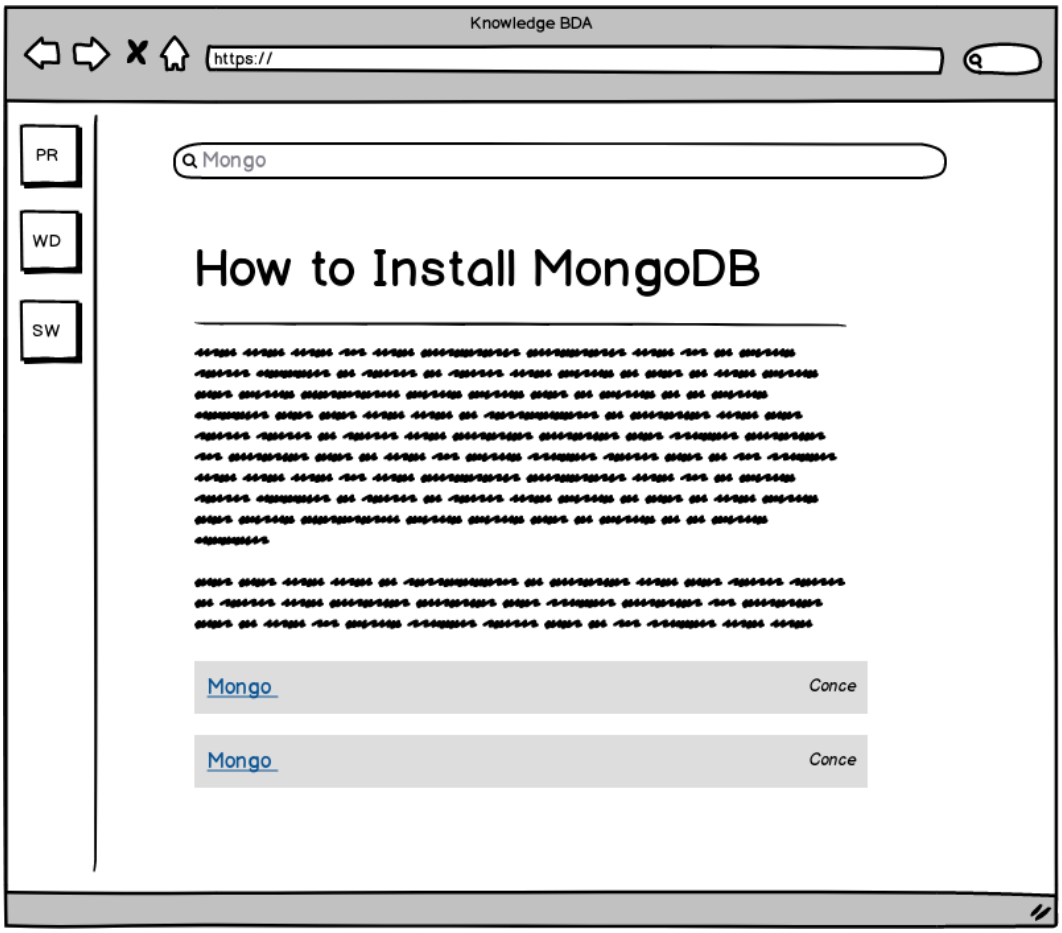
Description of problem

Currently there is no possibility to search content. The user has to look through all the articles by hand.

