

Erstellung von Personenprofilen im Internet durch Big Data

Studienarbeit

des Studiengangs Informatik
an der Dualen Hochschule Baden-Württemberg Stuttgart

von

Lukas Berwanger
Mat.-Nr.: 7482367
HUGO BOSS AG, Metzingen

Niels Heidbrink
Mat.-Nr.: 8308611
M-Way Solutions GmbH, Stuttgart

Juni 2021

Bearbeitungszeitraum
Betreuer
Vertraulichkeit

300 Stunden p.P., Semester 5 und 6
Jürgen Schneider
Stufe 1 “nicht vertraulich”

Abstract

In the context of this thesis, an application was developed that collects data available on the Internet about a person and thus creates a person profile based on the defined search parameters. Telephone directories, social networks and sports portals serve as sources. Concepts of the Semantic Web and Knowledge Graphs were used to connect individual data sets. The implementation showed that the quality of the data sources is crucial for the later result. While it turned out that most of the results of the data sources can be put into a standardized format with moderate effort, their quality leaves much to be desired in some places. This disadvantage is further aggravated by the limitation that data is not persisted over a longer period of time.

Fundamentally, this work nevertheless shows how important it is, even as a private individual, to be careful about what personal data you share on social networks, in forums or on other platforms. While some of these providers are legally bound to protect the data as best as possible, it is still the responsibility of the user to take appropriate measures (for example, in the form of privacy settings) to minimize the digital fingerprint.

While its productive use is rather difficult from a legal and economic point of view, the concept nevertheless shows some potential. In particular, individuals who may wish to control their digital fingerprint non-commercially or privately can build on this project.

Zusammenfassung

Im Rahmen dieser Arbeit wurde eine Applikation entwickelt, welche im Internet verfügbare Daten über eine Person sammelt und somit ein Personenprofil anhand der definierten Suchparameter erstellt. Als Quellen dienen Telefonbücher, soziale Netzwerke oder auch Sportportale. Zur Verbindung einzelner Datensätze kamen Konzepte des Semantic Web und Knowledge Graphs zum Einsatz. Die Umsetzung hat gezeigt, dass die Qualität der Datenquellen ausschlaggebend für das spätere Ergebnis ist. Während sich zwar herausgestellt hat, dass sich die meisten Ergebnisse der Datenquellen mit moderatem Aufwand in ein standardisiertes Format bringen lassen, lässt deren Qualität stellenweise durchaus zu wünschen übrig. Weiter verstärkt wird dieser Nachteil durch die Limitierung, dass Daten nicht über einen längeren Zeitraum persistiert werden.

Grundlegend zeigt diese Arbeit dennoch wie wichtig es ist, auch als Privatperson achtsam zu sein, welche persönlichen Daten man auf sozialen Netzwerken, in Foren oder auf weiteren Plattformen teilt. Während einige dieser Anbieter rechtlich daran gebunden sind, die Daten bestmöglich zu schützen, obliegt es zuletzt trotzdem der Verantwortung des Benutzenden dafür zu sorgen, entsprechende Maßnahmen (beispielsweise in Form der Privatspäreeinstellungen) zu unternehmen, um den digitalen Fingerabdruck zu minimieren.

Während ein produktiver Einsatz rechtlich und wirtschaftlich eher schwierig ist, zeigt das Konzept dennoch ein gewisses Potential. Vor allem Personen, die gegebenenfalls nicht-kommerziell oder privat ihren digitalen Fingerabdruck kontrollieren möchten, können auf diesem Projekt aufbauen.

Inhaltsverzeichnis

Abkürzungsverzeichnis	IV
Abbildungsverzeichnis	V
Tabellenverzeichnis	VI
Listings	VII
1 Einleitung	1
1.1 Problemstellung	1
1.2 Bereits bestehende Technologien	2
1.3 Erwartungen	2
2 Grundlagen	4
2.1 Architektur	4
2.2 Datenquellen und Scraper	5
2.3 API	11
2.4 Datenbank	15
2.5 Frontend	19
2.6 Monitoring	20
3 Konzept	22
3.1 Anforderungen	22
3.2 Architektur	24
3.3 Datenquellen und Scraper	25
3.4 API	28
3.5 Frontend	38
3.6 Monitoring	39
3.7 Bereitstellung	40
4 Umsetzung	42
4.1 Datenquellen und Scraper	42
4.2 API	49
4.3 Frontend	54
4.4 Bereitstellung	56
5 Fazit	59
5.1 Anforderungen	59
5.2 Haben sich die Erwartungen bewahrheitet?	60
Literatur	62

Abkürzungsverzeichnis

API	Application Programming Interface
BLOB	Binary Large Object
BSON	Binary JSON
CI/CD	Continuous Integration/Continuous Delivery
DOM	Document Object Model
GCP	Google Cloud Platform
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
NoSQL	Not only SQL
OWL	Web Ontology Language
RDFS	Resource Description Framework Schema
REST	Representational State Transfer
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
URL	Uniform Resource Locator
UUID	Universally unique identifier
XML	Extensible Markup Language
YAML	YAML Ain't Markup Language

Abbildungsverzeichnis

2.1	Grundlegende Übersicht der notwendigen Komponenten	4
2.2	Beispiel einer Ontologie	6
2.3	Beispiel eines Knowledge Graphs [9]	7
2.4	Node Event Loop [20]	11
2.5	Aufbau einer Message Queue	14
2.6	Komplexität der Datenstruktur pro Datenbanktyp [4]	18
2.7	Interne Prometheus Architektur [28]	20
3.1	Architekturübersicht	24
3.2	Speicherarchitektur	28
3.3	Aufbau des Graphen	31
3.4	Suche auf dem Graphen, Fall 1	32
3.5	Suche auf dem Graphen, Fall 2	32
3.6	Suche auf dem Graphen, Fall 3	33
3.7	Suche auf dem Graphen, Fall 4	33
3.8	Suche auf dem Graphen, Fall 5	34
3.9	Übersicht des Monitorings	39
4.1	Aufbau der Suchanfrage im Graphen	50
4.2	Ausschnitt des gefüllten Graphen	52
4.3	Screenshot der Login-Seite	54
4.4	Screenshot des Suchverlaufs	54
4.5	Screenshot der Suchparametereingabe	55
4.6	Screenshot der aktuellen Suchergebnisse	55
4.7	CI/CD Pipeline	57

Tabellenverzeichnis

3.1	Beispielhafte Bereinigung von Attributwerten	30
3.2	Darstellung der Berechnung zur Genauigkeit eines Werts	36
4.1	Übersicht der WebSocket Kanäle	56
5.1	Erfüllung der Anforderungen	59

Listings

2.1	Beispiel von Daten im XML Format	8
2.2	Beispiel von Daten im JSON Format	9
2.3	Beispielhafte SOAP Anfragedaten	12
3.1	Nutzdaten einer Suchanfrage	25
3.2	Beispiel einer REST API Antwort	26
3.3	Beispielkonfiguration einer Datenquelle	26
3.4	Beispiel für ein Scraper-Ergebnis	29
3.5	Pseudocode der Validatorfunktion	35
4.1	Antwort der fupa.net API	43
4.2	Konfiguration einer JSON Datenquelle	43
4.3	Konfiguration einer XML Datenquelle	44
4.4	Implementierung des generischen Scrapers (gekürzt)	46
4.5	Hinzufügen eines Jobs zur Message Queue	48
4.6	Empfangen der Jobs im Worker	48
4.7	Funktionen zum Login in der API	50
4.8	Query zum Hinzufügen eines Attributs	51
4.9	Unterscheidung der Art des Attributs	52
4.10	Abfrage aller Ergebnisse, die Regel 1 und 2 erfüllen	53
4.11	Docker Compose Konfigurationsdatei	57

1 Einleitung

In den letzten Jahren hat sich die im Internet stattfindende Kommunikation stark erhöht. Verschiedenste Dienste werden von nahezu allen Mitgliedern moderner Gesellschaften verwendet. Laut einer Statistik von Januar 2021 benutzen über 1 Milliarde Menschen Netzwerke wie WeChat, Instagram oder Facebook Messenger, über 2 Milliarden Menschen haben ein Benutzerkonto bei WhatsApp, YouTube oder Facebook [33]. Die dort hinterlegten Daten sind meist für einen unabsehbar langen Zeitraum gespeichert und erzeugen in Summe einen klar differenzierbaren Fingerabdruck, wodurch wenige Informationen ausreichen, um eine Person eindeutig zu identifizieren. Weiter werden über soziale Netzwerke diverse Informationen über Personen und deren Leben verfügbar gemacht und verteilt. Teilweise sind diese Informationen sogar ohne jeglichen Schutz (wie beispielsweise Authentifizierung oder Privatsphäreinstellungen) mit Suchmaschinen auffindbar.

1.1 Problemstellung

Viele Benutzende des Internets bzw. der angebotenen Dienste sind sich über das Ausmaß der über sie auffindbaren Daten nicht im Klaren. Diese Arbeit soll zeigen, mit welchem Aufwand und welcher Komplexität sich eindeutige Personenprofile anhand dieser Daten erstellen lassen. Hierzu werden einige Datenquellen zu Rate gezogen, welche üblicherweise viele personenbezogenen Daten von Benutzenden enthalten. Als Beispiel sind soziale Netzwerke wie Facebook oder Instagram aber auch digitale Telefonbücher oder Online-Jobbörsen wie LinkedIn zu nennen. Personenbezogene Daten schließen dabei unter anderem den Vor- und Nachnamen, den Wohnort, Telefonnummern oder auch verwendete Pseudonyme, also Benutzernamen, ein. Die im Rahmen dieser Arbeit entwickelte Applikation soll Benutzenden ermöglichen, über die Eingabe von einigen wenigen Suchparametern angereicherte Informationen über die Person zu erhalten, sprich das Personenprofil. Es können beliebig viele und beliebige Arten von Datenquellen angebunden werden. Ein Algorithmus verbindet die von den Datenquellen gelieferten Datenpunkte und erstellt hieraus das geforderte Personenprofil. Idealerweise können so mit nur wenigen bekannten Parametern stark angereicherte Datensätze erstellt werden. Beispielsweise könnte eine Suche nach Max Mustermann den Wohnort, den Arbeitsplatz und die Telefonnummer zurückliefern, obwohl diese dem Suchenden vorher nicht bekannt und auch nie Teil der initialen Suche waren.

1.2 Bereits bestehende Technologien

Es existieren bereits ähnliche Dienste und Technologien, welche es Kunden anbieten, durch die Angabe von einigen Suchparametern ein Portfolio mit potenziell übereinstimmenden Personen zu erhalten. Diese Arbeit grenzt sich in vier relevanten Punkten von diesen Diensten ab:

1. Die Suche und Ergebnisse sind (kosten-)frei zugänglich.
2. Die Suche basiert auf einer leeren Datenbasis.
3. Die Suche betrachtet nur öffentliche Quellen.
4. Das Ziel ist keine Persönlichkeitsanalyse im Sinne von Charakterzügen oder Emotionen.

Während die angebotenen Dienste natürlich nicht preisgeben, wie die Ergebnisse zusammengestellt werden ist davon auszugehen, dass mit qualitativ hochwertigen Bestandsdaten gearbeitet wird. Potenziell werden Personendaten über mehrere Iterationen angereichert und kuriert, um ein möglichst sauberes Datenbild zu erschaffen. So ist es möglich, die von einem Kunden angegebenen Suchparameter effizient mit der Datenbasis abzugleichen. Weiter wird auch keine Angabe gemacht, von welchen Quellen die gefundenen Daten stammen. Sie könnten von öffentlichen Quellen stammen, es könnte aber auch verschiedene Datenverarbeitungsabkommen und private Quellen geben, von welchen diese Daten stammen. Die Suche solcher kurierter Datenquellen erfordert allerdings großen wirtschaftlichen Aufwand und liegt entsprechend nicht im Rahmen dieser Arbeit.

1.3 Erwartungen

Aufgrund der oben angesprochenen Einschränkungen ist die initiale Erwartung an das Ergebnis des Projekts, dass es vermutlich schwer ist, ein qualitativ hochwertiges und vollständiges Personenprofil für jede Suchanfrage zu liefern. Die Datenquellen sind in ihrem Umfang stark limitiert und die Betreiber teilweise selbst an der Privatsphäre ihrer Benutzenden interessiert, was den öffentlichen Zugriff einschränkt oder ganz verhindert. Weiter stellt die Verbindung der gefunden Ergebnisse verschiedener Quellen eine Hürde dar, da der Zusammenhang potenziell nicht trivial hergestellt werden kann. Vor allem bei häufigen Vorkommnissen beispielsweise eines Namens ist die finale Auswahl der für die Suche tatsächlich relevanten Person unter Umständen komplex. Da weiter, wie erwähnt, nicht auf Bestandsdaten zurückgegriffen werden kann ist die Erwartung hier, dass eine gewisse intelligente Entscheidung oder die Eingabe spezifischerer Suchparameter des

Benutzenden erfolgen muss, um die gefundenen Ergebnisse endgültig zu kurieren. Sprich ein Benutzer wird bei beendeter Suche aufgefordert, die Ergebnisse auf Plausibilität zu prüfen und gegebenenfalls korrekte von inkorrekten Ergebnissen zu unterscheiden.

2 Grundlagen

In diesem Kapitel werden die im Rahmen des Projekts verwendeten Grundlagen und Technologien erklärt. Um das Verständnis für die Zusammenhänge der einzelnen Komponenten zu verbessern, wird zu Beginn die Architektur vorgestellt.

2.1 Architektur

Abbildung 2.1 zeigt die Architektur des Projekts. Die Daten fließen von rechts nach links, von den externen Datenquellen über das Application Programming Interface (API) und die Datenbank zu den Benutzenden im Frontend.

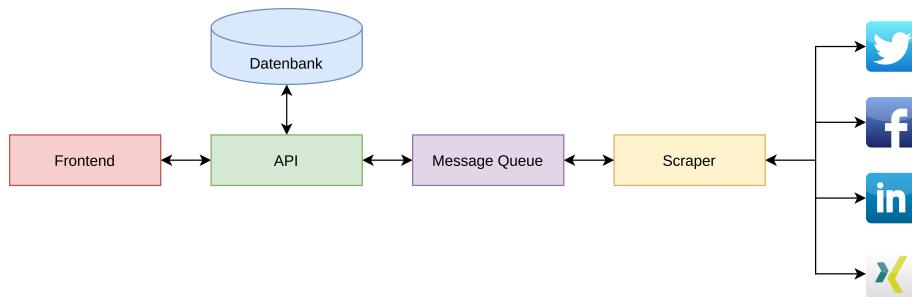


Abbildung 2.1: Grundlegende Übersicht der notwendigen Komponenten

2.1.1 Microservice-Architektur

Es handelt sich bei diesem Aufbau um eine Microservice-Architektur. Dies bedeutet, dass jede Komponente einen sogenannten Microservice darstellt. Dieser erfüllt im Idealfall eine dedizierte Aufgabe und hat fest definierte Schnittstellen und Schemata für Eingaben und Ausgaben.

Es gelten fünf Prinzipien für die Entwicklung von Microservices: [30]

- Wart- und Testbar
- Lose Kopplung zwischen Microservices
- Unabhängig bereitstellbar
- Fokussiert auf die Geschäftsprozesse

- Verwaltet von einem kleinen Team

In der Praxis bieten sich für eine Microservice-Architektur Container an. Diese ermöglichen durch die Verwendung verschiedener Prozess-Namespaces für die einzelnen Container, dass keine Dateien oder ähnliches zwischen den Containern ausgetauscht werden, sofern nicht gewünscht. Gleiches gilt für Netzwerkkommunikation oder umgebungsspezifische Einstellungen, wie beispielsweise Variablen. Weiter basieren Container immer auf einem Basis-Image. Änderungen die während der Laufzeit des Containers an Dateien gemacht werden, gehen nach dem Ableben des Container-Prozesses verloren. Auch kann durch diese Images sichergestellt werden, dass nach jedem Start des Containers exakt der gleiche Stand erreicht wird, unabhängig von Bibliotheken oder Applikationen, welche potenziell auf dem Host-Betriebssystem installiert wurden. Docker¹ ist eine Software, welche die Container- und Imageverwaltung mittels benutzerfreundlicher Befehle ermöglicht.

2.2 Datenquellen und Scraper

Nachdem nun die verwendete Architektur grob gezeigt wurde, werden im Folgenden die Datenquellen und Scraper beschrieben. Da dieses Projekt primär auf Konzepten von Big Data und der zugehörigen Datenanalyse basiert, werden diese Grundlagen an dieser Stelle auch kurz erläutert.

2.2.1 Big Data

Bei Big Data handelt es sich um die Verarbeitung (sehr) großer Datenmengen. Dabei ist die Unterscheidung von “normalen” Daten, dass das Volumen von üblichen Verarbeitungslösungen nicht in einer angemessenen Zeit verarbeitet werden kann. Für die Definition von Big Data finden sich meist drei Kriterien: [21, 22, 24]

- Umfang (“Volume”): Die Menge der zu analysierenden und verarbeitenden Daten.
- Varietät (“Variety”): Die verschiedenartige Struktur verschiedener Daten.
- Schnelllebigkeit (“Velocity”): Die Geschwindigkeit, in welcher Daten produziert werden.

Je nach Quelle werden weiter die folgenden zwei Kriterien in die Definition aufgenommen:

- Richtigkeit (“Veracity”): Die Qualität der Daten. Unvollständige oder falsche Daten erschweren die Verarbeitung.

¹ Siehe auch: <https://www.docker.com/>

- Wert (“Value”): Der Unternehmenswert soll durch die Verarbeitung der Daten gesteigert werden.

Da in diesem Projekt mit Datenquellen mit potenziell sehr großen Datensätzen (zu nennen sind zum Beispiel große soziale Netzwerke wie Facebook oder Twitter) gearbeitet wird, ist die Betrachtung von Big Data hier auch relevant. Es sei allerdings gesagt, dass durch spezifische Suchen auf diesen Quellen die Datensetze initial reduziert werden kann, wodurch nicht zwangsläufig eine spezifische Big Data Lösung verwendet werden muss.

2.2.2 Semantic Web und Knowledge Graph

Durch die Entwicklung des Internets, und zugehörig auch die des World Wide Web, ist der Bedarf dafür, Wissen in einem maschinenlesbaren Format anzubieten, stark gestiegen. Dies ist auch dadurch bedingt, dass beispielsweise durch Suchmaschinen relevante Daten aus größtenteils unstrukturierten und für Menschen gedachten Daten extrahiert werden müssen. Eine vorgeschlagene Lösung für dieses Problem ist der Einsatz von künstlicher Intelligenz, um die Daten zu analysieren und Erkenntnisse hieraus zu ziehen. Dies ist allerdings in der Hinsicht problematisch, dass sich aufgrund der schieren Menge der täglich im Internet generierten Daten deren Verarbeitung aufwändig und kostspielig darstellt. Aus diesem Grund wurde ein zweiter Ansatz entwickelt: Wissensrepräsentation mittels des Semantic Web. Abbildung 2.2 zeigt, wie eine Ontologie (vereinfacht gesagt eine Wissensbasis) in einem Semantic Web aussehen kann. Ein Subjekt wird mittels eines Prädikats mit einem Objekt verbunden. In diesem Fall ist die Aussage, dass der Springer Verlag das Buch Semantic Web verlegt. [10]



Abbildung 2.2: Beispiel einer Ontologie

Während dies ein sehr einfaches Beispiel ist, sollte die Erweiterbarkeit und Mächtigkeit dieses Ansatzes dennoch deutlich werden. Je mehr dieser Tripel (Subjekt-Prädikat-Objekt) in einem solchen Semantic Web zusammengeführt werden, desto mehr Aussagen lassen sich am Ende über dieses ableiten, sowohl explizit als auch implizit. Während weitere, im Kontext des Semantic Web eingesetzte Paradigmen, wie beispielsweise das **Resource Description Framework Schema (RDFS)** oder die **Web Ontology Language (OWL)**, notwendig sind, um komplexere Aussagen abzuleiten, sollen diese hier nicht weiter beschrieben werden, da in dieser Arbeit nur die grundlegende Idee Anwendung findet. [10]

Eine Erweiterung des Semantic Web ist der Knowledge Graph. Wie der Name bereits suggeriert, handelt es sich ebenfalls um eine Wissensbasis, explizit in Form eines Gra-

phen. Ein wichtiger Unterschied ist, dass ein Knowledge Graph nicht zwangsläufig den Beschränkungen eines Semantic Web folgen muss. Somit kann jedes Semantic Web als Knowledge Graph dargestellt werden, allerdings ist nicht jeder Knowledge Graph gleichzeitig ein Semantic Web im engeren Sinne. Ein Beispiel für einen Knowledge Graph ist in Abbildung 2.3 dargestellt. Hier können Personen Filme bewerten und sollen anhand dieser weitere Filme, welche sie gegebenenfalls interessieren könnten, vorgeschlagen bekommen. [15, 23]

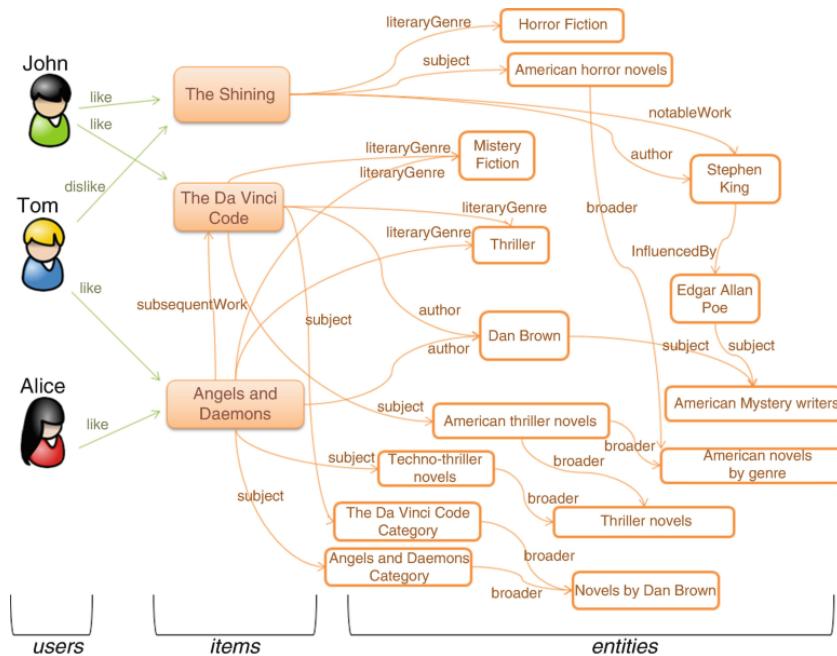


Abbildung 2.3: Beispiel eines Knowledge Graphs [9]

Wie auch in der Abbildung gezeigt wird, existieren nicht nur direkte Verbindungen zwischen den einzelnen Personen und den jeweiligen Filmen, sondern auch zu deren jeweiligen Attributen (unter anderem Thema oder Kategorie). Durch diese indirekten Verbindungen ist es möglich, weitere Informationen aus wenigen Datenpunkten zu generieren. Da auch in diesem Projekt Daten verschiedener Personen zusammengetragen werden sollen, deren Relation zueinander allerdings nicht von Beginn an klar ist, kann sich ein auf Knowledge Graphen basierter Ansatz durchaus anbieten.

