MARTIN-LUTHER-UNIVERSITÄT
HALLE-WITTENBERG

MAX-PLANCK-INSTITUT
FÜR ETHNOLOGISCHE FORSCHUNG

Planung und Entwicklung einer Datenbankanwendung für das Forschungsdatenmanagement

Max Brauer (ma.brauer@live.de)

Bachelor für Informatik

Juli 2022

Erstgutachter: Dr. Stefan Brass

Zweitgutachter: Sebastian Ehser

Inhaltsverzeichnis

1 Einleitung

Diese Arbeit wurde am Max-Planck-Institut für ethnologische Forschung (MPI) und dem Institut der Informatik der Martin-Luther-Universität Halle-Wittenberg erstellt. Das Ziel ist, das Forschungsdatenmanagement von ethnologischen Instituten (im diesen Fall das MPI) mit Hilfe aktueller Techniken und Software zu verbessern.

Die Forscher des MPI begeben sich, bedingt durch ihre Tätigkeit, auf weltweite Reisen, um mit einer großen Vielfalt an Personen Interviews zu führen und Daten für ihre Forschung zu sammeln. Hauptsächlich werden diese Interviews in Papierform (handschriftliche Notizen) oder als einfache Audioaufnahmen dokumentiert. Letzteres wird händisch als Textdatei transkribiert und alles zusammen in einer großen Sammlung an Word-Dokumenten abgespeichert und archiviert.

Eine weitere Verarbeitung mit diesen Dokumenten erfolgt dann nur noch über das Programm Microsoft Word und/oder mit den Ausdrucken in Papierform. Diese Dokumente erfahren häufig keine richtige Versionierung (Änderungen sind nicht mehr oder schwer nachvollziehbar) und spätestens beim Teilen mit Forschungskollegen entstehen Duplikate die später nur sehr aufwändig händisch vereinigt werden.

Ein weiterer wichtiger Aspekt ist die Sicherheit der aufgenommenen Daten. Die Daten waren bisher ungesichert auf Papier oder auf elektronischen Datenträgern hinterlegt und es war bisher schwierig diese vor unbefugten Einsehen oder vor Zerstörung zu schützen. Dies kann mitunter an Grenzübertritten in Staaten wie China (#TODO: weitere Beispiele) geschehen, wo das Gepäck durchsucht oder einbehalten wird. In dieser Arbeit wurde dieser Aspekt der Sicherheit sehr hoch angesehen und es wurden verschiedene Leitlinien für den sicheren Umgang mit Daten umgesetzt.

Zur Sicherheit gehört auch der Datenschutz selbst. Nur berechtigte Personen sollen Zugang zu den Daten haben, diese einsehen oder modifizieren können. Dazu wurde in erster Linie eine Zwei-Faktor-Authentifizierung eingeführt. Zum Datenschutz gehört es auch, dass bestimmte Daten ab einen bestimmten Zeitpunkt nicht mehr für die Forschung relevant und deshalb für den Forscher nicht mehr zugänglich sein sollen. Als Beispiel lassen sich hier personenbezogene Daten, wie Name oder Kontakt der Befragten nennen.

Eine strukturiertes Abspeichern der Daten und eine schnelle Suche in diesen ist der zweite große Punkt, der in dieser Arbeit verfolgt wird. Dem Forscher soll es möglich sein, seine Daten schnell, einfach und sicher eingeben können und sehen was sich wann geändert hat. Dadurch wurde auch der Grundsatz der Versionierung berücksichtigt.

Der letzte große Punkt ist das einfache Teilen und Erstellen eines Backups der Daten selbst. Diese wurde mit einfachen Import und Exports der Datenbanken selbst geregelt.

Diese Arbeit ist um den Entwicklungsprozess der Anwendung strukturiert. Im ersten Kapitel nach dieser Einleitung geht es um die inhaltlichen und technischen Anforderungen, die an die Anwendung gestellt werden und eine Auswahl der Komponenten wird getroffen. Im nächsten Kapitel geht es um die Sicherheit, dem Datenschutz und die technische Basis dazu. Danach wird das technische System entworfen und die einzelnen Komponenten werden genauer beleuchtet. Schließlich geht es um die Umsetzung und die Stolpersteine, die dabei aufgetreten sind. Im darauffolgenden Kapitel geht es um die Tests, welche genutzt wurden und welche Vor- bzw. Nachteile diese gebracht hatten. Anschließend geht es darum, wie die Anwendung an alle Nutzer ausgerollt wird und welche Zyklen dabei durchlaufen werden. Nach diesen Kapitel kommt dann der Ausblick, wo die Nachnutzung, Erweiterungen und

Wartung und Updates genauer beleuchtet werden. Schlussendlich kommt der Schluss mit dem Fazit über diese Arbeit.

2 Anforderungen

In den Gesprächen mit dem Forschern und dem Auftraggeber wurde eine Liste an Anforderungen erstellt, die für das Projekt zu erfüllen sind. Dazu gehören inhaltliche, welche besagen was die Anwendung für den Forscher primär leisten soll. Aber auch strukturelle, wie mit dem Daten, die der Forscher bei seinen Forschungen erstellt oder welche bei der Arbeit mit dem Programm entstehen, verarbeitet und gespeichert werden sollen. Weiterhin müssen auch technische Anforderungen über das System, die Speicherdauer und Architektur erfüllt werden.

2.1 Inhaltliche Anforderungen

Die Anwendung soll die Daten, welche bei den Forschungsreisen und Interviews eines Forschers anfallen aufnehmen und übersichtlich darstellen und organisieren. Diese Daten beinhalten folgendes:

- **Informationen über den Interviewpartner**

  Dazu zählen Kontaktinformationen wie Name, Adresse und Telefonnummer. Außerdem müssen je nach Interviewpartner auch bestimmte Dokumente hinterlegt werden, wie die unterschriebene Datenschutzerklärung.

- **Arten der Interviewpartner**

  Diese sind je nach Forschungsthema unterschiedlich und können zum Beispiel Patient, Arzt, Krankenpfleger, familiäre Angehörige und religiöse Beistehende sein.

- **Familiäre Beziehungen zwischen den Interviewpartnern**

  Dies ist auch je nach Forschungsthema unterschiedlich, ob diese überhaupt aufgenommen werden sollen oder dürfen.

- **Informationen wann und wo ein Interview stattgefunden hat und wer dort interviewt wurde**

- **Antworten auf bestimmte Fragen**

  Dies ist vom Forschungsthema abhängig und können nicht vorher bestimmt werden. Die Fragen sind aber meist von der Art des Interviewpartners abhängig.

- **Dateien zu den Interviewpartnern**

  Das können Familienfotos, Tonbandaufnahmen, Fotos vom Gehöft oder Skizzen sein. Hier ist die Art der Dokumente je nach Interviewpartner unterschiedlich. Zum Teil lassen sich auch mehrere Dateien gruppieren.

Ein Teil der Daten unterliegt besonderen Regeln des Datenschutzes (z.B. personenbezogene Daten) und dürfen zu bestimmten Zeitpunkten nicht mehr für die Forschung genutzt werden. Diese müssen anonymisiert werden und Daten, welche Personen zurückschließen lassen, entfernt. Die entfernten Daten müssen an einen sicheren Platz für eine spätere Einsicht aufbewahrt werden. Dies betrifft zum Beispiel den Klar-Namen und Kontaktdaten des Betroffenen. Für die Aufnahme und Durchführung des Interviews sind diese Daten noch relevant. Bei der Auswertung der Daten sollen diese nicht mehr verfügbar sein und spätestens bei der Veröffentlichung darf nichts mehr enthalten sein, was eine spätere Identifikation der Person ermöglicht (z.B. “Chefarzt im Krankenhaus der Stadt X, welcher einen Schnurrbart trägt, . . . ”). Die betroffene Person kann später immer die Löschung

der eigenen Daten verlangen. Dafür werden wieder die Kontaktdaten und eine Zuordnung, welche anonymisierte Daten dadurch betroffen sind, gebraucht. Daher ist ein einfaches Löschen der Zuordnungen nicht möglich.

In der Praxis werden Zuordnungstabellen zu den Kontaktinformationen erstellt. Diese werden dann in gedruckter Form oder als digitales Medium in einem sicheren physischen Tresor verwahrt. Für die Forschung steht die Person dann nur noch als anonyme ID zur Verfügung.

2.2 Technische Anforderungen

2.2.1 System

An dem technischen System werden bedingt durch das Nutzungsszenario eine große Vielfalt an Bedingungen gestellt, die dies erfüllen muss:

1. Die Forscher sollen die Software auf ihren Forschungsreisen nutzen, weswegen es vorkommen kann, dass es dabei zweitweise keinen Internetzugriff gibt. Daher müssen alle Daten offline verfügbar sein. Dennoch können online Backups erstellt werden, sobald eine Internetverbindung wieder aufgebaut wird.
2. Es ist von sehr großen Datenmengen auszugehen. Die Forscher werden auf ihren Reisen mehrere Hundert bis Tausend Bildern und Videos aufnehmen. Daher muss das Zielgerät entsprechend Speicherplatz für die Anwendung bereitstellen. Für die externen Geräte (z.B. Kamera) müssen außerdem Anschlüsse vorhanden sein, um Daten übertragen zu können.
3. Des weiteren muss für den Verschlüsselungsprozess auch die benötigte Rechenleistung zur Verfügung stehen.
4. Der Forscher wird mit vielen Daten gleichzeitig arbeiten müssen. Daher ist eine Oberfläche erforderlich, die dies einfach und effizient ermöglicht.
5. Die Geräte müssen relativ kostengünstig sein, da diese auf Forschungsreisen kaputt, verloren oder gestohlen werden können und daher leicht zu ersetzen sein müssen.
6. Das Gerät sollte dem Forscher vertraut sein, damit es den Arbeitsfluss erleichtert.

Aus diesen Anforderungen ergibt sich nach aktuellen Stand ein günstiger Laptop. Ein Mobiltelefon, derzeit weiter Verbreitung gefunden hat, kommt aus folgenden Punkten leider derzeit nicht in Frage:

1. Die 2. Bedingung kann bei vielen Mobiltelefonen nur bedingt erfüllt werden. Es werden große Datenmengen von deutlich mehreren GB (hauptsächlich durch Bilder, Videos) erwartet, welche durch den begrenzten internen Speicher nur schlecht gespeichert werden können. Es gibt zwar Möglichkeiten der SD-Karten Erweiterung, welche aber immer seltener werden. Stattdessen bieten Hersteller immer größeren internen Speicher zu einen schlechterem Preis-Leistungsverhältnis an.
2. Da hier günstige Geräte erwartet werden, kann die benötigte Rechenleistung nur bedingt bereitgestellt werden. Zwar sind einzelne Ver- und Entschlüsselung relativ günstig, dafür aber viel aufwändiger, wen von mehreren Hundert Megabyte bis Gigabyte geredet wird. Dies ist besonders bei der Arbeit mit einer verschlüsselten Datenbank der Fall.
3. Mobiltelefone haben relativ kleine Oberflächen, wodurch die Übersichtlichkeit stark eingeschränkt wird. Außerdem ist die Arbeit mit der virtuellen Tastatur langsamer als mit einer realen. Zwar kann ist es hier möglich externes Zubehör bereitstellen (Maus, Tastatur und Bildschirm über spezielle Hubs), welche aber die Kompaktheit

reduzieren und auch wieder Geld kosten.

Zwar lassen sich die oberen drei Kontrapunkte leicht widerlegen, indem dafür spezielle Systeme und Oberflächen aufgebaut werden, die dafür ausgelegt sind, das erhöht aber nur den Umfang der Arbeit enorm. Dies kann aber eine Möglichkeit der zukünftigen Fortentwicklung sein.

Es gäbe noch die Möglichkeit neben der Bereitstellung auf einem Laptop oder Mobiltelefon dies auch als WebApp bereitzustellen. Dies beinhaltet leider das Problem, dass die Daten auch offline verfügbar sein müssen. Zwar ist es hier möglich den Browser-Cache und -Speicher zu nutzen, um bestimmte Daten zwischenzuspeichern, dieser ist aber leider in seiner maximalen Größe sehr stark begrenzt und es ist nicht möglich die komplette Datenbank dort unterzubekommen. Aufgrund der Komplexität wird dieses Möglichkeit daher derzeit nicht in Betracht gezogen.

Das Max-Planck-Institut für ethnologische Forschung stellt seit Jahren seinen Forschern Windows-Laptops, wenn diese sich auf eine Forschungsreise begeben. Nach dem aktuellen Mobile-Device-Management Plan sollen IPhone-Mobiltelefonie zum Reporteau hinzugefügt werden. Aus oben genannten Gründen wird diese IPhones vorerst nicht berücksichtigt. Das primäre Entwicklungsziel ist daher ein Laptop mit dem Betriebssystem Windows.

2.2.2 Backups

Es ist erforderlich in regelmäßigen Abständen Backups von den Daten erstellen zu können. Dies beinhaltet die komplette Datenbank inklusiver Metadaten, damit bei einen Ausfall, Verlust, etc. diese leicht wiederhergestellt und daran weitergearbeitet werden kann.

Hierzu ist die Nutzung eines Cloudspeicherdienstes geplant, welcher die Daten aus einem lokalen Ordner automatisch mit der Cloud synchronisiert. Eine Herausforderung hierbei ist, dass es zu Problemen mit der Synchronisierung kommen kann, wenn die gleiche Datenbank auf zwei Geräten offen ist und über die gleiche Cloud synchronisiert wird. Hier kann es zu Kollisionen kommen, welche sich nur schwer beheben lassen. Das ist vor allem deshalb der Fall, weil die Daten nur verschlüsselt und binär vorliegen und sich daher eher schlecht vergleichen lassen.

2.3 Wahl der Komponenten

2.3.1 Front-End (UI)

Für die Oberfläche wurde eine Web-Oberfläche gewählt. Dies bietet den Vorteil, dass ohne viel Änderungen an der Codebasis die Oberfläche auf verschiedenen Geräten und Systemen angewandt werden kann. Außerdem modularisiert dies die Codebasis mehr, was die Austauschbarkeit und Wartung verbessert. Zudem lässt sich dadurch leichter für verschiedene Platformen entwickeln (Windows, Linux, Mac, Android, ...).

Im Browser, der die Web-Oberfläche darstellt, gibt es derzeit zwei Technologien, die genutzt werden können, um Code auszuführen: JavaScript und WebAssembly (kurz: WASM). JavaScript ist eine Skriptsprache, welche früher vom Browser interpretiert wurde (derzeit wird sie meistens vor der Ausführung übersetzt) und hat den vollen Umfang der Browser APIs. WASM ist eine neuere Technologie, welche als Byte Code von einer virtuellen Maschine des Browser ausgeführt wird. WASM hat nur Zugang zu den meisten Browser APIs, indem es über eine JavaScript-Schnittstelle kommuniziert.

Tabelle 1: Übersicht Programmiersprachen

Sprache	C#	Java	C/C++	Rust	Java-Script	Elm	Type-Script
Erscheinungs-jahr	2001	1995	1985	2015	1995	2012	2012
Browser	WASM	WASM	WASM	WASM	nativ	JS	JS

Bei Browser wird aufgelistet welche Technologie genutzt wird, um den Code im Browser auszuführen. WASM steht für WebAssembly und JS als JavaScript. Nur JavaScript selbst kann direkt im Browser ausgeführt werden und muss nicht erst in eine andere Sprachen übersetzt werden.

Für die Gestaltung der Web-Oberfläche gibt es eine Vielzahl an Werkzeugen und Systemen (siehe Tabelle 1), die alle ihre Vor- und Nachteile haben und sich mehr oder weniger für diese Projekt eignen. Bei der Vorauswahl wurde mit Absicht nur eine Teilmenge der Möglichkeiten herausgesucht, da es den Rahmen dieser Arbeit sprengen würde, wenn alle berücksichtigt werden. In dem Vergleich wurden in Erster Linie Programmiersprachen herausgesucht mit dem der Author vertraut ist oder sich relativ schnell aneignen kann. In diesem Vergleich kommen C#, Java, C/C++, Rust, JavaScript, TypeScript und Elm zum Einsatz.

C# ist eine objektorientierte Sprache, welche 2001 von Microsoft veröffentlicht wurde. Über die Umgebung Blazor ist es möglich diese als WASM im Browser auszuführen. Dazu gibt es Nuget, einem Paketmanager für Dotnet Bibliotheken (dazu gehört auch C#), welcher es ermöglicht einfach eine Vielzahl bereits vorhandender Bibliotheken einzubinden. Dies erleichtert die Entwicklung deutlich, da so eine Fokussierung auf die individuellen Projektanforderungen möglich ist.

Java ist 1995 von Sun Microsystems veröffentlicht wurden. Früher wurde dies gern für Java Applets im Browser genutzt aber seit der Einführung von HTML 5 wird es mehr und mehr von Browsern nicht mehr unterstützt. Einen großen Einfluss darauf hatte auch, dass dies generell nicht an Mobilgeräten genutzt werden konnte. Mittlerweile lässt sich dies über extra Build-Prozesse in WASM übersetzen und im Browser ausführen.

C/C++ ist die älteste Programmiersprache, welche in dieser Auswahl einbezogen wurde, und wurde 1972 (C) und 1985 (C++) veröffentlicht. Diese Programmiersprache ist sehr hardwarenah und ist quasi das Schweizer Taschenmesser für alle möglichen Zwecke und Umgebungen. Über einen speziellen Compiler lässt sich dies in WASM übersetzen.

Rust ist 2015 von Mozilla veröffentlicht wurden und soll genauso wie C/C++ hardwarenahen Code erlauben aber auch wie C# oder Java sehr abstrakt sein können. Rust arbeitet sehr effizient mit dem Arbeitsspeicher und das ohne Garbage Collection oder Runtime (wie in C# oder Java). In Rust geschriebene Programme sind sehr Speichersicher, da der Compiler viele Fehler entdeckt und prüft oder es durch die Programmiersprache nicht möglich ist diese zu machen. Zu solchen Fehlern zählt auch noch benutzten Speicher freizugeben oder auf die gleiche Speicherstelle von unterschiedlichen Stellen gleichzeitig zuzugreifen. Ein großes Argument für Rust ist also die Nähe am Arbeitsspeicher und die hohe Abstraktion ohne zusätzliche Laufzeit, die zur Verfügung gestellt wird. Zudem kann der Compiler direkt nach WASM übersetzen.

JavaScript (1995) ist eine Script-Sprache, welche für Anwendungen im Browser entwickelt wurde. Diese ist dynamisch typisiert und prototypisch. Dies führt leicht zu schwer erkenn- und wartbaren Fehlern, da oftmals der genaue Typ und die Daten erst bei der Ausführung bekannt ist. Zudem lassen sich die Prototypen zur Laufzeit jederzeit bearbeiten, was zu ungewünschten Seiteneffekten führen kann.

TypeScript wurde 2012 von Microsoft als Erweiterung zu JavaScript herausgebracht, wo versucht wurde ein paar Probleme aus JavaScript anzugehen. Zum Beispiel ist es jetzt möglich Typen zu annotieren, welche von einen Compiler geprüft werden. Es ist aber möglich TypeScript und JavaScript beliebig zu mischen, wodurch es leicht wieder möglich ist die oben genannten Fehler einzubauen oder zu verursachen. Für JavaScript und TypeScript gibt es einige Frameworks, die die Entwicklung vereinfachen sollen, aber hier nicht weiter eingegangen wird.

Elm wurde im gleichen Jahr wie TypeScript veröffentlicht und ist funktional und an die Programmiersprache Haskell angelehnt. Elm wurde für den Browser entworfen, wodurch der Compiler direkt in JavaScript übersetzt. Es gibt derzeit Anstrebungen auch nach WASM zu übersetzen, was aber derzeit noch nicht vollständig ist (siehe [3]). Elm hatte sich auch das Ziel gesetzt die Probleme in JavaScript anzugehen. Zum Beispiel schließt dazu der Elm Compiler Laufzeitfehler generell aus und die Sprache ist typsicher. Weiterhin übersetzt der Compiler den Quellcode in optimierten JavaScript Code, welcher möglichst kompakt und schnell ausgeführt werden kann (siehe [6]).

Diese Technologien sind alle sehr unterschiedlich weit verbreitet, was unter anderem auch daran liegt, wer dahinter steht und diese gepusht hatte. Hinter C# und TypeScript zum Beispiel steht Microsoft und fördert seit langem die Verbreitung. Dies zeigt sich an der Menge an verfügbaren Bibliotheken, Dokumentation und existierenden Projekten.

Hinter Rust stehen primär Mozilla und die Rust Foundation und gelangt in den letzten Jahren an immer mehr Popularität. Es wird vor allem durch die hohe Performance gelobt, hat auf der anderen Seite eine äußerst steile Lernkurve.

Es ist zwar, wie oben gesagt, möglich Java und C/C++ im Browser auszuführen, aber dies ist eher weniger dokumentiert und es gibt vergleichsweise weniger Projekte dazu.

Im Gegensatz dazu steht Elm, was sehr gut dokumentiert ist, eine große Bibliothek hat und recht leicht zu erlernen ist. Die Verbreitung dahinter ist vergleichsweise eher gering einzuschätzen.

Für diese Arbeit wurden folgende Kriterien festgelegt:

1. Es muss im Browser ausführbar sein.
2. Es muss sicher und möglichst ohne Laufzeitfehler funktionieren.
3. Es muss leicht zu debuggen und zu warten sein.
4. Es muss eine gute Dokumentation existieren.

Zumindest Punkt 1 und 4 lässt sich bei allen mit “Ja” beantworten. Punkt 2 lässt sich aus Erfahrungssicht des Autors nur bei Rust und Elm mit “Ja” beantworten, es ist aber möglich bei den anderen Programmiersprachen dies mit mehr oder weniger Aufwand zu erreichen.

Bei Punkt 3 muss hier eine Unterscheidung getroffen werden. Das Problem ist hier WASM, was nur über eine virtuelle Maschine im Browser ausgeführt wird und daher keinen direkten Zugang hat. Hierfür muss erst einmal eine spezielle Umgebung eingerichtet werden, was

je nach Programmiersprache mehr oder weniger aufwändig ist. Klar im Vorteil ist hier JavaScript, da die meisten modernen Browser genügend Werkzeuge mitliefern.

Ein Ausreißer ist hierbei Elm, da es von einer anderen Sprache in JavaScript übersetzt wird, sind die Werkzeuge des Browser teilweise nicht hilfreich (zumindest die, die auf den Code genauer eingehen) oder nutzlos (da das Konzept von Elm diese nicht braucht). Dafür ist Elm stark modularisiert und funktional aufgebaut und bietet für sein Modell-Update-View Konzept genügend eigene Werkzeuge um zu debuggen.

Insgesamt wurde sich hier für Elm entschieden, da es zum einen alle Kriterien erfüllen kann und zum anderen der Entwicklungsprozess damit vergleichsweise leicht und schnell vonstatten gehen kann.

2.3.2 Back-End (Server)

Die Anwendung ist modular aufgebaut mit einer Web-Oberfläche und einen dazugehörigen Server. Dieser kümmert sich um die Verwaltung der Datenbanken, die Verschlüsselung, Verifikation und Backups. Das ist eine vergleichsweise große Palette an Aufgaben. Außerdem muss der serverseitige Teil eine Schnittstelle zur Oberfläche bereitstellen, damit diese auch Daten austauschen können.

Tabelle 2: Übersicht Programmiersprachen

Sprache	C#	Java	C/C++	Rust	Java-Script	Elm	Type-Script
Erscheinungs-jahr	2001	1995	1985	2015	1995	2012	2012
Windows	nativ	nativ	nativ	nativ	Node.js		Node.js
Linux	nativ	nativ	nativ	nativ	Node.js		Node.js
Docker	nativ	nativ	nativ	nativ	Node.js		Node.js

Hier wurde geschaut, wie Programme in der Programmiersprache auf dem Rechner ausgeführt werden kann. Nativ kann ohne externe Werkzeuge und Node.js nur mit dem Programm Node.js.

Hierfür wurden verschiedene Programmiersprachen verglichen (siehe Tabelle 2). Dabei gilt hier, dass die Programme auf dem Computer des Nutzers und nicht im Web-Browser ausgeführt werden soll. Ein Teil der Programmiersprachen kann nativ ohne externe Werkzeuge auf dem Zielrechner ausgeführt werden. C# und Java übersetzen hierbei aber nicht in Maschinensprache, sondern in eine Zwischensprache, die von einer Art virtuellen Maschine ausgeführt wird. Alle anderen Sprachen, die nach JavaScript übersetzen oder es selbst schon sind, können über das Program NodeJS direkt auf dem Rechner ausgeführt werden.

Ansonsten bleibt vieles an den Vergleichen zur Nutzeroberfläche gleich. Mit einer Ausnahme, dass Elm hierzu ungeeignet ist, da diese Programmiersprache nicht dafür designt wurde. Es ist zwar theoretisch möglich Programme in Elm auf dem Rechner ohne Web-Browser ausführen zu lassen, aber dies ist mit enormen Aufwand verbunden, da hier ein Großteil der nötigen Bibliotheken fehlt (z.B. zum Netzwerk oder zum Dateisystem).

Auch hier wurden ein paar Kriterien für die Auswahl definiert:

1. Es muss auf einem Windows- und Linux-Rechner ausführbar sein.

2. Es muss recht schnell und zuverlässig seine Aufgaben ausführen.
3. Es muss sicher und möglichst ohne Laufzeitfehler funktionieren.
4. Es muss leicht zu debuggen und zu warten sein.
5. Es muss eine gute Dokumentation existieren.
6. Es muss ein großer Umfang an Bibliotheken existieren, um die Aufgaben zu bewältigen.
7. Es muss multi-threaded arbeiten, um die gesamte Rechenleistung für die Aufgaben nutzen zu können.
8. Es muss möglichst weit verbreitet sein, damit das Projekt langfristig gewartet werden kann.

Hier treffen fast alle Kriterien auf alle Kandidaten zu. Punkt 3 lässt sich am Besten bei Rust umsetzen. Hier wird der Entwickler durch Sprache und Compiler dazu gezwungen den Speicher sicher und sauber zu halten und es kommt nicht zu Laufzeitfehlern (was nicht heißt, dass das Programm abstürzen "panicken" kann). Dafür ist es aber eher weniger verbreitet (Punkt 8) und hat eine steile Lernkurve und braucht daher einiges an Einarbeitungszeit.

Der Punkt 7 ist in NodeJS zwar möglich, aber umständlicher zu erreichen als bei den anderen Kandidaten. Daher eignen sich hier Programmiersprachen, die nicht darauf angewiesen sind.

Zwar ist es praktisch, wenn der Entwickler ein Programmiersprache hat, die sehr maschinennah arbeitet, um den letzten Funken Geschwindigkeit rauszuholen, aber für dieses Projekt ist es auch in Ordnung, wenn eine andere genutzt wird. Solange die gesamte Reaktionszeit sich im akzeptablen Rahmen hält. Da für diese Arbeit die Entwicklungszeit relativ knapp bemessen ist, wird hier auch eine Programmiersprache bevorzugt, die einfacher und schneller zu schreiben ist.

Unter Berücksichtigung der oben genannten Punkte hat sich der Autor für die Programmiersprache C# entschieden. Diese ist mit über eine Millionen GitHub Repositories sehr weit verbreitet, arbeitet schnell und zuverlässig und kann auch in den restlichen Anforderungen gut punkten. Durch den Compiler und die Interpretationsschicht ist sie gleichzeitig auch soweit von der Maschine abstrahiert, dass es möglich ist schnell Code zu schreiben, ohne auf die zugrunde liegende Maschine genauer eingehen zu müssen.

2.3.3 Datenbank

Die Datenbank ist ein kritisches Thema, da sie alle sensiblen Daten des Forschers enthält und speichern muss. Gleichzeitig reicht es für die Datenbank nicht aus alle Daten einfach zu speichern, sondern diese muss diese Daten auch in angemessener Zeit wieder bereitstellen, damit die Daten weiterverarbeitet werden können. Dies kann eine einfache Anzeige der Daten oder eine Suche nach bestimmten Datensätzen sein.

Daher wurden folgende Kriterien für die Auswahl des Datenbankenmanagementsystems (DBMS) festgelegt:

1. Sicherheit: Die Daten müssen verschlüsselt auf der Festplatte vorliegen. Auch der Schlüssel muss sicher sein.
2. Zugriff: Der Server braucht einen schnellen und einfachen Zugriff auf die Daten. Dies bedeutet, dass die API einfach für den Server und Programmierer zu nutzen und zu verstehen sein muss. Gleichzeitig müssen die Daten in angemessener Zeit verfügbar sein, damit nicht lange Wartezeiten verursacht werden.
3. Backups: Sicherheitskopien müssen sich leicht erstellen lassen.