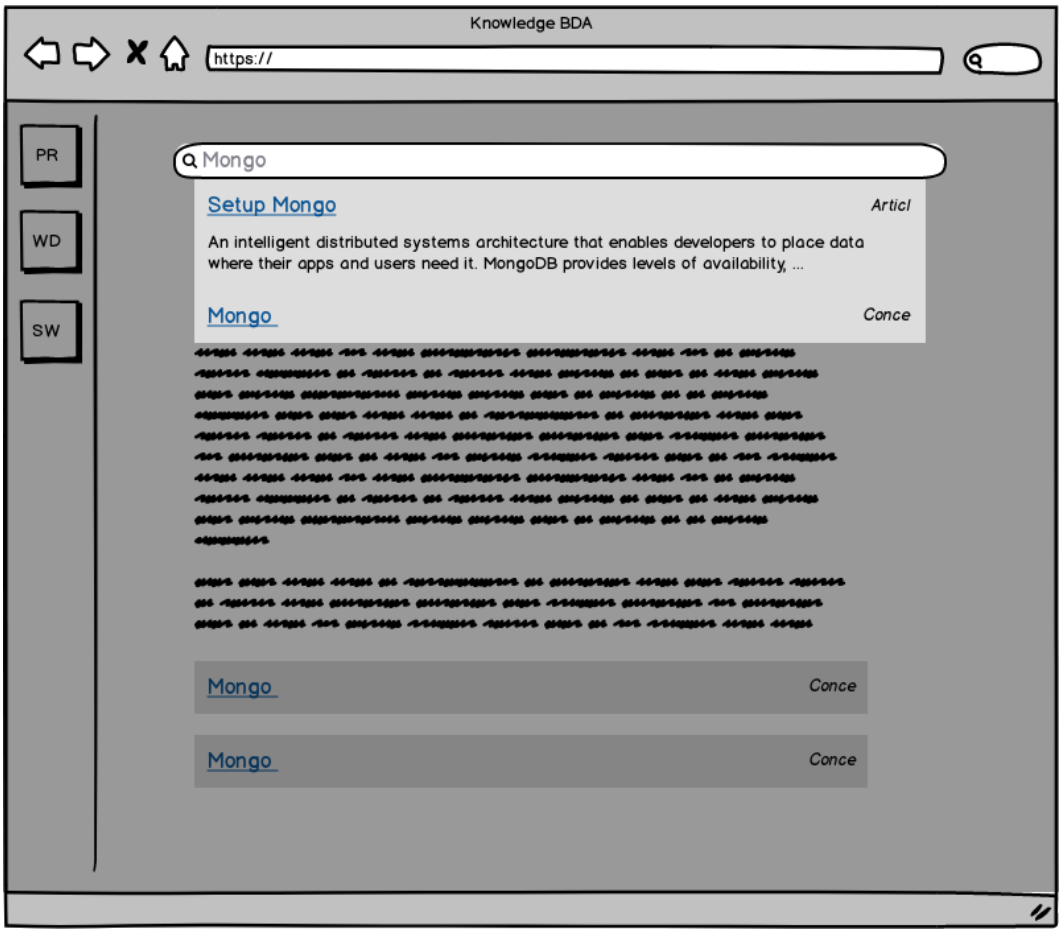
Solution

> Search bar

There needs to be a search bar:



The search bar is shown in every workspace as an overlay:



The overlay is triggered if the user interacts with the search bar. The search bar can be selected by pressing the keys `Ctrl+F`. Tutorial for potential solution To deselect the search bar the user has to click beside the bar or search results. The search query is persisted in the search bar.

> Search Results

The search results are nodes of the knowledge network. If the user presses enter or clicks on the search icon, a list of all results is displayed. Minimal length of the search query is >= 1. The search results contain: - Nodes where the title fuzzy-matches the search query. - Nodes matching a full text search. Every word of the search query is matched on its own. Fuzzy if possible. - Nodes adjunct to title results. If the user clicks on result, the node is loaded. The results are displayed as follow: - If the node contains text, the first fifty characters are displayed. - If the node contains no text, five adjunct concepts are displayed as badges. Results are selectable by arrows key and enter.

> Autoload Results

The search results are loaded while the user is typing if the following conditions are met: - The search query is at least three characters long. - The last keystroke is older than X milliseconds. Preventing from loading results for each keystroke.

> Technical stuff

Doing fuzzy stuff with Neo4j doesn't seem to be super easy. It's probably easier to build a basic search index

in the Postgres database. How to build a fuzzy, weighted search index with postgress

Criteria for acceptance

- [x] A search bar is displayed on the top of the page
- [x] Search results are returned according to the defined **Criteria
- [x] Only search results from the current workspace are returned
- [x] Results are shown while typing if the defined conditions are met
- [x] Results are selectable
- [x] Minor misspellings are accepted