2.2.3 Datenquellen

Bei den Datenquellen handelt es sich, wie am Anfang des Kapitels bereits erwähnt, um diverse online verfügbare Dienste und Portale. Dazu zählen soziale Netzwerke, Jobbörsen oder auch Telefonbücher. Die Form, in welcher diese die Nutzdaten zur Verfügung stellen unterscheidet sich dabei relativ stark. Sowohl im tatsächlichen Inhalt als auch in der äußeren Struktur. So gibt es als bekanntere Datenformate beispielsweise Extensible Markup

Language (XML) (mit der Spezialisierung Hypertext Markup Language (HTML)) und JavaScript Object Notation (JSON). Weiter unterscheidet sich die Art in welcher die Daten abgefragt werden. Handelt es sich bei der Datenquelle um eine API sind die gelieferten Daten mit hoher Wahrscheinlichkeit bereits vorstrukturiert. Im Gegensatz dazu müssten kosmetische Daten und Nutzdaten bei Webseiten erst voneinander getrennt werden.

XML und HTML

Listing 2.1 zeigt den beispielhaften Aufbau einer XML Nachricht. Jedes Element wird dabei von öffnenden und schließenden Elementen (den sogenannten Tags) umschlossen. Durch dieses Format ist es möglich, auch zusammengesetzte Strukturen abzubilden. Komplexer wird die Darstellung von Auflistungen. Praktisch wird dies oft über die Mehrfachnennung eines Tags im Elterntag gelöst, XML selbst sieht allerdings keine spezifische Syntax für Strukturen dieser Art vor. Ein syntaktisch prinzipiell korrekter String kann semantisch fehlerhafte oder doppeldeutige Interpretationen haben. Entsprechend ist das Parsing einer solchen Struktur mit einigen Schwierigkeiten verbunden. Nicht zuletzt muss der Parser jederzeit den letzten gelesenen öffnenden Tag speichern, um die Nutzdaten dem korrekten Element zuordnen zu können.

```
1 <root>
2   <person>
3     <givenName>Max</givenName>
4     <familyName>Mustermann</familyName>
5   </person>
6 </root>
```

Listing 2.1: Beispiel von Daten im XML Format

Durch die Implementierung des sogenannten XPaths wird dieses Problem zumindest teilweise gelöst. Dieser erlaubt durch die Verwendung von vorgegebener Syntax, XML Strukturen zu traversieren. So könnte beispielsweise auf den Vornamen der Person (XML-Tag `givenName`) über den XPath `/root/person/givenName` zugegriffen werden. Diese Technik ist in der Behandlung von XML üblich und sogar, im Zusammenhang mit Webseiten, zur Manipulation des Document Object Model (DOM) in den meisten Browsern unterstützt. [26]

Da HTML auf XML basiert, gelten alle angesprochenen Strukturen und Limitierungen auch hier. Ein wichtiger Unterschied ist allerdings, dass HTML einen festen Namespace mit vorgegebenen Tags definiert, welche zur Entwicklung von Webseiten standardmäßig benutzt werden kann. Bei Bedarf ist die Erweiterung dieses Namespaces natürlich weiterhin möglich, allerdings weiß der verwendete Browser beim Einsatz der Standardelemente wie diese grundlegend darzustellen sind. Neben den Nutzdaten existieren noch diverse Tags

und Attribute zur Darstellung der Webseite selbst. Üblicherweise sind diese, im Kontext eines Datenprojekts, nicht sonderlich interessant und bei der Verarbeitung eher hinderlich. [34]

JSON

Listing 2.2 zeigt das vorherige Beispiel nun im JSON Format. Man sieht, dass Objekte hier auch ohne expliziten Namen existieren können und keine Doppelbenennung des Attributbezeichners vorkommt. Weiter sind komplexe Typen wie Listen oder Objekte vorab in JSON mit einer eigenen Syntax versehen. Dies ermöglicht die bessere maschinelle Verarbeitung. Zusätzlich wird, durch die Minimierung unnötiger Zeichen, die Lesbarkeit für Menschen wesentlich einfacher.

```
1 [
2   {
3     "givenName": "Max",
4     "familyName": "Mustermann"
5   }
6 ]
```

Listing 2.2: Beispiel von Daten im JSON Format

Ähnlich dem XPath existiert auch für JSON die Möglichkeit, Strukturen zu traversieren: den sogenannten JSONPath. Wenn erneut der Vorname der Person ausgegeben werden soll, sieht der JSONPath wie folgt aus: `\$.[*].givenName` Der `[*]`-Operator würde über jedes Element in der Elternliste iterieren und das gewünschte Attribut, hier `givenName`, zurückgeben.

2.2.4 Scraper

Da nun die Grundlagen zu den in dieser Arbeit betrachteten Datenformaten gelegt wurden, kann mit der Beschreibung der Scraper begonnen werden. Dazu zählt unter anderem, was ein Scraper ist, welche Aufgaben dieser erfüllen soll und wie die Datenquellen angebunden werden. Wie bereits angesprochen handelt es sich bei den Datenquellen um zwei grundlegende Typen: APIs und Webseiten. In jedem Fall wird entweder XML oder JSON zurückgegeben. Je nach Bedarf können bei der Umsetzung der verschiedenen Scraper-Arten die Stärken von verschiedenen Sprachen und Technologien gut ausgenutzt werden, da diese nur dadurch eingeschränkt sind, dass sie über standardisierte Schnittstellen kommunizieren können. Solange die ausgewählte Sprache dieser Anforderung genügt, kommt sie für die Umsetzung in Frage. Wie folgt wurde sich im Rahmen dieses Projektes allerdings dafür

entschieden, alle Scraper mit derselben Technologie (Node.js) umzusetzen, da bei dieser im Projektteam bereits Erfahrungen vorliegen, welche die Implementierung vereinfachen.

Ein Scraper ist in allgemeiner Form eine Applikation, welche (regelmäßig) die Inhalte des Internets abfragt und herunterlädt oder anderweitig verarbeitet. Dabei ist der Umfang variabel. Dies bedeutet, dass bei der Entwicklung eines Scrapers die Webseiten, welche abgefragt werden, auf die beschränkt werden können, welche von Interesse sind, weil sie beispielsweise für den Geschäftszweck interessante Daten beinhalten. Weiter ist es möglich, dass ein Scraper verlinkte Inhalte selbst verfolgt. Beispielsweise könnte eine LinkedIn Seite einer Person weiter Links zu Artikeln oder einer privaten Website enthalten, welchen der Scraper folgen kann. Ein Scraper kann weiter Quellen wie APIs oder ähnliches Abfragen. Dabei weicht er allerdings von der originalen Definition ab, da es sich in diesem Fall wohl eher um einen Fetcher statt um einen echten Scraper handelt (da die Daten bei einer API meist bereits strukturiert vorliegen). [3]

2.2.5 Node.js

Node.js ist ein im Jahr 2009 von Ryan Dahl entwickeltes Framework zur Entwicklung von Webapplikationen. Es basiert auf der V8 JavaScript Engine von Google und erlaubt damit JavaScript (oder kompatiblen, beispielsweise TypeScript) Code serverseitig auszuführen. Damit ist es nun auch möglich, sowohl das Frontend als auch APIs mit JavaScript zu schreiben, wo es vorher nur mit Technologien wie Ruby on Rails oder PHP möglich war. [11]

Es sollen im Speziellen fünf Punkte herausgestellt werden: [11]

- Asynchrone Programmierung: Im Gegensatz zur üblichen synchronen Abarbeitung von Funktionsaufrufen kann bei der asynchronen Programmierung nicht davon ausgegangen werden, dass ein folgender Funktionsaufruf die Ergebnisse eines vorherigen verwenden kann, da diese gegebenenfalls noch nicht vorliegen. Ein Beispiel hierfür wäre eine Netzwerkoperation, welche etwas Zeit benötigt, bis die Antwort des Servers empfangen wurde. Somit muss eine asynchrone Funktion entweder explizit darauf hinweisen, dass auf das Ergebnis gewartet werden muss oder ein Callback (wiederum selbst eine Funktion) wird angegeben, welches nach der Beendigung der asynchronen Operation aufgerufen wird.
- Asynchrones I/O: Ähnlich wie bei der asynchronen Programmierung ist auch die Verwendung von externen Systemen (beispielsweise das Dateisystem oder eine Datenbank) asynchron zu behandeln. Auch hier kommen Callbacks zum Einsatz, welche es ermöglichen, den Fluss der Applikation nicht zu unterbrechen.

- Einfachheit: JavaScript kann für Anfänger eine sehr geeignete Sprache sein. Somit fällt auch der Einstieg mit Node.js verhältnismäßig einfach. Dennoch gibt es durchaus auch Gegner von JavaScript und Probleme, welche historisch mit der Entwicklung von JavaScript und Node.js gewachsen sind.
- Integration mit **JSON**-basierten Services: **JSON**-basierte APIs oder Datenbanken wie beispielsweise MongoDB eignen sich sehr gut in Verbindung mit Node.js, da **JSON**-parsing standardmäßig in die Sprache und das Framework eingebaut ist.
- Erweiterbarkeit über Pakete: Ein riesiger Vorteil von Node.js und der direkten Umgebung ist die eingebaute Paketverwaltung. Durch diese können Applikationen um beliebige weitere Funktionalität erweitert werden, ohne großen Aufwand bei der Entwicklung zu erzeugen. Wobei angemerkt sei, dass es nicht immer einfach ist, das korrekte Paket zu finden, vor allem wenn mehrere Pakete dieselben Funktionen versprechen.

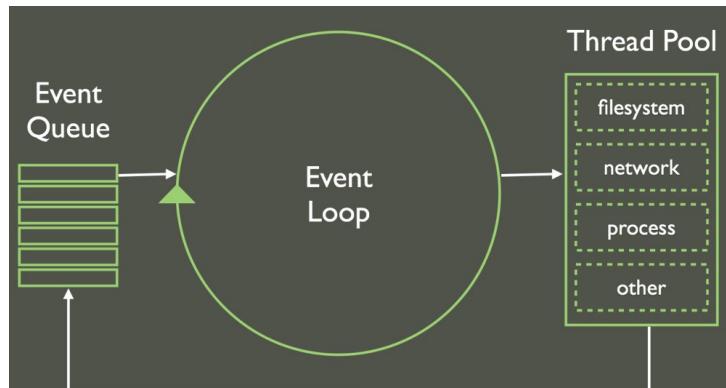


Abbildung 2.4: Node Event Loop [20]

Abbildung 2.4 zeigt den Aufbau des Event Loop, welches im Kern von Node.js steht. Wie bereits angesprochen ist Node.js ein asynchrones Framework. Entsprechend existiert die Event Queue, welche alle abzuarbeitenden Befehle enthält und reihum vom Event Loop abgearbeitet wird. Sollten aus dem Thread Pool weitere Daten vorhanden sein, werden diese in der Schleife auch abgefragt. Dies limitiert Node.js allerdings zu einem inhärent single-threaded Framework. [20]

2.3 API

Bei einer **API** handelt es sich um eine Schnittstelle, welche meist über das Netzwerk konsumiert wird. Üblicherweise können mit einer **API** diverse Funktionen abgedeckt werden, welche Daten (beispielsweise in einer Datenbank) lesen, erstellen oder bearbeiten. Es gibt keine festgelegte Technologie, in welcher eine **API** implementiert werden kann.

Weiter ist die Form der Daten frei wählbar, gewöhnlich wird allerdings mit **XML** oder **JSON** gearbeitet. So gibt es prinzipiell drei Arten von **APIs**: **Simple Object Access Protocol (SOAP)**, **Representational State Transfer (REST)** und **GraphQL**.

SOAP

SOAP APIs basieren auf dem Austausch von **XML** Nachrichten. Anfragen werden dabei immer über die **Hypertext Transfer Protocol (HTTP)** POST Methode gemacht, wobei sich die Nutzdaten im sogenannten Envelope befinden. In Listing 2.3 ist eine beispielhafte Anfrage an eine **SOAP API** dargestellt. In diesem Fall wurde der Benutzer mit dem Vornamen “Max” angefragt. Zurückgegeben wird ein entsprechender Envelope mit den Benutzerdaten, sofern vorhanden. Diesen gilt es mittels der im Kapitel 2.2.3 gezeigten Methode zu verarbeiten, um die Nutzdaten in ein einheitliches Format zu bringen. [36]

```
1 <soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
2   xmlns:m="http://www.example.org">
3   <soap:Header>
4   </soap:Header>
5   <soap:Body>
6     <m:GetUser>
7       <m:GivenName>Max</m:GivenName>
8     </m:GetUser>
9   </soap:Body>
10 </soap:Envelope>
```

Listing 2.3: Beispielhafte **SOAP** Anfragedaten

REST

REST wurde erstmal von Roy Fielding in seiner Dissertation [16] aus dem Jahr 2000 definiert. Es beschreibt eine Architektur für verteilte Netzwerkapplikationen. Fielding definiert sechs Anforderungen an eine solche **API**: [35]

- Client-Server: Anwender und Server müssen unabhängig voneinander funktionieren.
- Stateless: Die Anfrage muss alle relevanten Informationen, um diese auszuführen, selbst beinhalten.
- Layered System: Es werden verschiedene Schichten definiert (beispielsweise Proxy, Firewall, Gateway) welches unabhängig voneinander hinzugefügt, entfernt oder ausgetauscht werden können.

- Cache: Der Server soll definieren, ob die Antworten zwischengespeichert werden dürfen und, falls ja, unter welchen Bedingungen. Dies soll dazu führen, die Last des Servers zu reduzieren.
- Uniform Interface: Die Schnittstellen, welche zur Interaktion zwischen Anwender und Server verwendet werden, sollen einheitlich festgelegt sein. So kann sichergestellt werden, dass Änderungen entweder beim Anwender oder beim Server möglich sind, solange weiterhin die vereinbarte Schnittstellendefinition eingehalten wird.
- Code on demand: Anwender können ihre Funktionalität erweitern, indem der Server ausführbare Skripte zur Verfügung stellt. Diese Anforderung ist die einzige optionale und nicht jede API benötigt diese Flexibilität.

REST APIs können theoretisch mit beliebigen Technologien umgesetzt werden. Netzwerkinfrastruktur und Transportprotokoll sind also frei wählbar. Mittlerweile ist es allerdings üblich, diese APIs mittels **HTTP** bereitzustellen. Weiter ist das Datenformat durch die Architektur nicht limitiert, denkbar wären Implementierungen mit **XML**, **JSON** oder weiteren Formaten. In der Praxis findet sich im Zusammenhang mit REST häufig auch **JSON** als Datenformat. [35]

GraphQL

Die neueste Architektur für APIs findet sich in GraphQL. Das Ziel hierbei ist, Limitierungen vorheriger Ansätze schon in der Konzeption zu limitieren. Dabei unterscheidet sich zwar nicht das Datenformat, welches bei Antworten der API verwendet wird (normalerweise weiterhin **JSON**), sondern die Struktur der Anfrage. Konsumenten einer GraphQL API definieren für alle (lesenden) Anfragen exakt, welche Attribute für sie relevant sind. So kann von Beginn an der Overhead bei der Datenübertragung und Verarbeitung reduziert werden. Mittels dieses Schemas ist es auch möglich, schreibende Aktionen durchzuführen wobei allerdings gesagt sei, dass eine hybride Lösung, in welcher die Vorteile sowohl von GraphQL als auch von REST kombiniert werden, auch möglich ist. [17]

Nachdem nun die Typen der für das Projekt relevanten Datenquellen vorgestellt wurden, sollte ein grundlegendes Verständnis dafür entstanden sein, wie die Verarbeitung der Daten im Konzept erfolgen kann.

2.3.1 Message Queue

Um die Daten zwischen den einzelnen Scrapern und der zentralen Schnittstelle der API zu teilen, bietet sich eine Message Queue an. Diese erlaubt, Jobs zu erstellen und verteilt

diese automatisch an sogenannte Worker. Jobs können einer spezifischen Queue hinzugefügt werden, wodurch gesteuert wird, welche Worker-Node diesen Job prozessieren kann. Ein Beispiel hierfür wäre ein Bildverarbeitungsjob, welcher nur von einem Worker mit ausreichender Rechenkapazität durchgeführt werden kann, wobei Textverarbeitung von einem beliebigen freien Worker bearbeitet wird. Ein Beispiel für den Ablauf eines Jobs ist in Abbildung 2.5 dargestellt. [5, 29]

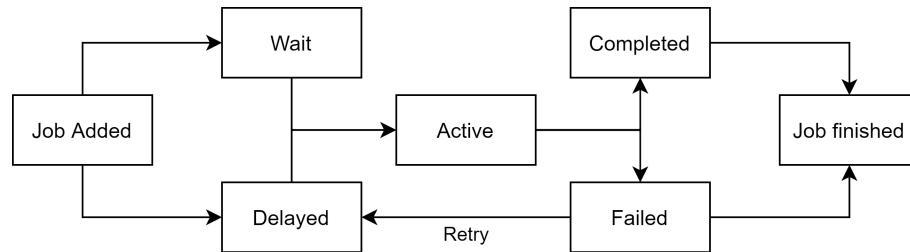


Abbildung 2.5: Aufbau einer Message Queue

Das Diagramm zeigt, wie ein Job erstellt wird und welche Stadien er durchläuft, beziehungsweise durchlaufen kann, bis er ein finales Ergebnis liefert. Der Einsatz einer Message Queue (oder vergleichbaren Technologie) birgt den Vorteil, dass Suchanfragen nicht manuell auf die einzelnen Scraper aufgeteilt werden müssen und Scraper nicht ausversehen den gleichen Job, also die gleiche Suchanfrage, doppelt bearbeiten. Durch die Implementierung existierender Lösungen in Form von Bibliotheken oder Produkten lässt sich der Aufwand weiter minimieren. Üblicherweise speichert eine Message Queue die Liste der Jobs sowie deren Status in einer Datenbank. So kann auch im Falle eines (Teil-)Ausfalls des Systems der korrekte Stand wiederhergestellt werden. [5, 29]

Alternativ könnte eine Worker-ähnliche Architektur auch mittels WebSockets implementiert werden. In diesem Fall hören alle Clients auf das Auslösen eines bestimmten Events und starten mit den empfangenen Daten die Verarbeitung. Dies kostet potenziell unnötige Ressourcen, falls mehrere gleichartige Worker dieselbe Suchanfrage bearbeiten. Dieses Problem verstärkt sich vor allem in einer Umgebung, in welcher Worker dynamisch skaliert werden. Zusätzlich ist die Überwachung der einzelnen Jobs sowie die Realisierung von Retry-Strategien im Fehlerfall bei WebSockets mit erhöhtem Aufwand verbunden.

2.3.2 WebSocket

WebSockets sind eine Technologie, welche bidirektionale Echtzeitkommunikation zwischen Client oder Server ermöglicht. Technisch wird dabei bei der initialen Verbindung eine Socketverbindung geöffnet, welche über die gesamte Dauer der Kommunikation bestehen bleibt. Daten können so ohne erneuten Verbindungsauflauf und ohne Polling des Servers auf Seiten des Clients in Echtzeit empfangen und gesendet werden. Häufig finden sich

sogenannte Events, auf welche sich ein Client registrieren kann. Sollten neue Daten vorliegen fügt der Server diese dem Event-Stream hinzu und alle Clients, welche sich für das Event registriert haben, erhalten die neuen Daten und können bei Bedarf antworten. So ergibt sich beispielsweise die Möglichkeit einer effizienten Chat-App, in welcher über ein `sendMessage` Event neue Nachrichten an den Server propagiert werden, der diese wiederum an alle registrierten Clients weiterleitet. [1, 31]

Auch wenn WebSockets in diesem Projekt für die Kommunikation zwischen API und Scraper nicht relevant sind, bietet sich der Einsatz für die Datenübertragung an das Frontend durchaus an, um Suchergebnisse live verfolgbar zu machen.

