MARTIN-LUTHER-UNIVERSITÄT
HALLE-WITTENBERG

MAX-PLANCK-INSTITUT
FÜR ETHNOLOGISCHE FORSCHUNG

Planung & Entwicklung einer Datenbankanwendung für das Forschungsdatenmanagement

Bachelor für Informatik

Juni 2022

Student: Max Brauer

Erstgutachter: Dr. Stefan Brass

Zweitgutachter: Sebastian Ehser

Inhaltsverzeichnis

1 Einleitung

Diese Arbeit wurde in Zusammenarbeit mit dem Max-Planck-Institut für ethnologische Forschung (MPI) und dem 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 auf einen Stand zu bringen, der vergleichbar mit anderen Forschungseinrichtungen oder -gebieten ist.

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 auf einem Papierblock) 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 man hat 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 diesem war der zweite große Punkt, der in dieser Arbeit verfolgt wurde. Dem Forscher soll es möglich sein, seine Daten schnell, einfach und sicher eingeben können und sehen was und wann sich 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.

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

2 Anforderungen

Für das Projekt sind eine Liste an Anforderungen zu erfüllen. Dazu gehören inhaltliche, welche besagen was die Anwendung für den Forscher primär leisten soll. Dazu zählen auch die Daten, die aufgenommen werden.

Des weiteren müssen auch technische Anforderungen über das System, die Speicherdauer und Architektur erfüllt werden.

TODO:

- Inhalt (Was?)
- Technische Sachen
  - System
  - Speicherdauer (Fristen?)
  - Architektur
- Auswahl der Programmiersprachen, Datenbanken, Schnittstellen
- DDI Format

2.1 Inhaltliche Anforderungen

Die Anwendung sollen die Daten, welche bei der Forschungsreise und Interviews eines Forschers anfallen aufnehmen und übersichtlich darstellen und organisieren. Zu diesen Daten gehören 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 man diese überhaupt aufnehmen möchte oder darf.
- 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 Skitzen 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 entfernt und an einen sicheren Platz für eine spätere Einsicht aufbewahrt werden. Dies betrifft zum Beispiel den Klarnamen und Kontaktdaten des Betroffenen. Für die Aufnahme und Durchführung 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 Schnurbart trägt, . . . ”). Die

betroffene Person kann und darf später aber immer ankommen und die Löschung der eigenen Daten verlangen. Dafür braucht man wieder die Kontaktdaten und eine Zuordnung welche anonymisierte Daten dadurch betroffen sind. Von daher ist ein einfaches Löschen der Zuordnungen nicht möglich.

In der Praxis werden Zuordnungstabellen zu den Kontaktinformationen erstellt. Diese werden dann in einen sicheren 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 Nutzungszenario eine große Vielfalt an Bedingungen gestellt, die dies erfüllen muss:

1. Die Forscher sollen die Software auf ihren Forschungsreisen nutzen und da ist die Wahrscheinlichkeit sehr groß, dass es dabei zweitweise keinen Internetzugriff gibt. Von daher müssen alle Daten offline verfügbar sein. Dennoch können online Backups erstellt werden, sobald eine Internetverbindung wieder aufgebaut wird.
2. Außerdem kann man von sehr großen Datenmengen ausgehen. Die Forscher werden auf ihren Reisen eine größere Anzahl an Bildern und Videos aufnehmen. Von daher muss das Zielgerät einen größeren 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. Von 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, was vielleicht eine größere Verbreitung 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, und größerer internen Speicher, welchen sich aber die Hersteller gut bezahlen lassen.
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, wenn man größeren Mengen redet. 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 man hier auch externen 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 man spezielle Systeme und Oberflächen aufbaut, 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 Webplatform bereitzustellen. Dies beinhaltet leider das Problem, dass die Daten auch offline verfügbar sein müssen. Zwar kann man hier auch den Browser-Cache und -Speicher 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 Windows betriebener Laptop.