4. Angriffsoberfläche: Je mehr Funktionen ein DBMS liefert, desto mehr Schwachstellen kann diese haben. Diese müssen alle einzeln abgesichert werden. Das kann erhöhten Konfigurationsaufwand für den Anwender bedeuten. Weniger oder das komplette Deaktivieren nicht benötigter Funktionen kann eine Erleichterung für die Einrichtung und Verwaltung einer Datenbank bedeuten.
5. Lizenzen: Die Lizenz muss mit dieser Arbeit kompatibel sein. Daher ist es schwierig, kommerzielle Lizenzen zu nutzen.

Tabelle 3: Übersicht der DBMS

Kategorie	MariaDB	MySQL	SQLite	MongoDB	LiteDB
Webseite	`mariadb.org` [16]	`mysql.com` [25]	`sqlite.org` [4]	`mongodb.com` [21]	`litedb.org` [9]
Relational	x	x	x		
Art	server-basiert	server-basiert	embedded	server-basiert	embedded
Lizenz	GPL 2.0	GPL 2.0*	public domain	SSPL v1 [23]	MIT
Open Source	GitHub [15]	GitHub [28]	Sqlite.org (Fossil) [5]	GitHub [22]	GitHub [10]
Preis	free	free*	free	free	free
Erster Release	29.10.2009	23.05.1995	17.08.2000	11.02.2009	17.09.2016

MySQL ist nur dann kostenlos und unter GPL 2.0, wenn das gesamte Produkt auch unter der GPL 2.0 veröffentlicht wird. Ansonsten handelt es sich um eine kostenpflichtige Lizenz. Siehe [26].

Für den Vergleich wurden MariaDB, MySQL, SQLite, MongoDB und LiteDB einbezogen (siehe Tabelle 3). Es mag noch mehr Möglichkeiten geben, aber diese fünf sind die Gebräuchlichsten, mit Bibliotheken für C#.

Punkt 1 ließ sich von keinen DBMS außer LiteDB zufriedenstellend erfüllen (siehe Tabelle 4). Zum Teil ist ein enormer Aufbau nötig, Schlüssel liegt unverschlüsselt auf der Festplatte, es müssen unbekannte Patches genutzt werden oder es ist nur möglich die Werte innerhalb der Zellen verschlüsseln. Was auch ein No-Go ist, ist dass Daten unverschlüsselt in Logs stehen können (MariaDB).

Punkt 2 ist nur bei MongoDB unzufriedenstellend. Bei allen anderen reicht es beim Verbindungsaufbau den Schlüssel auszutauschen oder zu authentifizieren und danach läuft alles transparent ab. Bei MongoDB muss dagegen bei jedem Einfügen, Bearbeiten oder Suchen ein Schlüssel übermittelt werden und die Anfrage muss gleichzeitig den Ver- und Entschlüsselungs-Prozess beinhalten.

Backups von einzelnen Datenbanken lassen sich bei Standalone DBMS schwieriger anlegen. Hier sind die Daten z.T. an mehrere Orte verteilt und es gibt aufwändige Prozesse lokale Backups wiederherzustellen. Alternativ ist es möglich alle Daten in ein allgemein lesbares Format (z.B. SQL) zu exportieren und später wieder importieren. Dies dauert aber in der Regel deutlich länger als eine einfache lokale Kopie der Daten. DBMS, die im Prozess der

Tabelle 4: Lokale Verschlüsselung der DBMS

Kategorie	MariaDB	MySQL	SQLite	MongoDB	LiteDB
komplette Datenbank	supported [13]	nur in kostenpflichtiger Enterprise-Version [27]	alternativer Fork [17]	aufwändig und nur Felder [20]	supported [8]
einzelne Tabellen	supported		nur komplette Datenbank	nur Felder	nur komplette Datenbank
Engines	XtraDB, InnoDB, teilw. Aria		alle	alle	alle
Speicherort der Schlüssel	lokale Datei	zentral in einer Schlüsselverwaltung	frei	frei	frei
Methode	AES	AES	AES	AES	AES

MariaDB: Bei aktivierter Verschlüsselung kann es zu Backup-Problemen kommen, da externe Programme die Logs nicht lesen können (mit Ausnahme von MariaDB Backup). Weiterhin ist der Galera gcache in der Community Version unverschlüsselt - in der kostepflichten Version dagegen schon. Die Logs `general query log`, `slow query log`, `aria log` und `error log` sind unverschlüsselt. Es müssen viele Konfigurationen am Server angepasst und ein Verschlüsselungs-Plugin installiert werden. Der Verschlüsselungs-Key muss in irgendeiner Art und Weise als lesbare, unverschlüsselte Datei auf der Festplatte liegen. (siehe [14])

Anwendung laufen, sind dagegen meist so gestrickt, dass 1-2 lokale Dateien mit allen Daten existieren, welche einfach nur kopiert werden müssen.

Tabelle 5: Autorisierung der DBMS

Kategorie	MariaDB	MySQL	SQLite	MongoDB	LiteDB
Nutzerautorisierung	x	x		x	
Mehrere Accounts	x	x		x	
Rollen	x	x		x	
Rechtemanagement	detailliert	detailliert		detailliert	

Ein weiterer Nachteil bei Standalone DBMS ist, dass die naturgemäß eine größere Angriffsoberfläche nach außen bietet, da sie eine Vielzahl von Nutzern Zugriff gewährt. Um dies möglichst gut abzusichern ist eine Konfiguration des Webservers und Einrichten von Accounts, Rollen und Rechten notwendig (siehe Tabelle 5). Die getesteten Embedded-Datenbanken weißen keines dieser besonderen Funktionen auf, da diese in der Regel nur von einer Anwendung genutzt werden und die sich um die Rechtevergabe kümmern muss. In diesem Projekt wird eine lokale Datenbank benötigt auf die nur ein Nutzer gleichzeitig Zugriff hat - und dies ist der Server. Daher reicht eine Datenbank aus, die im Anwendungsprozess läuft.

Vom Kostenfaktor sind alle bis auf MySQL kostenlos und unter Open-Source-Lizenzen verfügbar. MySQL ist nur dann kostenlos, falls alles unter GPL 2.0 veröffentlicht wird.

Unter Berücksichtigung aller Punkte wurde sich für LiteDB entschieden.

2.3.4 Interne Schnittstellen

Zwischen den einzelnen Modulen gibt es interne Schnittstellen, damit diese kommunizieren können. Zwischen Datenbank und Server wird dies über die verwendete Bibliothek geregelt und muss daher nicht weiter berücksichtigt werden. Zwischen FIDO Key und Web-Oberfläche wird hauptsächlich vom Browser übernommen. Übrig bleibt jetzt nur noch die Schnittstelle zwischen Server und Web-Oberfläche. Hierfür gibt es die Bedingung, dass die Schnittstelle über einen herkömmlichen Browser erreichbar, leicht erweiterbar und leicht verständlich sein soll.

Für den Großteil der Kommunikation wird eine WebSocket-Schnittstelle gewählt. Dazu wird ein Nachrichtentunnel zwischen Oberfläche und Server aufgebaut, in denen verschiedene JSON-formartierte Nachrichtenpakete hin und her verschickt werden können. Dieses Protokoll ist nicht Zustandsbasiert und die Kommunikation kann in beide Richtungen jederzeit erfolgen. Weiterhin ist der Tunnel zwischen beiden Teilnehmern sicher. Es kann sich keine dritte Partei (wenn hier der WebBrowser außen vor gelassen wird) einmischen und zur Laufzeit der Verbindung, was meist über die gesamte Dauer der Ausführung der Anwendung hinweg geht, sind beide Teilnehmer immer authentifiziert.

Über diese WebSocket-Schnittstelle werden so gut wie alle Nachrichten übermittelt. Das hat den Vorteil, dass beide Seiten sofort auf Ereignisse reagieren können.

Des weiteren gibt es eine kleine REST-Schnittstelle. Hier werden über verschiedene URLs hauptsächlich Dateien angeboten, da sie meist zu groß sind, um sie über WebSocket zu übertragen. Damit wird versucht die Latenzen über WebSocket möglichst gering zu halten und wichtige Pakete jederzeit den Vorrang geben zu könne.

Ein Problem besteht bei der REST-Schnittstelle, da bei jeden Aufruf eine neue Verbindung aufgebaut wird. Dadurch ist eine Authentifizierung nicht durchgehend möglich. Dafür wird ein kurzlebiger Authentifizierungstoken über die WebSocket-Schnittstelle mitgeteilt, den die Web-Oberfläche nutzen kann, um die REST Anfragen authentifizieren zu können.

3 Sicherheit

Dem Sicherheitsaspekt wurde einer hohen Bedeutung und Wichtigkeit zugewiesen. Dafür gibt es eine Liste an Gesetzen, an die sich zu halten, oder Richtlinien, welche zu folgen sind. Zu den zu betrachtenden und anzuwendende Gesetzen zählt das Bundesdatenschutzgesetz (BDSG) der Bundesrepublik Deutschland und die europäische General Data Protection Regulation (GDPR) der EU. Im Folgenden wird sich der Einfachheit halber auf das BDSG gezogen, da es die deutsche Implementierung der GDPR ist.

Zu den angewandten Richtlinien zählt die der Europäischen Kommission [24], welche zudem auf die Sicherheit und den Schutz der Forschungsdaten eingeht.

3.1 Grundsätze der IT-Sicherheit

Die IT-Sicherheit unterscheidet folgende Ziele: Vertraulichkeit, Integrität und Verfügbarkeit (siehe [12, S. 6–11]). Dazu kommen noch weitere kommen noch weitere Schutzziele, wie Authentizität, Verbindlichkeit, Zurechenbarkeit und Resilienz. Was diese Begriffe bedeuten und deren Vergleich zu den BDSG und Richtlinien der Europäischen Kommission wird in den folgenden Unterkapiteln eingegangen.

3.1.1 Vertraulichkeit

Die Daten selbst dürfen nur von autorisierten Nutzern gelesen und bearbeitet werden. Dies gilt auch für den Zugriff auf gespeicherte Daten oder die Übertragung dieser.

Für eine Autorisierung stehen einem eine große Liste an Möglichkeiten zur Auswahl. Hier ein paar Beispiele:

- Verwendung von Benutzername und Passwort
- Nutzung eines Auth-Tokens (z.B. ID-Karte mit Chip und/oder NFC, USB-Sticks)
- Biometrische Daten wie Gesichtserkennung oder Fingerabdrucksensor
- externe Anbieter und die Schnittstelle LDAP oder OAuth nutzen
- physische Liste mit Einmalpasswörtern (z.B. die TAN Liste, welche früher von Banken genutzt wurde)
- Anmeldecodes per SMS oder Email (wird meist zur Verifizierung als 2. Faktor genutzt)
- Kurzlebige Codes über Apps externer Anbieter (z.B. Google Auth, Microsoft Authenticator)

Es wird empfohlen mindestens zwei dieser Möglichkeiten zu verbinden (Zwei-Faktor-Authentisierung [30]), um einen möglichst guten Schutz erhalten.

Den Zugriff auf die gespeicherten Daten lassen sich mit folgenden Möglichkeiten absichern:

- physisches Gerät mit Daten vor unbefugten Zugriff schützen: z.B. Laptop nicht stehen lassen, Gerät nicht weitergeben
- Datenspeicher vor Zugriff schützen: Dies lässt sich z.B. mit den Rechtesystem des Betriebssystems erreichen.
- Daten vor Zugriff schützen: z.B. den kompletten Datenspeicher verschlüsseln und nur autorisierten Nutzern ermöglichen diesen Bereich zu entschlüsseln

Die sichere Übertragung der Daten geht vergleichsweise einfacher mit einer verschlüsselten Verbindung, auch wenn es hier ein paar Hürden gibt. So soll auf ein etabliertes System (wie

TLS) gesetzt werden [1]. Aber auch hier muss darauf geachtet werden, dass die verwendeten Verschlüsselungsmethoden noch aktuell sind (z.B. MD5 und SHA-1 gelten mittlerweile als veraltet) und die verwendeten Zertifikate noch gelten.

Zertifikate, solange diese von einer vertrauenswürdigen Stelle signiert sind, sind ein probates Mittel um den Schlüsselaustausch und die Authentizität der Gegenseite zu gewährleisten. Hier empfiehlt es sich unter Umständen sogar Zertifikate in der Anwendung mitzuliefern und sich nicht auf die installierten des Betriebssystems zu verlassen, da Nutzer (oder Viren) jederzeit unwissentlich ein kompromittiertes installieren können.

3.1.2 Integrität

> "Integrität bezeichnet die Sicherstellung der Korrektheit (Unversehrtheit) von Daten und der korrekten Funktionsweise von Systemen" [29, S. 34]

Es gibt eine große Vielzahl an Faktoren, welche die Integrität von Daten beeinträchtigen können. Eine große Liste hat das BSI in seinem Grundschutzkompendium im Jahre 2021 aufgelistet (siehe [29, S. 41–89]).

Für dieses Projekt sind folgende Gefahrenquellen als besonders wichtig anerkannt wurden und für diese wurden auch Gegenstrategien erstellt:

"Informationen oder Produkte aus unzuverlässiger Quelle" ([29, S. 62]) können die Integrität stark gefährden indem Daten zum einen unvollständig durch den Forscher oder externe Anwendungen aufgenommen werden. Dies kann z.B. passieren, wenn Bilder beim Upload abgebrochen oder manipuliert werden oder fehlerhafte Imports vorgenommen werden. Gleichzeitig können aber auch Drittprogramme die Schnittstellen der Anwendung fehlerhaft ansprechen.

Um hier den Schaden möglichst gering zu halten, werden alle Anfragen (egal ob vom Nutzer oder anderen Anwendungen oder auch sich selbst) generell nicht vertraut und geprüft. Das bedeutet zwar, dass in der Regel Daten mehrfach geprüft werden, erhöhen dafür aber die Garantie der Korrektheit. Gleichzeitig wird an jeder Schnittstelle davon ausgegangen, dass Daten fehlerhaft oder unberechtigt aufgenommen werden können und dafür gibt es dann entsprechende Fehlermeldungen und Behandlung der Anfragen. Falls Daten nicht vollständig sind, so wird dies dem Nutzer auch mitgeteilt und nur vollständige Datensätze werden bestätigt und weiterverarbeitet.

Ein weiteres Problem ist die "Manipulation von Hard- oder Software" ([29, S. 63]), bei der ein Nutzer oder Programm (Viren, Trojaner, ...) sich Zugang zum Datenspeicher oder der Anwendung verschaffen und manipulieren.

Sämtlich gespeicherte Daten sind permanent verschlüsselt auf der Festplatte und lassen sich auch ohne mehrstufige Anmeldung nicht entschlüsseln. Eine Manipulation der gespeicherten Daten kann aber dazu führen, dass Anmeldungen und somit Entschlüsselung der Datenbank fehlschlagen, da die benötigten Daten entfernt wurden. Auch die Manipulation der Datenbankdateien selbst kann im schlimmsten Fall dazu führen, dass die Datenbankdatei nicht mehr lesbar wird und daher die Daten verloren sind. Gleiches gilt auch für die verschlüsselten Datenbanklogs und Dateien. Gegen dieses Integritätsverlust helfen nur Backups und eine passende Strategie.

[1] Die beliebte Messaging-App Telegram benutzt ein eigenes Verschlüsselungssystem MTProto für die Kommunikation mit ihren Servern. Leider gab es immer wieder wieder Datenlecks die systematisch bedingt durch das Protokoll sind. [1]

Bei einer “Fehlfunktion von Geräten oder Systemen” ([29, S. 68]) kann z.B. das komplette Gerät ausfallen und unzuverlässig arbeiten. Dies kann durch verschiedene Faktoren, wie Alter, Unfälle (wie z.B. Wasserschaden, siehe [29, S. 45]), fehlerhafte Programmierung oder unsachgemäße Benutzung des Nutzers geschehen. Diese haben dann meist zur Folge, dass die Daten fehlerhaft auf die Festplatte geschrieben werden oder unwiderruflich beschädigt oder verloren sind. Hier hilft nur eine gute Backupstrategie.

3.1.3 Verfügbarkeit

Systemausfälle müssen verhindert werden und die Daten sollen nach einen vorher vereinbarten Zeitrahmen wieder verfügbar sein. Systemausfälle lassen sich leider nicht immer vermeiden und die Sorgfalt der Forscher hat einen großen Einfluss darauf, da sich dagegen kaum Vorbereitungen treffen lassen. Wofür sich Vorbereitungen treffen lassen ist die Wiederherstellung der Daten durch Backups. Indem regelmäßig Backups erstellt werden, ist es relativ schnell wieder möglich die Daten darüber wiederherstellen. Es besteht zwar immer noch das Problem, dass beides gleichzeitig ausfallen kann, aber dafür wird die Wahrscheinlichkeit als extrem gering angesehen.

Ein Problem bei der Wiederherstellung durch Backups ist, dass dies nur ein altes Abbild der Daten selbst darstellt. Sämtliche Daten, die danach generiert wurden, sind somit unwiederbringlich verloren. Hier ist Abhilfe nur darüber möglich, indem das Backupzeitfenster relativ kurz wählt, damit der Umfang an verlorenen Daten relativ klein ist.

Für dieses Projekt soll ein Cloudspeicherdienst genutzt werden, welcher für den Nutzer einen ausreichend großen Speicher zur Verfügung stellt.

Als Backupzeitfenster wird 1 Tag empfohlen. Das ist ein guter Kompromiss zwischen zu vielen Backups (Cloudspeicher wird schnell voll) und der Menge an Daten die verloren gehen können. Im schlimmsten Fall verliert der Forscher ein Tag seiner Arbeit.

3.1.4 Authentizität

Die Daten müssen auf Echtheit und Vertrauenswürdigkeit geprüft werden können. Dies erfolgt in erster Linie dadurch, dass sämtliche Daten verschlüsselt auf der Festplatte liegen. Sämtliche Schlüssel lassen sich nur erhalten, indem sich der Nutzer an der Anwendung anmeldet und somit die Daten frei legt. Ist eine Anmeldung nicht möglich, so kann dies an ungültigen Anmeldedaten oder der Authentizität der gespeicherten Daten liegen.

Eine weitere Stelle, wo die Authentizität geprüft wird, ist die Kommunikation zwischen Oberfläche und Hintergrundserver. Die meiste Kommunikation erfolgt über eine WebSocket-Schnittstelle. Da dies einen festen Tunnel darstellt, wird hier die Authentizität beim Aufbau der WebSocket-Verbindung geprüft. Auch hier gilt: Dem Nutzer wird nicht vertraut. Sämtliche Anfragen werden geprüft, ob der Nutzer überhaupt befugt ist die Anfragen zu machen. Die Anmeldung erfolgt über den gleichen Tunnel und stellt somit sicher, dass alles zusammengehört und sich kein Dritter einmischen kann.

Nicht jede Kommunikation zwischen Oberfläche und Hintergrundserver erfolgt über die WebSocket-Verbindung. Für den Zugriff auf einzelne Dateien und Up- oder Downloads gibt es eine REST-API. Hierfür gibt es kurzlebige Tokens, welche direkt mit einer WebSocket-Verbindung verknüpft sind und sich auch nur über diese erhalten lassen. Ohne diese Tokens wird die Anfrage nicht vertraut und die Anfrage wird nicht beantwortet.

Des weiteren werden externe Anfragen von anderen Geräten in der Standardkonfiguration

nicht vertraut. Daher ist der Hintergrundserver so eingestellt, dass nur Anfragen vom gleichen Gerät angenommen werden. Somit wird versucht sicher zu stellen, dass der aktuelle Nutzer möglichst vor dem Gerät sitzt. Dies garantiert einem zwar nicht, dass kein Proxy genutzt wird, dafür reduziert es aber ein potentielles Einfallstor für Angriffe.

3.1.5 Verbindlichkeit

Ein unzulässiges Abstreiten durchgeführter Handlungen ist nicht möglich. Sämtliche Aktionen werden durch den Nutzer induziert und werden auch nur von ihm akzeptiert. Sobald der Nutzer sich angemeldet und eine Datenbank geöffnet hat, ist er somit berechtigt diese auch zu bearbeiten. Jede Aktion wie erstellen, bearbeiten oder löschen von Daten wird direkt durch den Nutzer ausgelöst. Die Anwendung macht nichts ohne den Befehl des Nutzers.

Für kritische Aktionen, wie Löschen von Daten, sind entweder die Menüs so strukturiert, dass diese sehr übersichtlich sind, was gerade getan wird und der Nutzer dies bestätigen muss, oder es gibt Wiederherstellungsfunktionen.

Es gibt aber auch Aktionen, die indirekt durch den Nutzer ausgelöst werden. Dazu zählt in erster Linie die automatische Speicherung der Änderung von Einträgen. Dies verhindert Datenverlust und sorgt für eine bequemere Nutzung.

Eine zweite Aktion wäre die automatische Erstellung und Aktualisierung des Suchindexes. Dies ist notwendig, damit die Suche von Einträgen schnell voran geht und der Nutzer nicht gezwungen ist dies selbst nach jedem Bearbeitungsschritt durchzuführen. Dies geschieht aber nur nach Neuanlegen von neuen Einträgen oder der Speicherung von Bearbeitungen dieser.

3.1.6 Zurechenbarkeit

Eine durchgeführte Handlung lässt sich den Verantwortlichen jederzeit zuordnen. Da die Anwendung nur lokal auf dem Gerät des Forschers betrieben wird und auch jede Aktion auch nur vom dem einen Nutzer ausgehen darf, lässt sich diese Frage jederzeit beantworten: vom Nutzer selbst.

3.1.7 Resilienz

Das System muss Widerstandsfähig gegen Ausspähungen, irrtümliche oder mutwillige Störungen oder absichtlichen Schädigungen (Sabotage).

3.2 Datenschutz

3.2.1 Schutz vor Fremdzugriff

Die Daten befinden sich auf physischen Geräten (Laptop, Festplatte, USB-Stick, . . .) welche mit dem Forscher weltweit mitgenommen werden. Dabei kann es sein, dass die Datenträger in fremde Hände gelangen können. Um hierbei die Daten selbst zu schützen ist es zwingend erforderlich die Daten so zu schützen, dass sie selbst mit vertretbaren Aufwand nicht les- oder manipulierbar sind.

In erster Linie hierzu werden hierzu die gespeicherten Dateien verschlüsselt und sind somit ohne ihren Schlüssel nicht mehr auszulesen. Zusätzlich wird noch ein Hashwert über

die verschlüsselte Datei ermittelt, welcher es ermöglicht Manipulationen an der Datei zu entdecken, aber nicht zu beheben.

Die Namen, Pfade, Schlüssel und Hashwerte der Dateien sind alles Metadaten, welche alle in einer Datenbank hinterlegt werden. Mit Hilfe dieser Informationen erhält die Anwendung (oder auch der Angreifer) Zugriff auf alle Daten, die in den Dateien gespeichert wurden. Ohne diese ist auch gleichzeitig kein Zugriff möglich. Damit die Datenbank selbst wiederum geschützt ist, wird diese verschlüsselt. Hierfür gibt es verschiedene Möglichkeiten (siehe 2.3.3). Damit existiert nur noch eine Stelle, wo ein Schlüssel notwendig ist, welcher alles andere freischaltet.

Zusätzlich zur Verschlüsselung der Dateien und der Datenbank ist es auch möglich die komplette Festplatte (bzw. Partition) inklusive der Daten da drauf zu verschlüsseln. Dies geht unter Linux mit Luks und unter Windows mit BitLocker sehr gut. Das hat den Vorteil, dass ohne den Schlüssel für die Festplatte einen Zugriff auf die Datenbank oder Dateien nicht mehr möglich ist. Sobald die Festplatte aber geöffnet ist (Schlüssel wurde Luks, BitLocker, etc. übermittelt), dann sind alle Dateien darauf allen Prozessen vom Rechner transparent verfügbar, als wären die nicht verschlüsselt gewesen.

Eine Verschlüsselung der Festplatte schützt somit nur vor Fremden, die von außen versuchen auf das Gerät zuzugreifen. Andere Prozesse auf dem gleichen Gerät hätten nach wie vor Zugriff auf alle Daten, wenn diese nicht noch zusätzlich verschlüsselt wäre.

Ein weiteres Problem stellt der verwendete Schlüssel dar, der für die Ent- und Verschlüsselung der Festplatte, der Dateien und der Datenbank genutzt wird. Wird immer wieder der gleiche Schlüssel verwendet, somit es Fremden erleichtert diese einfach auszulesen und in Zukunft erneut einzugeben. Dies wird in der IT auch als Replay-Angriff bezeichnet. Eine Möglichkeit dies zu umgehen stellen Einmal-Passwörter (engl. One-Time-Pad) dar, wo für jede Ent- und Verschlüsselung ein neuer, bisher nicht verwendeter Schlüssel genutzt wird. In der Regel wird versucht, dass aus der Historie bisheriger Schlüssel der nächste Schlüssel nicht abgeleitet werden kann.

Und schließlich stellt sich für die Anwendung die Frage, wann diese den Schlüssel erhält, um die benötigten Daten zu entschlüsseln. Dies könnte direkt beim Start der Anwendung über ein Argument oder Konfigurationsdatei geschehen. Das hätte aber zur Folge, dass für den kompletten Lebenszyklus der Anwendung der Schlüssel bekannt wäre. Bei einer Konfigurationsdatei sogar über diesen hinaus. Alternativ könnte die Anwendung zu einen beliebigen Zeitpunkt den Schlüssel erhalten und diesen auch nur für die einzelne Aufgabe oder Sitzung des Nutzers verwenden. Das hätte den Vorteil, dass der Schlüssel möglichst kurz bekannt ist und die Anwendung dennoch Zugriff auf die Daten erhält, wenn sie gerade benötigt werden.

Eine weitere wichtige Möglichkeit zum Schutz vor Fremdzugriff ist eine 2-Faktor-Authentifizierung. Hiermit fragt mit zusätzlich zu einen Geheimnis, was in diesem Fall der Schlüssel für die Entschlüsselung sein kann, einen weiteren Faktor, z.B. Besitz, ab. Nur wenn der Nutzer den zweiten Faktor bereitstellt und dieser verifiziert werden kann, wird eine Anmeldung als erfolgreich angesehen und dem Nutzer die Daten bereitgestellt. Für den zweiten Faktor gibt es derzeit eine Fülle an Möglichkeiten. Ein paar ausgewählte sind:

- Versand eines kurzlebigen Einmal-Codes per SMS an eine bekannte Nummer. Wird derzeit noch von PayPal, Google und früher für Online-Banking genutzt.
- Vorher generierte Liste an Einmal-Codes, welche ausgedruckt vorliegt. Wurde früher gern für Online-Banking genutzt. Die Universität Halle-Wittenberg nutzt dies noch

heute für ihr Selbstverwaltungsportal.
- Versand eines kurzlebigen Einmal-Codes per Email an eine bekannte Adresse. Wird derzeit von vielen System gern verwendet (u.A. Microsoft, Steam).
- Bereitstellen von kurzlebigen Einmal-Codes per Handy-App. Das bekannteste und verbreitetste hierzu ist der Google Authenticator, wo beliebige Dienstleister (z.B. GitHub, Nintendo, Rockstar, Ubisoft, Discord, NVIDIA, ...) ihre Codes abfragen können. Viele Banken (z.B. Commerzbank, Postbank und Sparkasse) stellen hierzu eigene Apps bereit.
- Nutzung von Hardware-Tokens, die kryptografische Schlüssel austauschen und somit ihre Identität verifizieren. Dies gibt es in verschiedenen Form, unter anderem SIM-Karten (z.B. Autorisierung gegenüber dem Provider), elektronischen Ausweisen (z.B. Personalausweis) und USB-Sticks (z.B. FIDO).

Generell gilt, jeder zweite Faktor ist besser als gar kein zweiter Faktor. Manche brauchen mehr Aufwand in der Bereitstellung, da extra Server und Dienstleister kontaktiert werden müssen und andere können direkt zwischen Nutzer und Anwendung geregelt werden. Außerdem bieten alle auch eine unterschiedliche Sicherheit anderen gegenüber. Bei Emails wird sich auf die Sicherheit des Übertragungsweges, der Server und des Postfachs des Empfängers verlassen. Bei Handy-Apps auf den jeweiligen Dienstleister. Und bei Hardwaretokens vor Manipulation der Hardware (was mit erhöhten Aufwand verbunden ist).