2.4 Datenbank

Von den Scrapern gesammelte Daten werden idealerweise in einer Datenbank gespeichert. Hierfür gibt es diverse Möglichkeiten von Structured Query Language (SQL) oder Not only SQL (NoSQL) Datenbanken. Während beide Typen, wie erwartet, Daten persistieren können, unterscheidet sich die Art und Weise in welcher sie dies bewerkstelligen durchaus relativ stark. Um in der Konzeptionsphase eine passende Auswahl für die Datenbank treffen zu können, werden deshalb diverse Optionen erklärt und zuletzt Anforderungen definiert, anhand welcher die Auswahl der Datenbank geschehen kann.

Relational

Relationale Datenbanken folgen dem relationalen Modell. Sie werden häufig auch SQL Datenbanken genannt, begründet durch die Abfragesprache, welche zum Einsatz kommt. Diese Art von Datenbanken zeichnet sich dadurch ab, dass sie einem festen Schema folgen, welches für jede Tabelle die enthaltenen Spalten und zugehörigen Datentypen fest definiert. Weiter ist es über Fremdschlüssel möglich, Beziehungen zwischen diesen Tabellen zu definieren. Alle Instanzen innerhalb einer Tabelle folgen diesem Schema. Sollte zu einem späteren Zeitpunkt eine Anpassung der Spalten oder Tabelle stattfinden, muss darauf geachtet werden, dass bestehende Daten an das (potenziell) neue Format angepasst werden. [12, 25]

Dieser Aufbau der Datenbanken bringt allerdings inhärent auch Vorteile mit sich. Die Daten befinden sich, sofern korrekt modelliert wurde, immer in einem konsistenten Zustand. Die referentielle Integrität ist zu jedem Zeitpunkt sichergestellt. Sie bieten sich außerdem für Daten an, welche viele Beziehungen untereinander haben. Ein konkretes Beispiel wäre eine Datenbank für einen Onlineshop, in welcher Kundendaten und deren Bestellungen

gepflegt werden. Je nach Bedarf kann bei jeder Bestellung auf den Kunden geschlossen werden. Sollte ein Kunde entfernt werden können die Bestellungen anonymisiert oder auch aus dem System gelöscht werden. [12, 25]

NoSQL

Im Gegensatz zu **SQL** Datenbanken erweitern **NoSQL** Datenbanken das Spektrum der möglichen Datenmodelle deutlich. **NoSQL** schließt allerdings **SQL** Datenbanken nicht von vornherein aus, es bringt lediglich weitere Datenbanktypen zusammen, wie zum Beispiel dokumentorientierte oder Multi-Model Datenbanken so wie Key-Value Stores. Ein wichtiger Unterschied zu einer relationalen Datenbank ist, dass Daten in **NoSQL** Datenbanken oft keinem festen Schema folgen müssen. Es können nach Bedarf Felder hinzugefügt oder entfernt sowie deren Datentypen geändert werden. Innerhalb einer “Spalte” muss für ein Feld auch nicht zwangsweise der gleiche Datentyp gelten, sofern dies für den Datenbanktyp anwendbar ist.

Key-Value Stores speichern Daten anhand eines festgelegten Schlüssels. Während es sich prinzipiell auch bei dokumentbasierter Datenbanken um Key-Value Stores handelt (jedes Dokument hat eine eindeutige ID) sind klassische Key-Value Stores in der Datenkomplexität deutlich einfacher. Üblicherweise werden die Daten in Zeilen einer Tabelle dargestellt, wobei natürlich, wie bei **NoSQL** bekannt, weder die Spalten noch die Datentypen in der gesamten Tabelle konsistent sein müssen. Dies ergibt sich auch aus der internen Struktur einer solchen Datenbank, da der Value für dieses lediglich als **Binary Large Object (BLOB)** gespeichert ist. Dies bedingt auch, dass nicht ausgewählt werden kann, welche Spalten für eine Abfrage zurückgeliefert oder nach welchen Spalten gefiltert werden soll. [25]

Wide Column Datenbanken erweitern das Konzept von Key-Value Stores dadurch, dass der Value nun in echte Spalten aufgeteilt ist, welche entsprechend selektiert und gefiltert werden können. Dabei ist es weiterhin nicht notwendig, bei Erstellung der Tabelle ein festes Schema zu definieren. Spalten können für einzelne Datensätze dynamisch eingefügt oder ausgelassen werden. Dies ist, aufgrund der internen Architektur dieser Datenbanken, weiter für beliebig große Tabellen skalierbar. Die Spalten werden in einzelnen Dateien persistiert und erlauben so sowohl Skalierbarkeit als auch Partitionierung der Datenbank. [25]

Dokumentbasierte Datenbanken speichern Daten in einer JSON ähnlichen Form welche sich **Binary JSON (BSON)** nennt. Zur Organisation existieren innerhalb einer Datenbank sogenannte Collections, welche beliebig viele solcher BSON-Dokumente enthalten können. Dabei dienen die Collections lediglich als Pseudo-Struktur durch welche nicht zwangsweise eine Aussage über die Struktur der enthaltenen Dokumente gemacht werden kann. Ein Beispiel wäre eine Collection für User-Dokumente und eine für Order-Dokumente. Es ist technisch nicht eingeschränkt, dass User in die Order Collection eingefügt werden oder umgedreht. Dieser schemafreie Aufbau hat den Vorteil, dass beliebige Strukturen ohne Modellierungsaufwand und ohne Einschränkungen persistiert werden können. [25]

Graph Datenbanken stellen den letzten, in dieser Arbeit vorgestellten, Datenbanktyp vor. Im Vergleich zu relationalen Datenbanken erweitern Graphdatenbanken die Möglichkeiten durch den expliziten Fokus auf Beziehungen zwischen Entitäten deutlich. Während ein relationales Datenmodell mit moderatem Aufwand direkte Beziehungen zwischen einigen (wenigen) Entitäten darstellen kann, sind mehrstellige und indirekte Beziehungen zwischen Entitäten, nach Bedarf auch mit weiteren Attributen angereichert, ohne Probleme möglich. Ein Beispiel hierfür wäre ein Stammbaum, welcher effektiv beliebig breit und hoch werden kann. Während die Modellierung eines solchen Falls in einer relationalen Datenbank möglich ist, ist die Abfrage, um die Daten nutzbar zu machen, potenziell komplex. In einem Graphen ließe sich eine solche Struktur sowohl einfacher anlegen als auch abfragen. Solange die Daten vorhanden sind, lassen sich nahezu beliebige Fragestellungen beantworten. [4, 25]

Grundlegend basieren Graphdatenbanken auf der Graphentheorie. Es existieren Nodes und Beziehungen zwischen diesen. Nodes können sowohl Labels als auch Attribute besitzen, welche bei der Abfrage gefiltert werden können. Beziehungen können auch Attribute besitzen. Sie können uni- oder bidirektional erstellt werden, je nach Datenbanksystem. Abgefragt werden kann der Graph (oder Teile des Graphs) dann mit einer eigenen Abfragesprache. Hier existieren Unterschiede zwischen den einzelnen Datenbankanbietern, es gibt also nicht wie bei relationalen Datenbanken eine standardisierte Abfragesprache, welche für alle Datenbanksysteme gültig sein sollte. [4, 25]

Zusammenfassung

Wofür bieten sich die einzelnen Datenbanken im Speziellen an? Abbildung 2.6 zeigt eine Übersicht der vorgestellten Datenbanktypen sowie deren zugehörige Datenkomplexität. Während Key-Value Stores für sehr einfache Strukturen gut einsetzbar sind, befinden sich sowohl relationale Datenbanken als auch Graphdatenbanken am anderen Ende des Spektrums.

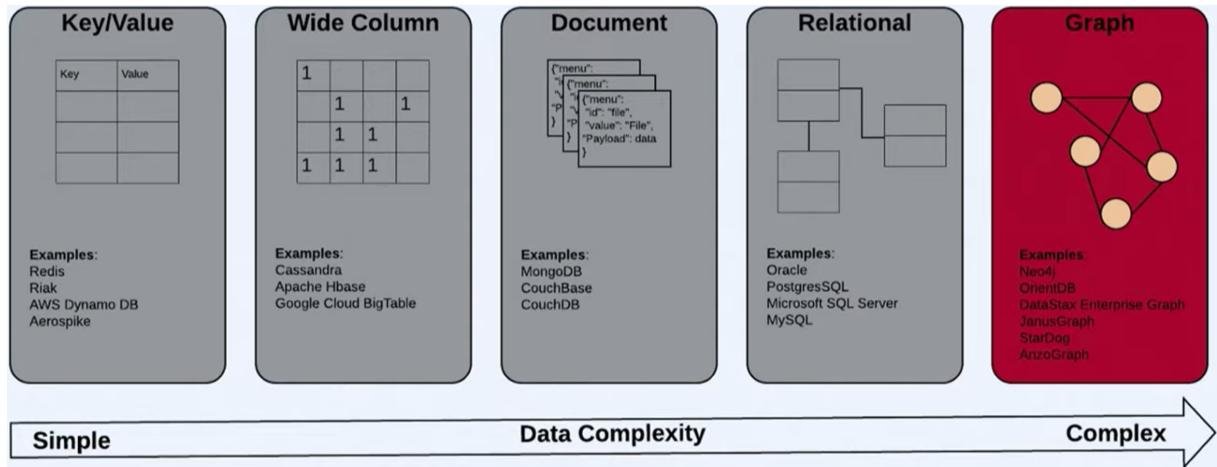


Abbildung 2.6: Komplexität der Datenstruktur pro Datenbanktyp [4]

Die erste Anforderung, welche die Auswahl des Datenbanksystems definiert, ist also die Datenstruktur bzw. -komplexität und ob diese von Beginn des Projekts an vorhersehbar ist oder potenzielle Änderungen auch im Nachhinein integriert werden können. Sollten die Daten sowohl eine hohe Komplexität als auch Dynamik aufweisen, bietet sich allerdings auch die Möglichkeit, eine dokumentbasierte Datenbank einzusetzen. Dies hat den Vorteil, dass auf nachträgliche Änderungen des Datensatzes reagiert werden kann, ohne das darunterliegende Schema anpassen zu müssen. Auf der anderen Seite müssen potenzielle Verbindungen zwischen Datensätzen oder Vermeidung von redundanter Datenhaltung auf Seiten der Applikation stattfinden und werden nicht von der Datenbank durchgesetzt.

Eine zweite Anforderung basiert auf dem Zusammenhang einzelner Datensätze, oder Entitäten. Wenn diese viele Beziehungen miteinander eingehen, bieten sich erneut relationale Datenbanken oder Graphdatenbanken an. Dabei ist zu unterscheiden, ob und wie die referentielle Integrität zwischen den Datensätzen einer Beziehung hergestellt werden kann und ob die Beziehung selbst mit weiteren Attributen angereichert werden soll. Wie beleuchtet liegt die Stärke von relationalen Datenbanken sowohl in der Schemagebundenheit als auch der impliziten Herstellung der referentiellen Integrität für jeden Datensatz in der Datenbank. Dies kann, wenn bekannt ist welche Entitäten Beziehungen miteinander eingehen, von Vorteil sein, da diese Validierung nicht auf Ebene einer oder potenziell auch mehrerer Applikationen stattfinden muss. Sollte es allerdings auch bei der Modellierung der Beziehungen eine relativ große Dynamik und Tiefe der Beziehungen auf mehreren Ebenen geben, fällt die Wahl eher auf eine Graphdatenbank.

Die nächste Anforderung ergibt sich aus der geplanten Abfrage der Daten. Prinzipiell lassen sich Abfragen oder auch Suchen auf allen gezeigten Datenbanksystemen bewerkstelligen, allerdings jeweils mit diversen Limitierungen. So kann in einem Key-Value Store auf einen Datensatz nur über den Primärschlüssel zugegriffen werden. Alle weiteren Datenbanksysteme erlauben Suchen auch auf Werten von nicht indexierten Spalten bzw. Attributen.

Während relationale Datenbanken mit [SQL](#) eine sehr mächtige Abfragesprache zur Verfügung stellen, sind die möglichen Fragestellungen, welche über die Daten beantwortet werden bei der Modellierung festgelegt und im Nachhinein gegebenenfalls nicht ohne weiteres erweiterbar. Es kann vorkommen, dass neue Fragestellungen nur über komplexe (und teilweise auch langsamere) Abfragen bewerkstelligt werden können. Diesen Nachteil hat eine Graphdatenbank nicht. Solange die Daten in der Datenbank vorhanden sind, ist eine Abfrage des Graphen mittels simplerer Abfragen (im Vergleich zu [SQL](#)) möglich. Vor allem für die Traversierung der Beziehungen zwischen Entitäten ist das naturgemäß ein enormer Vorteil für Graphdatenbanken im Vergleich zu anderen Datenbanksystemen.

Weiter mögliche Anforderungen können beispielsweise die Skalierbarkeit oder allgemeine Performanz des Datenbanksystems sein, dies soll allerdings im Rahmen dieser Arbeit nicht betrachtet werden, da es sich nur um ein akademisches Beispiel handelt.

Wie in Kapitel [2.2.2](#) angesprochen lässt sich auch die Struktur von Personen und deren Attributen (wie Vorname und Nachname, Geburtsdatum etc.) innerhalb eines Knowledge Graphen darstellen. Dies ist insbesondere dann hilfreich, wenn unabhängige Datensätze miteinander verbunden werden sollen. Somit schließt die erste Anforderung an die Datenkomplexität für dieses Projekt sowohl Key-Value Stores als auch Wide Column Datenbanken aus. Die zweite Anforderung reduziert die möglichen Datenbanksysteme weiter auf relationale oder Graphdatenbanken, da es sich wie angesprochen um potenziell sehr viele Beziehungen, sowohl zwischen Personen als auch deren zugehörigen Attributen handelt. Aufgrund der letzten Anforderung fällt die finale Wahl auf eine Graphdatenbank, da die Abfragen zur Traversierung der Beziehungen mittels einer relationalen Datenbank nicht oder nur schwierig darstellbar ist.

2.5 Frontend

Das Frontend, also die Weboberfläche, dient dazu die Suche zu ermöglichen und den Status sowie zuletzt die Ergebnisse zu präsentieren. Technologisch gibt es diverse Möglichkeiten, sofern diese die Anbindung an die [API](#) mittels WebSockets unterstützen. Denkbar sind beispielsweise React, Angular oder auch Flutter Web. In diesem Projekt wird, wie bei vorherigen Komponenten auch, auf eine Java-/TypeScript basierte Lösung gesetzt. Spezieller wird Angular ausgewählt, da hier im Projektteam bereits Knows-how vorhanden ist und sich nicht in eine neue Technologie eingearbeitet werden muss. Es sei aber, wie angesprochen, angemerkt, dass die Auswahl der Technologie prinzipiell frei erfolgen kann.

Angular (ursprünglich AngularJS) wurde von Google entwickelt und ist erstmals 2016 erschienen [2]. Es ist ein JavaScript Framework, welches seit seinem Beginn als AngularJS

eine Reihe an größeren Änderungen durchlaufen hat. Beispielsweise die Verwendung von TypeScript statt JavaScript oder die Einführung von Modulen für Teile von Webapplikationen. Prinzipiell teilt Angular die gesamte Applikation in drei wesentliche Teile: Komponenten, welche aus **HTML** Template und TypeScript Klasse zur Darstellung der Views bestehen, Services, welche die Geschäftslogik enthalten und Module, welche Komponenten und Services wiederverwendbar bereitstellen. Dies erlaubt einen entsprechend modularen und wiederverwendbaren Aufbau und reduziert so den Aufwand, sowohl bei der Entwicklung als auch bei der darauffolgenden Testung/Wartung der Applikation. [32]

2.6 Monitoring

Ein letzter Punkt, welcher für die Konzeption und Umsetzung relevant wird, ist das Monitoring, also die Überwachung der gesamten Applikation. Sollten Fehler auftreten ist es möglich, diese schnell zu identifizieren und entsprechend zu behandeln. In diesem Fall wird Prometheus zum Sammeln der Metriken aller einzelnen Komponenten sowie Grafana zur Aufbereitung und Visualisierung dieser vorgestellt. Beide sind als Open-Source Software verfügbar und am Markt stark verbreitet. Zu den Alternativen gehören unter anderem cAdvisor von Google oder AWS CloudWatch.

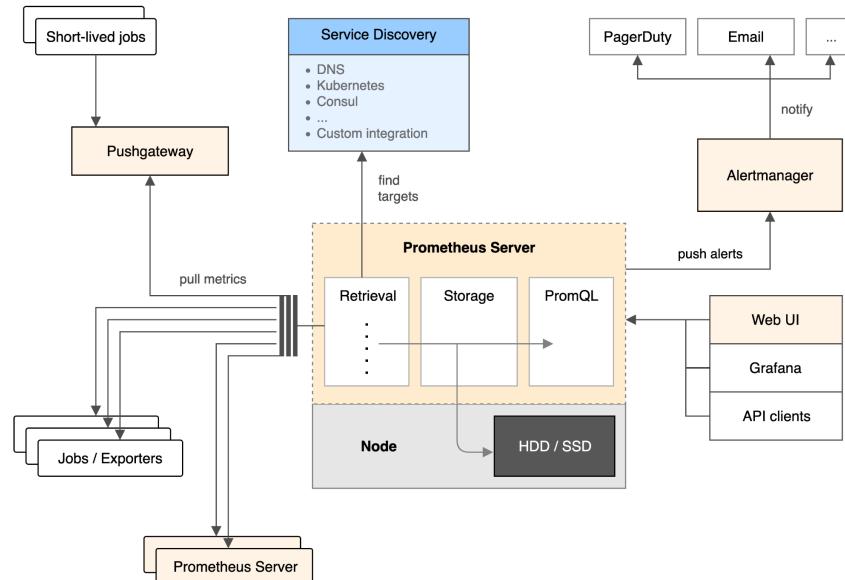


Abbildung 2.7: Interne Prometheus Architektur [28]

Abbildung 2.7 zeigt die Prometheus Architektur. Die relevantesten Teile sind die mit “pull metrics” beschrifteten Komponenten, welche die Endpunkte bereitstellen die regelmäßig von Prometheus abgefragt werden, der Prometheus Server als zentraler Punkt an welchem die Daten gesammelt, gespeichert und wiederum für Konsumenten bereitgestellt werden und die “Web UI”, also die Applikationen, welche die gesammelten Metriken visuell

beispielsweise auf Dashboards mit diversen Diagrammen und Texten darstellen. Man erkennt an der Abbildung auch, dass es durch den modularen Aufbau der Prometheus Architektur möglich ist, beliebige Teile anzupassen oder ganz auszutauschen. Beispielsweise können beliebige Applikationen Endpunkte mit Metriken bereitstellen, sofern diese dem von Prometheus interpretierbaren Schema genügen. Genauso kann eine eigene Lösung zur Visualisierung der Metriken zum Einsatz kommen und mittels der standardisierten Abfragesprache PromQL die Metriken nach Bedarf laden. [7]

3 Konzept

Aufbauend auf den im vorherigen Kapitel definierten Grundlagen wird im Folgenden das Konzept zur Umsetzung des Projekts erarbeitet. Zuerst sollen allerdings die funktionalen und nicht-funktionalen Anforderungen vorgestellt werden. Diese decken die grundlegenden Funktionen ab, welche für den erfolgreichen Abschluss des Projekts benötigt werden. Am Ende dieser Arbeit wird reflektiert, ob diese erfüllt wurden oder warum gegebenenfalls nicht.

3.1 Anforderungen

3.1.1 Funktionale Anforderungen

1. Frontend

Benutzende haben die Möglichkeit, eine Suchanfrage zu starten und entsprechend aufbereitete Ergebnisse dargestellt zu bekommen. Möglich hierbei sind beispielsweise Tabellen, Graphen oder Diagramme.

2. API

Die [API](#) behandelt eingehende Suchanfragen und verarbeitet die jeweiligen Ergebnisse. Sie ist auch dafür zuständig, die zugehörigen Such-Jobs der jeweiligen Scraper zu starten. Zum Einblick in den aktuellen Status des Systems existieren passende Schnittstellen.

3. Scraper

An die [API](#) sind verschiedenartige Scraper angebunden. Unabhängig von der Art kommunizieren diese über eine standardisierte Schnittstelle in einheitlichem Datenformat. Die folgenden Arten von Scrapern werden unterstützt:

HTML

Der [HTML](#) Scraper kann vorgegebene Webseiten auf die gesuchten Inhalte untersuchen. Die Suche ist zur Vereinfachung der Implementierung auf gewisse bekannte Teilbereiche einer Webseite eingegrenzt.

API

Die Anforderungen für [REST](#), [SOAP](#) und [GraphQL](#) [API](#) Scraper stellen sich prinzipiell

sehr ähnlich dar. Der Scraper kann Daten von vorgegebenen APIs anfragen und aufbereiten. Sofern notwendig findet eine Konvertierung des Datenformats statt.