#22 Document Surfing

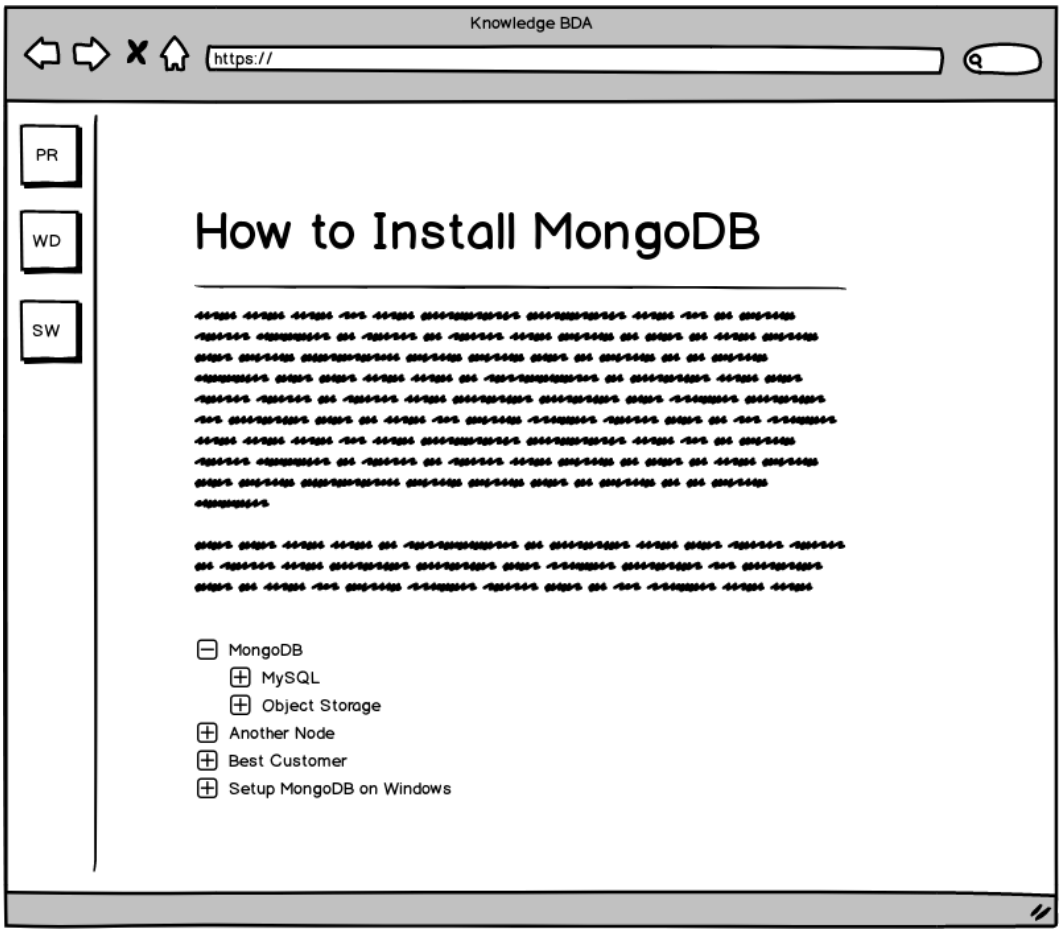
Description of problem

If the user looks at a node, adjunct nodes are displayed as batches. To see the adjunct nodes of these nodes, the user has to open them. Many screen changes can be confusing and the discovery possibilities are limited.

Solution

The nodes can be traversed by expanding a in a tree view.

Additional context



Criteria for acceptance

- [x] Adjunct nodes are displayed as a list
- [x] Nodes can be expanded
- [x] Max depth is five
- [x] Add link to corresponding node

#23 Support advanced Search Features

Description of problem

When searching for a node it should be possible to have some advanced queering features. Possible features: - exclude -> `jaguar -car` - must match -> `"jaguar"` - author -> `author:disperate` - creation & edit date -> `created:YYYY-MM-DD` (month and day optional) - place of search -> `in:title` or `in:body` - language of node -> `language:french`

Criteria for acceptance

- [ ] define features

- [ ] update user documentation

#24 Add Markdown support for Articles

Description of problem
Currently it's not possible to format articles. This is unfortunate as longer text are hard to read and there is no possibility to add images or other kind of media.

Solution
Add support for markdown.

Additional context
TBD: How is preview mode integrated?

Criteria for acceptance

- [ ] validate markdown to prevent xss
- [ ] preview mode
- [ ] formatting bar
- [ ] display markdown instead of simple text

#25 Concept Extraction per Workspace

Describe the bug
The concept extraction is not handled on a per workspace basis. It is possible that Articles form different Workspaces get linked together.

To Reproduce
Steps to reproduce the behavior: 1. Change into a different workspace than the default one 2. Create new article 3. Enter 'Git' as text 4. The article is now linked to the Git article

Expected behavior
The concept graph has to be built separately for each workspace. Articles from different workspaces should never be linked together.

#26 Workspace Switch should take you back to ArticleList-View

Describe the bug
Clicking on a workspace in the sidebar should take you to the overview page of that specific workspace. Currently you have to click 'Articles' in the top navigation to get back to the article list view.