3.2.2 Schutz der Betroffenen

In der Datenbank können Patienteninformationen oder persönliche Meinungen und Weltanschauungen vorhanden sein, welche nicht nachträglich mit dem Interviewten wieder verknüpft werden dürfen. Selbst für den Forscher dürfen diese Zusammenhänge nicht mehr nachträglich (alleinig über die Datenbank) gezogen werden. Dadurch wird auch das Datenbankschema maßgeblich beeinflusst.

TODO: (DSGVO)

- Europäische Datenschutzgrundverordnung DSGVO (GDPR)
- Bundesdatenschutzgesetz BDSG
- Landesdatenschutzgesetze?
- DSG der betroffenen Bürger?

3.3 Technische Basis

Aufgrund von den Grundsätzen der IT Sicherheit (siehe 3.1), und dem Datenschutz (siehe 3.2) lässt sich die technische Basis für die Sicherheit herleiten.

Der Forscher arbeitet mit hoch sensitiven Daten (Patientenakten, religiöse Einstellungen, persönliche Meinung, ...), welche den höchsten Schutz genießen und nicht in fremde Hände geraten oder missbraucht werden dürfen. Dies hat eine höhere Priorität als der Verlust der Daten an sich.

Daher wird die komplette Datenbank und die gespeicherten Dateien durch die Anwendung verschlüsselt und sind auch ohne den Schlüssel nicht entschlüsselbar. Es wird empfohlen zusätzlich die Festplatte zu verschlüsseln. Dies ändert nichts an der Verschlüsselung der Daten selbst, dafür wurde aber eine zusätzliche Hürde vor dem Zugriff von außen eingeführt.

Des weiteren muss sichergestellt werden, dass auch nur der Nutzer (in diesem Fall der Forscher) Zugriff auf seine Daten hat und auch sonst niemand anderes. Dafür übergibt der

Nutzer beim Start der Sitzung den Schlüssel an die Anwendung, welcher direkt wieder weggeschmissen wird, sobald der Nutzer die Sitzung beendet. Zusätzlich muss der Nutzer vor dem Start der Sitzung einen zweiten Faktor für die Autorisierung bereitstellen. Hier wird ein Hardwaretoken verwendet. Das hat den Vorteil, dass sich nicht auf externe Dienstleister oder Infrastruktur verlassen werden muss, da alles lokal am Rechner geprüft wird.

Die Daten aus der Datenbank (und den gespeicherten Dateien) werden nur dann entschlüsselt, wenn dies auch vom Nutzer gewünscht oder indirekt gefordert wird. Diese sind dann nur für den benötigten Zweck im Arbeitspeicher bereitgehalten und werden direkt im Anschluss wieder gelöscht.

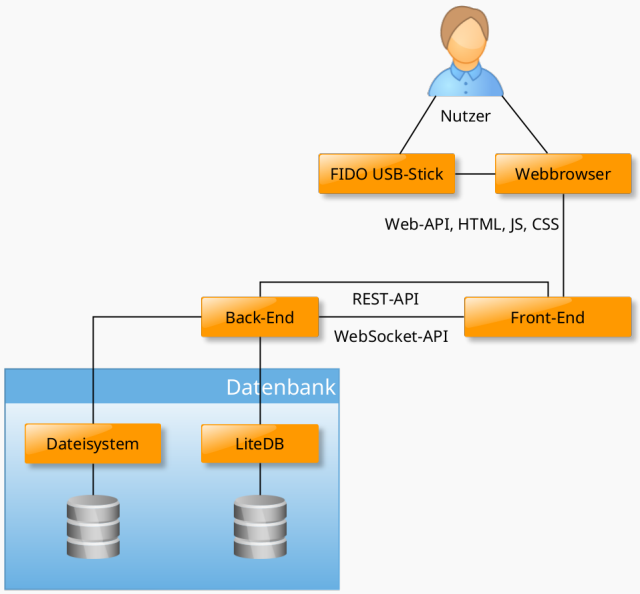
Daten können mit einen Forschungskollegen nur dann freigegeben werden, wenn dies explizit vom Forscher gewünscht wird. Ohne das ist das Einlesen der Daten nicht möglich. Insbesondere der Hardwaretoken verhindert den Zugriff über die gleichen Zugangsdaten von unterschiedlichen Standorten aus.

Damit der Zugriff zur Datenbank durch Verlust oder Beschädigung des Hardwaretokens oder das Vergessen des Passworts nicht verloren geht, wird im Vorfeld ein Masterpasswort für die Datenbank erzeugt. Dies allein reicht aus, um den Zugriff wiederherzustellen.

4 Technisches System

Hier wird ein Überblick (siehe Abbildung 1) über das technische System geliefert, dazu zählen die einzelnen Komponenten, die Apis, die Datenbank und wie ein Anmeldeprozess aussieht.

Abbildung 1: Überblick über das technische System



Diese Grafik zeigt einen Überblick über das technische System. Dazu sind alle Module und deren Schnittstellen gekennzeichnet. Die Module sind Back-End (siehe Kapitel 4.1.2), Front-End (siehe Kapitel 4.1.1) und Datenbank (siehe Kapitel 4.4). Dazu wurden die internen API von WebSocket und REST in Kapitel 4.1.4 und den Anmeldeprozess über FIDO in Kapitel 4.3 behandelt.

4.1 Überblick

Die gesamte Anwendung ist in mehrere Module geteilt, welche für sich abgeschlossen und auch austauschbar sind. Sie sprechen über eine einfache API miteinander und lassen sich separat voneinander testen.

Hier wird ein lokaler HTTP Webserver genutzt, welcher nur Anfragen von localhost entgegen nimmt, verarbeitet und die Antworten zurückliefert. Der Nutzer kann dann eine beliebige Anwendung (in den meisten Fällen ein moderner Webbrowser wie Firefox oder Chrome) nutzen und diesen Server ansprechen. In den folgenden Fällen wird vom diesen Modul als Back-End oder auch Server gesprochen.

Hinter dem Webserver wird eine lokale verschlüsselte Datenbank genutzt, wo die komplette Verwaltung im Prozess des Webservers eingebettet ist. Dazu wird eine Open Source Bibliothek genutzt, die sich um die komplette Verwaltung dazu kümmert.

Des weiteren gibt es das Front-End welches aus einer Webseite besteht, welche von einen beliebigen modernen Webbrowser dargestellt werden kann. Mit dieser Oberfläche wird der Nutzer hauptsächlich kommunizieren und von den Vorgängen im Hintergrund sollte dieser

eigentlich nichts mitbekommen. Im folgenden wird vom Front-End auch von der Oberfläche oder auch UI gesprochen.

4.1.1 Front-End (UI)

Für das Front-End wurde eine moderne HTML Oberfläche gewählt. Diese hat den Vorteil, dass sich diese auch später ohne großen Aufwand auf andere Plattformen oder Systeme übertragen lässt. (Webbrowser sind fast überall verfügbar.) Außerdem hat sich das Web mittlerweile so weit entwickelt, dass viele Office Tätigkeiten sich auch heute schon komplett im Browser erledigen lassen (z.B. Dokumente schreiben, Emails lesen, einfache Videobearbeitung, ...) und auf Spezialanwendungen verzichtet.

Als Programmiersprache für die UI hat der Autor die Sprache Elm gewählt. Besonders folgende Aspekte haben diese Sprache gegenüber Konkurrenten wie JavaScript oder TypeScript durchgesetzt:

- Es gibt ein statisches Typsystem wo jederzeit feststeht welche Daten welchen Typ haben. Es existiert kein Casten oder untypisierte Objekte. Es können leicht Veränderungen am Datenmodell vorgenommen werden und der Compiler hilft den Entwickler dies im gesamten Programm zu berücksichtigen.
- Die Programmiersprache ist funktional und hat keine Seiteneffekte. Dadurch lässt sich der Code leicht in kleinere, leicht verwaltbare Komponenten aufteilen.
- Es existieren keine Laufzeitfehler. Der Compiler zwingt den Entwickler alles schon zur Entwicklungszeit zu berücksichtigen. Dies erleichtert das Debuggen und Beheben von Problemen enorm. Außerdem wird somit auch eine große Fehlerklasse, wie die Null-Fehler, komplett ausgeschlossen.
- Im Vergleich zu Vue oder React sind die resultierenden Builds besonders klein und schnell. Der Compiler entfernt früh nicht verwendeten Code und baut bestehenden so um, dass dieser effizient wird.

Das Styling wird durch handgeschriebenes CSS gemacht. Hier wird kein Framework genutzt.

4.1.2 Back-End (Server)

Der Server ist eine kleine C# Anwendung, welche sich um alles Wichtige im Hintergrund kümmert. Sie nimmt alle Anfragen von der Oberfläche entgegen und informiert diese über Änderungen. Dann kümmert es sich um die Verwaltung der Datenbanken und der verschlüsselten Dateien. Hier ist der komplette Sicherheitsaspekt gelagert.

4.1.3 Datenbank

Auf die Datenbank wird im Kapitel 4.4 genauer eingegangen.

4.1.4 Interne APIs

All diese Module werden über verschiedene Apis zusammengehalten und darüber wird auch kommuniziert.

Der Server und die UI kommunizieren hauptsächlich über eine einzelne WebSocket-Verbindung (siehe RFC 6455 **??**). Das hat den Vorteil, dass die Authentifizierung nur einmal am Anfang erledigt werden muss und danach kann sich gegenseitig vertraut werden, solange die Verbindung nicht abbricht (z.B. wenn der Nutzer die Seite im Browser neulädt).

Außerdem können jederzeit Nachrichten vom Server zur UI und auch anders herum sendet werden, und so schneller auf neue Ereignisse reagieren.

Für Datei-Up- und Downloads wird zusätzlich eine kleine REST-API genutzt, damit zum einen hierfür die Verbindungskapazität der WebSocket-Verbindung nicht ausgelastet wird und zum anderen die Einbettung in die Oberfläche einfacher geschieht.

Der Server und die Datenbank kommunizieren über eine Open-Source-Bibliothek, welche im Prozess des Servers angesiedelt ist. Über die API der Bibliothek wird dann die Datenbank verwaltet. Es findet keine Inter-Process-Kommunikation statt - alle Daten sind sofort beim Server verfügbar und können nicht mit einfachen Mitteln ausgespäht werden.

4.2 Externe Apis

Um den Zugang der Daten zu anderen Anwendungen zu ermöglichen soll die Anwendung Schnittstellen bereitstellen. Die einfachste Form hierzu ist der Export der gesicherten Dokumente, welche im sicheren Speicher hinterlegt wurden. Nachdem diese exportiert (heruntergeladen) wurden, können diese in einem anderen Programm betrachtet werden.

Genauso funktioniert der Prozess für den Import von Daten anderen Programme in die Anwendung hinein. Hier übernimmt die Anwendung nur die Aufgabe eines sicheren Speichers, in dessen die Daten vor Manipulation von außen sicher sind.

Für die Daten in den Eingabefeldern gibt es bisher nur die Möglichkeit die aktuelle Seite als PDF über den Browser zu exportieren. Mehr Möglichkeiten sind hierzu als Erweiterungen (siehe 8.2) geplant.

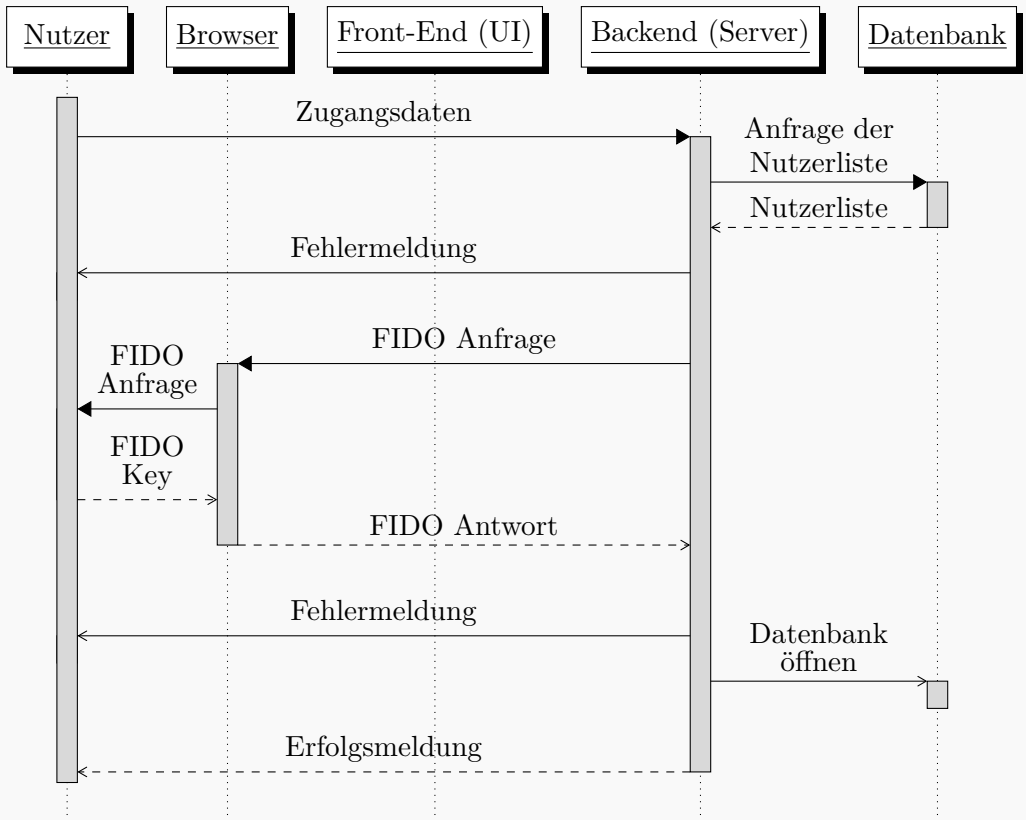
4.3 Anmeldung

Die zu speichernden Daten haben einen sehr hohen Schutzbedarf (#Belege) und müssen dementsprechend verschlüsselt gespeichert und übertragen werden und brauchen eine Zugriffskontrolle. Das BSI verlangt hierfür eine Zwei-Faktor-Authentifizierung.

Für die Anwendung wurde eine Zwei-Faktor-Authentifizierung ausgewählt, welche auf Wissen (Passwort) und Besitz (FIDO-Key) basiert. Die Authentisierung erfolgt in folgenden Schritten (vergleiche Abbildung 2):

1. Der Nutzer öffnet die Oberfläche in seinem Browser und wird nach Benutzername und Passwort gefragt.
2. Der Server prüft, ob eine Datenbank diesen Nutzer hinterlegt hat. Wenn nein gibt es eine Fehlermeldung und die Authentifizierung wird abgebrochen.
3. Dann wird geprüft, ob der Wert von `SHA256(Passwort + Salt)` mit dem gespeicherten Wert übereinstimmt. Der Salt ist ein zufälliger Wert, welcher bei der Erstellung der Datenbank angelegt wurde. Falls es hier ein Fehler gab, wird dies angezeigt.
4. Die Oberfläche fragt über die WebAuthn Schnittstelle des Browser (ist in jeden modernen Browser implementiert) nach dem FIDO Key. Das ist ein kleiner spezieller USB Stick, welcher eingesteckt werden muss.
5. Der FIDO Key bekommt den Wert von `SHA256(Passwort)` und soll diesen mit seinen lokalen privaten Key signieren. Die Signatur ist immer gleich, wenn die Eingabe gleich ist.
6. Der Browser lieft die Signatur an die Oberfläche und diese an den Server.
7. Es wird geprüft, ob `SHA256(Signatur)` mit den gespeicherten Wert übereinstimmt. Wenn nicht gibt es wieder eine Fehlermeldung und ein anderer FIDO-Key wird

Abbildung 2: Der Anmeldefluss mit Nutzername, Passwort und FIDO.



Der Übersichtlichkeit halber wurden einige Vermittlungsphasen rausgelassen. Die einzelnen Instanzen können nur mit den jeweiligen Nachbar direkt kommunizieren.

verlangt.

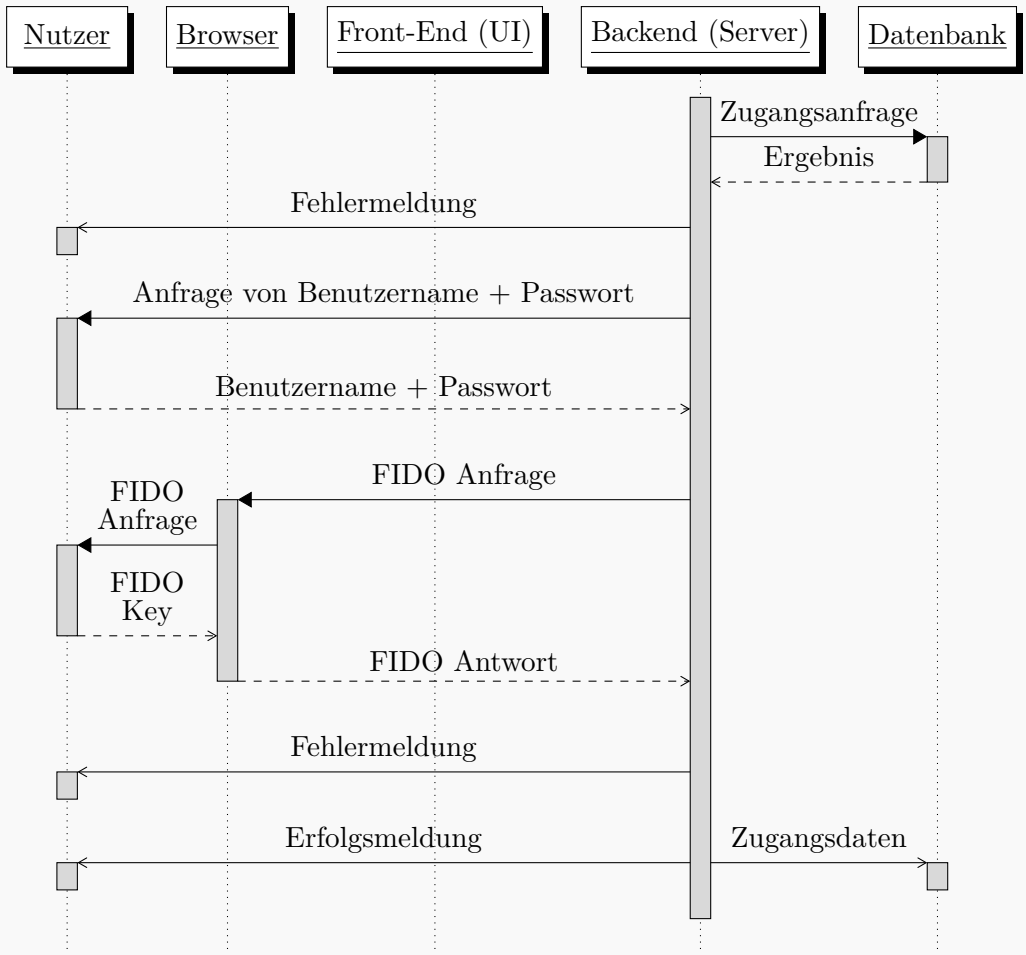
8. Die Datenbank wird geöffnet.

Es können in einer Anwendung mehrere Datenbanken hinterlegt werden, wo bei jeder Datenbank der Nutzer ein anderes Passwort oder FIDO-Key nutzen kann. Am Anfang wird mit jeder Datenbank verglichen und nach und nach werden die noch gültigen Datenbank ausgesiebt, welche noch damit angemeldet werden können. Sobald zu einen Zeitpunkt keine Datenbank mehr möglich ist, so wird dies als Fehler angegeben. Am Ende können mehrere Datenbanken gleichzeitig authentifiziert werden (sofern Nutzername, Passwort und FIDO bei allen gleich sind). Datenbanken, welche gerade nicht geöffnet werden konnten, können auch noch nachträglich geöffnet werden. Dazu wird der Anmeldeprozess wiederholt und schon offene Datenbanken ignoriert.

Das Neuanlegen einer Anmeldemöglichkeit (z.B. bei Neuerzeugung einer Datenbank oder Hinzufügen eines weiteren Nutzers) werden folgende Schritte abgehandelt (vergleiche Abbildung 3):

1. Zuerst wird geprüft, ob Zugang überhaupt besteht. Bei neuen Datenbanken ist dies implizit. Bei bestehenden muss der Nutzer sich erneut anmelden, da der Server den Entschlüsselungs-Key nicht permanent im RAM hält.
2. Der Nutzer muss eine Benutzernamen und Passwort angeben und bestätigen.
3. Nutzer wird von der Oberfläche nach einen FIDO-Key gefragt.

Abbildung 3: Erstellung neuer Datenbankzugangsdaten



Der Übersichtlichkeit halber wurden einige Vermittlungsphasen rausgelassen. Die einzelnen Instanzen können nur mit den jeweiligen Nachbar direkt kommunizieren.

4. Es wird der Public-Key und eine Prüfung abgefragt.
5. Der Server prüft das. Bei Misserfolg wird dies angezeigt.
6. Die Prüfsummen für die Anmeldung werden erzeugt und bei der Datenbank parallel hinterlegt.
7. Datenbank wird verbunden, sofern noch nicht geschehen.

Des weiteren wird bei der Anmeldung nicht der eigentliche Key für die Datenbank erzeugt. Stattdessen wird für jede Anmeldemöglichkeit der eigentliche Key für die Datenbank separat verschlüsselt und ist nur mit der Signatur aus der Anmeldung entschlüsselbar. Dadurch ist es auch relativ einfach möglich mehreren Nutzern Zugang zur gleichen Datenbank zu geben, ihnen zu erlauben ihre Passwörter zu ändern oder den Zugang wiederherzustellen, falls der mal verloren gegangen ist.

4.4 Datenbank

Die Datenbank umfasst verschiedene Arten von Daten. Darunter zählen die Einstellungen, Zugangsberechtigungen, die Dateien, die eingegebenen Daten und die Metadaten des

Forschers. Dadurch ist diese auch relativ groß und komplex. In den folgenden Unterkapiteln wird genauer auf das DBMS LiteDB, die Herleitung des Schemas, die Speicherung von Daten und die Arbeit mit diesen eingegangen.

4.4.1 LiteDB

LiteDB ist eine OpenSource ([10]) NoSQL Datenbank, welche als eingebettete Datenbank für eine Dotnet (u.a. auch C#) Anwendung fungiert. Diese ist selbst komplett in C# geschrieben ([9]) und bietet eine große Palette an Funktionen und Anbindungen für Dotnet.

LiteDB speichert seine Daten in einem BSON (Binary JSON) Format. Dieses ist an JSON angelehnt und soll die Verarbeitung erleichtern. Ein paar Beispiele sind in Abbildung 4 zu finden.

Abbildung 4: BSON Beispiel

{"hello": "world"}	→	\x16\x00\x00\x00 // total document size \x02 // 0x02 = type String hello\x00 // field name \x06\x00\x00\x00world\x00 // field value \x00 // 0x00 = type EOO ('end of object')
{"BSON": ["awesome", 5.05, 1986]}	→	\x31\x00\x00\x00 \x04BSON\x00 \x26\x00\x00\x00 \x02\x30\x00\x08\x00\x00\x00awesome\x00 \x01\x31\x00\x33\x33\x33\x33\x33\x33\x14\x40 \x10\x32\x00\xc2\x07\x00\x00 \x00 \x00

Diese Beispiele sind aus der BSON FAQ ([18]) und sollen zeigen wie das BSON Format binär aussieht.

LiteDB nutzt ein Subset der vom offiziellen BSON Schema [19] unterstützten Datentypen (siehe [7]). Das ist hauptsächlich aus dem Grund, damit sich die Arbeit mit den Erweiterungen, die BSON unterstützt, erspart wird.

LiteDB erlaubt die Umwandlung von BSON in JSON und anders herum. Zudem hat es Funktionen zur Serialisierung und Deserialisierung eingebaut, die BSON Daten in C# Objekte und anders herum überführen. Dies erleichtert dem Entwickler die Arbeit mit dem Daten enorm.

LiteDB verwaltet die Daten in mehreren Collections. Diese sind Ansammlungen von BSON Daten, die iteriert werden können. Um die Suche von bestimmten Einträgen zu erleichtern, gibt es zu jeder Collection eine oder mehrere Indexe, welche den exakten Wert eines Pfades innerhalb der jeweiligen BSON Objekte abbilden. Standardmäßig wird ein Index für `$.Id` (den Wert von `Id` im Wurzelknoten) angelegt. Es gibt neben einfachen Collections auch Collections für Dateien (es ist möglich größere binäre Daten in LiteDB zu hinterlegen) oder Systemcollections für die interne Verwaltung.

Um eine Datenbank in LiteDB zu öffnen ist folgender Befehl notwendig:

```
// Öffnet oder erstellt eine Datenbank mit zusätzlichen Einstellungen
Database = new LiteDatabase(new ConnectionString()
{
```

```
    Filename = "der Pfad zu der Datenbankdatei",
    // das Passwort, falls eins gesetzt wurde. Andernfalls einfach null.
    Password = "pa$$w0rd",
});
```

Um Daten auszulesen wird dann eine Collection von der Datenbank abgefragt. Dies geht allgemein ohne C# Typinformationen und der Entwickler erhält die BSON Rohdaten. Für die Entwicklung ist es aber praktischer dies direkt in Objekte überführt zu bekommen. Hier wird ein einfaches Beispiel genutzt. Ausführliches dazu gibt es im Quellcode der Arbeit in `server/Dacryptero/DB/DBSingleController.cs`.

```
// Beispielklasse für die Daten
public class Person
{
    // Id der Person. ObjectId wird von LiteDB bereitgestellt und ist
    // eine 12 Byte große Zahl.
    public ObjectId Id { get; set; }

    public string Name { get; set; }
}

// Abrufen der Collection der Datenbank. Dazu wird der Typ und der
// Name der Collection übergeben. Falls die Collection noch nicht
// existiert, so wird diese angelegt.
var people = Database.GetCollection<Person>("people");

// Hinzufügen der Person
var person = new Person { Name = "Max Mustermann" };
people.Insert(person); // Die Id wird automatisch von LiteDB gesetzt.

// Finden aller Personen dessen Name mit Max anfängt.
var peopleWithMax = people.Find(x => x.Name.StartsWith("Max"));

// Finden der ersten Person anhand der Id
var person2 = people.FindById(person.Id);
```

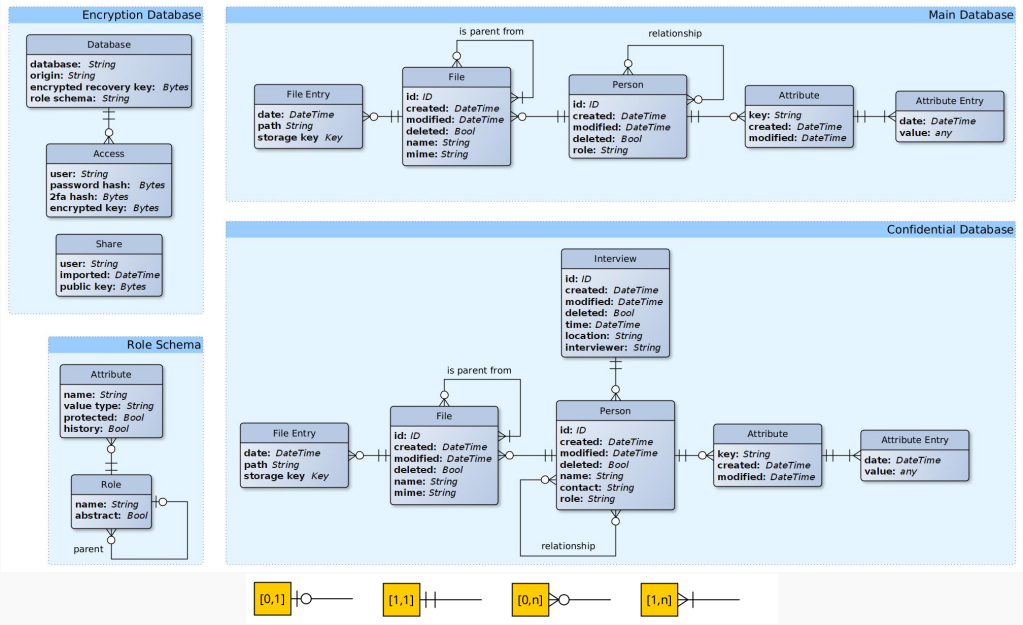
Wie an den Beispielen sehr leicht erkenntlich ist, ist die Nutzung der Datenbank recht leicht und die Abfragen können direkt im Nutzercode gemacht werden. Das Zusammenbauen einer Abfrage in textueller Form, wie SQL, ist nicht notwendig. Für komplexere Aufgaben gibt es weitere Funktionen, die in der Dokumentation [11] nachzulesen sind.