4. Standardisierung des Datenformats

Die Kommunikationsschnittstellen zwischen den Komponenten sind standardisiert. In Frage kommt hier sowohl die Kommunikation zwischen Frontend und API als auch zwischen API und Scraper.

5. Datenbank

Daten werden innerhalb einer Datenbank gespeichert.

6. Anreicherung der Inhalte

Aggregation

Ergebnisse der Scraper werden innerhalb der Datenbank persistiert. Dies erleichtert die Verarbeitung innerhalb der API und ermöglicht die Analyse komplexer Datenbestände durch die Verbindung mehrerer Datensätze. Es ist dabei jederzeit möglich, einen Datensatz der ursprünglichen Suchanfrage zuzuordnen, um die Trennung der Ergebnisse pro Benutzer zu ermöglichen.

Kombination

Die API kombiniert alle zur aktuellen Suchanfrage gehörenden Daten in einem gesamtheitlichen Personenprofil. Sobald die Suche des Frontends beendet ist, werden alle zugehörigen Daten aus der Datenbank entfernt.

Validierung

Ausgewählte Daten werden automatisiert auf Richtigkeit überprüft. Dabei ist zwischen zwei Arten zu unterscheiden. Einer grundlegenden Wert-Validierung (Beispiel: Postleitzahl in korrektem Wertebereich) und der Validierung der Richtigkeit des Ergebnisses (Beispiel: Vorname Max oder Moritz).

7. Monitoring

Jede Komponente bietet einen Endpunkt, welcher Auskunft über ihren aktuellen Zustand gibt. Durch diese wird eine gesamtheitliche Monitoring-Lösung möglich.

8. Continuous Integration/Continuous Delivery (CI/CD)

Sobald für eine einzelne Komponente eine neue Applikationsversion bereitsteht, wird diese automatisch in der Produktionsumgebung bereitgestellt.

3.1.2 Nicht-funktionale Anforderungen

1. Erweiterbarkeit

Es ist möglich, weitere Scraper an die API anzubinden. Dies erfolgt optional über eine graphische Benutzeroberfläche zur Administration.

2. Skalierbarkeit

Die jeweiligen Komponenten reagieren dynamisch auf sich verändernde Lastverhältnisse.

3. Portabilität

Die Subsysteme können auf einem beliebigen kompatiblen System bereitgestellt werden. Gegebenenfalls notwendige Konfigurationen sind im Quellcode dokumentiert.

4. Robustheit

Fehlerfälle werden korrekt behandelt und beeinträchtigen die Funktion des Gesamtsystems nicht. Im Falle eines systemkritischen Fehlers kann automatisch der letzte funktionierende Stand wiederhergestellt werden.

3.2 Architektur

Aus den im vorherigen Abschnitt definierten Anforderungen kann nun die benötigte Architektur abgeleitet werden. Diese ist in Abbildung 3.1 dargestellt.

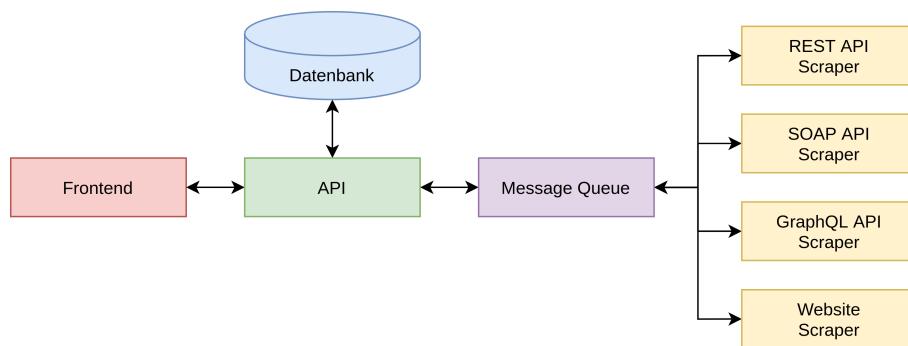


Abbildung 3.1: Architekturübersicht

Es handelt sich hierbei um eine Microservice-Architektur, wie in Kapitel 2.1.1 definiert. Im Zentrum steht die **API**, welche die grundlegende Kommunikation mit allen weiteren Komponenten bewerkstellt. So zum Beispiel mit dem Frontend, welches die Ergebnisse der entsprechenden Suche visualisiert.

Als Bindeglied zwischen der **API** und den jeweiligen Scrapern kommt eine Message Queue zum Einsatz. Die grundlegende Funktionsweise wurde in Kapitel 2.3.1 beschrieben und wird hier verwendet, um parallele Ausführungen zu ermöglichen und potenziell

fehlgeschlagene Suchanfragen zu wiederholen. Ein Scraper kann selbst keine Suchaufträge starten. Diese Strategie verhindert, dass mehrere Suchanfragen mit dem selben Ziel und Anfrageparametern gestartet werden. Außerdem lassen sich so auch Endlosschleifen verhindern, da die Management API zu jeder Zeit den Status des Gesamtsystems kennt.

Zur besseren Veranschaulichung der Architektur entspricht der Verlauf der nächsten Kapitel dem Verlauf der Daten zwischen den einzelnen Komponenten, beginnend mit den Scrapern.

3.3 Datenquellen und Scraper

Wie in den Anforderungen definiert, werden in dieser Arbeit vier Arten von Datenquellen betrachtet: REST, SOAP und GraphQL APIs sowie HTML Webseiten. Entsprechend werden verschiedene Arten von Scrapern benötigt. Es sei angemerkt, dass diese im Folgenden allgemein als Scraper bezeichnet werden, auch wenn diese Nomenklatur fachlich nur bedingt korrekt ist, da es sich nicht durchgängig um Webseiten handelt. Beispielsweise würde die Anbindung an eine REST API eher als Fetcher statt Scraper bezeichnet werden.

Um möglichst hohe Flexibilität zur Anbindung diverser Datenquellen zu ermöglichen, werden diverse Konfigurationsvariablen für alle Arten von Scrapern definiert. Dies erlaubt im Idealfall die Verwendung ähnlicher Datenquellen, ohne große Änderungen am Quellcode durchführen zu müssen. Zu diesen Konfigurationen gehört mindestens die Uniform Resource Locator (URL), an welche die Anfragen gesendet werden.

Suchanfragen enthalten für jede Anfrage ein query-Feld. Ein Beispiel hierfür ist in Listing 3.1 gezeigt. Konfigurierbar sind prinzipiell alle Felder, welche die Suche im Scraper verfeinern können.

```
1 "query": {
2   "schema.org/givenName": "John",
3   "schema.org/familyName": "Doe"
4 }
```

Listing 3.1: Nutzdaten einer Suchanfrage

Wie angesprochen sind hier prinzipiell zwei Arten von Datenquellen zu unterscheiden: APIs und HTML Webseiten. Dabei sind prinzipiell nur zwei relevante Datenformate zu betrachten: JSON und XML. Dies bedeutet, dass die Implementierung der Datenquellen generisch erfolgen kann. Dabei müssen lediglich einige grundlegende Attribute für jede Datenquelle definiert werden. Ein Beispiel hierfür wird im Folgenden detailliert.

```

1 GET https://example.com/api/users?firstName=Max&lastName=Mustermann
2 Authorization: secret
3
4 {
5     "results": [
6         {
7             "firstName": "Max",
8             "lastName": "Mustermann",
9             "birthDate": "05.07.1976",
10            "description": "Lorem ipsum dolor sit amet..."
11        }
12    ],
13    "page": 1,
14    "totalPages": 1,
15    "totalItems": 1
16 }
```

Listing 3.2: Beispiel einer REST API Antwort

Listing 3.2 zeigt die potenzielle Antwort einer REST API. Die gewünschten Ergebnisse befinden sich in diesen Fall im `results`-Array. Dies bedeutet, dass eine mögliche Konfiguration für diese Datenquelle wie in Listing 3.3 dargestellt aussieht.

```

1 {
2     "url": "https://example.com/api/users?firstName=Max&lastName=
3         Mustermann",
4     "root": ".results",
5     "attributes": {
6         "schema.org/givenName": ".firstName",
7         "schema.org/familyName": ".lastName",
8         "schema.org/birthDate": {
9             "path": ".birthDate",
10            "onlyAddIfNot": {
11                ".birthDate": null
12            }
13        }
14    }
```

Listing 3.3: Beispielkonfiguration einer Datenquelle

Dabei wird sowohl die URL der Datenquelle als auch das Root-Element, welches die gewünschten Daten enthält, definiert. Dies ist notwendig, falls eine Datenquelle komplexere und potenziell verschachtelte Strukturen zurückliefert, bei denen die Ergebnisse nicht auf der ersten Ebene zu finden sind. Die Pfade zu den Nutzdaten werden im `attributes`-Objekt definiert, wobei sich der Kontext der angegebenen Pfade auf das aktuelle Ergebnis aus

der Ergebnismenge bezieht. Hier gibt es einige Optionen, um die Konfiguration möglichst flexibel zu machen. In Zeile 5 und 6 werden die Attribute für Vor- und Nachnamen ohne Veränderungen oder Bedingungen auf die normalisierten Attributnamen übertragen. Die ausgewählten Attributnamen basieren soweit möglich auf der Standardisierung von Schema.org für eine Person [19]. Das Präfix schema.org wird hinzugefügt, um potenziell weitere Quellen der Standardisierung zu unterstützen ohne Namenskonflikte zu erzeugen. Im Gegensatz dazu soll das Geburtsdatum nur übernommen werden, wenn es von der Datenquelle tatsächlich zurückgeliefert wird. Hierfür ist die Prüfung in Zeile 9 vorgesehen. Sollte die Bedingung im `onlyAddIfNot`-Objekt nicht erfüllt sein, wird das Attribut zum Ergebnis hinzugefügt. Denkbar sind weitere Variationen, dieser Transformations-/Bedingungsfunktionen, zum Beispiel `onlyAddIf` oder die Anpassung eines Werts via `prefix` und `suffix` (dies kann unter anderem bei URLs relevant werden).

Die Aufbereitung von **XML** Datenquellen (also **HTML** Webseiten und **SOAP APIs**) stellt sich etwas komplexer, in den Grundzügen allerdings ähnlich dar. **XML** kann wie **JSON** auch über einen vordefinierten Pfad traversiert werden. Das größte Problem stellt sich bei der Auswertung von Webseiten, welche keiner einheitlichen Struktur folgen. Entsprechend wird auch für diese Datenquellen das oben definierte Schema befolgt und manuell gepflegt. Eine automatische Transformation oder Auswertung wird für den Rahmen dieses Projekts als zu komplex angesehen.

3.3.1 Kommunikation: Scraper ↔ API

Die Scraper selbst definieren verschiedene (eindeutige) Namen, um Prozesse für die jeweiligen Datenquellen erstellen zu können. Beispielsweise könnte die Queue für die Datenquelle Facebook den Namen “facebook” haben. Dies ist notwendig, da es potenziell kleine Variationen des Datenformats und der entsprechenden Anfragen gibt, welche es notwendig machen, teilweise angepasste Funktionen zu implementieren. Wie auch aus der Architektur hervorgeht, wird nur nach Typ des Scrapers unterschieden, nicht nach Datenquelle. Dies bedeutet, dass Datenquellen gleichen Typs vom gleichen Scraper bearbeitet werden können.

3.4 API

Nachdem die Daten von den Scrapern angefragt und aufbereitet wurden, kann die API diese für die weitere Verarbeitung verwenden. Dabei durchlaufen die Daten einige Schichten, in welchen verschiedene Arten von Transformationen und Bereinigungen durchgeführt werden. Dies erlaubt von Beginn an einen iterativen Ansatz, durch welchen die Ergebnisse bereinigt und mit weiteren Informationen angereichert werden können.

3.4.1 Speicherarchitektur

Abbildung 3.2 zeigt die oben angesprochene Speicherarchitektur. Die von den Scrapern gelieferten Ergebnisse durchlaufen erst einen Transformations-Layer, in welchem eine Normalisierung und Bereinigung der Daten stattfindet, um die spätere Verarbeitung der Daten zu erleichtern. Die normalisierten Ergebnisse werden danach in der Graphdatenbank persistiert. Verschiedene Datenbanken wurden in Kapitel 2.4 vorgestellt und verglichen. Aufgrund der stark dynamischen Inhalte einzelner Ergebnisse wird eine NoSQL-Datenbank empfohlen, genauer eine Graphdatenbank, um die Ergebnisse später effektiv verbinden und durchsuchen zu können.

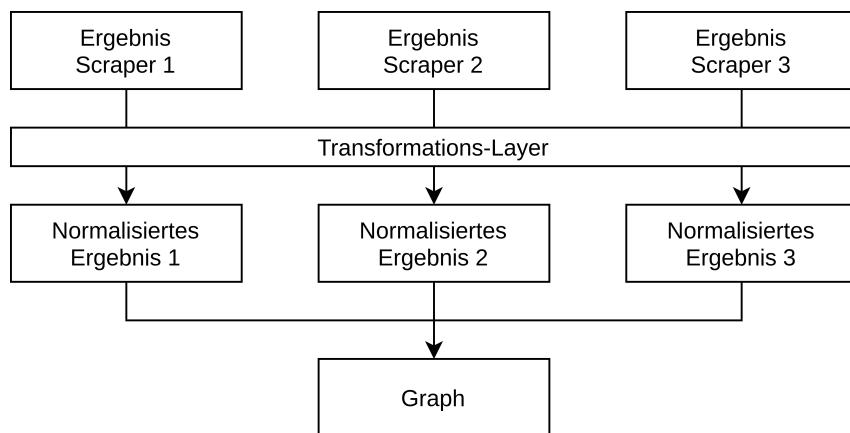


Abbildung 3.2: Speicherarchitektur

Sammeln der Ergebnisse

Listing 3.4 zeigt den beispielhaften Inhalt des Ergebnisses für einen einzelnen Scraper. `scraperId` entspricht hier der eindeutigen ID des Quell-Scrapers. `clientId` ist die ID des Benutzenden, welcher die Suche initiiert hat. Dies ist notwendig, um die Suchergebnisse eindeutig einem Benutzenden zuordnen zu können, ohne Gefahr zu laufen, Ergebnisse verschiedener Suchanfragen zu vermischen. `data` entspricht dem unveränderten Ergebnis,

welches vom Scraper geliefert wurde. Zuletzt können noch beliebige Metadaten an das Ergebnis angehängt werden, um perspektivisch weitere Analysen durchzuführen. In diesem Fall wurde die Gesamtzahl der Ergebnisse, die Scrape-Dauer sowie die Datenquelle und ursprüngliche Anfrage festgehalten.

```
1 {
2     "scraperId": 0,
3     "clientId": 0,
4     "data": [
5         {
6             "schema.org/givenName": "John",
7             "schema.org/familyName": "Doe",
8             ...
9         }
10    ],
11    "metadata": {
12        "source": "http://website.com",
13        "query": {
14            "schema.org/givenName": "John",
15            "schema.org/familyName": "Doe"
16        }
17    }
18 }
```

Listing 3.4: Beispiel für ein Scraper-Ergebnis

Transformation 1: Normalisierung der Ergebnisse

Wie in Abbildung 3.2 dargestellt, werden die Ergebnisse vor der Weiterverarbeitung transformiert. Dies entspricht einer Basisvalidierung und Bereinigung der Attributwerte. Aufgrund der generischen Konfiguration der Datenquellen kann davon ausgegangen werden, dass alle Attributnamen normalisiert vorliegen. Für die erste Validierung wird noch nicht überprüft, ob oder inwieweit der gelieferte Wert für das jeweilige Attribut korrekt ist, sondern nur, ob dieser im erlaubten Wertebereich liegt. Beispielsweise ist es in Deutschland nicht möglich, dass eine Postleitzahl Buchstaben beinhaltet. Gleichzeitig ist sie auf einen gewissen numerischen Wertebereich begrenzt, in diesem Fall muss der Wert zwischen 00001 und 99999 liegen. So kann verhindert werden, dass unplausible Daten weitergeleitet werden und so potenziell das Ergebnis verfälschen. Diese Validierung ist entsprechend für jedes relevante Attribut durchzuführen, sofern sie mit angemessenem Aufwand durchführbar ist. Beispielsweise ist die Validierung einer ganzen Adresse potenziell wesentlich komplexer als die vorher angesprochene Validierung der Postleitzahl. Sollte ein Ergebnis (teilweise) falsche Ergebnisse enthalten muss dieses von Anfang an nicht in den Datenbestand aufgenommen werden.

Um die Verbindung der Daten im Graphen zu erleichtern wird zusätzlich zur Validierung eine Bereinigung der Attributwerte durchgeführt. Dadurch können Konflikte durch unterschiedliche Schreibweisen eines Werts verhindert werden. Tabelle 3.1 zeigt einige Beispiele für die Bereinigung verschiedener Werte. Diese unterscheidet sich je nach Attributtyp. Timestamps werden beispielsweise zu einem UNIX-Timestamp transformiert, Telefonnummern ohne Sonderzeichen oder Gruppierungen dargestellt. Die bereinigten Werte werden in einem neuen Attribut gespeichert, um den Datensatz nicht zu verfälschen.

Von	Zu
Björn	Bjoern
LANGE STRAßE	Lange Straße
+1 234 567890	001234567890
05.09.1994	778716000

Tabelle 3.1: Beispielhafte Bereinigung von Attributwerten

Nach diesem Schritt liegen die Daten also normalisiert und bereinigt vor und können in der Datenbank persistiert werden. Dieser Schritt wird im Folgenden beschrieben.

Graph

Nachdem ein Scraper ein oder mehrere Ergebnisse liefert und diese die erste Transformation durchlaufen haben, können sie in der Graph-Datenbank gespeichert werden. Dafür wird für jedes Ergebnis eine Result-Node angelegt und mit einer **Universally unique identifier (UUID)** versehen. Zusätzlich wird für jedes gefundene Attribut eine zugehörige Attribut-Node angelegt, sofern noch keine existiert. Diese wird auch über eine **UUID** identifiziert. Von der Result-Node werden dann Relationen zu allen zugehörigen Attribut-Nodes erstellt. Der entstehende Graph ist in Abbildung 3.3 aufgezeigt.

`scraperId`, `clientId` und die bestehenden Metadaten werden vom Scraper-Ergebnis übernommen. Diese werden allerdings durch `normalizedValues` und `validationErrors` erweitert, welche respektive die Anzahl der bereinigten Werte beziehungsweise Fehler in der Basisvalidierung enthalten. `source` stellt weiter die Quelle des Ergebnisses dar.

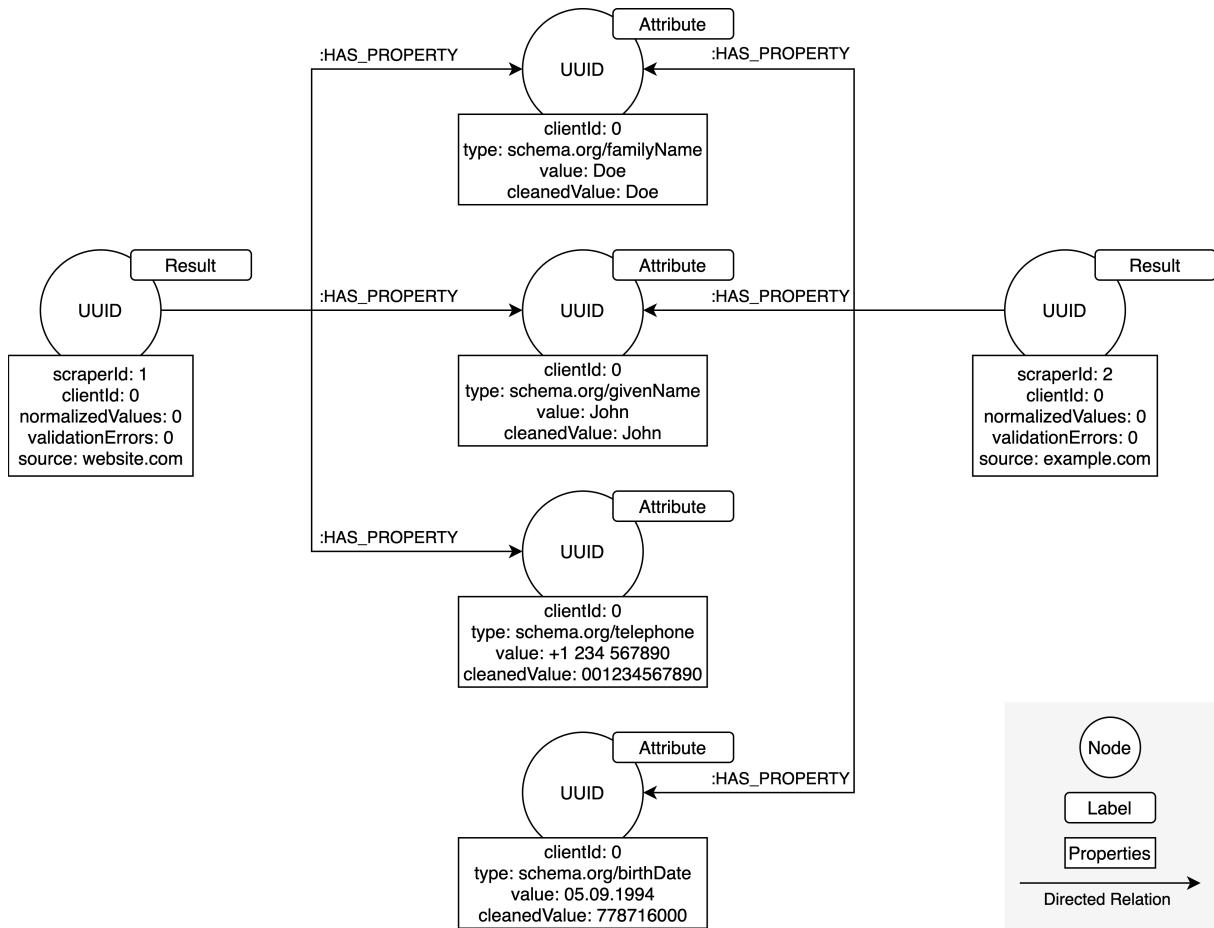


Abbildung 3.3: Aufbau des Graphen

Suche relevanter Ergebnisse im Graph

Nachdem die Ergebnisse aller Scraper im Graph enthalten sind, kann die eigentliche Suche beginnen. Dafür wird eine dedizierte Search-Node eingefügt. Diese hat Relationen zu allen Attribut-Nodes, welche den initialen Suchparametern entsprechen. Im Folgenden folgt die Erklärung der Suche für unterschiedliche Fälle. Für jeden Fall kann eine Regel für den zu implementierenden Algorithmus abgeleitet werden.