To Reproduce
Steps to reproduce the behavior: 1. Click on a article to open the article detail view 2. Click on a workspace icon in the side bar 3. Nothing happens

Expected behavior
Clicking on the workspace icon should take you to the overview page of that specific workspace.

#27 Detail View doesn't work in Firefox

Describe the bug
Opening the article detail view does not work and the following error is thrown: `TypeError: CSS2Properties doesn't have an indexed property setter for '0'`

To Reproduce
Steps to reproduce the behavior: 1. Open grafit.io in Firefox 63.0.3 2. Log in 3. Click on any article detail view 4. See error

Expected behavior
The article detail view is shown like in Chrome.
Additional context
I found a potential solution here and tried to apply the suggestions without any success. React—TypeError: CSS2Properties doesn't have an indexed property setter for '0'

#28 Properly handle XSS in search results

Describe the bug
The highlighting of the search results is currently done using dangerouslySetInnerHTML. This may be a XSS vulnerability and has to be fixed.

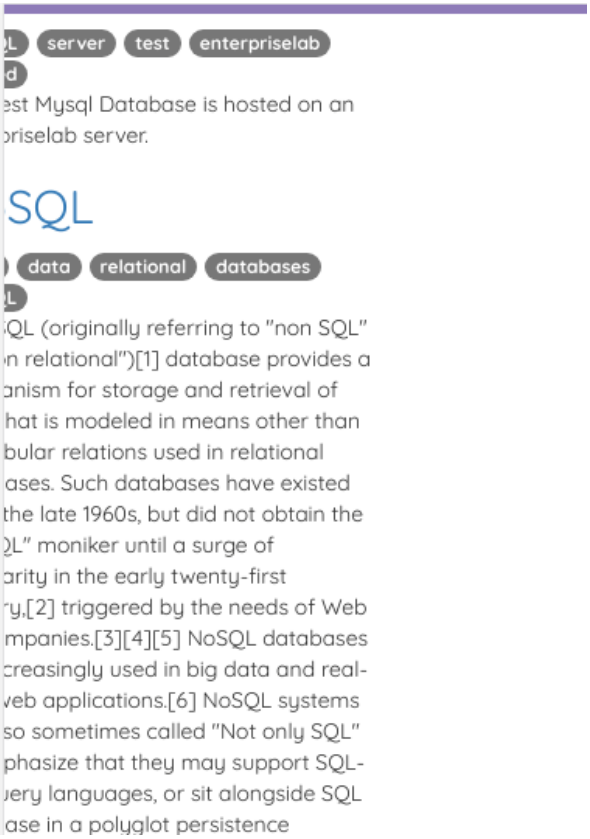
Fix
The highlighting of the search results has to be handled differently. Maybe with the same markdown render that we could use for the article content.

#29 Use First Line of Article as Title

Suggestion
Editing of articles could be made more seamless if instead of a separate input box the first line of the article would be interpreted as its title.

#30 Mobile Horizontal Scrolling

Describe the bug
The width of the webpage is set to wide on mobile devices. This allows the user to scroll horizontally.

Example: 

To Reproduce
Steps to reproduce the behavior: 1. Go to main page 2. Scroll horizontally

Expected behavior
The width of the page should match the width of the screen of the mobile device. Horizontal scrolling should not be possible.

#31 Handle cyclic Relations in Concept Surfing

Description of problem
Using the concept surfing mechanism it is possible to get into a cyclic tree view. E.g. the related node of a related node is the current node.

Solution
The nodes in the concept surfing mechanism should be filtered in a way that handles and hides cyclic relationships.

Criteria for acceptance

- [x] If the child of a node is the same as the parent , the child is not shown in the concept surfing tree

#32 Handle N-Grams in Concept Extraction

Description of problem
Currently the concept extraction algorithm does not take into regard n-grams. It should be possible for the concept extractor to find n-grams.

Solution
The keyphrase extraction algorithm has to be extended to be able to find n-grams. This is done by first tokenizing the text into sentences and then words. n-Grams are then generated from each sentence. Part of Speech Tagging is then used to tag each n-Gram (https://textblob.readthedocs.io/en/dev/quickstart.html#part-of-speech-tagging). The n-Grams are then filtered with the following three rules: - Each n-Gram has to contain a noun or foreign word - Each n-Gram has to start with a noun, foreign word or adjective - Each n-Gram has to end with a noun, foreign word or adjective After the filter process, the same TF-IDF algorithm as before should be used to determine the best keyphrases.