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 Synchronisationsproblemen kommen kann, wenn die gleiche Datenbank auf zwei Geräten offen ist und über die gleiche Cloud synchronisert 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 man ohne viel Änderungen an der Codebasis die Oberfläche auf verschiedenen Geräten und Systemen anwenden kann. Außerdem modularisiert dies die Codebasis mehr, was die Austauschbarkeit und Wartung verbessert.

Im Browser, der die Web-Oberfläche darstellt, gibt es derzeit zwei Technologien, die genutzt werden können, um Code auszuführen: JavaScript und WASM. JavaScript ist eine Scriptsprache, 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.

Für die Gestaltung der Web-Oberfläche gibt es eine Vielzahl an Werkzeugen und Systemen (siehe 10.3), 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 man

alle berücksichtigen würde. 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. Es gibt über Nuget eine große Sammlung an Bibliotheken, welche den Funktionsumfang deutlich vergrößern können.

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 Buildprozesse in WASM übersetzen und im Browser ausführen.

C/C++ ist der älteste Kandidat in dieser Runde und wurde 1985 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. Gleichzeitig wurde ein großes Augenmerk darauf gelegt, sehr sicher zu sein und viele Probleme, welche es in C/C++ gibt (u.a. Speicherzugriffe und -lebensdauer) nativ zu beheben. Der Compiler kann direkt in WASM übersetzen.

JavaScript (1995) und TypeScript (2012 von Microsoft) sind beides Sprachen, die direkt im Browser ausgeführt werden. Das Letztere ist ein Superset von JavaScript und es gibt einen Compiler, der ein paar Prüfungen vornimmt und dann in reines JavaScript umformt. TypeScript wurde notwendig, da JavaScript leider sehr wenig prüft und leicht fehleranfällig ist. Dadurch ist es auch recht umständlich den Code zu verwalten und zu warten.

Für JavaScript gibt es einige Frameworks, die die Entwicklung vereinfachen sollen, aber hier nicht weiter eingegangen wird.

Elm ist im gleichen Jahr wie TypeScript herausgekommen und ist rein funktional und an die Programmiersprache Haskell angelehnt. Hier hat man sich das Ziel gesetzt, eine äußerst sichere Programmiersprache für den Browser zu entwerfen, die keine Laufzeitfehler aufweist, da der Compiler alles im Vorfeld prüft und dann in optimierten JavaScript Code übersetzt.

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 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 flüssig im Browser funktionieren. [# TODO: Quellen mit Zahlen]
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.

Zumindest Punkt 1, 2 und 5 lässt sich bei allen mit "Ja" beantworten. Punkt 3 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 4 muss hier eine Unterscheidung getroffen werden. Das Problem ist hier WASM, was nur über eine virtuelle Maschine im Browser ausgeführt wird und man 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.

Hierfür wurde auch wieder der Vergleich der Programmiersprachen (siehe 10.3) wie in 2.3.1 zu Rate gezogen. Dabei gilt hier, dass das alles auf dem Computer des Nutzers und nicht im Web-Browser ausgeführt werden soll. Von daher ändert sich einiges, was im vorherigen Kapitel über WASM und JavaScript gesagt wurde. Alle genannten Programmiersprachen, die nach WASM übersetzen können direkt 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 man 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. Von daher eignen sich hier Programmiersprachen, die nicht darauf angewiesen sind.

Zwar ist es praktisch, wenn man ein Programmiersprache hat, die sehr maschinennah arbeitet, um den letzten Funken Geschwindigkeit rauszuholen, aber für dieses Projekt ist es auch in Ordnung, wenn man eine andere nimmt. 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 muss sie den Server auch erlauben schnell auf die Daten zuzugreifen und darüber suchen zu können.

Von daher wurden folgende Kriterien für die Auswahl der Datenbankenmanagementsysteme (DMS) festgelegt:

1. Sicherheit: Die Daten müssen verschlüsselt auf der Festplatte vorliegen. Auch der Schlüssel muss sicher sein.
2. Zugriff: Der Server braucht schnellen und unkomplizierten Zugriff auf Daten.
3. Backups: Sicherheitskopien müssen sich leicht erstellen lassen.
4. Angriffsoberfläche: Je kleiner desto besser.