Fall 1

Abbildung 3.4 zeigt den Standardfall. Es wurde nach einer Person namens "John Doe" gesucht. Diese wurde auch gefunden. Zusätzlich liefert Ergebnis 1 die Telefonnummer und Ergebnis 2 das Geburtsdatum dieser Person.

Regel 1: Die Ergebnismenge enthält alle Result-Nodes, die mindestens eine Verbindung zu den Attribut-Nodes haben, welche auch mit der Search-Node verbunden sind.

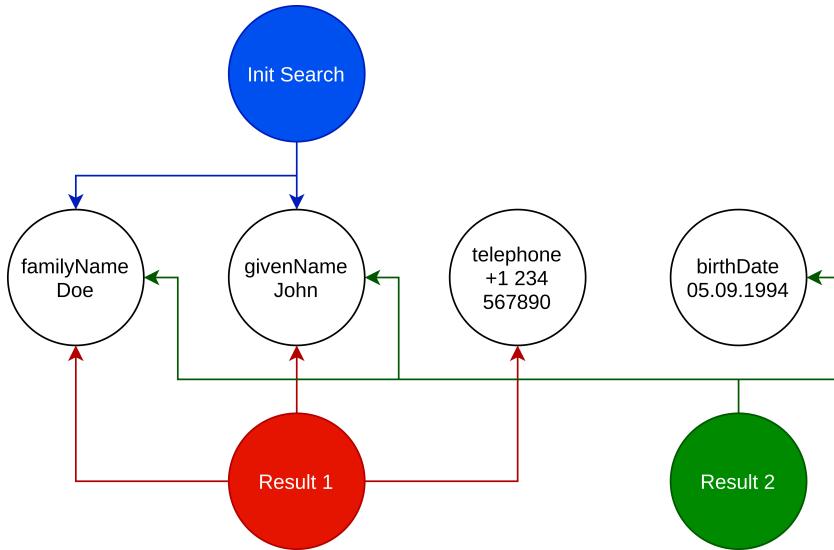


Abbildung 3.4: Suche auf dem Graphen, Fall 1

Fall 2

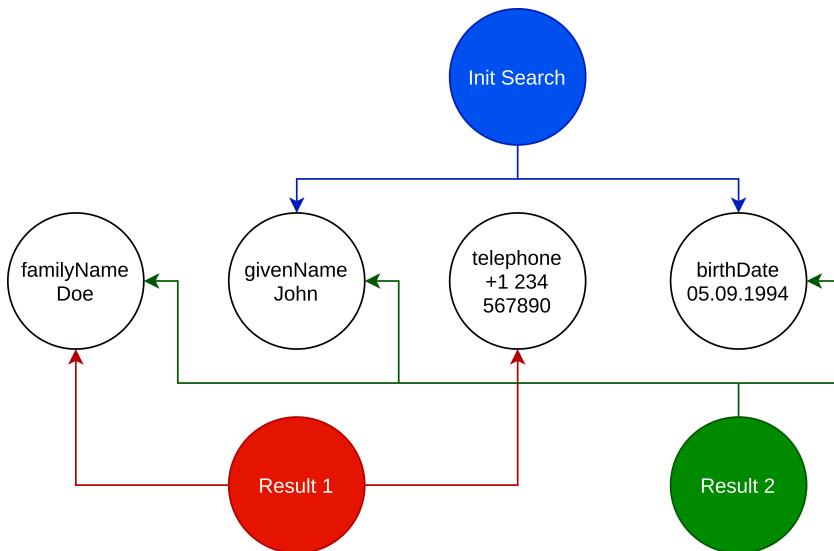


Abbildung 3.5: Suche auf dem Graphen, Fall 2

In Abbildung 3.5 ist der Vorname und das Geburtsdatum bekannt. Laut Regel 1 kommt Ergebnis 2 zur Ergebnismenge hinzu, da eine Verbindung zu dem passenden Vornamen und Geburtsdatum existiert. Zusätzlich liefert Ergebnis 2 den Familiennamen, welcher eine Verbindung zu Ergebnis 1 hat. Somit kann auch auf die Telefonnummer geschlossen werden.

Regel 2: Sollte eine gefundene Result-Node Relationen mit weiteren, nicht in der ursprünglichen Suche enthaltenen Attributen haben, werden deren zugehörige Result-Nodes auch der Ergebnismenge hinzugefügt.

Fall 3

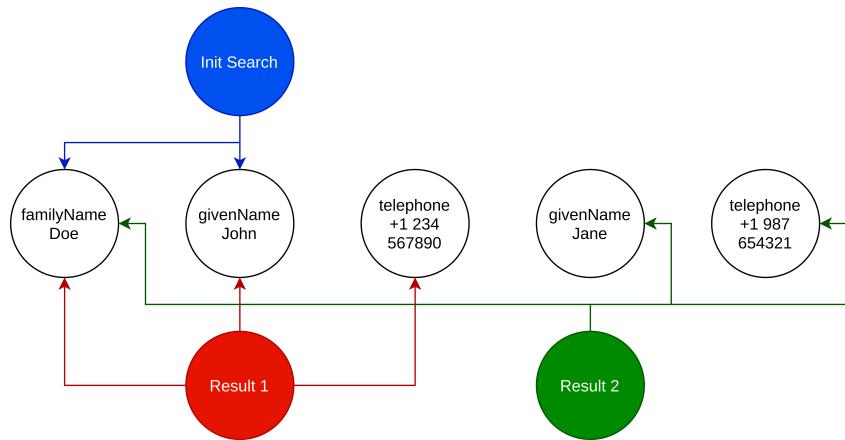


Abbildung 3.6: Suche auf dem Graphen, Fall 3

Abbildung 3.6 zeigt den Fall, dass zwei unterschiedliche Personen mit sich teilweise überschneidenden Attributen (in diesem Fall der Familienname) gefunden werden. Laut Regel 1 und 2 würden Ergebnis 1 und 2 der Ergebnismenge hinzugefügt werden, dies steht in diesem Fall allerdings im Widerspruch zu den Suchparametern. Richtig wäre, nur Ergebnis 1 zurückzugeben, da sowohl Vorname als auch Familienname übereinstimmen.

Regel 3: Eine Result-Node darf nur der Ergebnismenge hinzugefügt werden, wenn alle der Search-Node zugehörigen Attribut-Nodes auch in der Attributmenge der Result-Node existieren. Die Ausnahme hierfür ist, wenn eine Result-Node für einen Suchparameter keine Attribute-Node des gleichen Typs enthält, in welchem Fall der Match für dieses Attribut nicht hergestellt werden muss.

Fall 4

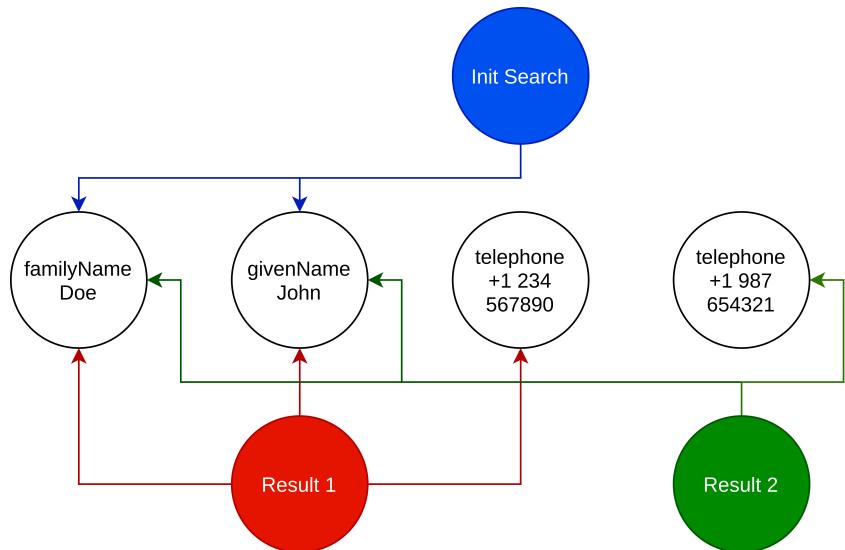


Abbildung 3.7: Suche auf dem Graphen, Fall 4

Im Beispiel von Abbildung 3.7 wurden zwei Personen mit denselben Attributen gefunden. Regel 1 und 2 definieren die Ergebnismenge als Ergebnis 1 und 2. Regel 3 kann keine Abhilfe schaffen, da die gesuchten Attribute tatsächlich für beide Ergebnisse übereinstimmen.

Regel 4: Für den Randfall, dass zwei Personen mit den gleichen Suchparametern gefunden wurden, werden alle, laut den Regeln 1 bis 3 passenden, Ergebnisse an den Benutzenden zurückgemeldet. Dieser muss manuell entscheiden, welche Attribute korrekt sind.

Fall 5

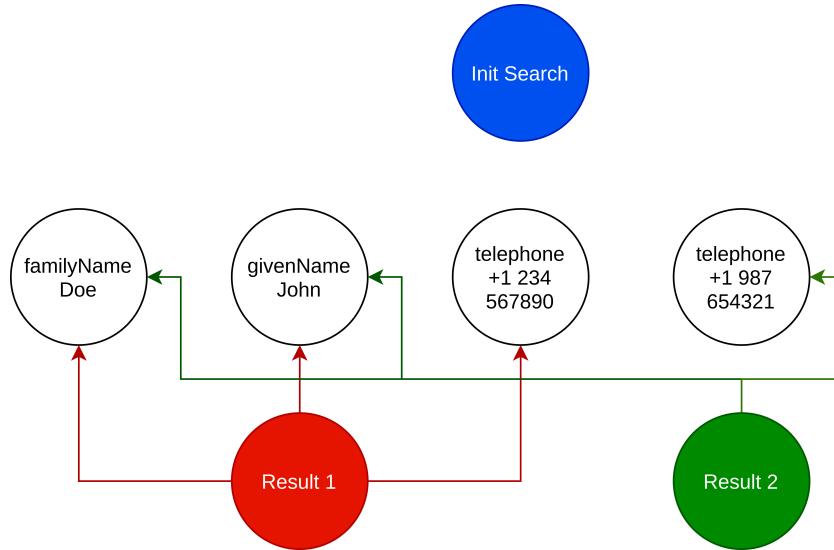


Abbildung 3.8: Suche auf dem Graphen, Fall 5

Der letzte Fall ist in Abbildung 3.8 gezeigt. Es kann keine Relation mit gefundenen Attributen hergestellt werden, da keiner der Suchparameter in den Ergebnissen existiert.

Regel 5: Für den Randfall, dass für mindestens einen Suchparameter eines Typs keine passende Attribut-Node im bestehenden Graph gefunden wird, muss der Benutzer für diese/-n Typ/-en einen vorhandenen Wert auswählen.

Erweiterung des Suchverfahrens

Nach der Definition oben ist das Suchverfahren darauf beschränkt, dass die Suchparameter exakt mit den gefundenen Attributen übereinstimmen. Dies verhindert potenziell, dass die korrekten Ergebnisse zurückgegeben werden. Auch ist es nicht möglich, ähnliche Ergebnisse zu betrachten.

Erweitert werden kann das Verfahren dadurch, dass für Attribute gleichen Typs jeweils die Genauigkeit im Vergleich zu einem Erwartungswert berechnet wird. Es werden dann nicht nur Relationen von der Search-Node zu den Attribut-Nodes mit exakt gleichem Wert

(Genauigkeit = 100%), sondern auch zu denen mit geringerer Genauigkeit, zum Beispiel $\geq 80\%$ erstellt. Dies erlaubt eine fehlertolerantere Suche und Benutzende können flexibler entscheiden, welche Ergebnisse tatsächlich korrekt sind.

```
1 algorithm getResultAccuracy(type, expectedValue, values)
2 { Errechnet anhand von type, values und expectedValue die Genauigkeit
   fuer jeden Wert. }
3 begin
4     if (expectedValue nicht gesetzt) {
5         Setze expectedValue auf das Vorkommnis, welches in values am
          haeufigsten vorkommt
6     }
7
8     Initialisiere accuracies
9
10    /*
11     * Moegliche Attributtypen: String, Number, Enum, Boolean, Timestamp
12     */
13    for (value in values) {
14        switch (type) {
15            case String:
16                accuracy entspricht Abweichung laut Dice Koeffizient.
17                break;
18            case Number:
19                accuracy entspricht der relativen Differenz von value und
                  expectedValue.
20                break;
21            case Enum:
22            case Boolean:
23                accuracy entweder 0% oder 100%.
24                break;
25        }
26
27        Fuege accuracy zu accuracies hinzu
28    }
29
30    return accuracies
31 end
```

Listing 3.5: Pseudocode der Validatorfunktion

Listing 3.5 zeigt den Pseudocode für die zugehörige Validatorfunktion. Diese kann für gefundene Ergebnisse die entsprechende Genauigkeit berechnen. Exemplarische Eingaben und zugehörigen Genauigkeiten sind in Tabelle 3.2 dargestellt.

Die Berechnung für die String-Genauigkeit erfolgt mittels des Dice Koeffizienten [37]. Entsprechend der Eingabewörter “Jonas” und Jonathan ergeben sich zwei Mengen von

Typ	Erwartungswert	Wert	Genauigkeit
String	“Jonas”	“Jonathan”	54,55%
Number	946681200 (01.01.2000)	978303600 (01.01.2001)	96,77%
Boolean	false	true	0%

Tabelle 3.2: Darstellung der Berechnung zur Genauigkeit eines Werts

Bigrammen: { jo, on, na, as } und { jo, on, na, at, th, ha, an }. Entsprechend der Formel $s = \frac{2n_t}{n_x+n_y}$ ergeben sich für die Unbekannten: $n_t = 3$ als Anzahl der Elemente in der Schnittmenge und $n_x = 4$ als Anzahl der Bigramme in Jonas bzw. $n_y = 7$ für Jonathan. Eingesetzt ergibt sich $s = \frac{2*3}{3+7} = \frac{6}{11} \approx 54,55\%$.

Entsprechend erfolgt die Berechnung der Genauigkeit einer Zahl (in diesem Fall ein UNIX Timestamp) anhand der Formel zur arithmetischen relativen Distanz. Diese lautet $d_r = \frac{|x-y|}{\max(|x|,|y|)}$, wobei die Einschränkung gilt, dass mindestens einer der Werte nicht gleich 0 ist. Eingesetzt ergibt sich in diesem Fall: $d_r = \frac{|946681200-978303600|}{\max(|946681200|,|978303600|)} \approx 3,23\%$. Um die Genauigkeit zu erhalten, muss dieses Ergebnis allerdings noch invertiert werden, da es sich bei d_r um die prozentuale Abweichung handelt.

Neben einfachen Vergleichen zwischen Strings und Zahlen, gibt es auch Attribute, deren Informationsgehalt sich nicht direkt anhand einer Zeichenkette ablesen lässt. Beispielsweise gibt es deutliche Unterschiede beim Standort im geografischen Sinn. Die Stadt Paris liegt zwar in Frankreich, existiert allerdings auch in den USA im Bundesstaat Texas. Um also bei einem Standort eine Genauigkeit angeben zu können, bedarf es einer Umwandlung in Längen- und Breitengrad. Je mehr Adressdaten über einen Standort bekannt sind, desto exakter lässt sich dessen geographische Position bestimmen. Paris in Frankreich ergibt beispielsweise den Wert [48.856614, 2.3522219]. Aus den einzelnen Längen- und Breitengraden lässt sich nun eine Distanz ableiten. Je näher die Standorte am Erwartungswert liegen, desto höher ist die ermittelte Genauigkeit. Diese Berechnung setzt allerdings den Zugriff auf eine API oder Datenbank voraus, welche so eine Umwandlung und Berechnung durchführen kann. Für Enums müssten die Erwartungswerte pro Attribut kuriert werden. Als Beispiel wäre der Enum-Typ `gender` zu nennen, für den die möglichen Werte “männlich” und “weiblich” entsprechen. Auch bei Arrays oder zusammengesetzten Datentypen stellt sich die Validierung wesentlich komplexer dar und wird im Rahmen dieses Projekts nicht genauer betrachtet.

Algorithmus zur Ermittlung der Ergebnismenge

Ein expliziter Algorithmus soll im Rahmen des Konzepts nicht konkretisiert werden, um die Möglichkeiten bei der Umsetzung nicht zu stark einzuschränken. Es wäre beispielsweise denkbar, die Regeln zur Suche im Code anzuwenden oder einen umfassenden Datenbank-Query zu definieren. Jeder Ansatz hat diverse Vor- und Nachteile, beispielsweise in der Laufzeit oder seiner Komplexität. Es ist nur wichtig, dass der gewählte Ansatz am Ende alle definierten Regeln abdeckt. Diese werden zur Übersicht folgend erneut dargestellt:

1. Die Ergebnismenge enthält alle Result-Nodes, die mindestens eine Verbindung zu den Attribut-Nodes haben, welche auch mit der Search-Node verbunden sind.
2. Sollte eine gefundene Result-Node Relationen mit weiteren, nicht in der ursprünglichen Suche enthaltenen Attributen haben, werden deren zugehörige Result-Nodes auch der Ergebnismenge hinzugefügt.
3. Eine Result-Node darf nur der Ergebnismenge hinzugefügt werden, wenn alle der Search-Node zugehörigen Attribut-Nodes auch in der Attributmenge der Result-Node existieren. Die Ausnahme hierfür ist, wenn eine Result-Node für einen Suchparameter keine Attribute-Node des gleichen Typs enthält, in welchem Fall der Match für dieses Attribut nicht hergestellt werden muss.
4. Für den Randfall, dass zwei Personen mit den gleichen Suchparametern gefunden wurden, werden alle, laut den Regeln 1 bis 3 passenden, Ergebnisse an den Benutzenden zurückgemeldet. Dieser muss manuell entscheiden, welche Attribute korrekt sind.
5. Für den Randfall, dass für mindestens einen Suchparameter eines Typs keine passende Attribut-Node im bestehenden Graph gefunden wird, muss der Benutzer für diese/-n Typ/-en einen vorhandenen Wert auswählen.
6. Erweiterung des Suchverfahrens: Die Search-Node wird nicht nur mit den Attribut-Nodes verbunden, welche exakt mit den Suchparametern übereinstimmen, sondern auch mit denen, welche einer gewissen Fehlertoleranz genügen.

Die Suche wird an dieser Stelle beendet. Die Ergebnisse werden dem Benutzenden angezeigt, welcher aus diesen die treffendsten Werte auswählen kann. Mit den neuen Erwartungswerten kann eine neue Suchanfrage gestartet werden. Dies hat den Vorteil, dass die gefundenen Werte von einer dritten Partei, nämlich dem Benutzenden, bestätigt werden. So kann nach wenigen Generationen eine höhere Gesamtgenauigkeit erzielt werden, als wenn automatisiert eine neue Suche gestartet würde. Ein Beispiel hierfür wäre, dass der Benutzende den Namen des Subjekts kennt, sich bei der Telefonnummer allerdings nicht sicher ist. Die Suche liefert

drei potenzielle Telefonnummern, anhand welcher der Benutzende die Korrekte auswählen kann, weil die Vorwahl bekannt ist.

3.5 Frontend

Das Frontend dient als graphische Schnittstelle für die Benutzenden und bietet Zugriff auf alle Funktionen, welche von der API angeboten werden. So werden diverse Seiten angeboten, mit welchen ein Benutzer interagieren kann, um Suchen zu starten oder Ergebnisse zu sichten.

Prinzipiell gestalten sich die Seiten relativ einfach. Auf der initialen Seite können Benutzende eine Suche starten, wobei als Eingabe die Felder und der zugehörige Suchwert angegeben werden können. Nach dem Start der Suche wird auf eine weitere Seite navigiert, welche in Echtzeit den aktuellen Suchfortschritt darstellt. Auf dieser werden noch keine Ergebnisse dargestellt, sie bietet lediglich Einblick in den Suchvorgang. Nachdem das Ergebnis vorliegt, wird auf eine neue Seite navigiert, welche dieses beispielsweise in Form von Tabellen oder Dropdown-Menüs visualisiert und den Benutzenden die Möglichkeit gibt, pro Feld einen der gefundenen Werte als neuen Suchwert zu übernehmen. Bei Bedarf kann hier auch ein eigener Wert für das Feld eingeführt werden. Nach der Anpassung der Suchparameter kann eine neue Suche gestartet werden.