4.4.2 Datenbankschema

Das Erstellen des Datenbankschemas stellte sich als schwierig heraus. Das größte Problem war, dass zuerst unklar war, was die Auftraggeber erwarten und welche Daten diese benötigen. Dabei entstanden verschiedene Iterationen des Datenbankschemas. Im späteren Verlauf haben sich Autor und Auftraggeber darauf geeinigt, das Datenbankschema allgemeingültig für verschiedene Forschungszwecke zu machen und dann über ein separates Schema dies zu konfigurieren und einzuschränken (siehe 4.4.3).

Was sofort am Schema auffällt ist die Aufteilung in verschiedene Bereiche. Es gibt hier die Main Database, die Confidential Database, die Encryption Database (siehe Kapitel

Abbildung 5: Entity Relationship Diagram



Diese Abbildung gibt es noch einmal im Anhang D.1 als große Grafik.

Die verschiedene Bereiche wurden mit unterschiedlichen Hintergrundfarben hervorgehoben. Zwischen den einzelnen Objekten gibt es verschiedene Beziehungen. Die Kardinalität ist auf der jeweiligen Seite des Pfeils durch Krähenfüße oder Striche angegeben. Dabei gibt es für jede Seite vier Fälle, die unten in der Legende noch einmal aufgelistet wurden.

4.4.4) und das Role Schema (siehe Kapitel 4.4.3). Die Aufteilung von Main Database und Confidential Database ergibt sich aus den Anforderungen siehe (**TODO** Kapitelnummer). Beide enthalten zusammen die Datensätze, die der Forscher im Laufe der Forschung eingibt. Diese Datensätze sind je nach Priorität entweder in der Confidential Database oder in der Main Database. Die Anwendung braucht auf jeden Fall die Main Database, um mit den Daten zu arbeiten. Die Confidential Database kann zu einen beliebigen Zeitpunkt entfernt oder wieder hinzugefügt werden.

Zentrale Objekte in der Datenbank ist die `Person`, `Interview`, `File` und `Attribute`, welche alle in Beziehung zueinander stehen. Alle vier Objekttypen finden sich in beiden Datenbanken wieder. Einzig `Interview` ist nur in der Confidential Datenbank enthalten. Von `Person` gibt es immer zwei Varianten, welche jeweils in der Main Database und in der Confidential Database enthalten sind. Beide Varianten sind über die gleiche Id verbunden.

`File` und `Attribute` sind als Objekte nur in einen von beiden Datenbanken enthalten, je nachdem ob sie derzeit confidential deklariert wurden oder nicht. Jeweils von `File` und von `Attribute` wird eine Historie angelegt, welche sich auch im Datenbankschema wiederspiegelt. Somit ist der Bearbeitungsverlauf der Daten immer einsichtlich.

Das Verhalten und die Nutzung von `Attribute` ist in Kapitel 4.4.3 näher erläutert.

Die Objekte von `File` enthalten nur die Metainformationen zu den Dateien. Die Inhalte selbst werden parallel zu den Datenbankdateien verschlüsselt gespeichert. Der Schlüssel für

die jede Datei ist einzigartig und für jede neu generiert. Diese werden alle in die Datenbank mit gespeichert.

4.4.3 Schemadateien

Schemadateien bieten den Nutzer das allgemeine Datenbankschema weiter einzuschränken und für seine Bedürfnisse anzupassen. In der Schemadatei wird angegeben, welche Rollen existieren, welche Attribute diese jeweils haben und wie diese verwaltet oder bearbeitet werden sollen.

Alle Schemadateien sind JSON Dateien und befinden sich in einem vom Nutzer konfigurierten Ordner. In diesem können sie in einem beliebigen Unterordner befinden. Alle JSON Dateien werden von der Anwendung als Schemadateien betrachtet. Bei der Namensgebung der Pfade haben das Institut und der Nutzer freie Wahl. Einzig `/internal/**/*.json` wird für interne Testzwecke für die Zukunft reserviert.

Der Aufbau der Schemadatei wird im Anhang unter D.4 genauer erklärt. Jede Schemadatei enthält eine Liste von Rollen mit Attributen, die von oben nach unten ausgewertet werden. Jede Person, welche vom Forscher in der Anwendung angelegt wird, entspricht genau eine dieser Rollen, welche derzeit auch nicht nachträglich geändert werden kann.

Falls mehrere Rollen gleiche Attribute enthalten, kann eine neue Rolle (Eltern-Rolle) angelegt werden, welche die gleichen Attribute enthält, und alle Rollen erben dann von der neuen Rolle. Dadurch übernehmen alle Kinder-Rollen (das sind die Rollen, die von der Eltern-Rolle erben) alle Attribute der Eltern-Rolle. Jede Rolle kann nur von einer anderen Rolle direkt erben, aber eine Rolle kann von einer Rolle erben, die schon von einer anderen Rolle erbt.

Eine Rolle kann nur von einer anderen Rolle erben, die im Schema vorher definiert wurde. Dadurch wird das Problem mit der zyklischen Vererbung umgangen. Falls die Eltern-Rolle nicht von Forscher direkt genutzt werden soll, dann empfiehlt es sich diese Eltern-Rolle als abstrakt zu markieren.

Das Rollensystem wurde direkt von verschiedenen Programmiersprachen wie C#, Java oder C++ abgeguckt und dementsprechend umgesetzt.

Jede Rolle hat eine Liste von Attributen. Ein Attribut ist eine Eigenschaft, die der Nutzer bei der Person einstellen kann. Diese bekommt exakt den Namen, der im Schema als Schlüssel definiert wurde. Als Typ lässt sich hier die Art des Editors bestimmen, welche wiederum auch den Wertetyp definiert (siehe Tabelle 6). Eine genauere Einschränkung für den Editor gibt es derzeit nicht.

Weiterhin lässt sich zu jedem Attribut sagen, ob dies protected ist, also in der Confidential Datenbank gespeichert wird. Diese Einstellung ist durch das Schema vorgegeben und lässt sich nicht im Nachhinein ändern. Sobald die Confidential Datenbank entfernt wurde, kann der Wert nicht mehr eingesehen werden, aber auch eine Bearbeitung ist nicht mehr möglich.

Außerdem lässt sich für jeden Attribut festlegen, ob es eine Historie anlegen soll. Falls diese Option aktiviert ist, wird jeder historische Zustand in der Datenbank abgespeichert und lässt sich durch den Nutzer ansehen. Ist die Option deaktiviert, so ist immer nur der aktuelle Wert zu sehen und eine Bearbeitung ist nicht möglich.

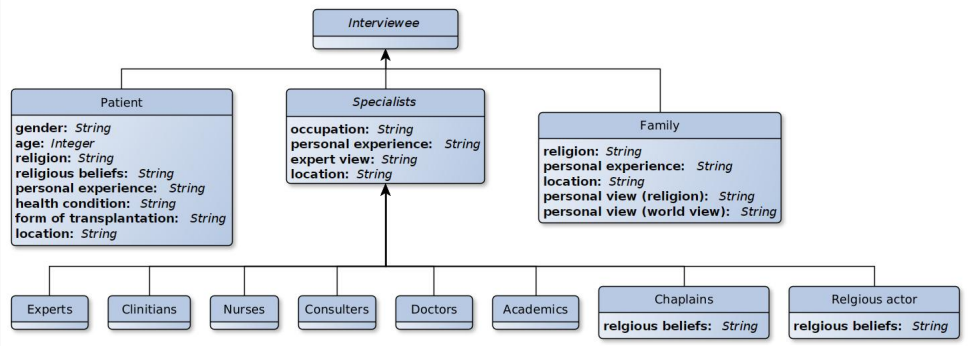
In Abbildung 6 wird das Rollenschema, welches für das Forschungsthema des Auftraggebers erstellt wurde, dargestellt. Hier ist deutlich zu erkennen, dass je nach Rolle unterschiedliche

Tabelle 6: Editortypen

Editortyp	Wertetyp	Beschreibung
`DateTime`	`string`, `<date-time>`	Angabe von Datum **und** Zeit. Angezeigt wird in lokaler Zeitzone des Anwenders. Gespeichert in UTC.
`SingleLine`	`string`	einzeiliger Text
`MultiLine`	`string`	mehrzeiliger Text
`Number`	`number`	Fließkommazahl
`Int`	`int`	Ganzzahl
`Bool`	`bool`	Checkbox mit Ja, Nein

Alle Editortypen für Attribute, die von der Oberfläche unterstützt werden.

Abbildung 6: Rollenschema für das Projekt



Dies ist eine grafische Visualisierung wie das Rollenschema für das Forschungsthema des Auftraggebers aussieht. In kursiv sind die Namen von abstrakten Rollen gekennzeichnet. Die Pfeile geben an, von welcher Rolle diese ihre Attribute erben. Bei den Attributen wurde nur der Wertetyp angegeben.

Informationen abgefragt und gespeichert werden.

4.4.4 Schlüsselverwaltung

Die Schlüssel für die Entschlüsselung der Datenbankdatei befindet sich im Datenbankordner in der `config.json` Datei. Hier sind alle Informationen enthalten, die benötigt werden, um Nutzer zu autorisieren und authentifizieren. Die Datei nach dem Schema im Kapitel D.3 (Anhang) aufgebaut.

Die Datei enthält einen Hash vom MasterKey, damit immer nachgeschaut werden kann, ob der aktuell eingegebene MasterKey der richtige sein könnte, bevor dieser direkt an der Datenbank angewandt wird.

Dann ist da eine Liste an Nutzerzugängen aufgelistet. Jeder Nutzer kann beliebig viele Zugänge hinterlegt haben. Diese können sich in Benutzername, Passwort oder Fido gleichen oder unterscheiden. Das macht für die Anwendung keinen Unterschied.

Zu jedem Passwort wird der gesalzene Hash hinterlegt. Gesalzene Hashes werden generiert, indem zu dem Passwort ein zufällig generierter Hash angehangen wird und dann zusammen

gehasht wird. In dieser Anwendung wird der Prozess 10.000 mal wiederholt. Der Prozess des gesalzenen Hashens hat den Vorteil, dass es deutlich erschwert wird mit Hilfe von den Hashes die Passwörter zu knacken (siehe [2]).

Zusätzlich dazu werden Informationen zum genutzten Fido-Key hinterlegt. Das Format dieser Werte ist durch die verwendete Fido-Bibliothek vorbestimmt und wird durch die Anwendung nur gespeichert und weitergereicht. Diese Daten werden genutzt, um den FIDO USB Stick zu authentifizieren und zu validieren.

Zum Schluss gibt es noch das verschlüsselte Datenbankpasswort, welches durch das Nutzerpasswort und einem Salt entschlüsselt werden kann. Das entschlüsselte Datenbankpasswort entspricht dem MasterKey, wird aber im späteren Verlauf durch die Anwendung nicht mehr bereitgestellt oder angezeigt.

Es ist mithilfe des Datenbankpassworts möglich neue Zugänge in die JSON Datei einzutragen. Existierende Einträge können jederzeit durch den Nutzer entfernt werden. Es ist technisch nicht nötig das Passwort hierzu zu wissen, da die JSON Datei einfach bearbeitet werden kann. Für die Anwendung wäre es dann so, als ob es die Zugangsdaten nie gegeben hätte.

Wenn die Datei vom Nutzer in anderer Art verändert wird (kein Hinzufügen von validen Einträgen oder Löschen dieser), dann ist es möglich, dass der Zugang zur Datenbank verloren geht. Die Anwendung kann damit umgehen, wenn an einer beliebigen Stelle die Authentifizierung fehlschlägt und meldet dies auch dem Nutzer.

Falls die Datei verloren gegangen ist, dann lässt sich der Zugang nur über den MasterKey wiederherstellen, da dieser das eigentliche Datenbankpasswort ist.

4.4.5 Dateisystem

Die Anwendung speichert verschiedene Dateien im Dateisystem für die Datenbanken. Für alles gibt es einen vom Nutzer konfigurierten Ordner. In diesen werden alle Datenbanken gespeichert. Für jede Datenbank gibt es hierfür einen eigenen Unterordner. Der Namen des Unterordners ist frei wählbar, nur dürfen sich zwei Datenbank nicht den gleichen Ordner teilen und die Unterordner dürfen nicht verschaltet sein.

In dem Datenbankordner gibt es dann eine Liste von Dateien und Ordnern, die von der Anwendung genutzt werden:

- `config.json`: Die Konfiguration der Datenbank. Genauere Informationen dazu im Kapitel 4.4.4.
- `config.json.backup`: Wird vor der Aktualisierung von `config.json` von der Anwendung automatisch angelegt, um Fehler zu vermeiden.
- `confidential.db`: Die LiteDB Datei zur Confidential Datenbank
- `confidential-log.db`: Wird von LiteDB automatisch für `confidential.db` angelegt.
- `confidential.index.db`: Der LiteDB Index zur Confidential Datenbank
- `confidential.index-log.db`: Wird von LiteDB automatisch für `confidential.index.db` angelegt.
- `data.db`: Die LiteDB Datei zur Main Datenbank
- `data-log.db`: Wird von LiteDB automatisch für `data.db` angelegt.
- `data.index.db`: Der LiteDB Index zur Main Datenbank
- `data.index-log.db`: Wird von LiteDB automatisch für `data.index.db` angelegt.

- `files/`: Die gespeicherten verschlüsselten Dateien. Mehr siehe unten
- `upload/`: Die Dateien, die gerade vom Nutzer zum Server hochgeladen werden.

Zu den LiteDB Dateien gibt es immer jeweils ein `-log.db` Datei. Diese wird automatisch vom System angelegt und dient zur Wiederherstellung ungesicherter Daten. LiteDB sichert zuerst alle Änderungen in diese Log Datei und schreibt nach einer gewissen Zeit diese dann in die eigentliche Datenbank rein. Dies geschieht alles automatisch, ohne dass der Entwickler eingreifen muss. Wenn Sicherungen angelegt werden, so sind die Logdateien auch immer mitzusichern, da diese noch ungesicherte Daten enthalten können.

Des weiteren gibt zur Main und Confidential Datenbank jeweils einen Index. Dieser kann jederzeit vom Nutzer gelöscht werden. Die Anwendung legt diesen dann wieder neu an, indem die komplette Datenbank neu indiziert wird. Dies kann je nach Größe der Datenbank und Rechenleistung einige Minuten bis eine Stunde dauern.

Wenn die Confidential Datenbank entfernt werden soll, so reicht es einfach aus die Dateien `confidential.db` und `confidential-log.db` zu entfernen (Anwendung muss dabei neugestartet werden). Die Anwendung kann dann problemlos damit weiterarbeiten. Ein einfaches Rücksichern ist möglich, indem diese Dateien jederzeit wieder hinzugefügt werden.

Wenn Dateien vom Nutzer hochgeladen werden, dann landen diese zuerst im Ordner `upload/`. Dazu wird ein zufälliger Schlüssel generiert und mittels AES in Echtzeit verschlüsselt. Die Daten sind also zu jeder Zeit verschlüsselt auf der Festplatte, auch wenn die Übertragung nicht vollständig ist. Als Dateiname wird Hex-Wert einer zufälligen 12 Byte großen Zahl gewählt. Fall der Upload abbricht wird einfach diese Datei gelöscht.

Sobald der Upload vollständig ist, wird die Datei in `files/` verschoben. Dafür wird zuerst der SHA512 Hashwert der verschlüsselten Datei ermittelt. Als Pfad wird der Hex-Wert des SHA512 Hash-Werts genutzt, wo die ersten beiden Bytes einen Unterordner und die nächsten beiden Bytes einen Unterunterordner bilden. Zum Beispiel: `files/0D48/60D6/FA4F66CCB0C0D35C9FEAC8B...` (der Pfad wurde gekürzt dargestellt). Die Wahrscheinlichkeit für eine Kollision ist hier äußerst gering. SHA512 kann $1,3 \cdot 10^{154}$ verschiedene Zahlen abbilden (zum Vergleich, es gibt ca. 10^{80} Atome im beobachtbaren Universum [31]). Und selbst für den äußerst unwahrscheinlichen Fall gibt es ein Fallback: Es wird an den Dateinamen ein Bindestrich ("-") und eine dezimale Zahl beginnend bei 1 angefügt, falls es zu einer Kollision kommt.

Nach dem Verschieben werden die Metainformationen in die Datenbank eingetragen und der Oberfläche mitgeteilt.

Die Dateien in `files/` werden nicht entfernt oder exportiert, wenn die Confidential Datenbank entfernt wird. Diese werden einfach von der Anwendung nicht mehr erkannt und können nicht gelesen werden, da der Schlüssel fehlt. Für den Nutzer oder externe ist also der Inhalt der Dateien nicht mehr erkenntlich. Einzig über die Dateigröße ließen sich Rückschlüsse ziehen. Dies wurde aber mit dem Betreuer und dem Auftraggeber besprochen und dies soll nicht geändert werden.

Theoretisch ist es möglich Dateien aus `files/` heraus zu löschen. Dies wird zwar nicht empfohlen, aber die Anwendung erkennt dies und zeigt auch an, dass die Datei nicht mehr verfügbar ist. Es würde nur für das Szenario, wo der Nutzer alte Dateien aus der Historie entfernt haben möchte, Sinn ergeben. Für diesen Fall ist derzeit kein Support in der Anwendung eingebaut.

5 Umsetzung

5.1 Hürden und Herausforderungen

Auch bei diesen Projekt gab es bei der Umsetzung ein paar Stolpersteine, die welchen in den folgenden Unterkapiteln genauer eingegangen wird.

5.1.1 WebAuthn

WebAuthn ist ein relativ neuer Standard. So neu, dass es zwar schon in jeden modernen Browser unterstützt wird, es aber derzeit keine offizielle Spezifikation gibt (nur Entwürfe). Des weiteren ist diese jetzt auch nicht so verbreitet, dass der Entwickler sofort auch die nötige Technik parat hat, um das zu testen.

So konnte der Autor sich am Anfang nur auf das eigene Mobilgerät verlassen, wo der eigene Fingerabdrucksensor als Hardwaretoken genutzt wird. Für die Umsetzung auf den Laptop wurden FIDO-USB-Sticks bestellt, da zum einen nicht jeder Laptop einen Fingerabdrucksensor hat und zum anderen der Sicherheitstoken als eigenständige physische Komponente genutzt werden soll.

Sobald die bestellten Hardwaretokens angekommen waren, ging es an die Implementierung der Schnittstelle. Hier war das größte Problem, dass es nur wenige Beispiele und eine ausführliche und leicht verwirrende Spezifikation online verfügbar ist. Ein weiteres Problem war die schier große Menge an Funktionen und Variabilität an Schlüsseln, die WebAuthn bereitstellt. Nicht alles wird von allen Tokens unterstützt und es ist schwierig herauszufinden was benötigt wird und was nicht. Schwierig wird es aber erst recht, wennsämtliche Möglichkeiten der unterstützten Schlüsselalgorithmen berücksichtigen werden sollen (z.B. ES256, EdDSA, HMAC in unterschiedlichen Varianten, AES in unterschiedlichen Varianten, ...). Und dann gibt es eine Authentifizierung in 24 Schritten.

Einen Großteil dieser Schwierigkeiten können verschiedene Bibliotheken (z.B. fido2-net-lib) übernehmen, welche aber alle wiederum eigene Schnittstellen und Schwierigkeiten haben.

5.1.2 Tests beim Nutzer

Es ist immer gut schon frühzeitig Versionen den zukünftigen Nutzern zu zeigen, damit Feedback schnell an den Entwickler gelangt. Dies wurde auch in dieser Arbeit versucht. Doch leider konnte durch zeittechnische Schwierigkeiten seitens Nutzers und Autors nicht genauer auf Probleme eingegangen werden.

6 Tests

Hier wird auf die Tests genauer eingegangen, die bei der Entwicklung der Anwendung durchgeführt wurden. Auf Grund der Größe und des Umfang des Projektes und gleichzeitig dem knappen Zeitrahmen wurde auf viele Tests verzichtet. Vieles wurde manuell getestet und es wurden so gut wie keine automatischen Tests vorgenommen. Dafür wurde bei der Entwicklung darauf geachtet das alles so zu entwickeln, dass ein Testen möglichst leicht ist und möglichst viele Fehler ausgeschlossen werden.

Es ist nicht auszuschließen, dass das derzeitige Produkt noch Fehler enthält. In der aktuellen Version läuft diese aber zuverlässig auf sämtlichen Testgeräten ohne Abstürze oder Datenverlust.

6.1 Automatisierte Tests

Ein großer Vorteil an den automatisierten Tests ist, dass die ohne menschliches Zutun gestartet, durchgeführt und ausgewertet werden können. Dadurch lassen die sich gut in die Entwicklungsprozesse einbinden, wo diese regelmäßig mit den neuesten Änderungen am Code ausgeführt werden. Das hat zur Folge, dass Probleme am Code relativ frühzeitig erkannt und somit auch behoben werden können.

Je nachdem, ob nun das Front-End, Back-End oder die API testen werden sollen, gibt es unterschiedliche Testarten, die sich mehr oder weniger gut dafür eignen. Aus zeitlichen Gründen wurde bei vielen darauf verzichtet diese im Projekt umzusetzen. Stattdessen wurden die manuell vom Entwickler durchgeführt. Daher wird nur auf die Testarten genauer eingegangen, die auch ihre Anwendung gefunden haben.

6.1.1 Kompilierung

Dies ist ein relativ einfacher Test, da nur geschaut wird, ob der Compiler syntaktische Fehler im Quellcode findet. Zum Teil kann der Compiler noch auf andere Fehlerquellen, wie zum Beispiel Typfehler, untersuchen. In der Ausführung wird der Compiler auf dem Quellcode gestartet und geschaut, ob es Fehler oder Warnungen gibt.

Die Compiler von den verwendeten Programmiersprachen C# und Elm sind sehr gut im Finden vieler Fehler und geben hilfreiche Fehlermeldungen aus. Übrig bleiben hauptsächlich logische Fehler, die sich dann manuell oder mit anderen Testarten testen lassen.

Bei diesem Projekt wurden alles korrigiert, was der Compiler als Fehler oder Warnung gemeldet hat.

6.2 Nutzungsdaten

Der Forscher wird im Laufe seiner Arbeit die Anwendung mit Daten füllen. Dabei wächst die Datenbank und zu Menge der zu verwaltenden Dateien. Ziel ist es, dass die Anwendung auch nach einer langfristigen Arbeit noch gefühlt genauso schnell arbeitet, wie Anfang auch.

Um die Last der Nutzungsdaten zu testen, wurde eine einheitliche Testumgebung und Grenzwerte für die Zeiten definiert. Als Testumgebung wurde ein 10 Jahre alter Laptop mit 4 Kernen, 8 GB RAM und 256 GB SSD festgelegt. Für alle Tests wurde immer wieder die gleiche Umgebung gewählt. Die Datenbank wurde mit Daten gefüllt und dann wurden verschiedene Suchanfragen gestellt oder durch die Daten navigiert. Der Vorteil an dieser

beschränkten Umgebung ist, dass die verwendeten Laptops der Forscher alle neuer und leistungsstärker sind, wodurch auszugehen ist, dass diese es leichter haben werden mit der Menge an Daten umzugehen.

Für die Grenzwerte bei den Zeiten wurde definiert, dass eine einfache Navigation nie länger als 1 Sekunde dauern darf. Bei einer Suchanfrage müssen schon nach maximal 3 Sekunden erste Ergebnisse zu sehen sein. Nach 10 Sekunden ist die Suche beendet. Ist der gewählte Suchausdruck komplexer, dann sind 60 Sekunden erlaubt. Das Öffnen der aggregierten Ansicht von Einträgen muss nach 3 Sekunden fertig sein. Die Oberfläche muss flüssig auf Nutzereingaben reagieren. Diese Schwellwerte wurden dahingehend festgelegt, dass diese zum einen für einen Nutzer vertretbar sind und zum anderen ein ungestörtes Arbeiten ermöglichen.

Nachdem die Umgebung und die Grenzwerte definiert sind, geht es darum festzustellen, wie viele Daten ein Forscher überhaupt anlegen wird. Dazu wurden die Forscher aus der Testgruppe befragt, wie sie sich das vorstellen und es wurden alte Projekte angesehen, um eine gute Abschätzung zu erhalten.

Tabelle 7: Größe der Testdatenbank

Kategorie	Erzeugte Daten-bank	Erwartete Realwerte	Kommentar
Anzahl von Interviews	1000	ca. 10-15	
Anzahl von Personen	5000	ca. 50	
Anzahl an genutzten Rollen	2	ca. 5-10	dies hat keine Auswirkung auf die Suchperformance
Anzahl an Attributen pro Rolle	7 und 182	ca. 10-15	jeweils zu ca. 50% genutzt
Anzahl an Attributen	ca. 472.000	ca. 625	
Länge der Texte	ca. 2,5 KB	ca. 50 KB	Für die Suche wurden kürzere Texte mit mehr Attributen gewählt
Anzahl der Einträge in der Historie	ca. 5	ca. 50	Historie wird für die Suche ignoriert
Speicher	ca. 2,3 GB	ca. 650 MB	

Für die Testfälle wurde dann diese Zahlen großzügig multipliziert (siehe Tabelle 7), um eine deutlich größere Datenbank zu erhalten. Für die Textgenerierung wurden Markov-Ketten von diversen Wikipedia-Artikeln genutzt. Dadurch werden einer natürlichen Sprache ähnliche Texte erzeugt, die beliebig lang sein können, aber keine Bedeutung haben. Die Texte wurden mit Absicht kürzer gewählt als in den erwarteten Werten, da die Textgenerierung über Markov-Ketten recht lange braucht (in der aktuellen Konfiguration schon 5-10 Minuten

für die komplette Datenbank). Die resultierende Datenbank ist trotzdem groß genug.

Dabei stellte sich heraus, dass der anfängliche naive Suchalgorithmus sehr langsam ist. Es werden 30 bis 120 Sekunden für Suchanfragen benötigt. Bei dem naiven Suchalgorithmus wurden alle Einträge aus der Datenbank einzeln ausgelesen, mit dem Suchquery abgeglichen und dann weitergereicht. Sobald alle Einträge überprüft wurden, wurde das an die Oberfläche weitergereicht. Auch das Abrufen der aggregierten Seiten ist unverhältnismäßig lang mit 5-10 Sekunden.

Danach wurden mehrere Optimierungen an den Algorithmus vorgenommen, die aber nicht aus zeitlichen Gründen nicht einzeln auf ihre Effektivität überprüft wurden. Stattdessen wurde jede Optimierung beibehalten und wirkte sich somit auch auf die weiteren Optimierungen aus.

Zuerst wurden die Verknüpfungen von Objekten nun auf beiden Seiten hinterlegt. Vorher wurde eine 1-n Verknüpfung so abgebildet, dass beim n-Element nur die ID des 1-Element hinterlegt wurde, anders herum nicht. Das bedeutete also, wenn alle Verknüpfungen des 1-Element abrufen werden sollen, musste die komplette Datenbank durchsucht werden. Nun ist zusätzlich beim 1-Element eine Liste der IDs hinterlegt. Das erhöht den Synchronisierungsaufwand dafür sind Beziehungen schneller verfügbar. Außerdem wurden verschiedene Elemente (Attribute, Attributs-Historie, Einträge von Dateien) aus den jeweiligen Elternelementen ausgegliedert und erhielten ihre jeweils eigenen Collections. Das erhöht auch wieder den Verwaltungsaufwand, da mehr Objekte verwaltet werden müssen.

Diese Änderungen hatten zur Folge, dass sich an der Größe der Datenbank nicht wirklich etwas geändert hat. Dafür sind die Abfragen für aggregierte Ansichten auf unter 200ms gesunken. An der Suchperformance hat dies nicht viel geändert.