5. Lizenzen: Die Lizenz muss mit dieser Arbeit kompatibel sein. Von daher ist es schwierig kommerzielle Lizenzen zu nutzen.

Für den Vergleich wurden MariaDB, MySQL, SQLite, MongoDB und LiteDB zu Rate gezogen (Details siehe 10.4). Es mag noch mehr Möglichkeiten geben, aber diese 5 sind die Gebräuchlichsten, wofür es Bibliotheken für C# gibt.

Punkt 1 ließ sich von keinen DMS außer LiteDB zufriedenstellend erfüllen. Zum Teil ist ein enormer Aufbau nötig, Schlüssel liegt unverschlüsselt auf der Festplatte, es müssen unbekannte Patches genutzt werden oder man kann nur 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 Entschlüsselungsprozess beinhalten.

Backups von einzelnen Datenbanken lassen sich bei Standalone DMS schwieriger anlegen. Hier sind die Daten z.T. an mehrere Orte verteilt und es gibt aufwändige Prozesse lokale Backups wiederherzustellen. Alternativ kann man auch alle Daten in ein allgemein lesbares Format (z.B. SQL) exportieren und später wieder importieren. Dies dauert aber in der Regel deutlich länger als eine einfache lokale Kopie der Daten. DMS, die im Prozess der Anwendung laufen, sind dagegen meist so gestrickt, dass es 1-2 lokale Dateien mit allen Daten existiert, welche einfach nur kopiert werden müssen.

Ein weiterer Nachteil bei Standalone DMS ist, dass die naturgemäß eine größere Angriffsoberfläche nach außen bietet, da sie eine Vielzahl von Nutzern Zugriff gewährt. In diesem Projekt wird eine lokale Datenbank benötigt auf die nur ein Nutzer gleichzeitig zugriff hat - und dies ist der Server. Von 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 man hier den WebBrowser selbst außen vor lässt) 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 versucht man 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 man sich zu halten, oder Richtlinien, welche man folgen sollte. 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 Kommision [3], welche zudem auf die Sicherheit und den Schutz der Forschungsdaten eingeht.

TODO:

- Grundsätze der IT-Sicherheit (Unterpunkte hier eingliedern)
  - Authenzität
  - Vertraulichkeit
  - Verfügbarkeit
  - Integrität
- Datenschutz
- Technische Basis ableiten
  - Warum
  - Vorteile
  - Alternativen (Nachteile)

3.1 Grundsätze der IT-Sicherheit

In der IT haben sich verschiedene Schutzziele [2, S. 6–11] etabliert. Dazu zählt die Vertraulichkeit, die Integrität, die Verfügbarkeit, die Authentizität, die Verbindlichkeit, die Zurechenbarkeit und die Resilienz. Was diese Begriffe bedeuten und deren Vergleich zu den BDSG und Richtlinien der Europäischen Kommision wird in den folgenden Unterkapiteln eingegangen.

3.1.1 Vertraulichkeit

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

Für eine Authorisierung 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 [5]), um einen möglichst guten Schutz erhalten.

Den Zugriff auf die gespeicherten Daten kann man 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 kann man z.B. mit den Rechtesystem des Betriebssystems erreichen.
- Daten vor Zugriff schützen: z.B. den kompletten Datenspeicher verschlüsseln und nur authorisierten 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 man darauf achten, 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" [4, 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 [4, 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" ([4, 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.

[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]

Ein weiteres Problem ist die “Manipulation von Hard- oder Software” ([4, 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.

Bei einer “Fehlfunktion von Geräten oder Systemen” ([4, 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 [4, 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.