Um diese Funktionen zu ermöglichen existieren einige WebSocket Schnittstellen. Die WebSocket-Technologie wurde in Kapitel 2.3.2 vorgestellt und wird eingesetzt, da bidirektionale ereignisbasierte Echtzeitkommunikation zur Umsetzung der beschriebenen Funktionen notwendig ist.

Verbindung

Mit Hilfe der Events `connect` und `disconnect` verbindet sich das Frontend mit der API und meldet sich nach der Sitzung wieder ab. Durch diese Events lässt sich außerdem im Frontend eine Anzeige implementieren, die alle aktiven Verbindungen der Benutzenden anzeigt. Diese Information dient also auch als ein Teil der Auslastungs-/Nutzungsmetrik.

Suchanfragen

Für den Start und Status der Suchanfragen werden weiter drei Events benötigt:

- `updateSearchRequest`: Der Nutzer startet oder aktualisiert eine Suchanfrage mit den ausgewählten Suchparametern.
- `getSearchResults`: Die API stellt Aktualisierungen der gefundenen Suchergebnisse bereit.

- `getScraperErrors`: Falls beim Bearbeiten einer Suchanfrage bei einem Scraper ein Fehler auftritt, wird dieser als Kurzversion im Frontend dargestellt.

Monitoring

Damit im Frontend auch gewisse Monitoring-Metriken, wie etwa die CPU, RAM oder Speicher Auslastung, von z.B. einer Datenbank, dargestellt werden können, bedarf es einer Verbindung zwischen Frontend und Prometheus:

- `getInstantMetric`: Gibt den letzten bekannten Wert einer bestimmten Metrik zurück.
- `getRangeMetric`: Gibt alle Werte einer bestimmten Metrik zurück. Zeitraum und Granularität sind gegebenenfalls anpassbar.

3.6 Monitoring

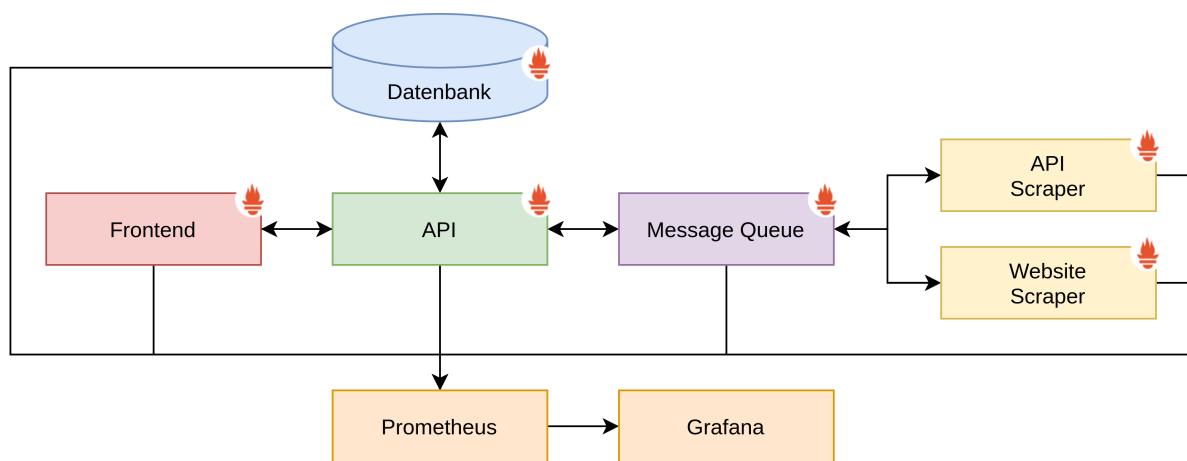


Abbildung 3.9: Übersicht des Monitorings

Um sicherstellen zu können, dass jede Komponente wie erwartet funktioniert, wird eine Monitoring-Lösung benötigt. Das Konzept hierfür ist in Abbildung 3.9 dargestellt. Jede Komponente stellt einen bekannten Metrik-Endpunkt in Prometheus-kompatiblem Format zur Verfügung. Dieser Endpunkt enthält je nach Funktion unterschiedliche Metriken. Wichtig ist in jedem Fall der System-Status, also CPU/RAM-Auslastung, Speicherverbrauch oder ähnliches. Weiter werden pro Komponente die Folgenden Metriken bereitgestellt:

Frontend

- Anzahl der Anfragen mit zugehörigen Status-Codes
- Antwortzeit

API

- Anzahl der verbundenen Clients
- Verbindungsduer eines Clients
- Antwortzeit
- Anzahl von (potenziell aufgetretenen) Fehlermeldungen

Scraper

- Dauer der Suchanfrage pro Datenquelle
- Anzahl der Anfragen mit zugehörigen Status-Codes an die Datenquelle
- Anzahl der parallel laufenden Suchanfragen pro Datenquelle
- Anzahl von (potenziell aufgetretenen) Fehlermeldungen

Datenbank

- Anzahl von ausgeführten Operationen
- Speicherbedarf pro Client
- Antwortzeit

Um die Daten zu visualisieren wird Grafana eingesetzt. Dies ermöglicht die Erstellung von diversen Dashboards, auf welchen die oben genannten Metriken passend visualisiert werden können.

3.7 Bereitstellung

Da es sich bei der Architektur, wie in Kapitel 3.2 dargestellt, um eine Microservice-Architektur handelt, bietet es sich zur Bereitstellung der Komponenten an, eine Container-Lösung wie Docker zu verwenden. Jede Komponente stellt ein Image bereit, sofern noch nicht vorhanden, welches die jeweilige Applikation beinhaltet. Nach Bedarf können Umgebungsvariablen definiert werden, welche Teile der Applikation anpassbar machen. Es ist allerdings angedacht, dass eine Basisversion ohne manuell angepasste Konfiguration lauffähig ist. Entsprechend kann die gesamte Applikation innerhalb einer Docker Compose Datei definiert und auf dem Zielsystem bereitgestellt werden. Als weiterer Schritt könnte eine Orchestrierungsplattform wie beispielsweise Docker Swarm oder Kubernetes eingesetzt werden, um spezielle Anforderungen an Skalierbarkeit und Verfügbarkeit zu erfüllen. Dies ist im Rahmen dieser Arbeit allerdings nicht notwendig, wäre aber für einen produktiven Einsatz interessant.

Die virtuelle Maschine, auf welcher das Projekt bereitgestellt wird, befindet sich in der bwCloud, da diese für Studierende der DHBW einfachen Zugriff auf diverse Rechenressourcen ermöglicht. Prinzipiell spielt es hier allerdings keine Rolle, welcher Anbieter oder welche Plattform als Grundlage verwendet wird.

Um automatisiert Aktualisierungen der Teilkomponenten bereitzustellen, soll eine [CI/CD](#) Pipeline eingesetzt werden. Sobald es zu Änderungen kommt, müssen lediglich die aktualisierten Komponenten, mittels der vorher erwähnten Docker Compose Konfiguration, neugestartet werden. Dies hat den Vorteil, dass die Konfiguration zentral und codenah erfolgt und es nicht zu Diskrepanzen zwischen Soll und Ist-Zustand kommt.

4 Umsetzung

Nachdem im vorherigen Kapitel das Konzept erarbeitet wurde, kann dieses nun umgesetzt werden. In diesem Kapitel werden relevante Ausschnitte der Implementierung gezeigt und erläutert.

4.1 Datenquellen und Scraper

Es wurde sich für die Umsetzung für eine beispielhafte Anbindung an die Plattformen dastelefonbuch.de [8], welche über 18,9 Millionen Festnetz Telefonnummern aus Deutschland samt Vor- und Nachnamen sowie Adressen bereitstellt, und fupa.net [18] entschieden, welche bereits 2016 auf über 925 Millionen Seitenaufrufe verteilt, verschiedenste Daten (Vor- und Nachname, Geburtstag, Größe, Gewicht, Nationalität oder letzte Spielaktivitäten) zu Fußballspielern und deren Vereinen veröffentlicht. Da Facebook die Suchfunktion der Graph API [13] seit 2017 [14] stark eingeschränkt hat, wurde von einer direkten Anbindung als Beispiel-Datenquelle abgesehen. Somit ergeben sich zwei Scrapertypen, welche implementiert werden sollen: Für [XML](#) (Das Telefonbuch) und für [JSON](#) (FuPa API). Im Rahmen dieser Arbeit wird explizit lediglich die Anbindung von fupa.net beschrieben, es sei allerdings gesagt, dass andere Datenquellen (auch Das Telefonbuch) ähnlich anzubinden sind.

4.1.1 Generische Scraper am Beispiel fupa.net

Eine Besonderheit von fupa.net ist, dass die Daten zweistufig geladen werden. Dies bedeutet, dass in einem ersten Schritt mittels der [JSON API](#) Schnittstelle Basisdaten der Spieler geladen werden. In einem zweiten Schritt werden dann, über den expliziten Aufruf der Profilseite des Spielers ([HTML](#)), weitere Informationen gesammelt und der Datensatz so angereichert. Da zuerst der Aufruf an die [API](#) passieren muss, können die Scraper für diese Datenquelle nicht parallel arbeiten. Der [XML](#) Scraper muss auf die Ergebnisse des vorherigen Jobs warten. Um möglichst viele Datenquellen mit wenig Aufwand anbinden zu können, bietet sich ein generischer Ansatz an. So wurde eine Lösung implementiert, welche anhand einer vorgegebenen einheitlichen Konfiguration die entsprechenden Daten von beliebigen Datenquellen aufgearbeitet zurückgeben kann. Listing 4.1 zeigt als Referenz den Aufbau der Antwort von fupa.net. Diese ist verhältnismäßig simpel aufgebaut und

enthält nur ein Array mit den jeweiligen Objekten, welche die Attribute einzelner Spieler enthalten.

```

1 [ 
2   {
3     "type": "player",
4     "id": 1632417,
5     "name": "Markus Mueller",
6     "club": {
7       ...
8     },
9     ...
10 },
11 ...
12 ]

```

Listing 4.1: Antwort der fupa.net API

```

1 { 
2   client_id: -1,
3   url: 'https://search.fupa.net/v2/query?...',
4   options: {
5     arrayRoot: '$.*',
6     attributes: {
7       'schema.org:name': '$.name',
8       'schema.org:homeTeam': {
9         attributes: {
10           'schema.org:image': {
11             prefix: 'https://cdn.fupa.net/club/jpeg/200x200/',
12             value: '$.club.image.baseName',
13             onlyAddIf: {
14               '$.club.image.svg': false,
15             },
16             },
17           },
18         onlyAddIfNot: {
19           '$.club': 'undefined',
20         },
21       },
22     },
23   }
24 }

```

Listing 4.2: Konfiguration einer JSON Datenquelle

Listing 4.2 zeigt nun die zugehörige Konfiguration, um die Daten in ein standardisiertes Format zu bringen. In Zeile 3 wird die URL definiert, an welche der Scraper die Anfrage schicken soll. Das options Objekt enthält dann die Informationen zum Startpunkt der

Daten (Zeile 5) und zu den gewünschten Attributen (Zeile 6-22). `arrayRoot` dient als Ausgangspunkt für die iterative Sammlung der Daten. Dies ist notwendig, wenn die Nutzdaten in der Antwort der Datenquelle weiter verschachtelt sind. Die Attribute werden dann relativ zu dem jeweils gelesenen Objekt verarbeitet. So ist der Spielername im Spielerobjekt im Attribut `name` zu finden (Zeile 7). Weiter ist es durch diese Struktur auch möglich, komplexe Attribute oder zusammengesetzte Objekte zu verarbeiten. In diesem Fall wird für das Heimteam des Spielers das Objekt `club` durchsucht (Zeile 8). Auch hier werden die gewünschten Attribute definiert. Eine Besonderheit hierbei sind die Modifikatoren, welche für jedes Attribut bei Bedarf angegeben werden können. In Zeile 11 wird mittels des `prefix` Modifikators definiert, dass zu dem gefundenen Wert ein konstanter Prefix hinzugefügt werden soll. Gleichzeitig soll das gesamte Attribut nur dem Resultat hinzugefügt werden, sofern die Bedingung in Zeile 14 erfüllt ist. Hierfür kann der `onlyAddIf` Modifikator verwendet werden. Entsprechend wird das Heimteam auch nur dann dem Resultat hinzugefügt, sofern der Spieler tatsächlich einem zugehörig ist (Zeile 18-20, `onlyAddIfNot`). Alle Konfigurationsparameter können rekursiv gesetzt werden und so potentiell beliebige Datenstrukturen darstellen.

```

1 {
2   client_id: -1,
3   url: profileUrl,
4   options: {
5     httpNotFoundStatusCode: 404,
6     root: '#app',
7     rootIsArray: false,
8     attributes: {
9       'schema.org:name': /* gekürzter XML Path */,
10      'schema.org:homeTeam': {
11        attributes: {
12          'schema.org:url': {
13            prefix: 'https://www.fupa.net',
14            useHtmlAttribute: 'href',
15            value: /* gekürzter XML Path */,
16          },
17        },
18        onlyAddIfNot: {
19          /* gekürzter XML Path */: '-',
20        },
21      },
22    },
23  },
24 }
```

Listing 4.3: Konfiguration einer XML Datenquelle

Während die oben gezeigte Spielerliste zwar grundlegende Informationen enthält, können über den Aufruf der Detailseite weitere Daten eingeholt werden. Die Scraper Logik durchsucht also für jeden gefundenen Spieler die Profil [URL](#), welche eine Webseite im [HTML](#) Format liefert. Entsprechend kommt hier der generische [XML](#) Scraper zum Einsatz. Listing 4.3 zeigt die Konfiguration hierfür, welche im Aufbau allerdings sehr ähnlich zu der vorher gezeigten [JSON](#) Konfiguration ist. Der relevante Unterschied ist, dass in diesem Fall für die Definition der Position einzelner Attribute nicht der JSONPath sondern der XPath eingesetzt wird. Weiter kommt in Zeile 5 die Information hinzu, welchen [HTTP](#) Status Code der Scraper als `Not Found` interpretieren soll und ob es sich bei dem Starttag bereits um ein Array handelt oder nicht (Zeile 7).

4.1.2 Implementierung des generischen JSON Scrapers

Der Kern der generischen Scraper liegt in der `getAttributes` Funktion, gezeigt in Listing 4.4. Angemerkt sei, dass sich die Implementierungen für **JSON** und **XML** im Grundprinzip nicht unterscheiden, weshalb nur der **JSON** Scraper vorgestellt wird.

```

1 private getAttributes(searchObj, options): any {
2   const searchResultItem = {};
3
4   if (options.attributes) {
5     Object.keys(options.attributes).forEach(key => {
6       const valueSelector = options.attributes[key];
7
8       searchResultItem[key] = jp.value(searchObj, valueSelector);
9
10      const attributeObj = (options.attributes[key] as any);
11
12      if (attributeObj.prefix !== undefined) {
13        searchResultItem[key] =
14          attributeObj.prefix + searchResultItem[key];
15      }
16
17      if (attributeObj.attributes !== undefined) {
18        searchResultItem[key] =
19          this.getAttributes(searchObj, attributeObj);
20      }
21
22      if (attributeObj.onlyAddIf !== undefined) {
23        Object.keys(attributeObj.onlyAddIf).forEach(
24          conditionSelector => {
25            try {
26              if (jp.value(searchObj, conditionSelector) !==
27                  attributeObj.onlyAddIf[conditionSelector]) {
28                delete searchResultItem[key];
29              }
30            } catch(e) {}
31          }
32        );
33      }
34    });
35  }
36
37  return searchResultItem;
38}

```

Listing 4.4: Implementierung des generischen Scrapers (gekürzt)

Die Funktion `getAttributes` erhält zwei Parameter: Das von der Datenquelle gelieferte (singuläre) Datenobjekt und die manuell definierten Konfigurationsoptionen. Sofern diese Optionen die Attributbehandlung definieren (Zeile 4) wird über jedes Element des Objekts iteriert (Zeile 5), um die entsprechenden Werte zu finden (Zeile 8) und Modifikatoren anzuwenden. Sollte der `prefix` Modifikator (Zeile 12-15) eingesetzt werden, wird dem gefundenen Wert der definierte Prefix hinzugefügt. Ähnlich funktioniert die Behandlung des `onlyAddIf` Modifikators (Zeile 22-33) mit dem Unterschied, dass der Wert entfernt wird, sollte die Bedingung gebrochen werden. Hierfür wird über alle definierten Konditionen iteriert (Zeile 23) und der gefundene Wert auf Ungleichheit mit dem Erwartungswert geprüft (Zeile 26-27). Sollte dies der Fall sein, wird das Attribut aus dem Ergebnisobjekt entfernt (Zeile 28). Es wäre an dieser Stelle auch möglich, die Modifikatoren zuerst zu prüfen, bevor die jeweiligen Attribute in das Ergebnisobjekt geschrieben werden. Es wurde sich allerdings gegen diesen Ansatz entschieden, da in der implementierten Version dynamischer neue Modifikatoren eingefügt werden könnten und auch die Mehrfachverwendung von Operatoren für ein Attribut einfacher möglich ist. Zuletzt wird in Zeile 17-20 die Möglichkeit für die Verarbeitung komplexer Strukturen gelegt. Sollten weitere Kindattribute in der Konfiguration bestehen, so wird die Funktion rekursiv mit den angepassten Parametern aufgerufen.

4.1.3 Jobverwaltung mittels Message Queue

Wie angesprochen kommt für die Verteilung der einzelnen Jobs eine Message Queue zum Einsatz. In diesem Fall wird BullMQ eingesetzt, welche im Hintergrund eine Redis Datenbank zur Zustandsverwaltung verwendet. Listing 4.5 zeigt die Funktion mittels welcher die Jobs in die Message Queue gebracht werden. Ein Job entspricht hierbei einer Anfrage an eine spezifische Datenquelle. Anschaulich gesprochen bedeutet dies für eine Konfiguration mit zwei Datenquellen (beispielsweise LinkedIn und Facebook), dass bei einer Suchanfrage für Max Mustermann zwei Jobs erstellt werden, welche jeweils von dem Worker für LinkedIn beziehungsweise Facebook abgearbeitet werden. Die Funktion erhält die gewünschte Queue, zu welcher der Job hinzugefügt werden soll, sowie einen Namen und die entsprechende Konfiguration als Parameter (Zeile 2-4). Es wird dann zunächst geprüft, ob die Queue freie Worker hat, welche den Job abarbeiten könnten (Zeile 7-8). Sollte dies der Fall sein kann der Queue ein neuer Job angefügt werden (Zeile 10).

```

1 private async process(
2   queue: bull.Queue,
3   jobName: string,
4   jobOptions: any
5 ): Promise<any> {
6   return new Promise(async (resolve, reject) => {
7     if ((await queue.getWorkers()).length === 0) {
8       reject(`#${queue.name} - No workers available...`);
9     } else {
10       const job = await queue.add(jobName, jobOptions);
11
12       job.finished().then((data) => {
13         resolve(data);
14       }).catch((err) => {
15         reject(err);
16       });
17     }
18   });
19 }

```

Listing 4.5: Hinzufügen eines Jobs zur Message Queue

```

1 @Processor('scrapers-api')
2 export class ScrapersQueueConsumer extends BaseQueue {
3   @Process({
4     concurrency: 100,
5     name: 'genericJsonApi',
6   })
7   private async processGenericJsonApi(job: Job<JobDto>): Promise<any> {
8     job.progress(10);
9
10    const result = await this.genericJson ApiService.getSearchEntries(job
11      .data.url, job.data.options).catch((error) => {
12      throw new Error(error);
13    });
14
15    result.scrapers_id = 0;
16    result.client_id = job.data.client_id;
17
18    job.progress(100);
19    return result;
20  }

```

Listing 4.6: Empfangen der Jobs im Worker

Auf der anderen Seite der Message Queue, bei den Workern, ist je nach Queue eine eigene Funktion definiert, welche passende Jobs abarbeiten kann. Listing 4.6 zeigt die Funktion für den generischen API Scraper. So wird in der Queue `scrapers-api` (Zeile 1) mittels der Funktion `processGenericJsonApi` ermöglicht, dass Jobs, welche zu dieser Queue hinzugefügt werden, von dieser Funktion bearbeitet werden können. Es ist möglich, mittels des definierten Namens im `@Process` Dekorator (Zeile 5) die Auswahl der Funktion noch weiter zu verfeinern (dies entspricht dem oben genannten `jobName`). Weiter wird für diese Funktion definiert, wie viele Jobs gleichzeitig bearbeitet werden dürfen, in diesem Fall 100 (Zeile 4). Der Inhalt der Funktion ist, rein im Sinne der Message Queue, prinzipiell vollkommen variabel. In diesem Fall ist relevant, dass die Konfiguration an die Funktion übergeben wird (Zeile 7) und dass das entsprechende Ergebnis abgefragt und zurückgegeben wird (Zeile 10 & 18). Optional wird an dieser Stelle auch ein Fortschritt zurückgegeben, wobei dieser hier nicht repräsentativ zur Dauer der einzelnen Bearbeitung ist (Zeile 8 & 17).

Nachdem die Daten nun von den einzelnen Datenquellen geladen wurden, können diese innerhalb der API verarbeitet und verknüpft werden. Die Beschreibung hierfür folgt im nächsten Kapitel.