Criteria for acceptance

- ☑ The text ist tokenized into sentences and words
- ☑ N-Grams are generated on a per sentence basis with a n of 1,2 and 3
- ☑ Each word in each n-Gram is PoS-Tagged
- ☑ The n-Grams are filtered using the three rules above
- ☑ The n-Grams are then fed into the TF-IDF algorithm

#33 Short Text in Articlelist View

Description of problem
Currently the whole text of an article is displayed in the Articlelist View.

Solution
The Articlelist view should only display the first few lines of an article, no matter how long the article is.

Criteria for acceptance

- ☑ The Article list only displays the first three sentences or first 400 characters (whichever is shorter)

#34 Pagination of Articlelist View

Description of problem
Currently all articles are loaded into the overview page. This will slow down performance if there are a lot of articles in the workspace.

Solution
Use the DRF pagination to add pagination to the Articlelist View.

Criteria for acceptance

- ☑ Only the first 25 articles are loaded into the overview page
- ☑ A "load more" button or infinite scroll enables the user to load more articles

C Code Snippets

C.1 Beispiel Integrationstest

```python
class ArticleTest(APITestCase):
    def setUp(self):
        test_user = User.objects.create_user('testuser',
        ↪ 'test@example.com', 'testpassword')
        workspace = Workspace(name="Testworkspace", initials="TE")
        workspace.save()
        workspace.users.add(test_user)
        Article.objects.create(title="TestTitle", text="TestText",
        ↪ workspace=workspace)
        self.client = APIClient()
        self.user = User.objects.get(username='testuser')
        self.client.force_authenticate(user=self.user)
        self.workspace = workspace

    def test_article_list_workspace_permission(self):
        other_user = User.objects.create_user('test2',
        ↪ 'test2@example.com', 'testpassword')
        other_workspace = Workspace(name="Testworkspace",
        ↪ initials="TE")
        other_workspace.save()
        other_workspace.users.add(other_user)
        Article.objects.create(title="TestTitle2",
        ↪ text="TestText2", workspace=other_workspace)
        raw_response = self.client.get(
            reverse('article-list'),
            format="json")
        response = raw_response.data['results']
        self.assertEqual(raw_response.status_code,
        ↪ status.HTTP_200_OK)
        self.assertEqual(len(response),
        ↪ Article.objects.filter(workspace__users=
        ↪ self.user).count())
```

Listing 18: Integrationstest - Workspace Berechtigungen

C.2 Article Model

```python
class Article(models.Model):

    [...]

    title = models.CharField(max_length=250)
    text = models.TextField(blank=True)
    workspace = models.ForeignKey(Workspace,
    ↪  on_delete=models.CASCADE)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    web_content = models.TextField(null=True, default=None)
    update_web_content = models.BooleanField(default=True)

    @property
    def related(self):
        relatedNodes = []
        try:
            ownNode = GraphArticle.nodes.get(uid=self.id)
            relatedGraphNodes = ownNode.related
            for node in relatedGraphNodes:
                rel = ownNode.related.relationship(node)
                if not rel.hidden:
                    relatedNodes.append({
                        "id": int(node.uid),
                        "title": node.name,
                        "label": rel.label
                    })
        except:
            # article node does not exist yet
            pass

        return relatedNodes

    [...]
```

Listing 19: Article Model - Verkürzt

C.3 Docker Compose Files

```yaml
version: "2.3"

services:
  db:
    image: postgres

  neo4j:
    image: neo4j
    environment:
      NEO4J_AUTH: "neo4j/test"

  backend:
    image: grafitio/grafit-backend
    ports:
      - 8000:8000
    restart: always
    depends_on:
      - db
      - neo4j

  frontend:
    image: grafitio/grafit-frontend
    ports:
      - 3000:80
    restart: always
    depends_on:
      - backend

  documentation:
    image: grafitio/grafit-docs
    restart: always
    ports:
      - 8001:8001
```

Listing 20: docker-compose.yml - Produktiv

```yaml
version: "2.3"

services:
  db:
    image: postgres

  neo4j:
    image: neo4j
    ports:
      - 7474:7474
      - 7687:7687
    environment:
      NEO4J_AUTH: "neo4j/test"

  backend:
    image: grafit-backend
    build:
      context: ./backend
      target: base
    ports:
      - 8000:8000
    restart: always
    volumes:
      - ./backend:/code
    depends_on:
      - db
      - neo4j
[...]
```