TODO:

- Einpflegen von BSI G 0.46

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 man meist auf die Sorgfalt der Forscher angewiesen, da sich dafür kaum Vorbereitungen treffen lassen. Wofür sich Vorbereitungen treffen lassen ist die Wiederherstellung der Daten durch Backups. Indem man regelmäßig Backups erstellt kann man relativ schnell wieder 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. Man kann sich hier nur so abhelfen, dass man die Backupzeitfenster relativ kurz wählt, damit der Umfang an verlorenen Daten relativ klein ist.

Für dieses Projekt soll der Cloudspeicher der MPG namens Keeper genutzt werden. Es ist für Archivierungen ausgelegt und den Nutzern steht derzeit ein ausreichend großer Speicher von 1TB zur Verfügung.

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, somit stellt man 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. Von daher ist der Hintergrundserver so eingestellt, dass nur Anfragen vom gleichen Gerät angenommen werden. Somit versucht man 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 man 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.

TODO:

- Verschlüsselung der Datenbank ?
- Verschlüsselung der Dateien?
- Festplattenverschlüsselung, Luks, BitLocker?
- One-Time-Pad (Einmal Passwörter)?
- Authorisierung beim Start der GUI?
- 2-Faktor-Authentifizierung?

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:

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

3.3 Technische Basis

TODO:

- Warum
- Vorteile
- Alternativen (Nachteile)

4 Technisches System

TODO:

- Überblick
- Einzelkomponenten
  - Front-End
  - Back-End
  - Datenbank
  - Zusammenhalt
- Vor- und Nachteile bei ausgewählten Sachen, Vergleiche
- Api zu DDI (und/oder andere Standards)
  - Warum? Vorteile, Nachteile
- Anmeldung
  - Sicherheit
- Software-/Hardwarearchitektur
  - 4-Tier: Entwicklung, Staging, Test, Produktion

Unterrubrik:

- Metadatenstandards
  - Hergang zu DDI
    - Gibt es Schnittstellen, Hergang
  - DDI

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 mitlerweile 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 man auf Spezialanwendungen verzichten kann.

Als Programmiersprache für die UI hat der Autor die Sprache Elm gewählt. Besonders folgende Auspekte 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 gesammten Programm zu berücksichtigen.
- Die Programmiersprache ist funktional und hat keine Seiteneffekte. Dadurch lässt sich der Code leicht in kleinere, leicht verwaltbare Kompenten 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

Eine Datenbank enthält all ihre Einstellungen, Zugangsberechtigungen, Dateien und eingegebene Daten und Metadaten des Forschers. Diese wird verschlüsselt auf der Festplatte des Endgeräts hinterlegt und besteht in den meisten Fällen aus mehreren Dateien. Die Schlüssel selbst werden vom Server erzeugt.

4.1.4 Interne Schnittstellen

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. 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