Als nächstes wurde ein Index aufgebaut. Dazu werden alle aktuellen Texte (die aus der Historie werden derzeit ignoriert) in Tokens aufgespalten und umgewandelt. Ein Token ist ein Folge von Kleinbuchstaben und Zahlen. Großbuchstaben werden in Kleinbuchstaben umgewandelt. Akzente und Sonderzeichen werden entfernt. Der Index enthält dann zu jeden Token den jeweiligen Fundort. Beim Fundort wird nur die ID und Art des Eintrags berücksichtigt. Jeder Suchstring wird auch in die entsprechenden Tokens umgewandelt und dann wird für jeden Suchtoken herausgesucht, welche Einträge in Frage kommen. Die Mengen an Einträgen für jedes Suchtoken werden dann Mengentheoretisch zusammengefasst. Heraus kommt eine Liste an in Frage kommenden Einträgen.

Da der Index unter Umständen auch vertrauenswürdige Daten enthält, wird für die Confidential und die normale Datenbank jeweils ein Index angelegt.

Jeder Token kann wieder eine Menge an kürzeren Tokens enthalten, indem vom Anfang und Ende eine beliebige Menge an Zeichen entfernt wird. Dies ermöglicht die Suche nach Teilwörtern, ohne das ganze Wort zu kennen. Experimentell hat sich herausgestellt, dass es unpraktisch ist als minimale Länge eines Tokens 1 zu nehmen. Hierbei ist der Index auf die 10-fache Größe der Datenbank angewachsen und das noch bevor ein Zehntel der Datenbank gelesen wurde. Stattdessen wurde als minimale Länge 3 genommen, da dies ein guter Vergleich zwischen Größe des Index und der Datenbank ist (2,3 GB Datenbank und 1,0 GB Index).

Der Einsatz eines Index hatte nun zur Folge, dass bei Suchanfragen, die Schlüsselwörter genutzt haben, nun eine Antwort in Sekundenbruchteilen zu sehen ist. Falls allgemeinere Sachen benötigt werden, so dauert eine Suche immer noch genauso lang.

Bis zu diesen Zeitpunkt wurden die Ergebnisse einer Suche auf dem Server zurückgehalten, um sie dann zu sortieren und im Anschluss an die Oberfläche weiterzureichen. Diese wurde damit behoben, dass sämtliche Ergebnisse sobald sie verfügbar sind, nun direkt mit ein paar Hinweisen zur Sortierung an die Oberfläche weitergereicht werden. Die Oberfläche sortiert dann selbstständig die Ergebnisse und ordnet sie schon der Menge bekannter Ergebnisse ein. Dies hat zur Folge, dass erste Ergebnisse schnell zu sehen sind, die aber mit der Zeit weiter verbessert werden können.

Nach diesen Umstrukturierungen und Anpassungen haben sich Suche und die Anzeige aggregierter Ansichten stark verbessert. Zum Schluss hin konnten alle zeitlichen Grenzen auf der Testumgebung erreicht werden. Es gibt Ideen die Suche noch weiter zu verbessern, welche all in zukünftigen Erweiterungen (siehe 8.2.4) kommen können.

7 Rollout

Dies ist der Weg, um das Produkt vom Entwicklungsstation bis zum Endnutzer zu bringen. Dies beinhaltet auf der einen Seite die Installation beim Nutzer an sich und zum anderen der Weg, den eine Änderung beim Produkt (ein neues Feature, eine Fehlerkorrektur, ...) macht um dann beim Nutzer anzukommen.

7.1 Installation

Dies sind die Schritte, die notwendig sind, damit das Produkt auf dem Rechner des Nutzer läuft. Dafür gibt es verschiedene Methoden, die auch im Laufe der Entwicklung durchgelaufen sind. Vom Prinzip her bauen diese aufeinander auf und nehmen immer mehr manuelle Arbeit ab.

7.1.1 Manuelle Installation

Am Anfang der Entwicklung wurde alles manuell installiert. Dies hat den Vorteil, dass der Entwickler direkt sehen kann, was, wie und wo benötigt wird. Außerdem erleichtert dies die Konfiguration selbst. Es ist von Vorteil sich diese Schritte irgendwie zu notieren, da dies für die spätere Automatisierung benötigt wird.

Bei diesem Produkt bedeutete dies, dass folgende Produkte installieren bzw. Schritte durchgeführt werden müssen:

1. Herunterladen des Quellcodes in einen beliebigen temporären Ordner
2. Installation des Elm Compilers
3. Compilieren des Elm Codes
4. Installation von JavaScript Komprimierungswerkzeugen
5. Komprimierung des JavaScript Codes
6. Installation von .NET SDK
7. Compilieren des Server C# Codes
8. Compilieren von Server Tools und Ausführung dieser
9. Anlegen der Programmverzeichnisstruktur
10. Kopieren der compilierten Server- und JavaScript-Dateien in die Programmverzeichnisstruktur
11. Kopieren der statischen Inhalte für die Web-Oberfläche in die Programmverzeichnisstruktur
12. Anlegen der Konfigurationsdatei
13. Anlegen der Verknüpfung zum starten der Anwendung

Diese Schritte sind vom Prinzip her unter Windows und Linux gleich, auch wenn im Detail leicht unterscheiden (z.B. Installationsort des Programms).

7.1.2 Halbautomatische Installation mit Docker

Bei der manuellen Installation sind einige Schritte, die vereinfachen lassen, schon im Vorfeld compiliert, um sie dann fertig auf den Zielrechner runter zu laden. Dazu eignet sich eine CI-Pipeline, so wie sie auch in dieser Arbeit genutzt wurde. Das ist ein spezielles Script, welches von einen Server gestartet wird, wenn ein Entwickler neuen Code auf die Codeverwaltung (in diesem Fall GitLab, geht aber auch mit anderen wie GitHub) hochlädt.

Der Server startet dann verschiedene Dockercontainer, was in sich abgeschlossene und konsistente Umgebungen sind, und führt darin vordefinierte Befehle aus. Das Ziel von

Dockercontainern, dass immer die gleichen Bedingungen (installierte Software, Konfiguration, etc.) herrschen und daher genau ersichtlich ist, was genau getan werden muss.

Die Befehle sind zusammengefasst die Schritte 1 bis 8 aus 7.1.1. Dadurch fallen diese Schritte auch bei der Installation beim Nutzer weg, da diese schon fertig auf den Server existieren. Dafür werden diese 8 Schritte beim Nutzer durch den Download der fertig gebauten Sachen und Installation von .NET Runtime (schmalere Version von .NET SDK) ersetzt. Auf dem Server kommt noch hinzu, dass alle notwendigen Dateien noch einmal zusammengefasst werden, damit sie besser für die Installation geeignet sind.

Einen weiteren Vorteil hat diese Vorgehensweise auch. So ist relativ früh erkenntlich, ob es Probleme beim Compilieren und Zusammenstellen gibt und das bevor die Installation beim Nutzer durchführt wird. Außerdem lässt sich auf dem Server noch automatische Tests ausführen und die Versionierung erleichtern.

7.1.3 Vollautomatische Installation mit Wix (Windows)

An Automatisierung fehlen nur noch die letzten Schritte 9 bis 13 aus 7.1.1. Unter Windows eignet sich das von Microsoft veröffentlichte Softwaretool Wix. Hiermit wird eine XML-Datei angelegt, die alle Anweisungen enthält, die für die Installation notwendig sind. Danach gibt es einen Compiler, der die XML-Datei mit Anweisungen und alle zu installierenden Dateien einliest, zusammenpackt und eine ausführbare EXE- oder MSI-Datei erstellt.

Diese Schritte lassen sich auch automatisch auf dem Server in einen Dockercontainer ausführen, so dass am Ende nur noch die EXE- oder MSI-Datei übrig bleibt. Daher lässt sich die Installationsroutine am Nutzer so zusammenfassen:

1. Installer herunterladen
2. Installer starten und abwarten. Eventuell Konfiguration vornehmen
3. Fertig

7.1.4 Vollautomatische Installation unter Linux

Genauso wie sich die Installation am Nutzer bei Windows zusammenfassen lässt, geht dies auch unter Linux nur mit anderen Mitteln. Unter Linux gibt es aber eine große Palette an Werkzeugen, da je nach Linux Distribution einige Sachen anders anders funktionieren. So ist z.B. die Paketverwaltung (wird benötigt um Abhängigkeiten zu installieren) bei einem Debian-Linux `apt` und bei einem Arch-Linux `pacman`, welche natürlich jeweils andere Formate sehen wollen.

Als Entwickler lässt sich dies vereinfachen, indem dieser sich ein Shellscript schreibt, was alle Installationsanweisungen enthält und dieses im Detail nachschaut unter welcher Distribution es sich derzeit befindet.

Dies bedeutet natürlich aber auch viel Arbeit und das wurde aus Zeit- und Prioritätsgründen nicht vom Autor praktisch umgesetzt.

7.2 Stages

Von der Entwicklung des Codes bis zum Nutzer durchlaufen Änderungen an der Codebasis verschiedene Stages (Stadien). Üblicherweise wird hier mit einem Modell gearbeitet, was drei (Entwicklung, Staging, Produktiv) oder vier (Entwicklung, Test, Staging, Produktiv) Stages enthält.

Die Entwicklungs-Stage findet direkt beim Entwickler statt. Diese kann jederzeit vom Entwickler kaputt gemacht und wieder komplett neu aufgebaut werden. Hier kann und darf es passieren, dass sämtliche Nutzerdaten zerstört werden.

Danach geht es in die Test-Stage, wo geschaut wird, ob das ganze Produkt noch funktioniert und ob es Fehler gibt. Falls hier Probleme auftreten, dann werden diese direkt zurück zum Entwickler kommuniziert. Üblicherweise wird hier mit Daten gearbeitet, die Realdaten sehr ähnelt, um sämtliche Szenarien besser abbilden zu können.

Danach geht es in die Staging-Stage, die Änderungen enthält, die kurz vor Veröffentlichung stehen.

Und Schlussendlich kommen die Änderung in die Produktiv-Stage, wo sie dann direkt beim Nutzer installiert werden. Hier sollten keine Fehler mehr in den Änderungen existieren, da hier mit realen Nutzerdaten gearbeitet wird, die nicht verloren gehen dürfen.

Der Autor hat sich für seine Entwicklung für das 3-Stage-Modell entschieden, da die Entwicklung noch am Anfang ist und die zusätzliche Test-Stage erhöhten Aufwand bedeutet. Die vierte Stage kann jederzeit nachträglich eingeführt werden.

Die 3 Stage spiegeln sich auch in der Quellcode-Organisation des Projekts wieder. Die Entwicklungs-Stage sind sämtliche Feature- oder Fix-Branche, die der Autor in seiner Entwicklung anlegt. Da drin kann und darf alles passieren. Sobald die Änderungen an einem Branch abgeschlossen und in sich getestet sind, werden diese in den `develop`-Branch überführt. Dies entspricht derzeit der Staging-Stage. Hier wird alles insgesamt nochmal getestet. Oftmals mehere Änderungen aus der Entwicklungs-Stage gleichzeitig. Nachdem alle Änderungen hier bestanden haben, werden diese in den `master`-Branch (Produktiv-Stage) überführt und eine neue Versionsnummer wird erstellt.

8 Ausblick

Nachdem das Projekt abgeschlossen ist, geht es darum, wie danach damit verfahren wird. Wird es weiterhin im Einsatz sein (siehe Nachnutzung 8.1)? Gibt es Erweiterungen und von welcher Art wäre vorstellbar (siehe 8.2)? Und wie wird mit Updates und Wartungen verfahren (siehe 8.3)?

8.1 Nachnutzung

Derzeit ist eine Nutzung am ethnologischen Institut der Max-Plank-Gesellschaft geplant und dafür war auch das Projekt erstellt wurden. Während der Entwicklung zeigte sich schon früh der Bedarf und das Interesse an dem Produkt und wünschte sich möglichst früh erste Ergebnisse zu sehen. So kam es auch zu einen regen Austausch zwischen dem Autor und den Bedarfsträgern bzw. Forschern, welche auch als erste Testgruppe fungiert.

Nach dem Abschluss des Projekts wird im mit den Auftraggeber und den Bedarfsträgern über den Erfolg erneut diskutiert und besprochen, wie die weitere Entwicklung (siehe 8.2) geschehen und das Produkt eingesetzt wird.

Eine Verbreitung an die anderen Institute ist derzeit noch nicht vorgesehen, aber in einem zukünftigen Stadium geplant.

8.2 Erweiterungen

8.2.1 Webseite

Im Rahmen dieser Arbeit ist eine Webseite leider nicht zustande gekommen. Eine Webseite wäre ein guter Ort, um neuen Nutzern einen ersten Eindruck über das Produkt, seine Funktion und die Nutzung zu vermitteln. Hier können auch Updates und Downloads bereitgestellt werden.

8.2.2 Schnittstellen

Schnittstellen sind Möglichkeiten, um Daten aus anderen Programmen für dieses oder auch anders herum bereitzustellen. Dazu können diese Daten importiert, exportiert oder auch direkt nativ unterstützt werden, um ein nahtloses Arbeiten mit den Programmen zu ermöglichen.

Dabei gibt es verschiedene Formate zur Gestaltung dieser. Ein bekanntes, welches sich in der ethnologischen Forschung etabliert hat, wären die Datenaustauschformate der DDI Alliance. Diese hat verschiedene Schemas für XML, JSON und andere Dateiformate bereitgestellt, damit sich hier Daten leichter von einer Anwendung in eine andere übertragen lassen.

Diese Formate zu unterstützen würde somit den Forscher erlauben Daten aus beliebigen kompatiblen Programmen in dieses und auch anders herum zu übertragen und die Arbeit stark zu erleichtern, da die Daten nicht mehrfach händisch neu angelegt werden müssen.

Es gibt aber weitere Formen von Daten, mit den ein Forscher eines ethnologischen Instituts häufig zu tun hat. Dazu zählen auch die Programme Word und Excel der Office-Suite aus dem Hause Microsoft. Hier können Erweiterungen bezüglich des Einlesen oder Generierung von Dokumenten entstehen. Auch Plugins, damit direkt über die Programme Word oder Excel auf die Anwendung zugegriffen werden kann, wären möglich.

8.2.3 Cloud-Dienst

Derzeit ist die Anwendung nur lokal auf dem Laptop des Nutzers installiert und alle Daten sind auch nur dort verfügbar und müssen auch von gesichert werden. Dies kommt mit der Einschränkung, dass die Daten nur umständlich geteilt und gesichert werden können. Eine Möglichkeit zu Erweiterung steht hier die Bereitstellung eines Servers, über die alles zentral gesichert und verwaltet wird.

Hier sind alle Daten zentral gespeichert und die Nutzer haben dann die Wahl, ob sie mit den Live-Daten vom Server oder mit der lokalen Kopie (falls keine Internetverbindung besteht) arbeiten möchten.

Besondere Vorteile an dieser Erweiterung sind:

- Daten lassen sich leichter durch die IT sichern und es lassen sich Backup-Richtlinien stringend durchsetzen
- Das Teilen von Daten unter den Forschern untereinander ist leichter. Auch das gleichzeitige Arbeiten an gleichen Datensätzen wäre somit möglich.
- Der Forscher muss nicht mehr alle Daten lokal bereithalten. Damit würden auch die Anforderungen an das Endgerät gelockert werden. Eine Nutzung von Handys oder Tablets wäre somit möglich.
- Eine Installation ist nicht mehr zwingend notwendig, da die Weboberfläche auch zentral bereitgestellt werden kann.
- Updates lassen sich leichter durchsetzen.

An dieser Erweiterung entstehen aber auch Probleme, die dann direkt berücksichtigt und behandelt werden müssen:

- Die Daten müssen auf dem Server sicher vor externer Manipulation und Zugriff sein.
- Wie mit gleichzeitigen Änderungen umgegangen, die gegenseitig sich im Konflikt stehen?
- Wie lassen sich Rechte für die Datensätze vergeben? Wie granular geht das und wie wird das eingehalten? Das ist besonders für das Teilen der Datensätze notwendig.
- Wie werden nachträglich Daten synchronisiert, wenn der Nutzer wieder online ist?

8.2.4 Suchoptimierungen

Die Suche lässt sich über verschiedene Wege noch weiter optimieren. Wichtige ist, dass bei allen Optimierungen darauf geachtet werden muss, dass sämtliche Daten weiterhin den gleichen Sicherheitsstandard genießen, wie sie zuvor auch hatten. Ein paar Möglichkeiten dies zu erreichen werden im Folgenden kurz ausgearbeitet.

Die erste Idee wäre einen weiteren Suchindex für andere Datentypen aufzubauen. Derzeit existiert nur einer für Strings, der auch Substrings ab der Länger 3 abdeckt. Daher wäre es hilfreich, wenn für Ganzzahlen, Fließkommazahlen oder Datumsangaben ein weiterer Index existiert, der auch Bereiche zwischen zwei Werten oder ähnliche Werte abdeckt. Die Werte sind zum Teil direkt durch die Eingabefelder direkt zu ermitteln und zum Teil befinden die sich irgendwo in längeren Texten.

Ein weiterer Weg wäre für besonders kurzen Strings einen Index aufzubauen, damit auch nach Sachen wie “EU” gesucht werden kann.

Die dritte Idee wäre eine passende Speicherstruktur für den Index zu entwerfen. Derzeit wird der Index in einer Datenbank hinterlegt. Diese ermöglicht es zwar schnell einen

bestimmten Schlüssel abzurufen, muss aber alle Einträge abfragen, wenn der Nutzer alle Schlüssel unter oder über einen Wert haben möchte.

Weiterhin ist noch ein Index, welcher auch historische Werte berücksichtigt, hilfreich. Mit diesen können dann Forscher nach alten Werten suchen, bevor sie eine Änderung durchgeführt haben.

Des weiteren, weniger eine Geschwindigkeitsoptimierung, dafür mehr eine für die Nutzerfreundlichkeit wäre ein einfacher Editor für die Suchanfragen. Somit muss ein Anfänger nicht erst verstehen und lernen wie diese aufgebaut sind, und was mit diesen alles möglich ist.

8.3 Wartung und Update

Geplant sind zukünftige Updates, welche Probleme und Fehler im aktuellen Produkt beheben, sowie weitere Funktionen (siehe 8.2) hinzufügen. Zu Verbreitung der Updates wird ein ähnlicher Mechanismus wie bei der Installation (siehe 7.1) verfolgt. Die Pakete werden zusammengepackt und online zum Download bereitgestellt. Das Produkt überprüft selbstständig im Hintergrund auf Updates, informiert den Nutzer und leitet ihn zum Download und Installation an.

Auch eine automatische Installation ist möglich, auch wenn dies unter Einschränkungen durch die Art der Installation stehen kann. Dies ist zum Beispiel der Fall, wenn die Anwendung im Programmverzeichnis installiert wurde und derzeit kein befugter Administrator verfügbar ist, der das Passwort eingeben kann. Auch hierfür existieren Lösungen (z.B. über einen autorisierten Updatedienst), welche aber speziell noch eingerichtet werden müssen.

In welcher Art und Weise die Wartung und Updates erfolgen ist derzeit noch Bestandteil der in der Nachnutzung (siehe 8.1) geklärt wird.

9 Abschlussbetrachtung

TODO:

- gezogene Schlüsse
- positive, negative Erkenntnisse

Abschnitt aus Einleitung, umformulieren:

Abschließend lässt sich sagen, dass das Ziel, die Arbeit mit Interviewdaten von einer Feldforschung für Forscher an einem ethnologischen Institut, erreicht wurde. Aber es besteht viel Potential für zukünftige Erweiterungen.

A Literatur und Quellen

[1] Martin R. Albrecht. *Security Analysis of Telegram (Symmetric Part)*. 2021. URL: https://mtpsym.github.io/ (besucht am 23. 06. 2022).

[2] Dan Arias. *Adding Salt to Hashing: A Better Way to Store Passwords*. URL: https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/ (besucht am 24. 06. 2022).

[3] Brian Carroll. *Porting Elm to WebAssembly*. URL: https://dev.to/briancarroll/porting-elm-to-webassembly-2lp4 (besucht am 25. 06. 2022).

[4] SQLite Consortium. *SQLite Home Page*. URL: https://sqlite.org/index.html (besucht am 22. 06. 2022).

[5] SQLite Consortium. *SQLite: Top-level Files of trunk*. URL: https://sqlite.org/src/dir?ci=trunk (besucht am 22. 06. 2022).

[6] Evan Czaplicki. *Blazing Fast HTML*. URL: https://elm-lang.org/news/blazing-fast-html-round-two (besucht am 25. 06. 2022).

[7] Maurício David. *Data Structure - LiteDB :: A .NET embedded NoSQL database*. URL: http://www.litedb.org/docs/data-structure/ (besucht am 24. 06. 2022).

[8] Maurício David. *Encryption - LiteDB :: A .NET embedded NoSQL database*. URL: https://www.litedb.org/docs/encryption/ (besucht am 22. 06. 2022).

[9] Maurício David. *LiteDB :: A .NET embedded NoSQL database*. URL: https://www.litedb.org/ (besucht am 22. 06. 2022).

[10] Maurício David. *mbdavid/LiteDB: LiteDB - A .NET NoSQL Document Store in a single data file - https://www.litedb.org*. URL: https://github.com/mbdavid/litedb (besucht am 22. 06. 2022).

[11] Maurício David. *Overview - LiteDB :: A .NET embedded NoSQL database*. URL: http://www.litedb.org/docs/ (besucht am 24. 06. 2022).

[12] Claudia Eckert. *IT-Sicherheit: Konzepte – Verfahren – Protokolle*. Oldenbourg Verlag, 2012. ISBN: 978-3-486-70687-1.

[13] MariaDB Foundation. *Data-at-Rest Encryption Overview - MariaDB Knowledge Base*. URL: https://mariadb.com/kb/en/data-at-rest-encryption-overview/ (besucht am 22. 06. 2022).

[14] MariaDB Foundation. *File Key Management Encryption Plugin - MariaDB Knowledge Base*. URL: https://mariadb.com/kb/en/file-key-management-encryption-plugin/ (besucht am 22. 06. 2022).

[15] MariaDB Foundation. *MariaDB*. URL: https://github.com/MariaDB/ (besucht am 22. 06. 2022).

[16] MariaDB Foundation. *MariaDB Foundation - MariaDB.org*. URL: https://mariadb.org/ (besucht am 22. 06. 2022).

[17] SQLite Consortium und Frank A. Krueger. *praeclarum/sqlite-net: Simple, powerful, cross-platform SQLite client and ORM for .NET*. URL: https://github.com/praeclarum/sqlite-net (besucht am 22. 06. 2022).

[18] MongoDB Incorporated. *BSON (Binary JSON): FAQ*. URL: https://bsonspec.org/faq.html (besucht am 24. 06. 2022).

[19] MongoDB Incorporated. *BSON (Binary JSON): Specification*. URL: https://bsonspec.org/spec.html (besucht am 24. 06. 2022).

[20] MongoDB Incorporated. *MongoDB Data Encryption | MongoDB*. URL: https://www.mongodb.com/basics/mongodb-encryption (besucht am 22. 06. 2022).

[21] MongoDB Incorporated. *MongoDB: Die Plattform Für Anwendungsdaten | MongoDB*. URL: https://www.mongodb.com/de-de (besucht am 22. 06. 2022).

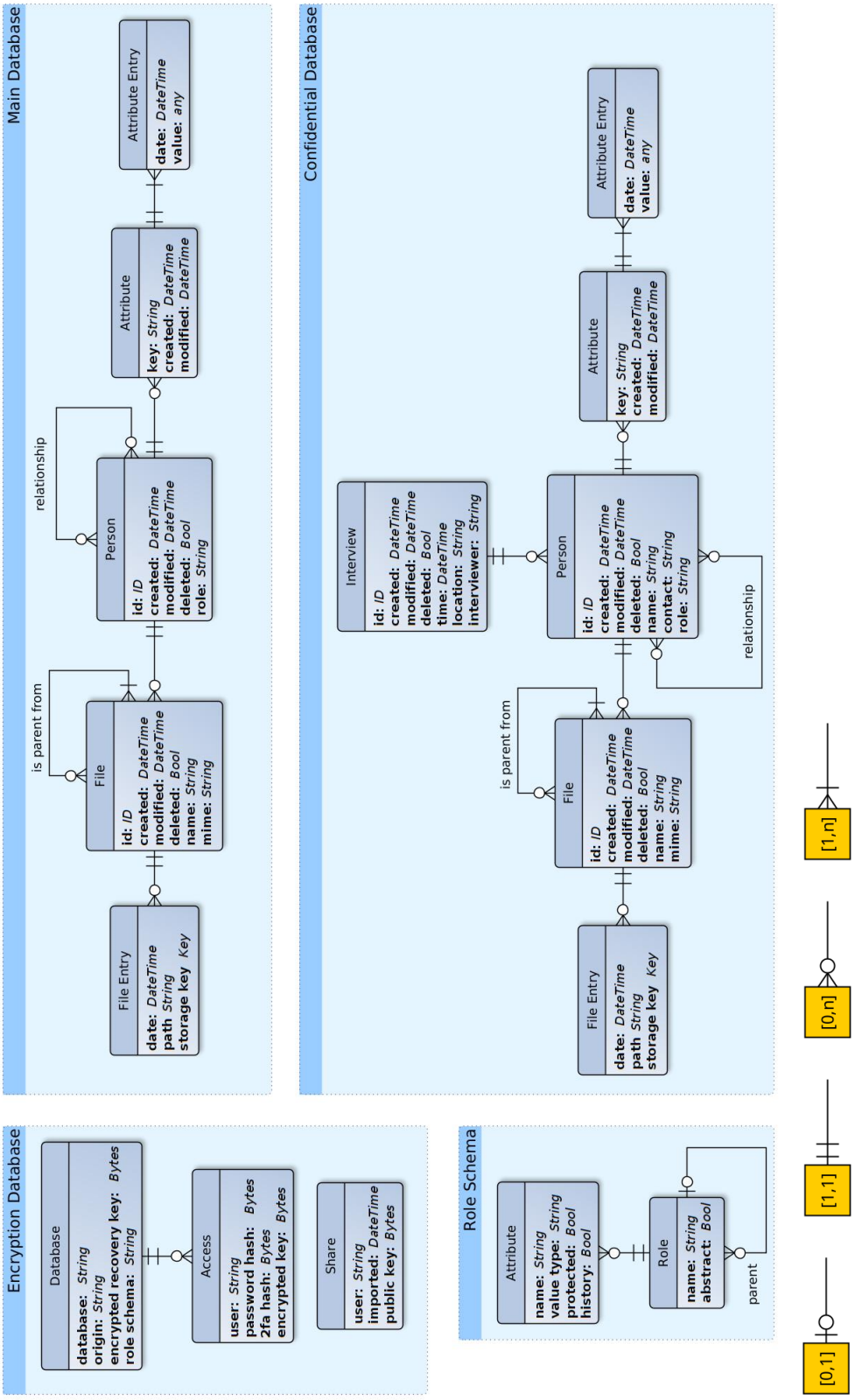
[22] MongoDB Incorporated. *mongodb/mongo: The MongoDB Database*. URL: https://github.com/mongodb/mongo (besucht am 22. 06. 2022).
[23] MongoDB Incorporated. *Server Side Public License - MongoDB*. URL: https://github.com/mongodb/mongo/blob/master/LICENSE-Community.txt (besucht am 22. 06. 2022).
[24] Europäische Kommission. *Ethics and data protection*. 2021. URL: https://ec.europa.eu/info/funding-tenders/opportunities/docs/2021-2027/horizon/guidance/ethics-and-data-protection_he_en.pdf (besucht am 23. 06. 2022).
[25] Oracle. *MySQL*. URL: https://www.mysql.com/de/ (besucht am 22. 06. 2022).
[26] Oracle. *MySQL :: Commercial License for OEMs, ISVs and VARs*. URL: https://www.mysql.com/about/legal/licensing/oem/ (besucht am 22. 06. 2022).
[27] Oracle. *MySQL :: MySQL Enterprise Transparent Data Encryption (TDE)*. URL: https://www.mysql.com/products/enterprise/tde.html (besucht am 22. 06. 2022).
[28] Oracle. *mysql/mysql-server: MySQL Server, the world's most popular open source database, and MySQL Cluster, a real-time, open source transactional database.* URL: https://github.com/mysql/mysql-server (besucht am 22. 06. 2022).
[29] Bundesamt für Sicherheit in der Informationstechnik (BSI). *IT-Grundschutz-Kompendium (Edition 2021)*. 2021. URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Grundschutz/Kompendium/IT_Grundschutz_Kompendium_Edition2021.pdf?__blob=publicationFile&v=6 (besucht am 23. 06. 2022).
[30] Bundesamt für Sicherheit in der Informationstechnik (BSI). *Zwei-Faktor-Authentisierung*. 2018. URL: https://www.bsi.bund.de/DE/Themen/Verbraucherinnen-und-Verbraucher/Informationen-und-Empfehlungen/Cyber-Sicherheitsempfehlungen/Accountschutz/Zwei-Faktor-Authentisierung/zwei-faktor-authentisierung_node.html (besucht am 23. 06. 2022).
[31] Universe Today. *How Many Atoms Are There in the Universe? - Universe Today*. URL: https://www.universetoday.com/36302/atoms-in-the-universe/ (besucht am 24. 06. 2022).