4.2 API

Die API ist die zentrale Schnittstelle, um mit den Datenquellen und der Datenbank zu interagieren. Das Frontend kann entsprechend Anfragen über festgelegte Schnittstellen schicken, um Daten zu empfangen oder auch zu senden. Relevant ist in diesem Fall, wie die Suchanfrage innerhalb der API bearbeitet wird und wie final die Ergebnisse an das Frontend zurückgeliefert werden. Zuerst ist es allerdings nötig eine Login-ähnliche Logik zu implementieren, damit jedem Benutzenden eine oder mehrere Suchen eindeutig zugeordnet werden können.

4.2.1 Login und Statusmanagement

Für den Login sind drei Funktionen notwendig, wie in Listing 4.7 gezeigt. `addClient` ist dafür zuständig, neue Verbindungen zu akzeptieren und für jeden Benutzenden eigene Nodes innerhalb der Datenbank anzulegen. Dieser Node werden später, sobald eine Suchanfrage gestartet wird, weitere Details über diese angehängt (beispielsweise Suchparameter). Um zu prüfen, ob sich ein Benutzer bereits angemeldet hat, wird die Funktion `isClient` eingesetzt. Diese prüft lediglich, ob bereits eine Node mit den entsprechenden Daten existiert oder nicht. Diese Funktion ist notwendig, um initial zu prüfen, ob ein Benutzer neu angelegt

werden muss oder der bestehende erweitert werden kann. Zuletzt liefert die Funktion `getAllClients` alle aktuell bekannten Benutzer. Diese Funktion dient allerdings eher zur Administration und hat für die explizite Suche keine Auswirkung.

```

1 public addClient(clientId: string): Promise<neo4j.Record[]> {
2   const options = '{ clientId:' + clientId + '}';
3   return this.executeStatement('MERGE (n:Client ' + options + ') ON CREATE
4     SET n.created = timestamp() RETURN n;');
5
6 public async isClient(clientId: string): Promise<boolean> {
7   const options = '{ clientId:' + clientId + '}';
8   const result = await this.executeStatement('MATCH (n:Client ' + options +
9     ') RETURN n.clientId as clientId LIMIT 1;');
10  return result.length != 0;
11 }
12 public async getAllClients(): Promise<string[]> {
13   const result = await this.executeStatement('MATCH (n:Client) RETURN n.
14     clientId as clientId LIMIT 20;');
15 }
```

Listing 4.7: Funktionen zum Login in der API

Nachdem sich die Benutzenden im Frontend eingeloggt haben und die entsprechenden Nodes in der Datenbank angelegt wurden, ist die API bereit, Suchanfragen zu bearbeiten. Dieser Schritt wird im Folgenden erklärt.

4.2.2 Aufbereitung der Daten

Bei Erstellung einer Suchanfrage wird zuerst für den zugehörigen Benutzer eine `SearchNode` erstellt. Diese erhält einen zufälligen Identifikator und Beziehungen zu den jeweiligen Suchparametern. Dieser Prozess ist in Abbildung 4.1 dargestellt.

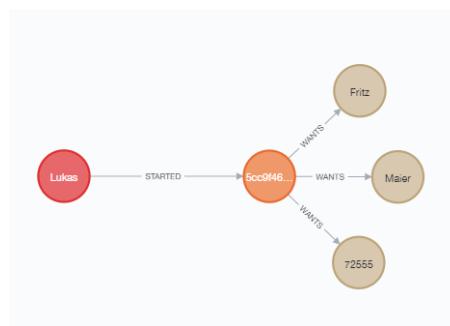


Abbildung 4.1: Aufbau der Suchanfrage im Graphen

Mit den vorgegebenen Erwartungswerten wird dann die Suche gestartet. Die Ergebnisse der Scraper werden, sobald sie geliefert werden, grundlegend bereinigt und folgend in der Datenbankpersistiert. Die Bereinigung erfolgt für jedes Attribut mittels der `cleanAttributes` Funktion. Leere Werte (`undefined` || `''` || `' '`) werden entfernt und valide Strings getrimmt (führende und folgende Leerzeichen entfernt). Die Funktion `addAttributeNode` ist in Listing 4.8 gezeigt und dient dazu, die bereinigten Attribute und Ergebnisse in die Datenbank einzufügen.

```

1 public addAttributeNode(
2   clientId: string,
3   type: string,
4   value: string,
5   cleanedValue?: string
6 ): Promise<neo4j.Record[]> {
7   const options = `{
8     title:'${value}',
9     clientId:'${clientId}',
10    type:'${type}'
11    '${cleanedValue ? ', cleanedValue:'${cleanedValue}' : ''}'
12  `;
13   return this.executeStatement(`
14     MERGE (n:Attribute ${options})
15     ON CREATE SET n.created = timestamp() RETURN n;
16   );
17 }
```

Listing 4.8: Query zum Hinzufügen eines Attributs

Anhand der Eingabeparameter werden die Parameter der Node erstellt. Diese enthalten die `clientId`, `type` und `cleanedValue`. Durch die Bedingung in Zeile 11 wird verhindert, dass der Parameter `cleanedValue` gesetzt wird, obwohl er nicht vorhanden ist. Optional wurde auch `title` hinzugefügt. Dies dient einerseits dazu, den originalen Wert weiterhin verfügbar zu haben, falls ein Benutzer sich diesen im Frontend erneut anschauen lassen möchte. Andererseits verwendet die Software Neo4j Desktop¹ das `title`-Attribut, um den Nodes verständliche Namen zu geben, statt eine generische ID. Im Kern der Funktion steht entsprechend der Datenbankquery. Das Schlüsselwort `MERGE` wird an dieser Stelle eingesetzt, um nicht manuell prüfen zu müssen, ob bereits Attribut-Nodes mit dem gleichen Wert (also denselben Parametern) existieren. Die Datenbank prüft dies bei Ausführung des Queries und erstellt entweder eine gänzlich neue Node oder vereint eine bestehende Node mit den neuen Optionen. Zusätzlich wird der Zeitpunkt der Erstellung in den Parametern der Node festgehalten.

¹ Siehe auch: <https://neo4j.com/download/>

```
1 if (typeof value === 'object') {  
2     await this.neo4jService.addComplexAttributeNode(clientId, key, value);  
3     await this.neo4jService.addResult2ComplexAttributeRelationship(...);  
4 } else {  
5     await this.neo4jService.addAttributeNode(clientId, key, value);  
6     await this.neo4jService.addResult2AttributeRelationship(...);  
7 }
```

Listing 4.9: Unterscheidung der Art des Attributs

Listing 4.9 zeigt eine weitere Unterscheidung beim Anlegen der Attribut-Nodes. Während im Konzept angedacht war, alle Attribute flach darzustellen, wurde sich in der Umsetzung dazu entschieden, die Zwischenebene `ComplexAttribute` einzuführen. Dies hat den Vorteil, dass semantisch zugehörige Attributgruppen weiterhin unterscheidbar sind. Deutlich wird dies beispielsweise bei einer Adresse. Wenn alle Attribute flach dargestellt würden, geht der Mehrwert, welcher initial durch die Gruppierung der Einzelattribute vorhanden war, verloren. Es ist prinzipiell davon auszugehen, dass die Einzelattribute, welche für ein Ergebnis von der Datenquelle geliefert werden, auch in diesem Kontext zusammengehören.

Nachdem die von den Scrapern gelieferten Ergebnisse mit den zugehörigen Attributen auf die oben gezeigte Art in den Graph eingefügt wurden, sieht das Ergebnis wie in Abbildung 4.2 angedeutet aus. An dieser Stelle kann nun begonnen werden, die tatsächlich relevanten Ergebnisse von den irrelevanten zu trennen.

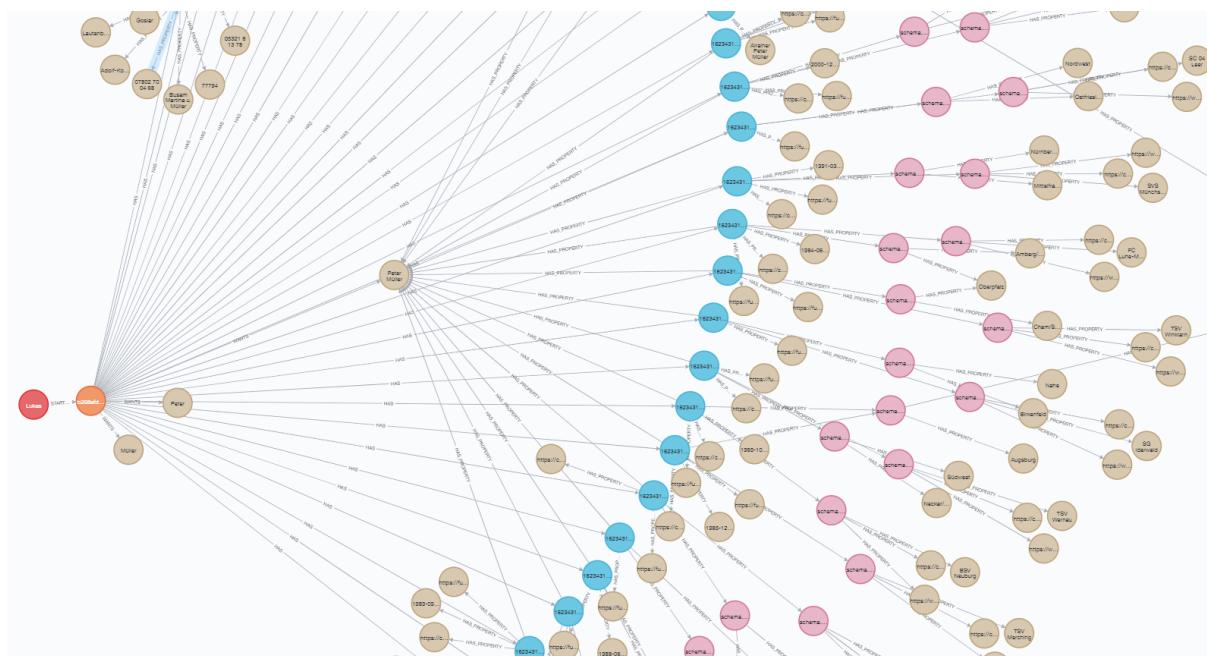


Abbildung 4.2: Ausschnitt des gefüllten Graphen

4.2.3 Generierung des Ergebnisdatensatzes

Anhand der im Konzept definierten Regeln wurden entsprechende Datenbankabfragen definiert, welche die Ergebnismenge korrekt zurückliefern. Um alle definierten Regeln korrekt behandeln zu können, ergibt sich der folgende Ablauf:

1. Ergebnismenge generieren, die Regel 1 und 2 erfüllt.
2. Wiederhole für jedes Ergebnis:
 - a) Auslesen der zugehörigen Attribute für das Ergebnis.
 - b) Regel 3 erfüllt? Wenn Nein: Ignorieren des Ergebnisses.
 - c) Regel 4 erfüllt? Wenn Ja: Speichern des Ergebnisses in separater Liste.
 - d) Ansonsten wird das Ergebnis in der Liste `resultSet` gespeichert.

```

1 MATCH (s:Search { searchId:'${searchId}' })-[:WANTS]->(a:Attribute)
2 MATCH (s)-[:HAS]->(rule1:Result)-[:HAS_PROPERTY]->(a)
3
4 WITH collect(rule1) AS rule1List
5
6 MATCH (s:Search { searchId:'${searchId}' })-[:WANTS]->(a:Attribute)
7 MATCH (s)-[:HAS]->(rule1:Result)-[:HAS_PROPERTY]->(a)
8 MATCH (rule1)-[:HAS_PROPERTY]->(b:Attribute)
9 MATCH (s)-[:HAS]->(rule2:Result)
10 WHERE NOT (rule2)-[:HAS_PROPERTY]->(a)
11 MATCH (rule2)-[:HAS_PROPERTY]->(b)
12
13 WITH rule1List + collect(rule2) as rule12
14
15 UNWIND rule12 as resultNodes
16 RETURN distinct ID(resultNodes);

```

Listing 4.10: Abfrage aller Ergebnisse, die Regel 1 und 2 erfüllen

Listing 4.10 zeigt die Datenbankabfrage, welche alle laut den Regeln 1 und 2 relevanten Ergebnisse liefert. Dafür wird mittels der `MATCH` Anweisung zuerst die Liste aller Ergebnisse erstellt, welche direkt über ein Attribut mit den Suchparametern verbunden sind (Zeile 1 & 2). Diese werden in der Zwischenvariable `rule1List` gespeichert. Um die Ergebnisse der zweiten Regel zu erhalten, werden zuerst alle weiteren Attribute abgefragt, auf welche die vorherigen Ergebnisse von Regel 1 zeigen (Zeile 7 & 8). Für die Ergebnisse der zweiten Regel kommen dann die Nodes in Frage, welche nicht bereits eine Verbindung zum Suchparameter haben (Zeile 10) sondern die, die eine Verbindung zu einem Attribut aus der Ergebnismenge

der ersten Regelanwendung haben (Zeile 11). Zuletzt werden die Ergebnisse beider Regeln zusammengefasst und als einzelne Elemente zurückgegeben (Zeile 13 - 16).

Es sei angemerkt, dass mit dieser Implementierung keine explizite Prüfung für Regel 5 erforderlich ist. Wenn von den Suchparametern keine Beziehungen zu einem passenden Attribut hergestellt werden können, erhält der Benutzende einen leeren Graph wodurch im Frontend die Option zur Verfeinerung der Suche gestartet werden kann.

4.3 Frontend

Sobald ein Benutzer die Webseite der Applikation besucht, wird er aufgefordert, einen neuen oder bestehenden Benutzernamen auszuwählen. Diese Seite ist in Abbildung 4.3 gezeigt. Sie besteht aus einem Dropdown mit gespeicherten Benutzern oder der Option, per Texteingabe einen neuen Benutzer zu erstellen.

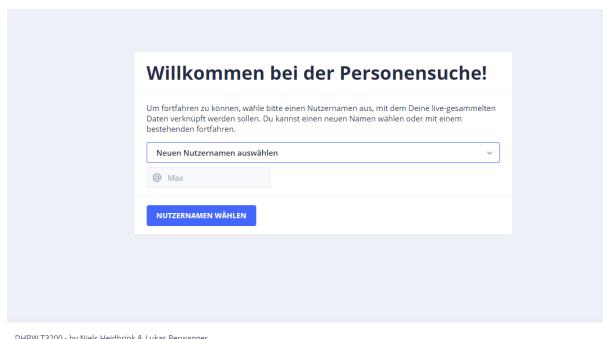


Abbildung 4.3: Screenshot der Login-Seite

Sobald der Login erfolgt ist, wird der Benutzende auf die in Abbildung 4.4 gezeigte Seite weitergeleitet. Auf dieser wird, sofern vorhanden, der Suchverlauf dargestellt. Für jede vergangene Suche können die Ergebnisse auf Wunsch erneut gesichtet werden. Alternativ kann von dieser Seite aus auch eine neue Suche gestartet werden.

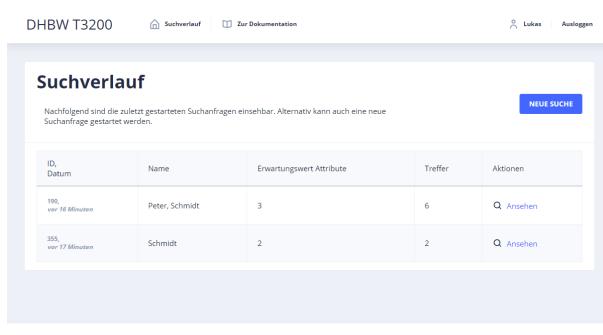


Abbildung 4.4: Screenshot des Suchverlaufs

4 Umsetzung

Wenn sich ein Benutzender entscheidet, eine neue Suche zu starten wird er aufgefordert, die entsprechenden Suchparameter, also Erwartungswerte, zu definieren. Dies passiert auf der Seite in Abbildung 4.5. In der Menüführung erhält der Benutzende die Möglichkeit, verschiedene Attribute, gruppiert nach der jeweiligen Kategorie, für die Suche anzugeben.

The screenshot shows a web-based search configuration interface. At the top, there are navigation links: 'DHBW T3200', 'Sicherheit', 'Zur Dokumentation', 'Lukas', and 'Ausloggen'. Below this, a search ID is displayed: 'searchId: 3359231f-4a1f-47ef-9588-48c8d0ff9e7152'. The main section is titled 'Erwartungswert' (Expectation Value). It contains a note: 'In diesem Schritt kann der Erwartungswert für eine neue Suchanfrage angepasst werden.' Below this, there are three tabs: 'Stammdaten' (selected), 'Social Media', and 'Interessen'. Under 'Stammdaten', there are fields for 'Name' (Name), 'Vorname' (First Name), 'Relevante Zwischennamen?' (Relevant middle names?), and 'Nachname' (Last Name). There are also fields for 'Ort' (Place), 'PLZ' (Postcode), and 'Stadt' (City). At the bottom of the form are two buttons: 'ZURÜCK' (Back) and 'WEITER' (Next).

Abbildung 4.5: Screenshot der Suchparametereingabe

Sobald die Suche aktiv ist, wird der Benutzende in Echtzeit über den aktuellen Suchfortschritt informiert. Die zugehörige Seite ist in Abbildung 4.6 gezeigt. Hier hat er auch die Möglichkeit, den Aufbau und die Suche im Graphen mitzuverfolgen, sowie einige Metadaten wie die Anzahl der Ergebnisse und eingesetzten Scrapper zu überprüfen.

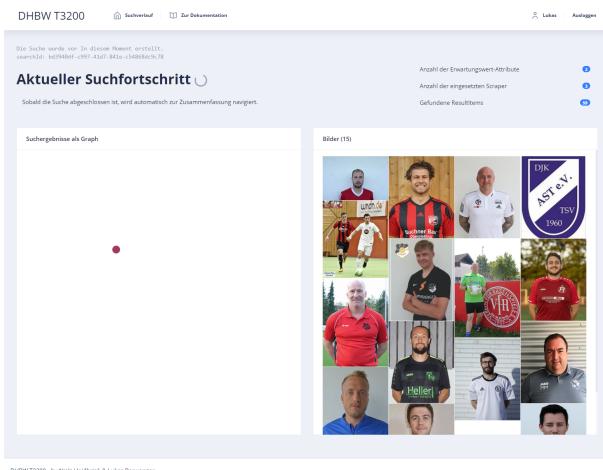


Abbildung 4.6: Screenshot der aktuellen Suchergebnisse

Die Kommunikation zwischen Frontend und API erfolgt, wie im Konzept beschrieben, über WebSocket Kanäle. Dabei werden die in Tabelle 4.1 gezeigten Kanäle eingesetzt.

`drawGraph` und `getAllImageUrls` dienen dazu, den Suchfortschritt im Frontend darstellen zu können. Die finalen Ergebnisse werden mittels `finishedSearch` propagiert. Etwaige Fehler können im Frontend mittels der in `scraperError` gesendeten Daten dargestellt werden. Der einzige WebSocket Kanal, auf welchem das Frontend Daten sendet ist `updateSearchRequest`. Dieser Kanal verwaltet neue und aktualisierte Suchanfragen und deren Parameter. Eine

	Empfangen	Senden
drawGraph	×	
finishedSearch	×	
getAllImageUrls	×	
scraperError	×	
updateSearchRequest		×

Tabelle 4.1: Übersicht der WebSocket Kanäle

Suchanfrage kann aktualisiert werden, wenn nach Beendigung der Suche neue Parameter eingefügt oder bestehende angepasst werden.

4.4 Bereitstellung

Während im Konzept die Verwendung der bwCloud zur Bereitstellung der Applikation vorgeschlagen wurde, wurde aufgrund von Stabilitätsproblemen von deren Einsatz abgesehen. Stattdessen kommt die [Google Cloud Platform \(GCP\)](#)¹ mit ihrer Kubernetes Engine² zum Einsatz. Auch hier steht Studenten ein gewisses Nutzungskontingent kostenfrei zur Verfügung. Es sei angemerkt, dass sowohl die Wahl eines Cloud Providers als auch Kubernetes³ als Orchestrierungsplattform optional sind und die Funktionalität des Projekts prinzipiell nicht beeinflussen.

Zur automatischen Bereitstellung wurde die in Abbildung 4.7 gezeigte [CI/CD](#) Pipeline entwickelt. Die Anforderung hierbei ist, dass bei Änderungen des Quellcodes automatisch die aktuelle Version getestet und öffentlich bereitgestellt wird. Aufgrund des Rahmens des Projekts wurde von einer Aufteilung in Test- und Produktivumgebungen abgesehen.

Wie die Abbildung zeigt, handelt es sich um drei sogenannte Stages, wobei für jeden Service erst definierte Tests durchgeführt und folgend das zugehörige Docker Image für die Applikation gebaut und in der Cloud bereitgestellt werden. Bei den Services handelt es sich um das Frontend, die [API](#) und die beiden Scraper für [HTML](#) und [JSON APIs](#).