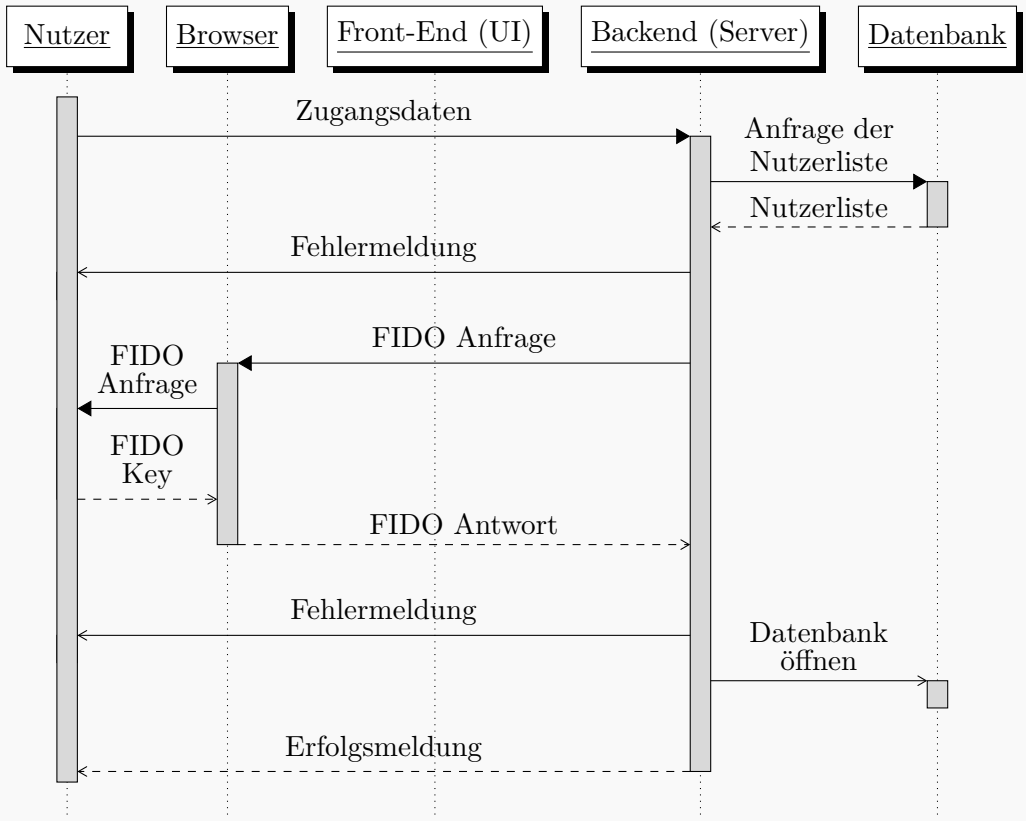
Um den Zugang der Daten zu anderen Anwendungen zu ermöglichen soll die Anwendung Schnittstellen bereitstellen.

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 Authenfizierung erfolgt in folgenden Schritten (vergleiche Abbildung 1):

Abbildung 1: 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.

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 verlangt.
8. Die Datenbank wird geöffnet.

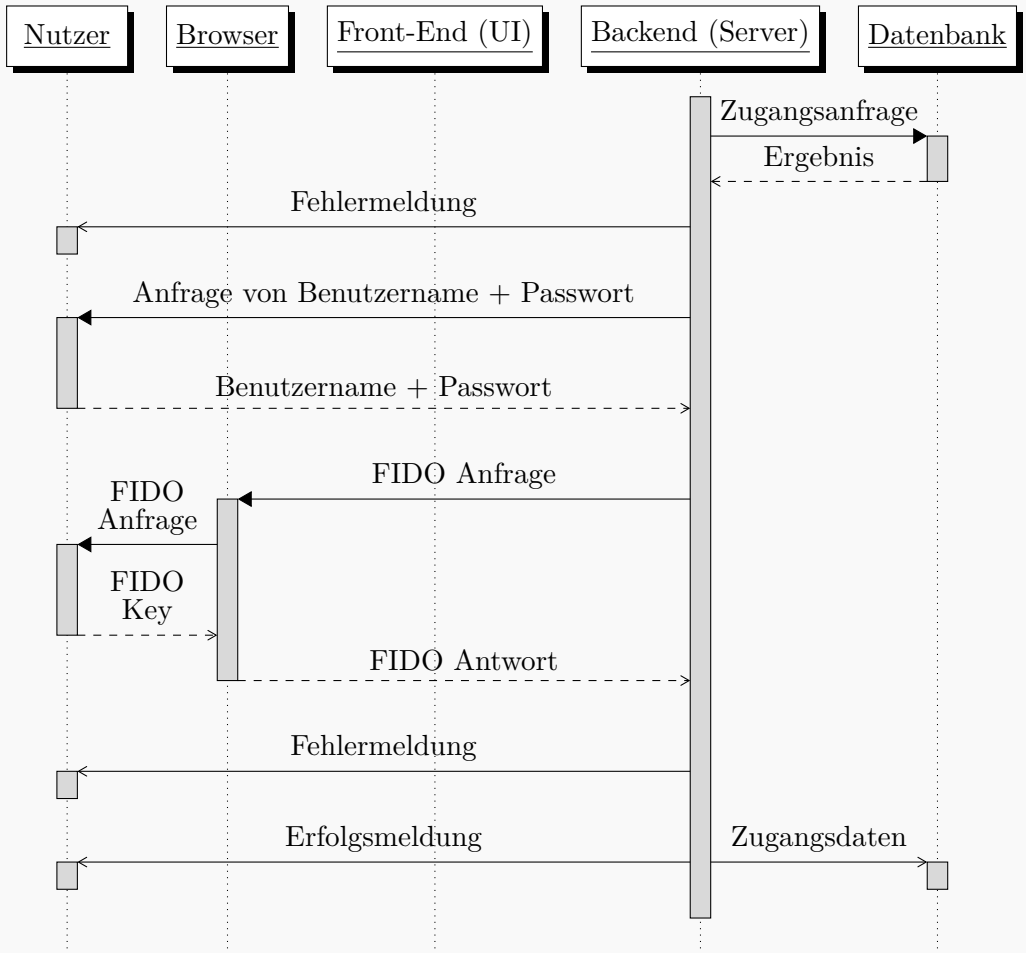
Es können in einer Anwendung mehrere Datenbanken hinterlegt werden, wo bei jeder Datenbank der Nutzer ein anderes Passwort oder FIDO-Key nutzen kann. Am Anfang wird mit jeder Datenbank verglichen und nach und nach werden die noch gültigen Datenbank ausgesiebt, welche noch damit angemeldet werden können. 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 2):