B Abbildungsverzeichnis

C Tabellenverzeichnis

D Anhang

D.1 Entity Relationship Diagramm v0.6



Erklärung siehe Abbildung 5.

D.2 Projektsteckbrief

Projekttitel	Planung und Entwicklung einer Datenbankanwendung für das Forschungsdatenmanagement		
Projektnummer	2022-SWD-19526		
Projektleitung	Max Brauer		
Auftraggeber	Sebastian Ehser, Max-Planck-Institut für ethnologische Forschung		
Projektnutzen	Verwaltung von Forschungsdaten		
Projektumfeld			
Projektinhalt / -ziele	Erstellen eine Datenbankanwendung zur Aufnahme von Forschungsdaten für Forschende eines ethnologischen Instituts. Diese Anwendung soll den Forschenden auf seinen Reisen begleiten und die Arbeit mit seinen Daten erleichtern. Dabei soll ein besonderes Augenmerk auf die Sicherheit gelegt werden. Als Nichtziel wurden zusätzliche Relationen zwischen den Interviewten (außer den vorgeschriebenen) festgelegt. (prototypisch Start mit Raza und das Ziel das auf das gesamte Instit umzusetzen)		
Projektnutzen	• Vereinfachung in der Arbeitsweise der Forschenden durch zentrale und strukturierte Speicherung und Darstellung von Daten • Sicherung der Daten durch Backups und Zugriffskontrollen • Weniger Papierverbrauch bei der Arbeit mit den Daten		
Klärungs-/ Unterstützungsbedarf	• Genaues Modell der zu speichernden Daten • Aktuelle Arbeitsweise der Forschenden		
Start / Ende	Januar 2022 - Juni 2022		
Zwischentermine	• Anmeldung der Bachelorarbeit: offen • Vorstellen der Zwischenergebnisse: im 2-4 Wochen Rythmus • Verteidigung der Arbeit: 5-6 Monate nach Anmeldung der Arbeit • Alphaversion: • Betaversion: • Releaseversion: ähnlich zur Verteidigung		
Aufwand	Software: 112-150 Stunden	Schriftliche Arbeit: 300 Stunden	Gesamt: 450 Stunden
Beteiligte	• Max Brauer (Entwickler und Verfasser der Arbeit) • Sebastian Ehser (Auftraggeber, Projektberater MPI) • Christian Kieser (technischer Berater MPI) • Farah Raza (Bedarfsträgerin)		
Risiken	• Krankheit oder sonstige Ausfälle • Software wird nicht rechtzeitig fertig • Software entspricht nicht oder nicht komplett den Wünschen der Kunden		

D.3 config.json Schema

Informationen und Erklärungen sind im Kapitel 4.4.4.

Path	Flags	Type/Value	Description
`name`	R	*string*	Der Name der Datenbank. Muss der gleiche wie der Ordner sein.
`recovery-key-hash`	R	*string*	Der Base64 codierte SHA512 Hash des Master-Keys.
`role-schema`	R	*string*	Der Pfad zum Rollenschema innerhalb des Schemaordners. Dieses Schema gilt für die gesamte Datenbank.
`access`	R	*array*	Eine Liste von Zugangsdaten, die Zugriff auf diese Datenbank haben
`access[]`	R	*object*	Die Einstellungen zu einen speziellen Zugriff
`access[].user`	R	*string*	Der Nutzername des Nutzers, der Zugriff hat. Dieser darf mehrfach verwendet werden.
`access[].user-id`	R	*string*, `<guid>`	Eine GUID für diesen Nutzerzugang. Wird für die Fido-Authentifizierung genutzt.
`access[] .password-salt`	R	*string*	Base64 codierte zufällige Byte-Folge, die zusammen mit dem Passwort verarbeitet wird.
`access[] .password-hash`	R	*string*	Base64 codierter Hash aus Passwort und Salt
`access[] .fido-descriptor`	RN	*object*	Gespeicherte Informationen zum Fido USB Stick. Format ist durch die Bibliothek bestimmt.
`access[] .fido-public-key`	RN	*string*	Base64 codierter Public Key vom Fido USB Stick
`access[] .fido-user-handle`	RN	*string*	Base64 codierter User Handle vom Fido USB Stick
`access[] .fido-counter`	RN	*int*	Der Counter Wert bei der letzten Anmeldung.
`access[].enc-salt`	R	*string*	Base64 codierte zufällige Bytefolge, die mit dem Passwort gehasht wird.
`access[].enc-key`	R	*string*	Base64 codierte, mit AES verschlüsselte Datenbankschlüssel. Um diesen zu entschlüsseln ist der Hash aus enc-salt und Passwort nötig.
`access[].created`	R	*string*, `<date-time>`	Der Zeitstempel an dem dieser Zugriff angelegt wurde.
`contains-index`	R	*bool*	Gibt an, ob ein Index für diese Datenbank existiert. Wird nicht mehr verwendet.

Flags:

- *R*: This field is required. This path must always exists.
- *N*: The value of this field can be `null`.

```
{
  "name": "test-db",
  "recovery-key-hash": "abdef01234567890...",
  "role-schema": "abc",
  "access": [
    {
      "user": "mustermann",
      "user-id": "d19482e9-e5df-4826-a824-235b5119dafc",
      "password-salt": "abdef01234567890...",
      "password-hash": "abdef01234567890...",
      "fido-descriptor": {
        "Type": 0,
        "Id": "ABDEFabcdef5678",
        "Transports": null
      },
      "fido-public-key": "abdef01234567890...",
      "fido-user-handle": "abdef01234567890...",
      "fido-counter": 1024,
      "enc-salt": "abdef01234567890...",
      "enc-key": "abdef01234567890...",
      "created": "2001-01-01T15:17:19.001"
    }
  ],
  "contains-index": "true"
}
```

D.4 Rollenschema

Genaue Erklärung dazu siehe Kapitel 4.4.3.

Path	Flags	Type/Value	Description
`name`	R	*string*	Der komplette Name inklusive lokalen Pfad des Schemas.
`online`	RN	*string*, `<url>`	Die online URL, wo das Schema abgerufen werden könnte. Wenn das nur ein lokales Schema ist, dann ist dieser Wert null.
`roles`	R	*array*	Alle einzelnen Rollen zu diesem Schema
`roles[]`	R	*object*	
`roles[].name`	R	*string*	Der Name der Rolle
`roles[].abstract`	R	*bool*	Bestimmt ob diese Rolle als abstrakt deklariert wurde. Abstrakte Rollen können nicht instantiert werden.
`roles[].parent`	RN	*string*	Der optionale Name der Elternrolle von dem alle Eigenschaften geerbt werden sollen.
`roles[] .attributes`	R	*object*	Alle Attribute dieser Rolle. Die Namen der Attribute sind frei wählbar, dürfen aber nicht doppelt vorkommen.
`roles[] .attributes.*`	A	*object*	Ein einzelnes Attribut für die Rolle
`roles[] .attributes.* .type`	R	`"DateTime"`, `"SingleLine"`, `"MultiLine"`, `"Number"`, `"Int"`, `"Bool"`	Die Art und somit auch der Typ des Eingabefeldes
`roles[] .attributes.* .protected`	R	*bool*	Gibt an, ob das Attribut geschützt (confidential) ist.
`roles[] .attributes.* .history`	R	*bool*	Gibt an, ob eine Historie zu diesem Attribut angelegt werden soll.

Flags:

- *R*: This field is required. This path must always exists.
- *N*: The value of this field can be `null`.
- *A*: This is an additional property. You can insert any not previously defined name to add your own properties. The schema of this property is predefined has to be used as above.

```
{
  "name": "/test/schemas/v0.json",
  "online": "http://schemas.example.com/test/schemas/v0.json",
  "roles": [
    {
      "name": "Foo",
      "abstract": false,
      "parent": "Bar",
      "attributes": {
        "*foo-bar-baz*": {
          "type": "DateTime",
          "protected": false,
          "history": false
        },
        "*foo-bar-baz*-2": {
          "type": "SingleLine",
          "protected": false,
          "history": false
        },
        "*foo-bar-baz*-3": {
          "type": "MultiLine",
          "protected": false,
          "history": false
        },
        "*foo-bar-baz*-4": {
          "type": "Number",
          "protected": false,
          "history": false
        },
        "*foo-bar-baz*-5": {
          "type": "Int",
          "protected": false,
          "history": false
        },
        "*foo-bar-baz*-6": {
          "type": "Bool",
          "protected": false,
          "history": false
        }
      }
    }
  ]
}
```


D.5 Dacryptero - Internal WebSocket API

Version: 1.0.0

This is the internal api used by the ui and the main webserver. See in the development notes for more details.

D.5.1 PUB `/ws/` Operation

The messages that are sent from the UI to the server

Accepts **one of** the following messages:

D.5.1.1 AccountLogin *Login with the provided credentials*

Path	Flags	Type/Value	Description
`$type`	R	`"AccountLogin"`	
`username`	R	*string*	
`password`	R	*string*	

Flags:

- *R*: This field is required. This path must always exists.

Examples:

```
{
  "$type": "AccountLogin",
  "username": "abc",
  "password": "abc"
}
```

D.5.1.2 AccountRegister *Register with new credentials*

Path	Flags	Type/Value	Description
`$type`	R	`"AccountRegister"`	
`database`	R	*string*	
`key`	RN	*string*	The key that is used for creating new databases. For existing databases this key is not used and null.
`username`	R	*string*	
`password`	R	*string*	

Flags:

- *R*: This field is required. This path must always exists.
- *N*: The value of this field can be `null`.

Examples:

```
{
  "$type": "AccountRegister",
  "database": "db-main",
  "key": null,
  "username": "abc",
  "password": "abc"
}
```

```
{
  "$type": "AccountRegister",
  "database": "db-main",
  "key": "abc",
  "username": "abc",
  "password": "abc"
}
```

D.5.1.3 DatabaseAddFinish *Finalize the DB creation*

Finalize the DB creation and sends every user the new database

Path	Flags	Type/Value	Description
`$type`	R	`"DatabaseAdd Finish"`	
`id`	R	*`string`*	The key that was used for the db creation process

Flags:

- *R*: This field is required. This path must always exists.

Examples:

```
{
  "$type": "DatabaseAddFinish",
  "id": "#id-0123456789abcdef"
}
```

D.5.1.4 DatabaseAddRequest *Starts the process of the db creation*

Starts the process of the db creation

Path	Flags	Type/Value	Description
`$type`	R	`"DatabaseAdd Request"`	
`database`	R	*`string`*	The new database name. The format has to match the pattern `^[\\w-]+$`
`schema`	R	*`string`*	

Flags:

- *R*: This field is required. This path must always exists.

Examples:

```
{
  "$type": "DatabaseAddRequest",
  "database": "new-db-name",
  "schema": "abc"
}
```

D.5.1.5 DatabaseCancelCreation *Cancel DB Creation Process*

Cancel the DB creation process and discard all collected data.

Path	Flags	Type/Value	Description
`$type`	R	`"DatabaseCancelCreation"`	
`key`	R	*`string`*	The key that is used to identify the db creation process

Flags:

- *R*: This field is required. This path must always exists.

Examples:

```
{
  "$type": "DatabaseCancelCreation",
  "key": "abc"
}
```

D.5.1.6 DatabaseConnect *Connect to Database*

Starts the connection process to an existing database. If required it will ask for user login.

Path	Flags	Type/Value	Description
`$type`	R	`"DatabaseConnect"`	
`name`	R	*`string`*	The database name

Flags:

- *R*: This field is required. This path must always exists.

Examples:

```
{
  "$type": "DatabaseConnect",
  "name": "abc"
}
```

D.5.1.7 DatabaseRemoveUser *Removes User from Database*

Tells the server to remove a user from the database configuration. This process is not revertible.

Path	Flags	Type/Value	Description
`$type`	R	`"DatabaseRemoveUser"`	
`database`	R	*`string`*	
`user-id`	R	*`string`*, `<guid>`	

Flags:

- *R*: This field is required. This path must always exists.

Examples:

```
{
  "$type": "DatabaseRemoveUser",
  "database": "db-main",
  "user-id": "guid"
}
```

D.5.1.8 DatabaseUnlock *Unlocks Database*

Unlocks the database to be able to add new access. This can be done using the provided master key. Otherwise the server will start an authentication process.

Path	Flags	Type/Value	Description
`$type`	R	`"DatabaseUnlock"`	
`database`	R	*`string`*	
`master-key`	RN	*`string`*	

Flags:

- *R*: This field is required. This path must always exists.
- *N*: The value of this field can be `null`.

Examples:

```
{
  "$type": "DatabaseUnlock",
  "database": "db-main",
  "master-key": null
}
```

```
{
  "$type": "DatabaseUnlock",
  "database": "db-main",
  "master-key": "abc"
}
```

D.5.1.9 DataInterviewAddRequest *Add Interview Request*

Creates a new interview entry at the server and fill with empty data. The server will respond with the created interview entry

Path	Flags	Type/Value	Description
`$type`	R	`"DataInterviewAdd Request"`	

Path	Flags	Type/Value	Description
`database`	R	*string*	

Flags:

- *R*: This field is required. This path must always exists.

Examples:

```
{
  "$type": "DataInterviewAddRequest",
  "database": "db-main"
}
```

D.5.1.10 DataInterviewGet *Request the data of a single interview*

Get the full data of a single interview

Path	Flags	Type/Value	Description
`$type`	R	`"DataInterview Get"`	
`id`	R	*string*	

Flags:

- *R*: This field is required. This path must always exists.

Examples:

```
{
  "$type": "DataInterviewGet",
  "id": "#id"
}
```

D.5.1.11 DataInterviewListPersonRequest *Request the list of persons in an interview*

Request the full list of all persons that are assigned to an interview.

Path	Flags	Type/Value	Description
`$type`	R	`"DataInterview ListPerson Request"`	
`id`	R	*string*	

Flags:

- *R*: This field is required. This path must always exists.

Examples:

```
{
  "$type": "DataInterviewListPersonRequest",
  "id": "#id"
}
```

D.5.1.12 DataPersonAddRequest *Add Person Request*

Creates a new person entry at the server and fill with empty data. The server will respond with the created interview entry.

Path	Flags	Type/Value	Description
`$type`	R	`"DataPersonAdd Request"`	
`database`	R	*string*	
`interview`	R	*string*	
`role`	R	*string*	

Flags:

- *R*: This field is required. This path must always exists.

Examples:

```
{
  "$type": "DataPersonAddRequest",
  "database": "db-main",
  "interview": "#id",
  "role": "abc"
}
```

D.5.1.13 DataPersonGet *Request the data of a single person*

Get the full data of a single person with history of attributes

Path	Flags	Type/Value	Description
`$type`	R	`"DataPersonGet"`	
`id`	R	*string*	

Flags:

- *R*: This field is required. This path must always exists.

Examples:

```
{
  "$type": "DataPersonGet",
  "id": "#id"
}
```

D.5.1.14 DatabaseUnlock *Locks unlocked Database*

With `DatabaseUnlock` the Database transition in the unlocked state (if no master key was provided after the login process). The transition the database back to the safe locked state this message can be sent. To add new access again you have to send `DatabaseUnlock`.

Path	Flags	Type/Value	Description
`$type`	R	`"DatabaseUnlock"`	
`database`	R	*string*	

Flags:

- *R*: This field is required. This path must always exists.

Examples:

```
{
  "$type": "DatabaseUnlock",
  "database": "db-main"
}
```

D.5.1.15 EditAttributeSend *Send the edit of an attribute of an person.*

This sends the the edit of an attribute. If the attribute doesn't exists before it will be creates as long the schema of the user allows it.

Path	Flags	Type/Value	Description
`$type`	R	`"EditAttribute Send"`	
`id`	R	*string*	
`key`	R	*string*	
`value`	R	*string*, *number*, *bool*	

Flags:

- *R*: This field is required. This path must always exists.

Examples:

```
{
  "$type": "EditAttributeSend",
  "id": "#id",
  "key": "abc",
  "value": "abc"
}
```

```
{
  "$type": "EditAttributeSend",
  "id": "#id",
  "key": "abc",
```

```
  "value": 3.14
}
```

```
{
  "$type": "EditAttributeSend",
  "id": "#id",
  "key": "abc",
  "value": false
}
```

D.5.1.16 EditPersonSend *Send the edit of the person*

This sends the the edit of an person. The editable fields are optional. They are only updated if they are set.

Path	Flags	Type/Value	Description
`$type`	R	`"EditPersonSend"`	
`id`	R	*string*	
`delete`		*bool*	If set to true this will delete this person
`name`	N	*string*	
`contact`	N	*string*	

Flags:

- *R*: This field is required. This path must always exists.
- *N*: The value of this field can be `null`.

Examples:

```
{
  "$type": "EditPersonSend",
  "id": "#id"
}
```

D.5.1.17 FidoReceive *Receives Information from FIDO USB Token*

Sends the Information from the FIDO USB Token to the server

Path	Flags	Type/Value	Description
`$type`	R	`"FidoReceive"`	
`method`	R	*string*	The method that was used for the fido request
`value`	R		Arbitary JSON data that was provided from the FIDO USB token

Flags:

- *R*: This field is required. This path must always exists.

Examples:

```
{
  "$type": "FidoReceive",
  "method": "abc",
  "value": {
    "arbitary": "data"
  }
}
```

D.5.1.18 FileChangeConfidential *Change the confidential setting of a single file*

Change the confidential setting of a file or folder. If a file is moved to non-confidential all confidential parent directories are also moved to non-confidential to allow further access to this file after the the confidential database is removed. No other files are touched.

If a folder is moved to confidential all containing files and folders are also moved to confidential.

Path	Flags	Type/Value	Description
`$type`	R	`"FileChange Confidential"`	
`id`	R	*`string`*	
`confidential`	R	*`bool`*	

Flags:

- *R*: This field is required. This path must always exists.

Examples:

```
{
  "$type": "FileChangeConfidential",
  "id": "#id",
  "confidential": false
}
```

D.5.1.19 FileDelete *Deletes file or folder*

Deletes a file or a folder with all containing files. This will mark the files as deleted and they are not really removed from disk.

Path	Flags	Type/Value	Description
`$type`	R	`"FileDelete"`	
`id`	R	*`string`*	

Flags:

- *R*: This field is required. This path must always exists.

Examples:

```
{
  "$type": "FileDelete",
  "id": "#id"
}
```

D.5.1.20 FileFetch *Fetch a single file*

This will fetch a single file by its id

Path	Flags	Type/Value	Description
`$type`	R	`"FileFetch"`	
`id`	R	*string*	

Flags:

- *R*: This field is required. This path must always exists.

Examples:

```
{
  "$type": "FileFetch",
  "id": "#id"
}
```

D.5.1.21 FileMove *Move one file to another folder*

Changes the parent directory of one file or folder to a different one. Both the file/folder and the new target has to be attached to the same resource. No circular references are allowed.

Path	Flags	Type/Value	Description
`$type`	R	`"FileMove"`	
`id`	R	*string*	
`parent`	RN	*string*	

Flags:

- *R*: This field is required. This path must always exists.
- *N*: The value of this field can be `null`.

Examples:

```
{
  "$type": "FileMove",
  "id": "#id",
  "parent": "#id"
}
```

D.5.1.22 FilePersonFetch *Fetch a list of files from an person*

This will send the list of all files attached to an person. If the `root` field is not set it will send all files attached to the root (they have no parent files). Otherwise it will send the direct decendends of this root

Path	Flags	Type/Value	Description
`$type`	R	`"FilePersonFetch"`	
`person`	R	*string*	
`root`	N	*string*	

Flags:

- *R*: This field is required. This path must always exists.
- *N*: The value of this field can be `null`.

Examples:

```
{
  "$type": "FilePersonFetch",
  "person": "#id"
}
```

D.5.1.23 FileRename *Rename a file*

Renames an existing file. If the provided name is empty or already used this request is silently ignored. If its succeeds a `file-updated` message will be sent.

Path	Flags	Type/Value	Description
`$type`	R	`"FileRename"`	
`id`	R	*string*	
`name`	R	*string*	

Flags:

- *R*: This field is required. This path must always exists.

Examples:

```
{
  "$type": "FileRename",
  "id": "#id",
  "name": "abc"
}
```

D.5.1.24 InfoRequest *Request the current info from the server*

After the login the ui can request the current information from the server

Path	Flags	Type/Value	Description
`$type`	R	`"InfoRequest"`	

Flags:

- *R*: This field is required. This path must always exists.

Examples:

```
{
  "$type": "InfoRequest"
}
```

D.5.1.25 PerformReindex *Do a re-index of the database*

Path	Flags	Type/Value	Description
`$type`	R	`"PerformReindex"`	
`database`	R	*string*	

Flags:

- *R*: This field is required. This path must always exists.

Examples:

```
{
  "$type": "PerformReindex",
  "database": "db-main"
}
```

D.5.1.26 SchemaRequest *Requests the data of a schema*

The schema needs to be registered at the server before. It is recommended to cache the result because it is unlikely that it will change.

Path	Flags	Type/Value	Description
`$type`	R	`"SchemaRequest"`	
`name`	R	*string*	

Flags:

- *R*: This field is required. This path must always exists.

Examples:

```
{
  "$type": "SchemaRequest",
  "name": "abc"
}
```

D.5.1.27 SearchCancel *Cancel a search request*

If the user changes its query or leaves the page the search should be canceld. This is not necessary if the results are already finished.

The server will send a last `SearchResponse` message to notify the cancellation

Path	Flags	Type/Value	Description
`$type`	R	`"SearchCancel"`	
`query`	R	*string*	
`uiId`	N	*string*	This is a field that can be set by the UI to identify the messages later.

Flags:

- *R*: This field is required. This path must always exists.
- *N*: The value of this field can be `null`.

Examples:

```
{
  "$type": "SearchCancel",
  "query": "abc"
}
```

D.5.1.28 SearchSend *Send a new search request to the server*

The search query is a complex string that allows detailed sorting and filtering

Path	Flags	Type/Value	Description
`$type`	R	`"SearchSend"`	
`query`	R	*`string`*	
`uiId`	N	*`string`*	This is a field that can be set by the UI to identify the messages later.

Flags:

- *R*: This field is required. This path must always exists.
- *N*: The value of this field can be `null`.

Examples:

```
{
  "$type": "SearchSend",
  "query": "abc"
}
```

D.5.2 SUB `/ws/` Operation

The messages that are sent from the server to the UI

Accepts **one of** the following messages:

D.5.2.1 AccountFidoRequest *Sends a request for the FIDO token*

Path	Flags	Type/Value	Description
`$type`	R	`"AccountFido Request"`	
`command`	R	*`string`*	
`database`	R	*`string`*	
`value`	R		The arbitary data that is forwarded to the fido library

Flags:

- *R*: This field is required. This path must always exists.

Examples:

```
{
  "$type": "AccountFidoRequest",
  "command": "abc",
  "database": "db-main",
  "value": {
    "arbitary": "data"
  }
}
```

D.5.2.2 DatabaseAddRequestInfo *Sends info for db creation process*

This message sends the current state of the database creation step.

Path	Flags	Type/Value	Description
`$type`	R	`"DatabaseAdd RequestInfo"`	
`info`	R	*object*	
`info.key`	R	*string*	The key to identify this creation process (the can be multiple ones at the same time).
`info.database`	R	*string*	
`info.schema`	R	*string*	
`info.access`	R	*array*	
`info.access[]`	R	*object*	
`info.access[] .user`	R	*string*	
`info.access[] .user-id`	R	*string*, `<guid>`	
`info.access[] .fido`	R	*bool*	Tells if an FIDO authentification is used for this.
`info.access[] .created`	R	*string*, `<date-time>`	

Flags:

- *R*: This field is required. This path must always exists.

Examples:

```
{
  "$type": "DatabaseAddRequestInfo",
  "info": {
    "key": "abc",
    "database": "db-main",
```

```
    "schema": "abc",
    "access": [
      {
        "user": "abc",
        "user-id": "guid",
        "fido": false,
        "created": "2001-01-01T15:17:19.001"
      }
    ]
  }
}
```

D.5.2.3 DataInterviewAddSend *Add Interview send*

The answer of the server with the created interview

Path	Flags	Type/Value	Description
`$type`	R	`"DataInterviewAddSend"`	
`id`	R	*string*	

Flags:

- *R*: This field is required. This path must always exists.

Examples:

```
{
  "$type": "DataInterviewAddSend",
  "id": "#id"
}
```

D.5.2.4 DataInterviewListPersonSend *Send the list of persons in an interview*

This is the response to the `DataInterviewListPersonRequest` message. This will only contain a partial list. The ui has to merge the results.

There is no last message field. It is possible that responses are sent twice if two requests are sent at the same time.

Path	Flags	Type/Value	Description
`$type`	R	`"DataInterviewListPersonSend"`	
`id`	R	*string*	
`persons`	R	*array*	
`persons[]`	R	*object*	
`persons[].id`	R	*string*	
`persons[].role`	R	*string*	

Flags:

- *R*: This field is required. This path must always exists.

Examples:

```
{
  "$type": "DataInterviewListPersonSend",
  "id": "#id",
  "persons": [
    {
      "id": "#id",
      "role": "abc"
    }
  ]
}
```

D.5.2.5 DataInterviewSend *Sends the data of an interview*

This message is a response of `data-interview-get` and will provide the full interview. If no interview was found the `data` field will be `null`.

Path	Flags	Type/Value	Description
`$type`	R	`"DataInterview Send"`	
`id`	R	*string*	
`data`	RN	*object*	
`data.id`	R	*string*	
`data.database`	R	*string*	
`data.created`	R	*string*, `<date-time>`	
`data.modified`	R	*string*, `<date-time>`	
`data.deleted`	R	*bool*	
`data.time`	R	*string*, `<date-time>`	
`data.location`	R	*string*	
`data.interviewer`	R	*string*	

Flags:

- *R*: This field is required. This path must always exists.
- *N*: The value of this field can be `null`.

Examples:

```
{
  "$type": "DataInterviewSend",
  "id": "#id",
```

```
  "data": null
}
```

```
{
  "$type": "DataInterviewSend",
  "id": "#id",
  "data": {
    "id": "#id",
    "database": "db-main",
    "created": "2001-01-01T15:17:19.001",
    "modified": "2001-01-01T15:17:19.001",
    "deleted": false,
    "time": "2001-01-01T15:17:19.001",
    "location": "abc",
    "interviewer": "abc"
  }
}
```

D.5.2.6 DataPersonAddSend *Add Person send*

The answer of the server with the created person

Path	Flags	Type/Value	Description
$type	R	"DataPersonAdd Send"	
id	R	*string*	
data	RN	*object*	
data.id	R	*string*	
data.database	R	*string*	
data.created	R	*string*, <date-time>	
data.modified	R	*string*, <date-time>	
data.deleted	R	*bool*	
data.interview	N	*string*	
data.name	N	*string*	
data.altName	R	*string*	
data.contact	N	*string*	
data.role	R	*string*	
data.attributes	R	*array*	
data.attributes[]	R	*object*	
data.attributes[] .key	R	*string*	
data.attributes[] .created	R	*string*, <date-time>	
data.attributes[] .modified	R	*string*, <date-time>	

Path	Flags	Type/Value	Description
`data.attributes[].entries`	R	*array*	the complete history of this attribute
`data.attributes[].entries[]`	R	*object*	
`data.attributes[].entries[].date`	R	*string*, `<date-time>`	
`data.attributes[].entries[].value`	R	*string*, *int*, *number*, *bool*	
`data.attributes[].entry`	R	*object*	
`data.attributes[].entry.date`	R	*string*, `<date-time>`	
`data.attributes[].entry.value`	R	*string*, *int*, *number*, *bool*	

Flags:

- *R*: This field is required. This path must always exists.
- *N*: The value of this field can be `null`.