Listing 21: docker-compose.dev.yml - Entwicklung - Teil 1

```
[...]
  frontend:
    image: grafit-frontend
    build:
      context: ./frontend
      target: base
    ports:
      - 3000:3000
    restart: always
    volumes:
      - ./frontend:/usr/src/grafit
      - /usr/src/grafit/node_modules
    depends_on:
      - backend

  documentation:
    restart: always
    build: ./mkdocs
    volumes:
      - ./mkdocs:/code
    ports:
      - 8001:8001
```

Listing 22: docker-compose.dev.yml - Entwicklung - Teil 2

D Arbeitsjournal

Datum	Stunden Julian	Bemerkungen
2018-10-01	15	Diverse Arbeitsorte; Übertrag Kickoff, Erstellung Arbeitsjournal, Recherche Issue/Anforderungstool, Dokumentationssetup
2018-10-02	0	
2018-10-03	1	UI Mockup
2018-10-04	5	Besprechung und Beschreibung Use Case & UI Mockup
2018-10-05	0	
2018-10-06	3	Doku
2018-10-07	0	
2018-10-08	4	Interview Waldis & Sigfried, Use Case
2018-10-09	0	
2018-10-10	0	
2018-10-11	0	
2018-10-12	5.5	Setup Domain & Repository, Recherche Technologieentscheid, React Tutorial
2018-10-13	0	
2018-10-14	1.5	GitHub Issue Tempaltes, Labels, Milestones, created Issue on issue.sh
2018-10-15	6	Besprechung Meeting, Scope, Use Case
2018-10-16	0	
2018-10-17	5.5	Doku Nachführung, Meeting
2018-10-18	4	Nachbesprechung Meeting, Mockups for Sprint 1, Aufbau Testkorpus
2018-10-19	0	
2018-10-20	0	
2018-10-21	1.5	Recherche Frontend, Interview Nutzer
2018-10-22	5.5	Setup Entwicklungsumgebung, Basic Frontend
2018-10-23	0	
2018-10-24	6.5	Setup Travis CI, Testkorpus
2018-10-25	0	
2018-10-26	0	
2018-10-27	1	catch up
2018-10-28	7.5	Dockerise development env, docs, add CORS
2018-10-29	7.5	status update, travis multistage build & deplyment to Dockerhub
2018-10-30	1	Markup Login
2018-10-31	4	Sprintplanung, Meeting
2018-11-01	0	
2018-11-02	4	Besprechung CI CD
2018-11-03	0	
2018-11-04	0	
2018-11-05	11.5	Frontend, fixed FOOSA test
2018-11-06	1	Logojoy
2018-11-07	0	
2018-11-08	0	
2018-11-09	3	Frontend - Favicon, Logo, Style
2018-11-10	0	
2018-11-11	5.5	Workspace Sidebar
2018-11-12	8	Workspace API, Navbar collaps, some UI refactoring
2018-11-13	0	
2018-11-14	7	Besprechung weiteres Vorgehen, Meeting Planung, Literaturstudium, Doku
2018-11-15	4.5	Literatur, Meeting
2018-11-16	1	Test Coverage
2018-11-17	0	
2018-11-18	0.5	Coverage fix
2018-11-19	9.5	Koordination Zwischenprästentation, Issues Sprint 3, code coverage
2018-11-20	0	
2018-11-21	8	Präsentation, Search API and Searuch UI
2018-11-22	10	Search und Präsentation
2018-11-23	0	
2018-11-24	0	
2018-11-25	0	
2018-11-26	6	Styling Frontend

2018-11-27	7	Search und Firefox Bug, Präsi
2018-11-28	6.5	Zwischenpräsentation, Review, Testing Workspace
2018-11-29	0	
2018-11-30	1	Demo bei whatwedo
2018-12-01	0	
2018-12-02	0	
2018-12-03	0	
2018-12-04	0	
2018-12-05	9	Fuzzy Search
2018-12-06	6.5	Tests and Seach Doku
2018-12-07	0	
2018-12-08	0	
2018-12-09	0	
2018-12-10	6	Doku
2018-12-11	0	
2018-12-12	3	Doku
2018-12-13	2	Sprint Meeting
2018-12-14	0	
2018-12-15	0	
2018-12-16	0	
2018-12-17	0	
2018-12-18	0	
2018-12-19	6	
2018-12-20	4.5	Doku
2018-12-21	0	
2018-12-22	0	
2018-12-23	0	
2018-12-24	0	
2018-12-25	2	
2018-12-26	0	
2018-12-27	5	Doku
2018-12-28	7	Doku - Anwendungsfall
2018-12-29	2	Doku, Review, Epics
2018-12-30	4	
2018-12-31	0	
2019-01-01	0	
2019-01-02	8	Doku
2019-01-03	8.5	Doku
2019-01-04	4	Doku
2019-01-05	3	Web Crawling
2019-01-06	4.5	Web Crawling
2019-01-07	8.5	Web Crawling - Cloud Intrgration
2019-01-08	11	Meeting, Doku, Issue Export
2019-01-09	7.5	Doku Cloud Integration & Fix of search
2019-01-10	7	Doku
2019-01-11	8	Doku
2019-01-12	0	Krank
2019-01-13	0	Krank
2019-01-14	2	Krank
2019-01-15	9	Korrekturlesen, Doku Suchindex
2019-01-16	9	Setup CoC, Contribution, Repo ReadMe
2019-01-17	10	Korrektur und Abschluss Dokumentation
2019-01-18	6	Abgabe