1. Zuerst wird geprüft, ob Zugang überhaupt besteht. Bei neuen Datenbanken ist dies implizit. Bei bestehenden muss man sich erneut anmelden, da der Server den Entschlüsselungskey 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.
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.

Abbildung 2: Erstellung neuer Datenbankzugangsdaten



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

5 Umsetzung

TODO:

- Stolpersteine, Probleme
  - Welche, Warum, Wie behoben
  - Technollogie basiert, Produkt basiert, ...

5.1 Stolpersteine

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 man 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, wenn man sämtliche Möglichkeiten der unterstützten Schlüsselalgorithmen berücksichtigen möchte (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 man schnell an Feedback gelang. 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

TODO:

- automatisierte Tests
  - Front-End, Integration, Test Suite, Browser-Test (Pupetier, . . .)
  - Back-End, Modultests, Funktionen
- Vor- und Nachteile, Was deckt man ab, Grenzen, unentdeckte Fehlerquellen
- Temporale Tests, weitere Testarten
- Abschätzung der Nutzungsdaten in der Zukunft

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.

TODO:

- Pipeline, Container, Installskript, Webseite, ...
- Installation

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 man 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 man folgende Produkte installieren bzw. Schritte durchführen musste:

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, indem man sie schon im Vorfeld compiliert und dann fertig auf den Zielrechner runterlädt. 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 man immer die gleichen Bedingungen (installierte Software, Konfiguration, etc.) hat und man daher genau sieht, 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 man alle notwendigen Dateien noch einmal zusammenfasst, damit sie besser für die Installation geeignet sind.

Einen weiteren Vorteil hat diese Vorgehensweise auch. So sieht man relativ früh, ob es Probleme beim Compilieren und Zusammenstellen gibt und das bevor man die Installation beim Nutzer durchführt. Außerdem kann man 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 kann man eine XML-Datei anlegen, 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. Von 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.

Man kann sich dies vereinfachen, indem man 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 arbeitet man hier mit einem Modell, 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 man schaut, ob das ganze Produkt noch funktioniert und ob es Fehler gibt. Falls hier Probleme auftreten, dann werden diese direkt zurück zum Entwickler kommuniziert. Üblicherweise arbeitet man hier mit Daten, 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.