Examples:

```
{
  "$type": "DataPersonAddSend",
  "id": "#id",
  "data": null
}
```

```
{
  "$type": "DataPersonAddSend",
  "id": "#id",
  "data": {
    "id": "#id",
    "database": "db-main",
    "created": "2001-01-01T15:17:19.001",
    "modified": "2001-01-01T15:17:19.001",
    "deleted": false,
    "altName": "abc",
    "role": "abc",
    "attributes": [
      {
        "key": "abc",
        "created": "2001-01-01T15:17:19.001",
        "modified": "2001-01-01T15:17:19.001",
        "entries": [
          {
            "date": "2001-01-01T15:17:19.001",
            "value": "Lorem Ipsum"
```

```
          }
        ],
        "entry": {
          "date": "2001-01-01T15:17:19.001",
          "value": "Lorem Ipsum"
        }
      }
    ]
  }
}
```

D.5.2.7 DataPersonSend *Sends the data of an person*

This message is a response of `data-person-get` and will provide the full person. If no person was found the `data` field will be `null`.

Path	Flags	Type/Value	Description
`$type`	R	`"DataPersonSend"`	
`id`	R	*string*	
`data`	RN	*object*	
`data.id`	R	*string*	
`data.database`	R	*string*	
`data.created`	R	*string*, `<date-time>`	
`data.modified`	R	*string*, `<date-time>`	
`data.deleted`	R	*bool*	
`data.interview`	N	*string*	
`data.name`	N	*string*	
`data.altName`	R	*string*	
`data.contact`	N	*string*	
`data.role`	R	*string*	
`data.attributes`	R	*array*	
`data.attributes[]`	R	*object*	
`data.attributes[].key`	R	*string*	
`data.attributes[].created`	R	*string*, `<date-time>`	
`data.attributes[].modified`	R	*string*, `<date-time>`	
`data.attributes[].entries`	R	*array*	the complete history of this attribute
`data.attributes[].entries[]`	R	*object*	
`data.attributes[].entries[].date`	R	*string*, `<date-time>`	
`data.attributes[].entries[].value`	R	*string*, *int*, *number*, *bool*	

Path	Flags	Type/Value	Description
`data.attributes[].entry`	R	*object*	
`data.attributes[].entry.date`	R	*string*, `<date-time>`	
`data.attributes[].entry.value`	R	*string*, *int*, *number*, *bool*	

Flags:

- *R*: This field is required. This path must always exists.
- *N*: The value of this field can be `null`.

Examples:

```
{
  "$type": "DataPersonSend",
  "id": "#id",
  "data": null
}
```

```
{
  "$type": "DataPersonSend",
  "id": "#id",
  "data": {
    "id": "#id",
    "database": "db-main",
    "created": "2001-01-01T15:17:19.001",
    "modified": "2001-01-01T15:17:19.001",
    "deleted": false,
    "altName": "abc",
    "role": "abc",
    "attributes": [
      {
        "key": "abc",
        "created": "2001-01-01T15:17:19.001",
        "modified": "2001-01-01T15:17:19.001",
        "entries": [
          {
            "date": "2001-01-01T15:17:19.001",
            "value": "Lorem Ipsum"
          }
        ],
        "entry": {
          "date": "2001-01-01T15:17:19.001",
          "value": "Lorem Ipsum"
        }
      }
    ]
```

```
  }
}
```

D.5.2.8 DatabaseCreated *Database was created*

Sends that a single database creation process is finished and the database was created.

Path	Flags	Type/Value	Description
`$type`	R	`"DatabaseCreated"`	
`id`	R	*`string`*	The key of the database creation process

Flags:

- *R*: This field is required. This path must always exists.

Examples:

```
{
  "$type": "DatabaseCreated",
  "id": "#id-0123456789abcdef"
}
```

D.5.2.9 EditAccept *The successful answer of the server of an edit request*

After the client has sent an edit request this will be the answer if the edit was successful

Path	Flags	Type/Value	Description
`$type`	R	`"EditAccept"`	
`id`	R	*`string`*	
`key`	N	*`string`*	Is `null` if deleted
`target`	R	*`string`*, `"Interview"`, `"Person"`, `"Attribute"`	

Flags:

- *R*: This field is required. This path must always exists.
- *N*: The value of this field can be `null`.

Examples:

```
{
  "$type": "EditAccept",
  "id": "#id",
  "target": "Interview"
}
```

```
{
  "$type": "EditAccept",
```

```
  "id": "#id",
  "target": "Person"
}
```

```
{
  "$type": "EditAccept",
  "id": "#id",
  "target": "Attribute"
}
```

D.5.2.10 EditDenied *The edit cannot be done after an edit request*

After the client has sent an edit request which cant be fullfilled this will be the answer.

Path	Flags	Type/Value	Description
`$type`	R	`"EditDenied"`	
`id`	R	*`string`*	
`key`	N	*`string`*	Is `null` if deleted
`target`	R	*`string`*, `"Interview"`, `"Person"`, `"Attribute"`	
`reasonCode`	R	`"IdNotFound"`, `"SchemaNotFound"`, `"AttributeNot Available"`, `"Attribute Forbidden"`, `"InvalidValue Type"`, `"Generic"`	
`reason`	R	*`string`*	A simple description of this error in english.

Flags:

- *R*: This field is required. This path must always exists.
- *N*: The value of this field can be `null`.

Examples:

```
{
  "$type": "EditDenied",
  "id": "#id",
  "target": "Interview",
  "reasonCode": "IdNotFound",
  "reason": "abc"
}
```

```
{
  "$type": "EditDenied",
  "id": "#id",
  "target": "Person",
  "reasonCode": "IdNotFound",
  "reason": "abc"
}
```

```
{
  "$type": "EditDenied",
  "id": "#id",
  "target": "Attribute",
  "reasonCode": "IdNotFound",
  "reason": "abc"
}
```

D.5.2.11 FidoResult *Result of the FIDO request*

Sends the result of the FIDO request to the user

Path	Flags	Type/Value	Description
`$type`	R	`"FidoResult"`	
`is-success`	R	*`bool`*	
`error`	RN	*`string`*	The error message if success is false

Flags:

- *R*: This field is required. This path must always exists.
- *N*: The value of this field can be `null`.

Examples:

```
{
  "$type": "FidoResult",
  "is-success": false,
  "error": null
}
```

```
{
  "$type": "FidoResult",
  "is-success": false,
  "error": "abc"
}
```

D.5.2.12 FileMoved *Notify about moved files*

Notifies the UI about a moved file or folder.

Path	Flags	Type/Value	Description
`$type`	R	`"FileMoved"`	
`id`	R	*string*	
`database`	R	*string*	
`created`	R	*string*, `<date-time>`	
`modified`	R	*string*, `<date-time>`	
`deleted`	R	*bool*	
`person`	R	*string*	
`parent`	N	*string*	
`name`	R	*string*	
`mime`	R	*string*	The MIME type of this file. If this file is a folder the string `folder` is used instead.
`confidential`	R	*bool*	
`entries`	R	*array*	the complete history of this file. This is empty if its a folder
`entries[]`	R	*object*	
`entries[].date`	R	*string*, `<date-time>`	
`entries[].hash`	R	*string*	The Base64 encoded SHA512 Hash of the encrypted file
`entries[].exists`	R	*bool*	This flag describes if this file version is on the local disk available and ready for download.
`entry`	RN	*object*	the most reason file entry. This is empty for folders
`entry.date`	R	*string*, `<date-time>`	
`entry.hash`	R	*string*	The Base64 encoded SHA512 Hash of the encrypted file
`entry.exists`	R	*bool*	This flag describes if this file version is on the local disk available and ready for download.

Flags:

- *R*: This field is required. This path must always exists.
- *N*: The value of this field can be `null`.

Examples:

```
{
  "$type": "FileMoved",
  "id": "#id",
  "database": "db-main",
  "created": "2001-01-01T15:17:19.001",
  "modified": "2001-01-01T15:17:19.001",
  "deleted": false,
```

```
  "person": "#id",
  "name": "abc",
  "mime": "text/plain",
  "confidential": false,
  "entries": [
    {
      "date": "2001-01-01T15:17:19.001",
      "hash": "abc",
      "exists": false
    }
  ],
  "entry": null
}
```

```
{
  "$type": "FileMoved",
  "id": "#id",
  "database": "db-main",
  "created": "2001-01-01T15:17:19.001",
  "modified": "2001-01-01T15:17:19.001",
  "deleted": false,
  "person": "#id",
  "name": "abc",
  "mime": "text/plain",
  "confidential": false,
  "entries": [
    {
      "date": "2001-01-01T15:17:19.001",
      "hash": "abc",
      "exists": false
    }
  ],
  "entry": {
    "date": "2001-01-01T15:17:19.001",
    "hash": "abc",
    "exists": false
  }
}
```

D.5.2.13 FilePersonSend *Send the requested list of file of an person*

Send the list of files that the user has with `file-person-fetch` requested. If no files are found, the root or the person doesn't exists the list will be empty.

Path	Flags	Type/Value	Description
`$type`	R	`"FilePersonSend"`	
`person`	R	*`string`*	
`root`	RN	*`string`*	

Path	Flags	Type/Value	Description
`data`	R	*array*	
`data[]`	R	*object*	
`data[].id`	R	*string*	
`data[].database`	R	*string*	
`data[].created`	R	*string*, `<date-time>`	
`data[].modified`	R	*string*, `<date-time>`	
`data[].deleted`	R	*bool*	
`data[].person`	R	*string*	
`data[].parent`	N	*string*	
`data[].name`	R	*string*	
`data[].mime`	R	*string*	The MIME type of this file. If this file is a folder the string `folder` is used instead.
`data[].confidential`	R	*bool*	
`data[].entries`	R	*array*	the complete history of this file. This is empty if its a folder
`data[].entries[]`	R	*object*	
`data[].entries[].date`	R	*string*, `<date-time>`	
`data[].entries[].hash`	R	*string*	The Base64 encoded SHA512 Hash of the encrypted file
`data[].entries[].exists`	R	*bool*	This flag describes if this file version is on the local disk available and ready for download.
`data[].entry`	RN	*object*	the most reason file entry. This is empty for folders
`data[].entry.date`	R	*string*, `<date-time>`	
`data[].entry.hash`	R	*string*	The Base64 encoded SHA512 Hash of the encrypted file
`data[].entry.exists`	R	*bool*	This flag describes if this file version is on the local disk available and ready for download.

Flags:

- *R*: This field is required. This path must always exists.
- *N*: The value of this field can be `null`.

Examples:

```
{
  "$type": "FilePersonSend",
  "person": "#id",
  "root": "#id",
```

```
  "data": [
    {
      "id": "#id",
      "database": "db-main",
      "created": "2001-01-01T15:17:19.001",
      "modified": "2001-01-01T15:17:19.001",
      "deleted": false,
      "person": "#id",
      "name": "abc",
      "mime": "text/plain",
      "confidential": false,
      "entries": [
        {
          "date": "2001-01-01T15:17:19.001",
          "hash": "abc",
          "exists": false
        }
      ],
      "entry": null
    },
    {
      "id": "#id",
      "database": "db-main",
      "created": "2001-01-01T15:17:19.001",
      "modified": "2001-01-01T15:17:19.001",
      "deleted": false,
      "person": "#id",
      "name": "abc",
      "mime": "text/plain",
      "confidential": false,
      "entries": [
        {
          "date": "2001-01-01T15:17:19.001",
          "hash": "abc",
          "exists": false
        }
      ],
      "entry": {
        "date": "2001-01-01T15:17:19.001",
        "hash": "abc",
        "exists": false
      }
    }
  ]
}
```


D.5.2.14 FileSend *Send the requested info of a single file*

Sends the info of a single file if it was requested by an `file-fetch`. If this entry doesn't exists the `data` field will be `null`.

Path	Flags	Type/Value	Description
`$type`	R	`"FileSend"`	
`id`	R	*string*	
`data`	RN	*object*	
`data.id`	R	*string*	
`data.database`	R	*string*	
`data.created`	R	*string*, `<date-time>`	
`data.modified`	R	*string*, `<date-time>`	
`data.deleted`	R	*bool*	
`data.person`	R	*string*	
`data.parent`	N	*string*	
`data.name`	R	*string*	
`data.mime`	R	*string*	The MIME type of this file. If this file is a folder the string `folder` is used instead.
`data.confidential`	R	*bool*	
`data.entries`	R	*array*	the complete history of this file. This is empty if its a folder
`data.entries[]`	R	*object*	
`data.entries[].date`	R	*string*, `<date-time>`	
`data.entries[].hash`	R	*string*	The Base64 encoded SHA512 Hash of the encrypted file
`data.entries[].exists`	R	*bool*	This flag describes if this file version is on the local disk available and ready for download.
`data.entry`	RN	*object*	the most reason file entry. This is empty for folders
`data.entry.date`	R	*string*, `<date-time>`	
`data.entry.hash`	R	*string*	The Base64 encoded SHA512 Hash of the encrypted file
`data.entry.exists`	R	*bool*	This flag describes if this file version is on the local disk available and ready for download.

Flags:

- *R*: This field is required. This path must always exists.
- *N*: The value of this field can be `null`.

Examples:

```
{
  "$type": "FileSend",
  "id": "#id",
  "data": null
}
```

```
{
  "$type": "FileSend",
  "id": "#id",
  "data": {
    "id": "#id",
    "database": "db-main",
    "created": "2001-01-01T15:17:19.001",
    "modified": "2001-01-01T15:17:19.001",
    "deleted": false,
    "person": "#id",
    "name": "abc",
    "mime": "text/plain",
    "confidential": false,
    "entries": [
      {
        "date": "2001-01-01T15:17:19.001",
        "hash": "abc",
        "exists": false
      }
    ],
    "entry": null
  }
}
```

```
{
  "$type": "FileSend",
  "id": "#id",
  "data": {
    "id": "#id",
    "database": "db-main",
    "created": "2001-01-01T15:17:19.001",
    "modified": "2001-01-01T15:17:19.001",
    "deleted": false,
    "person": "#id",
    "name": "abc",
    "mime": "text/plain",
    "confidential": false,
    "entries": [
      {
        "date": "2001-01-01T15:17:19.001",
        "hash": "abc",
        "exists": false
      }
    ],
    "entry": {
      "date": "2001-01-01T15:17:19.001",
      "hash": "abc",
```

```
      "exists": false
    }
  }
}
```

D.5.2.15 FileUpdated *File was updated on the server*

If the user has updated a file with the REST Api the server will send this message after successfull upload.

Path	Flags	Type/Value	Description
`$type`	R	`"FileUpdated"`	
`id`	R	*string*	
`database`	R	*string*	
`created`	R	*string*, `<date-time>`	
`modified`	R	*string*, `<date-time>`	
`deleted`	R	*bool*	
`person`	R	*string*	
`parent`	N	*string*	
`name`	R	*string*	
`mime`	R	*string*	The MIME type of this file. If this file is a folder the string `folder` is used instead.
`confidential`	R	*bool*	
`entries`	R	*array*	the complete history of this file. This is empty if its a folder
`entries[]`	R	*object*	
`entries[].date`	R	*string*, `<date-time>`	
`entries[].hash`	R	*string*	The Base64 encoded SHA512 Hash of the encrypted file
`entries[].exists`	R	*bool*	This flag describes if this file version is on the local disk available and ready for download.
`entry`	RN	*object*	the most reason file entry. This is empty for folders
`entry.date`	R	*string*, `<date-time>`	
`entry.hash`	R	*string*	The Base64 encoded SHA512 Hash of the encrypted file
`entry.exists`	R	*bool*	This flag describes if this file version is on the local disk available and ready for download.

Flags:

- *R*: This field is required. This path must always exists.
- *N*: The value of this field can be `null`.

Examples:

```
{
  "$type": "FileUpdated",
  "id": "#id",
  "database": "db-main",
  "created": "2001-01-01T15:17:19.001",
  "modified": "2001-01-01T15:17:19.001",
  "deleted": false,
  "person": "#id",
  "name": "abc",
  "mime": "text/plain",
  "confidential": false,
  "entries": [
    {
      "date": "2001-01-01T15:17:19.001",
      "hash": "abc",
      "exists": false
    }
  ],
  "entry": null
}
```

```
{
  "$type": "FileUpdated",
  "id": "#id",
  "database": "db-main",
  "created": "2001-01-01T15:17:19.001",
  "modified": "2001-01-01T15:17:19.001",
  "deleted": false,
  "person": "#id",
  "name": "abc",
  "mime": "text/plain",
  "confidential": false,
  "entries": [
    {
      "date": "2001-01-01T15:17:19.001",
      "hash": "abc",
      "exists": false
    }
  ],
  "entry": {
    "date": "2001-01-01T15:17:19.001",
    "hash": "abc",
    "exists": false
```

```
  }
}
```

D.5.2.16 InfoSend *Send the current server info from the server to the ui*

This is send after the `info-request` message

Path	Flags	Type/Value	Description
`$type`	R	`"InfoSend"`	
`user`	R	*string*	
`database`	R	*array*	
`database[]`	R	*object*	
`database[].name`	R	*string*	
`database[].connected`	R	*bool*	
`database[].hasConfidential`	R	*bool*	This flag is only true if the database is connected and the confidential data is available
`database[].roleSchema`	R	*string*	
`database[].access`	R	*array*	
`database[].access[]`	R	*object*	
`database[].access[].user`	R	*string*	
`database[].access[].user-id`	R	*string*, `<guid>`	
`database[].access[].fido`	R	*bool*	Tells if an FIDO authentification is used for this.
`database[].access[].created`	R	*string*, `<date-time>`	
`database[].unlocked`	RN	*string*, `<date-time>`	The timestamp when this access was unlocked.
`schemas`	R	*array*	
`schemas[]`	R	*string*	
`uploadToken`	R	*string*	This upload token is used to verify the author of an upload to the REST Api. This token can be used for the following things: - upload new file - update file - download file

Flags:

- *R*: This field is required. This path must always exists.
- *N*: The value of this field can be `null`.

Examples:

```
{
  "user": "alice",
```

```
  "database": [
    {
      "name": "main",
      "connected": true
    },
    {
      "name": "bob",
      "connected": false
    }
  ],
  "uploadToken": "\u003Cinsert your token here\u003E"
}
```

D.5.2.17 LoginRequest *Request the UI to show the login box*

The server needs credentials and the user has to login

Path	Flags	Type/Value	Description
`$type`	R	`"LoginRequest"`	
`database`	R	*string*	

Flags:

- *R*: This field is required. This path must always exists.

Examples:

```
{
  "$type": "LoginRequest",
  "database": "db-main"
}
```

D.5.2.18 SchemaResponse *Sends the data of a schema*

This contains the data of the requested schema. If the name was invalid an empty schema will be returned.

Path	Flags	Type/Value	Description
`$type`	R	`"SchemaResponse"`	
`name`	R	*string*	
`schema`	R	*array*	
`schema[]`	R	*object*	
`schema[].name`	R	*string*	
`schema[].abstract`	R	*bool*	
`schema[].parent`	RN	*string*	
`schema[] .attributes`	R	*object*	
`schema[] .attributes.*`	A	*object*	

Path	Flags	Type/Value	Description

`schema[]`
`.attributes.*`
`.type`
	R	*`string`*	

`schema[]`
`.attributes.*`
`.protected`
	R	*`bool`*	

`schema[]`
`.attributes.*`
`.history`
	R	*`bool`*	

Flags:

- *R*: This field is required. This path must always exists.
- *N*: The value of this field can be `null`.
- *A*: This is an additional property. You can insert any not previously defined name to add your own properties. The schema of this property is predefined has to be used as above.

Examples:

```
{
  "$type": "SchemaResponse",
  "name": "abc",
  "schema": [
    {
      "name": "abc",
      "abstract": false,
      "parent": null,
      "attributes": {
        "*foo-bar-baz*": {
          "type": "abc",
          "protected": false,
          "history": false
        }
      }
    },
    {
      "name": "abc",
      "abstract": false,
      "parent": "abc",
      "attributes": {
        "*foo-bar-baz*": {
          "type": "abc",
          "protected": false,
          "history": false
        }
      }
    }
  ]
```

```
}
```

D.5.2.19 SearchResponse *Respond with the search result*

Response with a partial result. The server will continuesly send more of these frames if it has more data.

Path	Flags	Type/Value	Description
`$type`	R	`"SearchResponse"`	
`query`	R	*string*	
`uiId`		*string*	This is an exact copy of the `uiId` field of `SearchSend` and only set if it was set in the request.
`isLast`	R	*bool*	If this was the last message to this search query this field will be set `true`.
`order`	R	*int*	The internal message order of the server.
`elapsed-seconds`	R	*number*	The number of seconds since the start of the execution of the query
`sorting`	R	*array*	This object contains the necessary information for the ui to sort all incoming result entries. The entries are listed in their priority. Always sort according the first entry, if that matches according the second entry and so on. This array will never be empty.
`sorting[]`	R	*object*	
`sorting[].target`	R	`"None"`, `"Score"`, `"Attribute"`, `"Field"`	The kind of target that was sorted here
`sorting[].name`	RN	*string*	The name of the target
`sorting[] .direction`	R	`"Ascending"`, `"Descending"`	The direction that should be sorted for
`result`	R	*array*	
`result[]`	R	*object*	
`result[].id`	R	*string*	
`result[].database`	R	*string*	
`result[].type`	R	`"Interview"`, `"Person"`, `"File"`	
`result[].role`	RN	*string*	Only used if `type` is `Person`
`result[].title`	R	*string*	
`result[].sorting`	R	*array*	Contains the values for sorting in the same order as the provided search configuration.

Path	Flags	Type/Value	Description
`result[]` `.sorting[]`	RN	*string*, *number*	Contains the single value for the sortig. This supports the types string and number. If this field doesn't exists for this search result the value is null.
`result[].info`	R	*array*	
`result[].info[]`	R	*object*	
`result[].info[]` `.name`	R	*string*	
`result[].info[]` `.value`	R	*string*	
`result[].found`		*array*	
`result[].found[]`	R	*object*	
`result[].found[]` `.field`	R	*string*	
`result[].found[]` `.type`	R	`"Field"`, `"Attribute"`	
`result[].found[]` `.value`	R	*array*	
`result[].found[]` `.value[]`	R		
`result[].found[]` `.value[].<0>`	C	*object*	
`result[].found[]` `.value[].<0>.type`	R	`"Ellipsis"`	
`result[].found[]` `.value[].<1>`	C	*object*	
`result[].found[]` `.value[].<1>.type`	R	`"Context"`, `"Term"`	
`result[].found[]` `.value[].<1>` `.value`	R	*string*	

Flags:

- *R*: This field is required. This path must always exists.
- *N*: The value of this field can be `null`.
- *C*: This is one of multiple choices. You have to select one of the choices.

Examples:

```
{
  "$type": "SearchResponse",
  "query": "abc",
  "isLast": false,
  "order": 42,
  "elapsed-seconds": 3.14,
  "sorting": [
    {
```

```
      "target": "Field",
      "name": "Name",
      "direction": "Ascending"
    },
    {
      "target": "Score",
      "name": null,
      "direction": "Descending"
    }
  ],
  "result": [
    {
      "id": "#id",
      "database": "db-main",
      "type": "Person",
      "role": "Doctor",
      "title": "abc",
      "sorting": [
        null,
        "abc",
        3.14
      ],
      "info": [
        {
          "name": "abc",
          "value": "abc"
        }
      ]
    }
  ]
}
```

D.5.2.20 SendNotification *Sends a user notification*

Path	Flags	Type/Value	Description
`$type`	R	`"Send Notification"`	
`id`	RN	*string*	The notification id. That is only used if the server intent to update this notification in the future.
`img`	RN	*string*	The url of the icon that should be shown. `null` if no icon should be shown.
`title`	R	*string*	
`description`	R	*string*	
`progress`	R	*array*	
`progress.<0>`	C		Do not show progress at all
`progress.<0>[0]`	R	`"null"`	

Path	Flags	Type/Value	Description
`progress.<1>`	C		Show a progress but the state is undefined
`progress.<1>[0]`	R	`"undefined"`	
`progress.<2>`	C		Show a specific progress state. The value is a number between 0 and 1
`progress.<2>[0]`	R	`"progress"`	
`progress.<2>[1]`	R	*number*	
`close`	RN	*string*, `<date-time>`	The UTC timestamp when this notification should be closed. `null` if the notification should be persistent. The server will send an update later to remove it.

Flags:

- *R*: This field is required. This path must always exists.
- *N*: The value of this field can be `null`.
- *C*: This is one of multiple choices. You have to select one of the choices.

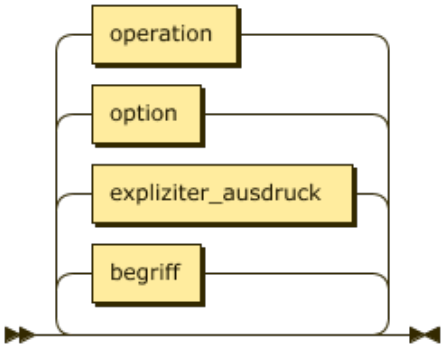
Examples:

```
{
  "$type": "SendNotification",
  "id": null,
  "img": null,
  "title": "Hello World",
  "description": "Have a nice day",
  "progress": [
    "null"
  ],
  "close": "2000-01-01T10:11:12"
}
```

```
{
  "$type": "SendNotification",
  "id": "download-1",
  "img": "/path/to/download/icon.png",
  "title": "Download",
  "description": "cat videos (50%)",
  "progress": [
    "progress",
    0.5
  ],
  "close": null
}
```

```
{
  "$type": "SendNotification",
  "id": "42",
  "img": null,
  "title": "Searching for the Answer of the ...",
  "description": "ultimate Question of Live, the Universe and Everything",
  "progress": [
    "undefined"
  ],
  "close": null
}
```

D.6 Suchquery



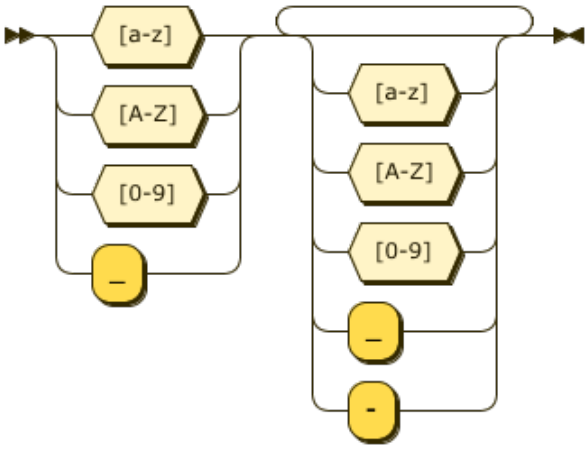
Die Anwendung erlaubt das Eingeben von speziell formatierten Suchanfragen. Diese werden in textueller Form von der Oberfläche aufgenommen, an den Server weitergeleitet, dort ausgewertet und dann die Ergebnisse zurück übermittelt. Diese textuellen Anfragen werden als Suchqueries oder auch kurz Query genannt.

Im folgenden Unterkapiteln wird auf dem Aufbau eines solchen Suchqueries genauer eingegangen.

D.6.1 Sonderzeichen

Sonderzeichen, sofern sie keinen speziellen Syntax bilden werden ignoriert und nicht in die Suche eingeschlossen.

D.6.2 Begriffe



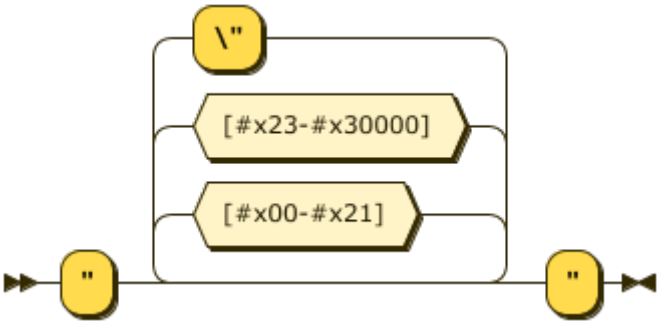
Begriffe werden mit einen Leerzeichen abgetrennt im Query definiert. Begriffe bestehen nur aus Zeichen der Zeichenklasse `\w`. Bindestriche dürfen zwischendrin vorkommen, aber nicht am Anfang. Groß- und Kleinschreibung wird ignoriert.

Es muss nur ein Begriff aus der Liste von Begriffen passen. Passen mehrere fällt das Ranking besser aus.