Datum	Stunden Robin	Bemerkungen
2018-10-01	5	Rotkreuz; Übertrag Kickoff, Lösungsfindung, Grundriss Architektur
2018-10-02	0	
2018-10-03	0	
2018-10-04	7	Rotkreuz; Besprechung Use Case, Lesen vorgängiger BDAs
2018-10-05	0	
2018-10-06	0	
2018-10-07	0	
2018-10-08	3	Interview Waldis & Sigfried
2018-10-09	0	
2018-10-10	0	
2018-10-11	4	Recherche Technologieentscheid
2018-10-12	5	Recherche Technologieentscheid, Test Django Rest Framework
2018-10-13	0	
2018-10-14	0	
2018-10-15	8.5	Epic Diagram, Besprechung Meeting 0
2018-10-16	0	
2018-10-17	5.5	Sprintplanung Sprint 1, Meeting
2018-10-18	4.5	Nachbesprechung Meeting, Aufbau Concept Extractor Modul
2018-10-19	0	
2018-10-20	1	Prototyp Django Rest Framework
2018-10-21	0	
2018-10-22	7	Setup Entwicklungsumgebung, Start with Concept extraction
2018-10-23	1	React Crashkurs
2018-10-24	6.5	TF-IDF implementation
2018-10-25	7	Aufbau DRF Backend, Docker compose initialisieren
2018-10-26	4	Postgres integration, improve Docker build with gunicorn
2018-10-27	1	Create Article Model and Endpoint, write initial data to DB
2018-10-28	0	
2018-10-29	10	Setup graph runner, article serializer, cors/ cofr stuff, frontend fetch articles from api
2018-10-30	0	
2018-10-31	5.5	Init React Router, Article Detail View, Meeting
2018-11-01	0	
2018-11-02	2	Evaluation Redux/ Mobx
2018-11-03	0	
2018-11-04	0	
2018-11-05	8.5	Login, Fix React Router, added Navbar, refactor API communication into services
2018-11-06	0	
2018-11-07	2	Nginx config for frontend prod build
2018-11-08	1	code formatting and minor fixxes
2018-11-09	8	editing and creating of new article nodes, registration form
2018-11-10	0	
2018-11-11	0	
2018-11-12	8.5	Neo4J init
2018-11-13	0	
2018-11-14	6	Besprechung weiteres Vorgehen, Meeting Planung
2018-11-15	2	
2018-11-16	0	
2018-11-17	0	
2018-11-18	0	
2018-11-19	9.5	Data from Neo4J, trigger keyword extraction on create/ edit
2018-11-20	0	
2018-11-21	5	IKC like concept surfing
2018-11-22	9	create workspace componente, research react state problem
2018-11-23	0	
2018-11-24	0	
2018-11-25	0	