¹ Siehe auch: <https://cloud.google.com/>

² Siehe auch: <https://cloud.google.com/kubernetes-engine>

³ Siehe auch: <https://kubernetes.io/>

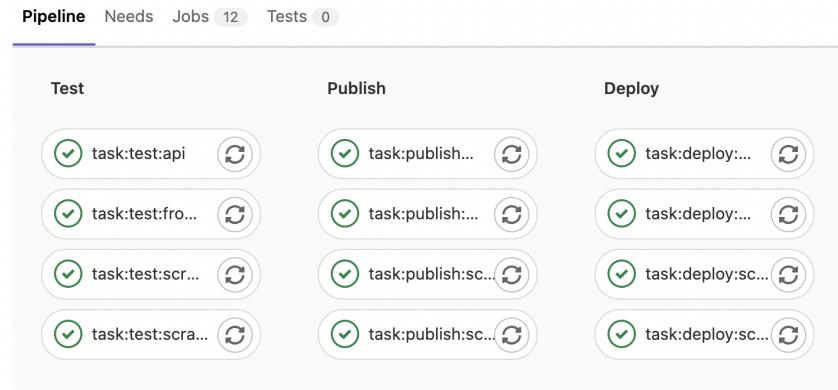


Abbildung 4.7: CI/CD Pipeline

Entwicklungshilfe Docker Compose

Für die lokale Entwicklung (oder die Bereitstellung auf einer virtuellen Maschine) kommt das Tool Docker Compose zum Einsatz. Dieses erlaubt mittels einer [YAML Ain't Markup Language \(YAML\)](#) Datei die Beschreibung und Konfiguration aller Services, welche zum Betrieb der Applikation benötigt werden. Listing 4.11 zeigt eine verkürzte Variante der Datei. Die Datenbanken verwenden das zugehörige (offizielle) Docker Image der Hersteller. Für das Frontend, die API und die Scraper wird jeweils zuerst das eigens definierte Dockerfile gebaut. Weiter sind einige Services mittels `links` verbunden. Dies bedeutet, dass die Netzwerkkommunikation zwischen ihnen möglich ist. Via `ports` werden auch nach außen Netzwerkports geöffnet. Dies ist unter anderem für das Frontend wichtig, damit es über das Internet erreichbar ist.

```

1 services:
2   frontend:
3     build: ./frontend/
4     ports:
5     - 80:80
6   api:
7     build: ./api/
8     links:
9     - bullmq-redis
10    - neo4j
11     ports:
12     - 3000:3000
13
14   scraper_html:
15     build: ./scraper/html/
16     links:
17     - bullmq-redis
18   scraper_api:
19     build: ./scraper/api/

```

```
20   links:  
21     - bullmq-redis  
22  
23 neo4j:  
24   image: neo4j  
25   ports:  
26     - 7474:7474  
27     - 7687:7687  
28 bullmq-redis:  
29   image: redis  
30   ports:  
31     - 6379:6379
```

Listing 4.11: Docker Compose Konfigurationsdatei

5 Fazit

Im diesem Kapitel wird das Fazit aus der Arbeit und den gewonnenen Erkenntnissen gezogen. Die an das Projekt gestellten Anforderungen werden evaluiert und ein Ausblick für einen potenziell produktiven Einsatz der Applikation gegeben.

5.1 Anforderungen

Zuerst werden die in Kapitel 3.1 definierten Anforderungen evaluiert. Tabelle 5.1 zeigt die funktionalen und nicht-funktionalen Anforderungen zusammengefasst auf.

	Titel	Erfüllt?
FA1	Frontend	Ja
FA2	API	Ja
FA3	Scraper	Ja
FA4	Standardisierung des Datenformats	Ja
FA5	Datenbank	Ja
FA6	Anreicherung der Inhalte	Ja
FA7	Monitoring	Nein
FA8	CI/CD	Ja
NFA1	Erweiterbarkeit	Ja
NFA2	Skalierbarkeit	Ja
NFA3	Portabilität	Ja
NFA4	Robustheit	Ja

Tabelle 5.1: Erfüllung der Anforderungen

Zu sehen ist, dass mit Ausnahme des Monitorings alle Anforderungen entsprechend der getroffenen Definitionen umgesetzt wurden. Die Scraper sammeln Daten von verschiedenen artigen Quellen, welche in der [API](#) verarbeitet werden. Die [API](#) speichert die Ergebnisse dann in einer Graphdatenbanken, in welcher durch die Verknüpfung mehrerer Datensätze ein Informationsnetz entsteht, in welchem zuletzt die tatsächlich relevanten Ergebnisse gefiltert werden. Das Frontend präsentiert den Benutzenden die entsprechenden Informationen

und Interaktionsmöglichkeiten, um das Gesamtsystem zu verwenden. Die Kommunikation zwischen den einzelnen Komponenten findet jeweils in einem einheitlich definierten Datenformat statt, wobei auch die Scraper-Ergebnisse vor der Verarbeitung normalisiert und bereinigt werden. Wenn Teile der Applikation aktualisiert werden, startet automatisch eine CI/CD-Pipeline, welche die neue Version auf dem Zielsystem bereitstellt. An dieser Stelle ist die Einschränkung zu machen, dass die Monitoring-Lösung aufgrund mangelnder Zeit nicht vollständig in die Applikation eingebaut werden konnte.

Die Erweiterbarkeit ist durch den Einsatz von generischen Scrapern gegeben, durch welche es möglich ist, mit geringem Konfigurationsaufwand neue Datenquellen anzubinden, um die Suche zu erweitern. Je nach verfügbarer Rechenkapazität ist es auch möglich, mehrere Suchanfragen gleichzeitig zu bearbeiten, ohne die Performanz zu verringern, was die Skalierbarkeit verbessert. Da es sich bei der Architektur um eine Microservice-Architektur handelt und die Teilkomponenten jeweils eigene Container Images anbieten, ist es möglich die Applikation auf einer beliebigen Maschine zu starten. Sollte widererwarten eine Teilkomponente ausfallen, wird diese automatisch neugestartet, um die restlichen Komponenten nicht negativ zu beeinträchtigen. Suchanfragen werden in diesem Fall wiederholt, es kommt zu keinem Datenverlust.

5.2 Haben sich die Erwartungen bewahrheitet?

Die in Abschnitt 1.3 formulierten Erwartungen haben sich prinzipiell bestätigt. Auch wenn die Anbindung verschiedener Datenquellen und die Normalisierung derer Daten prinzipiell mit moderatem Aufwand funktioniert, unterscheiden sie sich in ihrer Datenqualität teilweise deutlich. Auch durch die Verhinderung unautorisierter Zugriffe oder der inhärenten Limitierung der Daten, welche öffentlich über ein Benutzerprofil abgefragt werden können, liefert die Applikation nicht zwangsläufig die korrekte (gesuchte) Person. Auch die Erwartung, dass sich die Verbindung unabhängiger Datensätze komplex darstellt, wurde zumindest teilweise bestätigt. Während zwar gezeigt wurde, dass Ergebnisse anhand ähnlicher Attribute verbunden werden können, ist die Genauigkeit dieser gewonnenen Information zu gering, um ohne die Eingaben und ggf. Korrekturen der suchenden Person ein exaktes Personenprofil zu liefern. Einige dieser Limitierungen ergeben sich auch aus den Einschränkungen, welche vor Beginn der Arbeit definiert wurden. Könnte beispielsweise auf einen vorhandenen Datenbestand zugegriffen werden, um die Suchergebnisse zu erweitern, könnte potenziell schneller und genauer ein Ergebnis geliefert werden. Auch die Anbindung nicht-öffentlicher Quellen würde die Genauigkeit und Qualität der Daten deutlich anheben.

Zum produktiven Einsatz sei gesagt, dass die rechtliche Lage in Europa den Betrieb einer solchen Applikation (sowohl kommerziell als auch nicht-kommerziell) erschwert. Menschen, deren Daten verarbeitet werden, müssen prinzipiell über die Art und den Zweck dieser Verarbeitung informiert werden. Sollte das Unternehmen kein legitimes Interesse vorweisen können, ist weiter die explizite Zustimmung der betroffenen Personen erforderlich. Noch weiter eingeschränkt wird der produktive Einsatz dadurch, dass bei der Datensammlung auch keine technischen Schutzvorrichtungen umgangen werden dürfen. Hierunter könnte beispielsweise bereits die Login-Vorrichtung von Web-Plattformen fallen, hinter welcher weitere Profildaten verfügbar sind. Sollte ein automatisiertes System auf eine solche nicht-öffentliche Schnittstelle zugreifen, könnte auch das bereits eine Verletzung des geltenden Rechts darstellen. Dies ist insbesondere problematisch, da bereits festgestellt wurde, dass die Datenqualität solcher nicht-öffentlicher Datenquellen grundsätzlich höher ist als von anderweitig erreichbaren Quellen. [6, 27, 38]

Dennoch ist das Projekt in unseren Augen ein Erfolg. Während das Konzept aufgrund von mangelnder Zeit nicht vollständig in eine reale Applikation umgewandelt werden konnte, bietet diese Arbeit durchaus das Potenzial, Menschen auf ihren digitalen Fingerabdruck hinzuweisen und die Daten, welche im Internet über eine Person auffindbar sind, zentral darzustellen. So können Benutzende beispielsweise ihre Datenschutzeinstellungen bei diversen Anbietern anpassen oder entsprechende Maßnahmen ergreifen, um die vorhandenen Daten komplett entfernen zu lassen. Durch die Veröffentlichung des Quellcodes ist es auch möglich, die noch fehlenden Teile des Konzepts einzuarbeiten und die Datensammlung durch beliebig viele weitere Datenquellen anzureichern. Benutzende könnten bei Interesse die Applikation auf ihren eigenen Systemen starten, um so rechtskonform zu arbeiten.

Neben dem Erreichen der Projektziele enthielt das Projekt für das Team durchaus wertvolle Kenntnisse, wie beispielsweise die Arbeit mit einer Graphdatenbank oder die Vertiefung von Konzepten des Knowledge Graphen bzw. Semantic Web. Durch mehrere Iterationen des Konzepts wurden viele Technologien und Ansätze in Betracht bezogen, wodurch das im Team vorhandene Fachwissen deutlich erweitert werden konnte.

Literatur

- [1] Jorge Acetozi. „WebSocket“. In: *Pro Java Clustering and Scalability: Building Real-Time Apps with Spring, Cassandra, Redis, WebSocket and RabbitMQ*. Berkeley, CA: Apress, 2017, S. 55–58. ISBN: 978-1-4842-2985-9. DOI: [10.1007/978-1-4842-2985-9_9](https://doi.org/10.1007/978-1-4842-2985-9_9). URL: https://doi.org/10.1007/978-1-4842-2985-9_9.
- [2] *Angular, version 2: proprioception-reinforcement*. URL: <http://blog.angularjs.org/2016/09/angular2-final.html>.
- [3] Robert Baumgartner, Wolfgang Gatterbauer und Georg Gottlob. „Web Data Extraction System“. In: *Encyclopedia of Database Systems*. Hrsg. von LING LIU und M. TAMER ÖZSU. Boston, MA: Springer US, 2009, S. 3465–3471. ISBN: 978-0-387-39940-9. DOI: [10.1007/978-0-387-39940-9_1154](https://doi.org/10.1007/978-0-387-39940-9_1154). URL: https://doi.org/10.1007/978-0-387-39940-9_1154.
- [4] David Bechberger. *A Skeptics Guide to Graph Databases*. URL: <https://www.youtube.com/watch?v=y0YodfN84N4>.
- [5] Sara Bouchenak und Noël de Palma. „Message Queuing Systems“. In: *Encyclopedia of Database Systems*. Hrsg. von Ling Liu und M. Tamer Özsü. New York, NY: Springer New York, 2018, S. 2232–2233. ISBN: 978-1-4614-8265-9. DOI: [10.1007/978-1-4614-8265-9_1548](https://doi.org/10.1007/978-1-4614-8265-9_1548). URL: https://doi.org/10.1007/978-1-4614-8265-9_1548.
- [6] Benjamin Bremert und Harald Zwingelberg. *Text Mining öffentlich zugänglicher Daten*. URL: https://www.datenschutzzentrum.de/uploads/projekte/itesa/Bremert_Zwingelberg_Text-mining.pdf.
- [7] Mainak Chakraborty und Ajit Pratap Kundan. „Prometheus“. In: *Monitoring Cloud-Native Applications: Lead Agile Operations Confidently Using Open Source Software*. Berkeley, CA: Apress, 2021, S. 99–131. ISBN: 978-1-4842-6888-9. DOI: [10.1007/978-1-4842-6888-9_4](https://doi.org/10.1007/978-1-4842-6888-9_4). URL: https://doi.org/10.1007/978-1-4842-6888-9_4.
- [8] Das Telefonbuch-Servicgesellschaft mbH. *Das Telefonbuch digital – Telefonverzeichnis, Adressen & mehr*. URL: <https://www.dastelefonbuch.de/Online>.
- [9] Tommaso Di Noia und Paolo Tomeo. „Recommender Systems Based on Linked Open Data“. In: *Encyclopedia of Social Network Analysis and Mining*. Hrsg. von Reda Alhajj und Jon Rokne. New York, NY: Springer New York, 2018, S. 2064–2080. ISBN: 978-1-4939-7131-2. DOI: [10.1007/978-1-4939-7131-2_110165](https://doi.org/10.1007/978-1-4939-7131-2_110165). URL: https://doi.org/10.1007/978-1-4939-7131-2_110165.

- [10] „Die Idee des Semantic Web“. In: *Semantic Web: Grundlagen*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, S. 7–14. ISBN: 978-3-540-33994-6. DOI: [10.1007/978-3-540-33994-6_2](https://doi.org/10.1007/978-3-540-33994-6_2). URL: https://doi.org/10.1007/978-3-540-33994-6_2.
- [11] Fernando Doglio. „Node.js and REST“. In: *Pro REST API Development with Node.js*. Berkeley, CA: Apress, 2015, S. 47–63. ISBN: 978-1-4842-0917-2. DOI: [10.1007/978-1-4842-0917-2_3](https://doi.org/10.1007/978-1-4842-0917-2_3). URL: https://doi.org/10.1007/978-1-4842-0917-2_3.
- [12] Shakuntala Gupta Edward und Navin Sabharwal. „NoSQL“. In: *Practical MongoDB: Architecting, Developing, and Administering MongoDB*. Berkeley, CA: Apress, 2015, S. 13–23. ISBN: 978-1-4842-0647-8. DOI: [10.1007/978-1-4842-0647-8_2](https://doi.org/10.1007/978-1-4842-0647-8_2). URL: https://doi.org/10.1007/978-1-4842-0647-8_2.
- [13] Facebook Inc. *Advanced - Graph API - Documentation - Facebook for Developers*. URL: <https://developers.facebook.com/docs/graph-api/advanced/>.
- [14] Facebook Inc. *Using the Graph API - Documentation - Facebook for Developers*. URL: <https://web.archive.org/web/20170104062957/https://developers.facebook.com/docs/graph-api/using-graph-api/#search>.
- [15] Dieter Fensel u. a. „Introduction: What Is a Knowledge Graph?“ In: *Knowledge Graphs: Methodology, Tools and Selected Use Cases*. Cham: Springer International Publishing, 2020, S. 1–10. ISBN: 978-3-030-37439-6. DOI: [10.1007/978-3-030-37439-6_1](https://doi.org/10.1007/978-3-030-37439-6_1). URL: https://doi.org/10.1007/978-3-030-37439-6_1.
- [16] Roy Thomas Fielding. „Architectural Styles and the Design of Network-based Software Architectures“. Ph.D. dissertation. Irvine, CA 92697, United States: University of California Irvine, 2000. URL: <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>.
- [17] Thomas Frisendal. „Visual Design of GraphQL Data“. In: *Visual Design of GraphQL Data: A Practical Introduction with Legacy Data and Neo4j*. Berkeley, CA: Apress, 2018, S. 1–6. ISBN: 978-1-4842-3904-9. DOI: [10.1007/978-1-4842-3904-9_1](https://doi.org/10.1007/978-1-4842-3904-9_1). URL: https://doi.org/10.1007/978-1-4842-3904-9_1.
- [18] FuPa GmbH. *FuPa GmbH - Aus Liebe zum Fußball*. URL: <http://fupa.gmbh/>.
- [19] Google u. a. *Person - Schema.org Type*. URL: <https://schema.org/Person>.
- [20] Bo Hou. *Understanding nodejs event loop without a Computer Science degree*. URL: <https://medium.com/@bohou/understanding-nodejs-event-loop-without-a-computer-science-degree-e1c9250d583f>.
- [21] Anil Jain. *The 5 V's of big data*. URL: <https://www.ibm.com/blogs/watson-health/the-5-vs-of-big-data/>.

- [22] Stefanie King. „Theorie“. In: *Big Data: Potential und Barrieren der Nutzung im Unternehmenskontext*. Wiesbaden: Springer Fachmedien Wiesbaden, 2014, S. 29–74. ISBN: 978-3-658-06586-7. DOI: [10.1007/978-3-658-06586-7_2](https://doi.org/10.1007/978-3-658-06586-7_2). URL: https://doi.org/10.1007/978-3-658-06586-7_2.
- [23] Maksim Kolchin. *What is a Semantic Web Knowledge Graph? The main building blocks*. URL: <https://medium.com/datafabric/what-is-a-semantic-web-knowledge-graph-82078ea481bc>.
- [24] Mark Lahn. *The 5 V's of Big Data: Velocity, Volume, Value, Variety, and Veracity*. URL: <https://www.servermania.com/kb/articles/vs-of-big-data/>.
- [25] Andreas Meier und Michael Kaufmann. „NoSQL-Datenbanken“. In: *SQL- & NoSQL-Datenbanken*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, S. 221–240. ISBN: 978-3-662-47664-2. DOI: [10.1007/978-3-662-47664-2_7](https://doi.org/10.1007/978-3-662-47664-2_7). URL: https://doi.org/10.1007/978-3-662-47664-2_7.
- [26] Mozilla and individual contributors. *XPath*. URL: <https://developer.mozilla.org/en-US/docs/Web/XPath>.
- [27] Niklas Plutte. *BGH zum Screen Scraping: Auslesen fremder Websites*. URL: <https://www.ra-plutte.de/bgh-zum-automatisierten-auslesen-fremder-websites-via-screen-scraping/>.
- [28] Prometheus Authors. *Prometheus - GitHub*. URL: <https://github.com/prometheus/prometheus>.
- [29] Rob Reagan. „Message Queues“. In: *Web Applications on Azure: Developing for Global Scale*. Berkeley, CA: Apress, 2018, S. 343–380. ISBN: 978-1-4842-2976-7. DOI: [10.1007/978-1-4842-2976-7_9](https://doi.org/10.1007/978-1-4842-2976-7_9). URL: https://doi.org/10.1007/978-1-4842-2976-7_9.
- [30] Chris Richardson. *What are microservices?* URL: <https://microservices.io/> (besucht am 2020).
- [31] Chris Schaefer, Clarence Ho und Rob Harrop. „WebSocket“. In: *Pro Spring: Fourth Edition*. Berkeley, CA: Apress, 2014, S. 645–661. ISBN: 978-1-4302-6152-0. DOI: [10.1007/978-1-4302-6152-0_17](https://doi.org/10.1007/978-1-4302-6152-0_17). URL: https://doi.org/10.1007/978-1-4302-6152-0_17.
- [32] Preston So. „Angular“. In: *Decoupled Drupal in Practice: Architect and Implement Decoupled Drupal Architectures Across the Stack*. Berkeley, CA: Apress, 2018, S. 355–380. ISBN: 978-1-4842-4072-4. DOI: [10.1007/978-1-4842-4072-4_19](https://doi.org/10.1007/978-1-4842-4072-4_19). URL: https://doi.org/10.1007/978-1-4842-4072-4_19.

- [33] Statista Research Department. *Social Networks mit den meisten Nutzern weltweit 2021*. URL: <https://de.statista.com/statistik/daten/studie/181086/umfrage/die-weltweit-groessten-social-networks-nach-anzahl-der-user/>.
- [34] Tim Berners-Lee und weitere. *HTML - Living Standard*. URL: <https://html.spec.whatwg.org/multipage/>.
- [35] Balaji Varanasi und Sudha Belida. „Introduction to REST“. In: *Spring REST*. Berkeley, CA: Apress, 2015, S. 1–13. ISBN: 978-1-4842-0823-6. DOI: [10.1007/978-1-4842-0823-6_1](https://doi.org/10.1007/978-1-4842-0823-6_1). URL: https://doi.org/10.1007/978-1-4842-0823-6_1.
- [36] Eric Wohlstadter. „SOAP“. In: *Encyclopedia of Database Systems*. Hrsg. von Ling Liu und M. Tamer Özsu. New York, NY: Springer New York, 2016, S. 1–2. ISBN: 978-1-4899-7993-3. DOI: [10.1007/978-1-4899-7993-3_1189-2](https://doi.org/10.1007/978-1-4899-7993-3_1189-2). URL: https://doi.org/10.1007/978-1-4899-7993-3_1189-2.
- [37] Anthony D. Yao u. a. „Deep Learning in Neuroradiology: A Systematic Review of Current Algorithms and Approaches for the New Wave of Imaging Technology“. In: *Radiology: Artificial Intelligence* 2.2 (2020). PMID: 33937816, e190026. DOI: [10.1148/ryai.2020190026](https://doi.org/10.1148/ryai.2020190026). eprint: <https://doi.org/10.1148/ryai.2020190026>. URL: <https://doi.org/10.1148/ryai.2020190026>.
- [38] Zyte team. *GDPR compliance for web scrapers: The step-by-step guide*. URL: <https://www.zyte.com/blog/web-scraping-gdpr-compliance-guide/>.