Beispiele:

- `Apfel` sucht nach allem, wo Apfel vorkommt
- `Apfel Banane` sucht nach allem, wo entweder Apfel oder Banane vorkommt. Wenn beides vorkommt, so erhält dies eine höhere Priorität

D.6.3 Explizite Ausdrücke

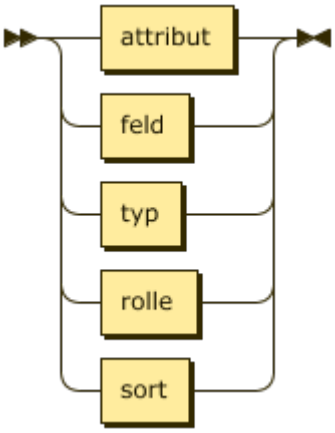


Explizite Ausdrücke werden von doppelten Anführungsstrichen `"` umschlossen. Alle Zeichen darin werden direkt gesucht und es wird nichts umformatiert. Anführungsstriche werden mit `\"` escaped.

Beispiele:

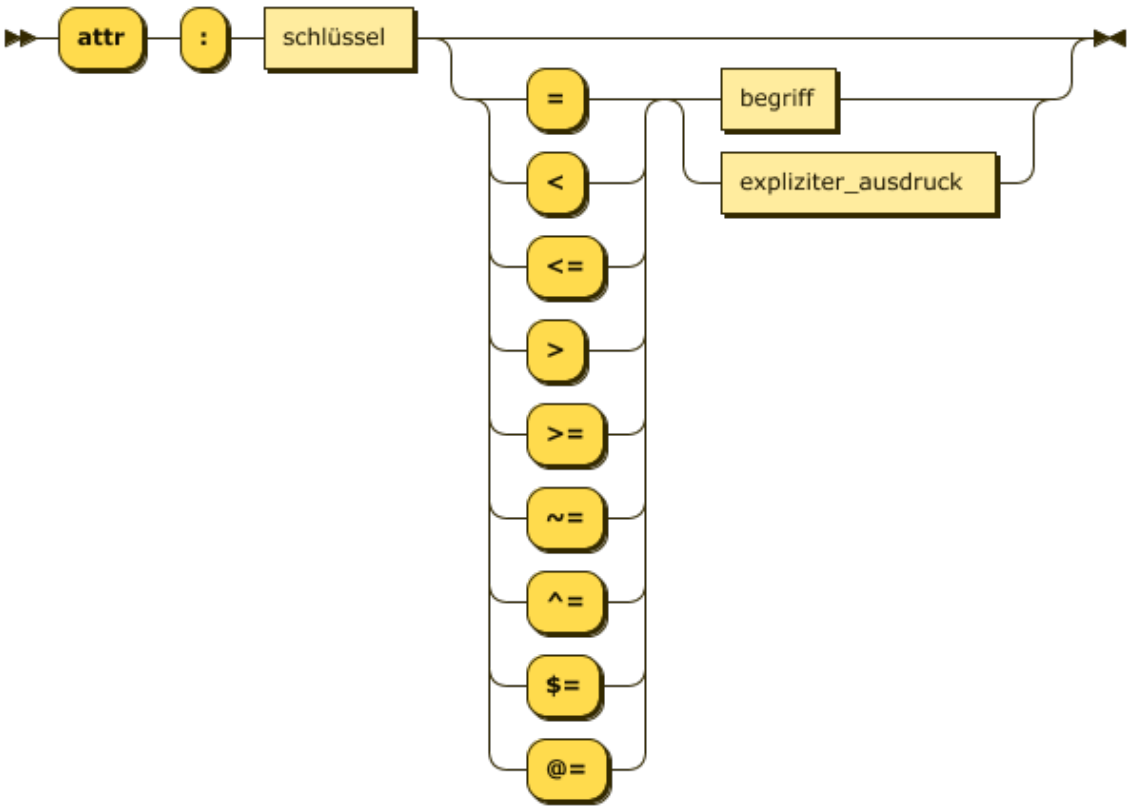
- `Äpfel, Banane"` sucht nach allem, wo der Substring “Apfel, Banane” in genau dieser Schreibung enthalten ist.

D.6.4 Optionen



Es gibt verschiedene Optionen. Alle Optionen sind mit den Schlüssel, einen Doppelpunkt und einen Wert versehen. Wenn Werte Leerzeichen enthalten, müssen diese mit doppelten Anführungsstrichen `"` umschlossen werden.

D.6.4.1 Attribute



Sucht direkt nach einen Attribut und vergleicht diesen.

> Attribute erzwingen den Typ `person`. Somit sind `type:person attr:age` und `attr:age` äquivalent.

Als Schlüssel für diese Operation gilt `attr`. Der Wert ist zweigeteilt. Er besteht aus den Attributsschlüssel, einen Vergleichsoperator und einen Wert.

Folgende Vergleichsoperatoren werden unterstützt:

Operator	Wertetyp	Beschreibung
`=`	alle	Das ganze Attribut muss den Wert entsprechen
`<=`, `<`, `>=`, `>`	`int`, `float`	Mathematischer Vergleichsoperator
`~=`	`string`	Attribut muss Wert enthalten
`^=`	`string`	Attribut muss mit Wert anfangen
`$=`	`string`	Attribut muss mit Wert aufhören
`@=`	`string`	Attribut muss einen Regulären Ausdruck genügen. Wenn dieser Anführungsstriche, Klammern oder Leerzeichen enthält, muss alles in Anführungsstriche gesetzt werden.

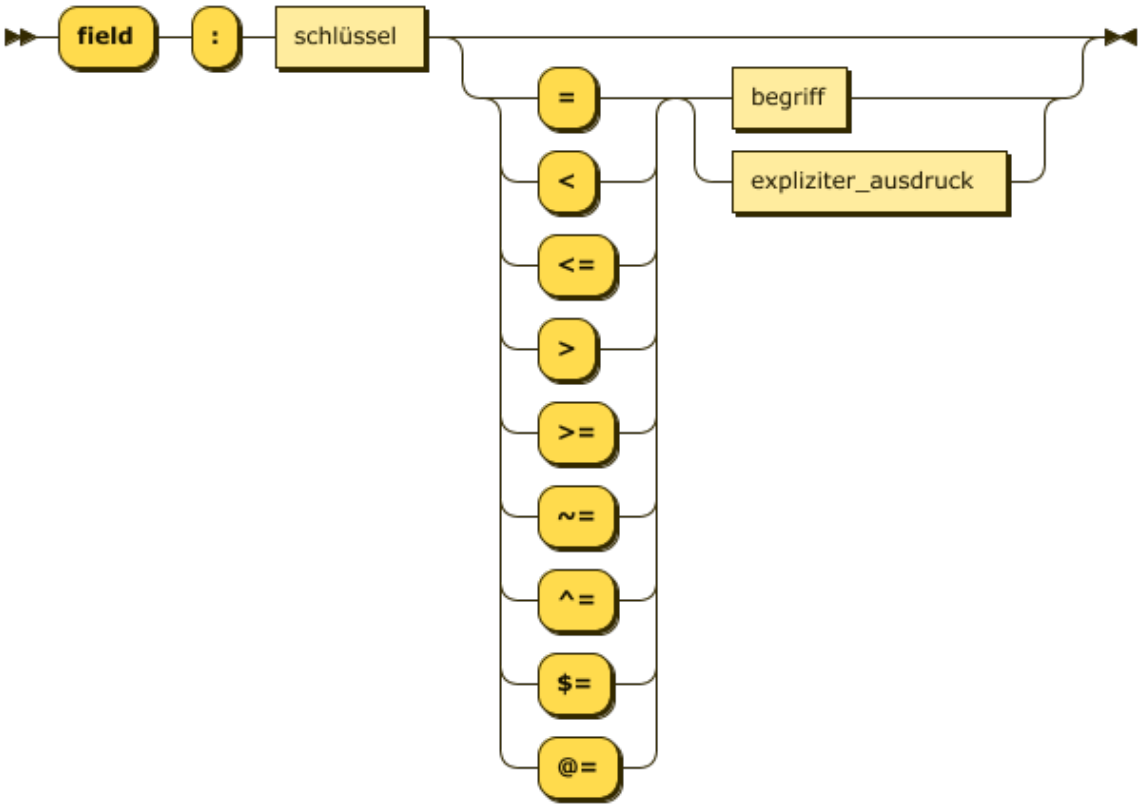
Es kann auch der Operator und Wert weggelassen werden, dann wird nur die Existenz des Attributs geprüft.

Beispiele:

- `attr:age>=5` Sucht nach allen Einträgen, dessen Alter gleich oder größer als 5 ist.
- `attr:opinion` Sucht nach allen Einträgen, die einen Wert für `opinion` haben
- `attr:hasConsent=true` Sucht nach allen Einträgen, die einen Consent hinterlegt haben.
- `attr:opinion@="do(es)?n't like m[ae]n"` Sucht nach allen Einträge mit einen Wert für `opinion` welcher einen bestimmten Regulären Ausdruck folgt.

Kann der Vergleichsoperator auf den Wert nicht angewandt werden, so schlägt dieser Teil immer fehl und trifft auf nichts zu. Dasselbe tritt auch ein, wenn der Reguläre Ausdruck fehlerhaft ist.

D.6.4.2 Felder



Sucht direkt nach einen Feld und vergleicht mit diesen.

Felder sind äquivalent so aufgebaut wie Attribute und arbeiten ähnlich, bieten aber die gleichen Optionen an.

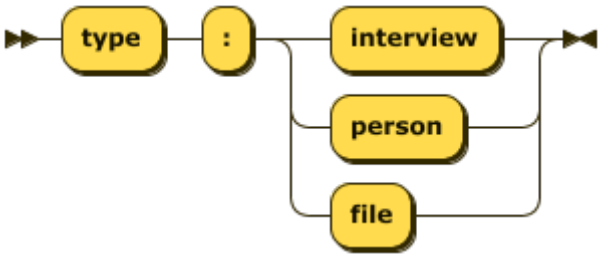
Felder können bei Interview und Person nur ausgewertet werden, wenn die Confidential Database verfügbar ist. Wenn nicht, schlägt dies immer zwingend fehl. Es können nur bestimmte Felder abgerufen werden:

- **Interview**: `time`, `location`, `interviewer`
- **Person**: `name`, `contact`
- **File**: `name`, `kind`

Beispiele:

- `field:name~=Ralf` Das Namensfeld enthält “Ralf”

D.6.4.3 Typ

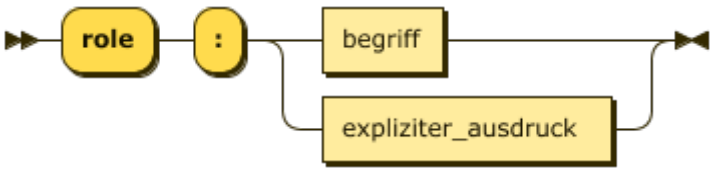


Sucht nur bestimmte Arten von Datensätzen ab.

Als Schlüssel gilt `type` und als Wert kann man einen von folgenden nutzen:

Wert	Beispiel	Beschreibung
`interview`	`type:interview`	Sucht nur nach Interviews
`person`	`type:person`	Sucht nach einen Interviewten
`file`	`type:file`	Sucht nach einer Datei (hier werden nur die Metadaten berücksichtigt)

D.6.4.4 Rolle



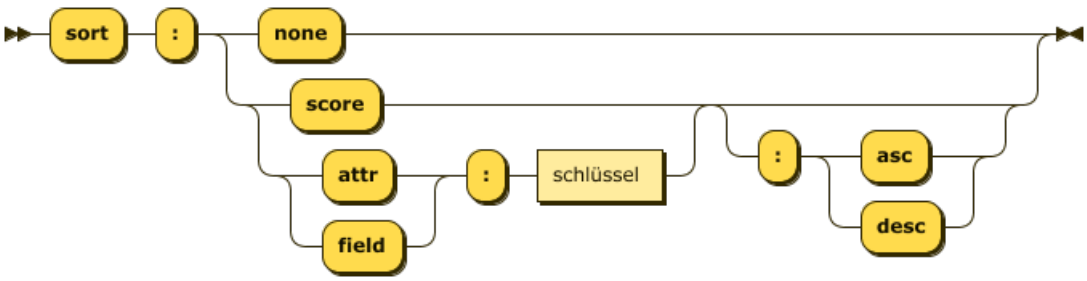
Sucht nur noch Interviewten, welche eine Rolle innehaben.

Als Schlüssel gilt `role` und als Wert wird der Schlüssel für die Rolle genutzt:

Beispiel:

- `role:doctor` Sucht nach allen Doktoren
- `role:family` Sucht nach allen Familienmitgliedern

D.6.4.5 Sortierung



Sortiert die Ergebnisse nach einen bestimmten Feld oder Score:

Sortierungen dürfen nur in der obersten Ebene definiert werden! Andernfalls werden diese ignoriert.

Als Schlüssel gilt hier `sort`. Der Wert ist in mehrere Teile geteilt, welche mit einen Doppelpunkt `:` abgetrennt sind.

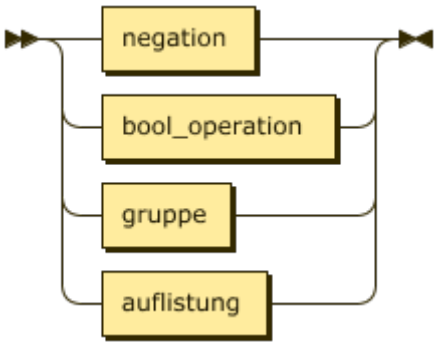
Als erstes kommt hier die Sorte, dann abhängig von der Sorte noch zusätzliche Optionen und zuletzt die Richtung.

Sorte	Extra Feld	Richtung	Beispiel	Beschreibung
`score`	*keins*	`desc`	`sort:score :desc`	Sortiert anhand des resultierenden Scores.
`attr`	Schlüssel	`asc`	`sort:attr :age:asc`	Sortiert anhand eines Attributes.
`field`	Schlüssel	`asc`	`sort:field :name:asc`	Sortiert anhand eines Standardfelds.
`none`	*keins*	*egal*	`sort:none`	Sortiert das Ergebnis nicht und gibt sofort alles aus. Dies ist relevant, wenn es schnelle Ergebnisse geht und keine großen Datenmengen erwartet werden.

Falls eine Richtung nicht angegeben wird, so wird die Standardmäßige Sortierung für die Sorte genommen.

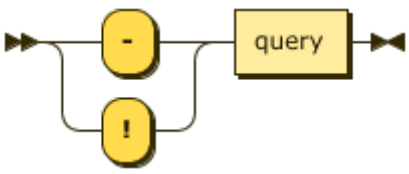
Hinweis: `field` ist bei Interview und Person sind nur verfügbar, wenn die Confidential Database verfügbar ist. Andernfalls wird diese Sortierung immer ignoriert.

D.6.5 Operatoren



Listen von Wörtern können mit Operatoren versehen werden, wodurch diese nur unter bestimmten Bedingungen erfüllt werden.

D.6.5.1 Negation



Ein Minus `-` direkt vor einen Ausdruck (es dürfen beliebig viele Leerzeichen zwischen Minus und Ausdruck stehen) negiert diese Behauptung. Nur solange der interne Ausdruck nicht zutrifft, gilt dies als erfolgreich.

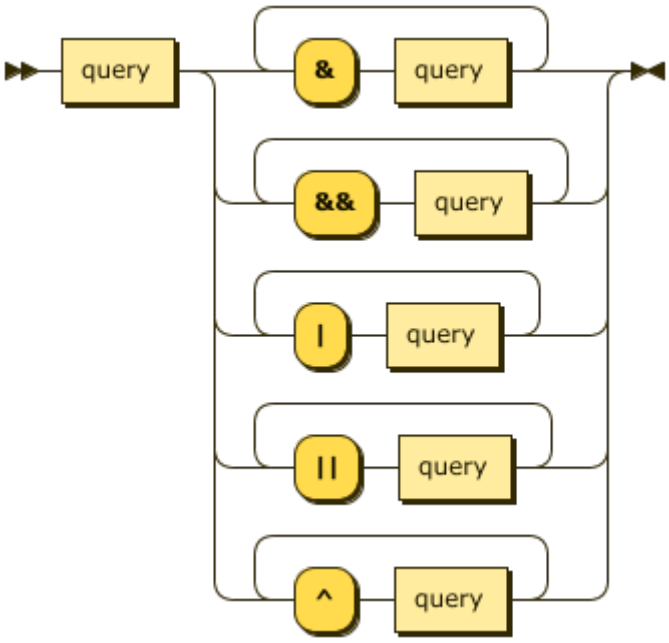
Alternativ zu einen Minus darf auch ein Ausrufezeichen `!` verwendet werden.

Folgenden mehrere Negationszeichen aufeinander werden diese solange aufgelöst, bis nur noch eins oder keins mehr übrig ist.

Beispiele:

- `-Apfel` sucht nach allem, was keinen Apfel enthält
- `Apfel - Banane` sucht nach allem was Apfel, aber keine Banane enthält.
- `-!--!Apfel` entspricht `-Apfel`

D.6.5.2 AND, OR, XOR



Erlaubt komplexere Ausdrücke, indem man spezielle Operatoren einbaut.

Operator	Beschreibung
`||` oder `|`	Der komplexe Ausdruck trifft nur zu, wenn eine von beiden Seiten zutrifft.
`&&` oder `&`	Der komplexe Ausdruck trifft nur zu, wenn beide Seiten zutreffen.
`^`	Der Ausdruck trifft nur zu, wenn entweder die linke Seite oder die rechte Seite zutrifft. Aber niemals beide gleichzeitig.

Die Operatoren dürfen gemischt werden, hierbei gilt aber folgende Priorität: Zuerst wird `^` ausgewertet, danach `&` und zum Schluss `|`. Falls eine andere Reihenfolge gewünscht ist, so sind Klammern zu verwenden.

Der Unterschied zwischen den einfachen und doppelten Operator ist beim Scoring. Beim doppelten wird nur der größte Score weitergegeben. Wobei beim einfachen die Summe genommen wird. Hierrüber lässt sich also die Priorität in den Suchergebnissen festlegen.

Beispiele:

- `Apfel && Banane` Sucht nach allem, wo Apfel **und** Banane gleichzeitig vorkommen.
- `role:doctor || Doctor` Sucht nach allen mit der Rolle Doktor oder wo das Wort Doktor irgendwo vorkommt.
- `Apfel ^ Banane` Sucht nach allem, wo entweder Apfel oder Banane vorkommt, aber niemals beides gleichzeitig.

D.6.5.2.1 Unterschiede bei der Sortierreihenfolge Man nehme an, wir haben folgende Texte, welche in der Datenbank existieren:

- `Apfel Kiwi Birne`
- `Apfel Kiwi`
- `Apfel Banane Birne Kirsche`
- `Apfel Kiwi Kirsche`

Diese werden je nach Query in einer unterschiedlichen Reihenfolge ausgegeben:

- `(apfel kiwi) | (birne | banane | kirsche)`:
  - `Apfel Banane Birne Kirsche`, Score: 4
  - `Apfel Kiwi Birne`, Score: 3
  - `Apfel Kiwi Kirsche`, Score: 3
  - `Apfel Kiwi`, Score: 2
- `(apfel kiwi) || (birne || banane || kirsche)`:
  - `Apfel Kiwi Birne`, Score: 2
  - `Apfel Kiwi`, Score: 2
  - `Apfel Kiwi Kirsche`, Score: 2
  - `Apfel Banane Birne Kirsche`, Score: 1

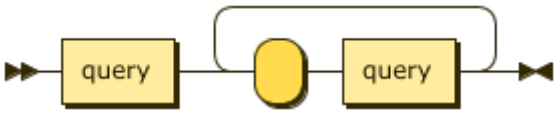
D.6.5.3 Gruppen

Runde Klammern `(` und `)` können Listen Wörtern gruppieren und somit komplexere Ausdrücke erstellen.

Falls es nicht genügend schließende Klammern gibt, so werden am Ende welche hinzugedacht. Gibt es zu viele schließende Klammern, werden die ignoriert.

Beispiele: - `Apfel && (Banane Kiwi)` Sucht nach allem, wo zwingend ein Apfel vorkommt und mindestens eins von Banane oder Kiwi

D.6.5.4 Auflistung



Wenn Ausdrücke nur von Leerzeichen begrenzt sind, so werden diese zu Listen zusammengefasst. Diese arbeiten nach speziellen Regeln:

1. Solange keine Optionen enthalten sind, so muss nur eins davon vorkommen. Dies entspräche der Logik, dass `Apfel Banane` und `Apfel | Banane` gleich arbeiten.

2. Wenn mehrere Ausdrücke in einer Auflistung zutreffen, so erhöht sich ihr Score. Bei einer einfachen Kette von `||` bleibt der Score erhalten.
3. Optionen werden in Auflistungen als zwingend angesehen. Die Auflistung kann somit nur noch erfolgreich passen, wenn diese Option erfolgreich sind. Die anderen Ausdrücke behalten ihre Logik bei.

   Somit entspräche `role:doctor Doctor House` `role:doctor && (Doctor House)`.
4. Bei der Abarbeitung werden die Optionen immer zuerst abgearbeitet. Erst dann werden sich die anderen Ausdrücke angeschaut. Hierbei werden auch Gruppen von Klammern beachtet.

   Somit entspräche `Doctor House role:doctor (attr:age)` `role:doctor && (Doctor House (attr:age))`.

Hinweis: Jeder Query ist in einer Auflistung eingeschlossen und auch ein leerer Query ist eine Auflistung. Eine leere Auflistung ist immer Erfolgslos.

D.6.6 Scoring

Es wird bei der Abarbeitung des Query's ein Score ermittelt. Hierbei gelten je nach Ausdruck folgende Regeln:

1. **Begriffe:** Wenn passen, dann `1.0` andernfalls `0.0`
2. **Optionen:** Erzwingt das Fehlschlagen oder Erfolgreich sein von Auflistungen.
3. **Operator:**
   1. **Negation:** Wenn interne Ausdruck großer als 0, dann ist das Ergebnis `0.0`, andernfalls `1.0`
   2. **AND, OR, XOR:**
      - Mit `&`, `|` und `^`: Ergebnis ist Summe von linken und rechten Ausdruck, wenn Ausdruck erfolgreich. Andernfalls `0.0`.
      - Mit `&&` und `||`: Ergebnis ist Maximum vom linken und rechten, wenn Ausdruck erfolgreich. Andernfalls `0.0`.
   3. **Gruppen:** Der Score ist immer der Score aus der internen Auflistung
   4. **Auflistung:**
      - Wenn einer der Attribute fehlschlägt: `0.0`
      - Wenn leer: `0.0`
      - Ansonsten Summe der enthaltenden Scoring

D.6.7 Query-Optimierung

Bevor der Query ausgewertet wird, wird sich die äußerste Auflistung angeschaut, ob bestimmte Abfragen schon auf der Ebene der Datenbank gemacht werden können. Hierbei werden zuerst die Queries nach folgenden Regeln umgebaut:

1. Wiederholte gleiche Operatoren, werden auf eine Hierarchie gebracht:
   - `apfel && (banane && kiwi)` entspricht `apfel && banane && kiwi`
2. Negation wird aufgelöst:
   - `-!-!-apfel` entspricht `-apfel`
   - `--apfel` entspricht `apfel`
3. Widersprüchliche Attribute werden aufgelöst:
   - `role:doctor role:family apfel` entspricht `()`

- `type:interview` wird zu `()` wenn die Confidential Database nicht mehr verfügbar ist.
- `type:interview attr:age` entspricht `()`
- `type:person attr:age` entspricht `attr:age`

4. Verundete Gruppen nur aus Attributen werden zusammengefasst:
   - `(role:doctor) && (attr:age=5)` entspricht `(role:doctor attr:age=5)`
5. Wenn eine Auflistung nur eine Gruppe enthält, so wird der Inhalt der Gruppe in die Auflistung gezogen:
   - `(((apfel banane)))` entspricht `apfel banane`
   - `(((role:doctor)))` entspricht `role:doctor`
   - `role:doctor ((attr:age))` entspricht `role:doctor (attr:age)`
6. Vorauswertung von booleschen Ausdrücken:
   - `apfel && ()` entspricht `()`
   - `apfel && -()` entspricht `apfel`
   - `apfel || ()` entspricht `apfel`

   Oder hier nocheinmal als übersichtliche Tabelle. `op` steht hierfür für den Operator.

Operator	`apfel op ()`	`apfel op -()`
`&`	`()`	`apfel & -()`
`&&`	`()`	`apfel`
`|`	`apfel`	`apfel | -()`
`||`	`apfel`	`apfel`
`^`	`apfel`	`-apfel`

Der Query wird auf unterschiedliche Tabellen ausgeführt, also wird der Query je nach Tabelle noch einmal extra umgebaut:

1. Wenn Confidential Database nicht verfügbar ist, wird folgendes optimiert:
   - `type:interview` zu `()` (immer)
   - `field:name` zu `()` (bei Person Tabelle)
2. Einschränkungen des Typs werden konkretisiert:
   - `type:interview` wird zu `()` bei einer Person Tabelle und zu `-()` bei einer Interview Tabelle
   - `attr:age` wird zu `()` bei einer Interview oder File Tabelle
   - `role:doctor` wird zu `()` bei einer Interview oder File Tabelle
3. Alle oben genannten Optimierungen werden nochmal geprüft
4. Sortieralternativen werden ausgewertet:
   - `(sort:field:name a | sort:attr:age b)` zu `sort:field:name (a | b)`
   - `(sort:field:name a || sort:attr:age b)` zu `sort:field:name (a || b)`
   - `(sort:field:name a & sort:attr:age b)` zu `sort:field:name sort:attr:age (a & b)`
   - `(sort:field:name a && sort:attr:age b)` zu `sort:field:name sort:attr:age (a && b)`
   - `(sort:field:name a ^ sort:attr:age b)` zu `a ^ b`
5. Unmögliche Sortierungen oder doppelte Sortierungen entfernt:
   - `sort:field:name:asc sort:field:name:desc a` zu `a`
   - `sort:field:name:asc sort:field:name:asc a` zu `sort:field:name:asc a`

Als nächstes werden sich die Optionen auf der obersten Ebene angeschaut. Diese können die Auswahl der Tabellen deutlich einschränken. Außerdem sind diese so strukturiert, dass

sich diese ohne weiteres problemlos auf der Datenbank ausführen lassen.

Danach werden sich die anderen Ausdrücke angeschaut. Für jeden wird separat nach folgenden Regeln entschieden, ob sich dieser auf der Datenbank ausführen lässt:

- Enthält in irgend einer Tiefe eine Option: Nein
- Enthält in irgend einer Tiefe eine Auflistung mit mehr als einen Ausdruck: Nein > Hier könnte es Probleme mit dem Scoring geben.
- Enthält in irgend einer Tiefe den Operator `&` oder `|`: Nein > Auch hier gibt es ein Scoringproblem.

Alles was sich auf der Datenbank ausführen lässt, wird auch auf dieser ausgeführt. Danach geht der Server die restlichen Ausdrücke durch, überprüft diese und berechnet einen Score.

Zum Schluss wird nach den Sortierregeln sortiert. Diese werden nur aus der obersten Ebene genommen und auch in der Reihenfolge abgehandelt. Falls keine Sortierung angegeben ist, so wird nach dem Score absteigend sortiert.

D.6.8 Beispiele:

Eine Sammlung von komplexen Beispielen:

- `attr:age>=30 attr:age<=50 (field:name sort:field:name || sort:field:age:desc)`

  Sucht sich alle Person zwischen 30 und 50 heraus. Solange die Confidential Datenbank noch verfügbar ist, wird nach Name sortiert, andernfalls nach dem Alter absteigend.

- `(type:interview field:location~=London && field:interviewer~=Bob) || attr:location~=Londyn`

  Sucht nach allen Interviews, welche von Bob in London geführt wurden, oder allen Personen, dessen `location` Attribut fälschlicherweise `Londyn` enthält.

- `sort:none We need a pretty long example with many words in it`

  Sucht nach allen Datensätzen, welches eins der angegebenen Wörter enthält. Die Sortierung `none` sorgt dafür, dass einfach sofort ausgegeben und nicht gewartet werden soll.

- `"mail@example.com" "http://localhost.de"`

  Sucht nach Feldern, wo eine spezifische Email-Adresse oder Link vorkommt. Da diese Sonderzeichen enthalten, müssen diese in doppelten Anführungsstrichen `"` gesetzt werden.