2018-11-26	6	Handle Alerts, Document Surfing
2018-11-27	10	Search, Vorbereitung Meeting
2018-11-28	6	Zwischenpräsentation, Sprint Review
2018-11-29	6	Pagination, Refactoring Frontend
2018-11-30	3	Create shorttexts for articlelist overview
2018-12-01	0	
2018-12-02	0	
2018-12-03	0	
2018-12-04	0	
2018-12-05	7	Refactor Article Detail Components
2018-12-06	8	Enable hiding of related articles
2018-12-07	0	
2018-12-08	0	
2018-12-09	0	
2018-12-10	9	Rewrite related tags component, create new connections between nodes autosuggest existing nodes, create node if not exists
2018-12-11	2	filter out cyclic relations in concept surfing tree
2018-12-12	6	Sprint Meeting Vorbereitung und Doku, filter out cyclic relations in concept surfing tree
2018-12-13	2	Sprint Meeting
2018-12-14	4	Doku
2018-12-15	0	
2018-12-16	0	
2018-12-17	0	
2018-12-18	0	
2018-12-19	4	Doku Stand der Technik
2018-12-20	4	Doku Stand der Technik
2018-12-21	0	
2018-12-22	0	
2018-12-23	0	
2018-12-24	0	
2018-12-25	0	
2018-12-26	0	
2018-12-27	3.5	Doku, Beschreibung IKC und Lokahi
2018-12-28	9	Doku, Einführungskapitel
2018-12-29	2	Doku Einleitung
2018-12-30	7	Doku Stand der Technik - NLP
2018-12-31	4	Doku Stand der Technik - NLP
2019-01-01	0	
2019-01-02	8	Dokumentation Architektur und Technologie Entscheide - Client/ Server
2019-01-03	8	Dokumentation Architektur und Technologie Entscheide - Graphdatenbank
2019-01-04	7	Dokumentation Architektur und Technologie Entscheide - Webframework und Programmiersprache
2019-01-05	3	Korrektur Dokumentation
2019-01-06	0	
2019-01-07	8	Entwicklung: Label Connections, Group by Label
2019-01-08	7	Dokumentation Implementierung - Keyphrase Extraction
2019-01-09	8	Dokumentation Implementierung - Keyphrase Extraction
2019-01-10	9	Dokumentation Implementierung - Generierung Knowledge Graph
2019-01-11	8	Dokumentation Implementierung - Qualitätssicherung & Buildpipeline
2019-01-12	6	Dokumentation Implementierung - Persistenzlösung
2019-01-13	9	Webabstract
2019-01-14	9	Dokumentation Implementierung - Systemübersicht
2019-01-15	7	Dokumentation - Aufbau Frontend/ Backend
2019-01-16	9	Dokumentation - Entwickler Doku/ Benutzerdoku
2019-01-17	8	Dokumentation Korrektur
2019-01-18	4	Abgabe

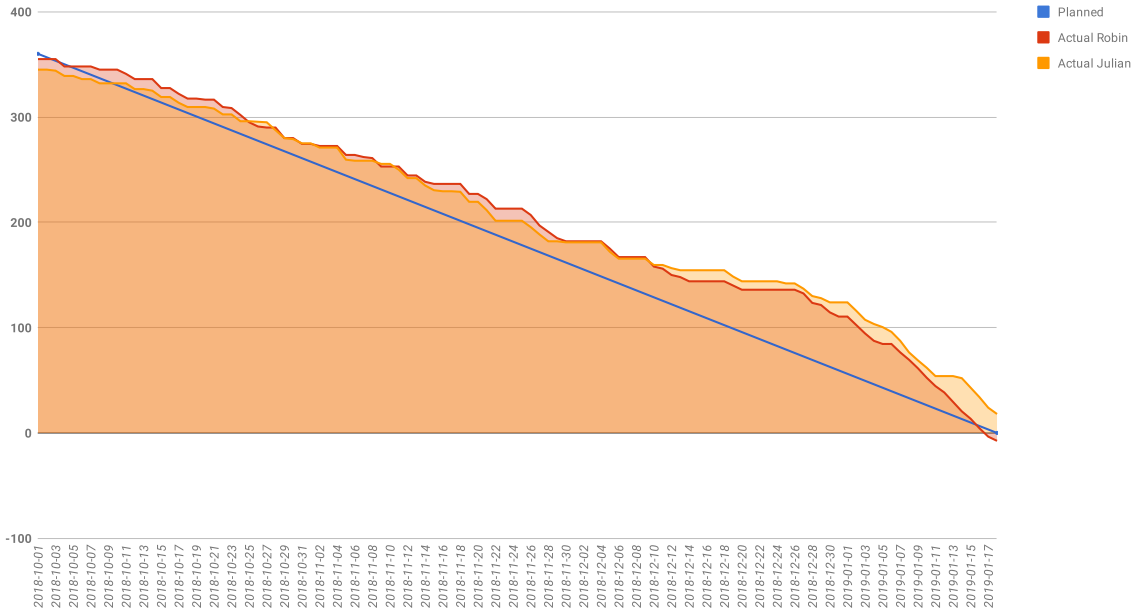


Abbildung 19: Arbeitszeit Burndown-Chart