TODO:

- nochmal genauer die Stages mit Quellen ausarbeiten

8 Ausblick

TODO:

- Nachnutzung
- Wartung
- Update
- Backup (Referenz zu Sicherheit)

8.1 Erweiterungen

8.1.1 Schnittstellen

TODO:

- DDI

DDI Code: In der ethnologischen Forschung haben sich die Datenaustauschformate der DDI Alliance etabliert. Diese stellt auf ihrer Webseite verschiedene Schemas für XML, JSON und andere Dateiformaten bereit, in der man seine Daten ex- und importieren kann.

---

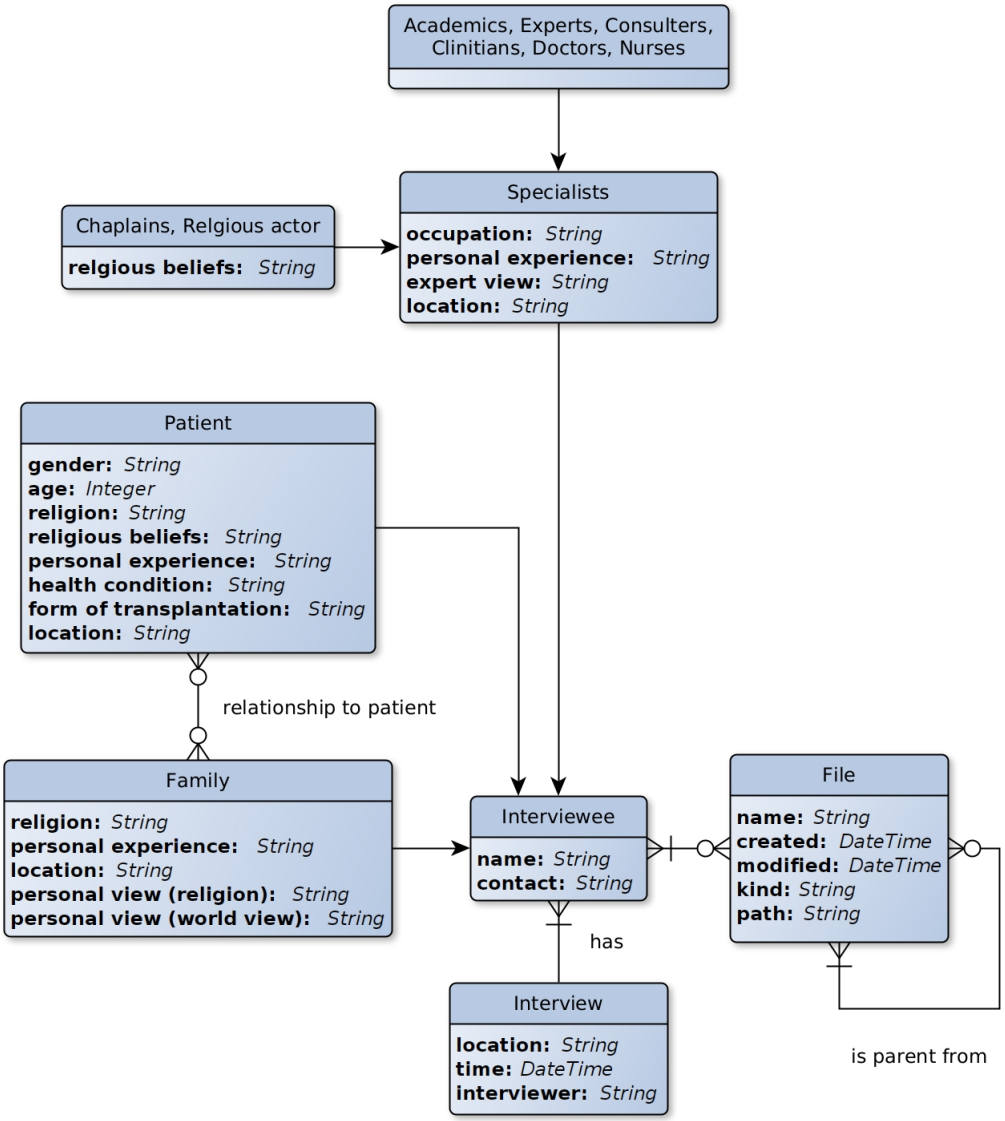
9 Abschlussbetrachtung

TODO:

- gezogene Schlüsse
- positive, negative Erkenntnisse

10 Anhang

10.1 Entity Relationship Diagramm



10.2 Projektsteckbrief

Projekttitel	Planung & 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		

10.3 Programmiersprachen

Sprache	C#	Java	C/C++	Rust	JavaScript	Elm	TypeScript
Erster Release	2001	1995	1985	2015	1995	2012	2012
Designer	Anders Hejlsberg	James Gosling, Sun Microsystems	Bjarne Stroustrup	Graydon Hoare	Brendan Eich	Evan Czaplicki	Microsoft
Entwickler	Microsoft	Sun Microsystems, Oracle	Bjarne Stroustrup	Mozilla, Graydon Hoare, Rust Fundation	Brendan Eich	Evan Czaplicki	Anders Hejlsberg, Microsoft
Zielsprache	IL	Java Byte Code	Maschinencode	Maschinen--code		JavaScript	JavaScript

10.3.1 Paradigmen

Quelle: Wikipedia

Sprache	C#	Java	C/C++	Rust	JavaScript	Elm	TypeScript
strukturiert	x		x	x			x
imperativ	x		x	x			x
deklarativ	x						
objektorientiert	x	x	x		x		x
ereignisorientiert	x						
funktional	x		x	x	x	x	x
generisch	x		x	x			
reflexiv	x						
parallel	x			x			
prozedural			x		x		
dynamisch typisiert					x		
prototypisch					x		x

10.3.2 Zielsystem

Sprache	C#	Java	C/C++	Rust	JavaScript	Elm	TypeScript
Server (Docker)	x	x	x	x	node	node	node
Browser	WASM		WASM	WASM	x	x	x

10.3.3 Verbreitung

Sprache	C#	Java	C/C++	Rust	JavaScript	Elm	TypeScript
PYPL-Index Oktober 2021	7,3%	17,18%	6,48%	1,08%	8,81%		1,91%
GitHub Repos (Stand: 19.10.2021)	1 010 441	3 830 935	1 823 839	47 611	4 464 883	9 564	353 697

PYPL Der Index wie oft eine Tutorial zu der Programmiersprache auf Google gesucht wurde.

10.4 Datenbankmanagementsysteme

Kategorie	MariaDB	MySQL	SQLite	MongoDB	LiteDB
Relational	x	x	x		
Standalone	x	x		x	
Lizenz	GPL 2.0	GPL 2.0 [2]	public domain	SSPL v1	MIT
Open Source	GitHub	GitHub	Sqlite.org (Fossil)	GitHub	GitHub
Preis	free	free [3]	free	free	free

10.4.1 lokale Verschlüsselung

Kategorie	MariaDB	MySQL	SQLite	MongoDB	LiteDB
komplette Datenbank	supported	nur in kostenpflichtiger Enterprise-Version	alternative	aufwändig und nur Felder	supported
einzelne Tabellen	supported		nur komplette Datenbank	nur Felder	nur komplette Datenbank
Engines	XtraDB, InnoDB, teilw. Aria		alle	alle	alle
Speicher der Schlüssel	Datei	zentral in einer Schlüsselverwaltung	frei	frei	frei
Overhead	3-5%		?	?	?
Methode	AES	AES	AES	AES	AES

[2] nur kostenlos, wenn alles unter GPL 2.0 veröffentlich wird. Dann ist MySQL auch unter GPL 2.0 verfügbar.

[3] nur kostenlos, wenn alles unter GPL 2.0 veröffentlich wird. Dann ist MySQL auch unter GPL 2.0 verfügbar.

10.4.2 Transportverschlüsselung

Kategorie	MariaDB	MySQL	SQLite	MongoDB	LiteDB
Art	TLS	TLS (bessere doc)	none (in process)	TLS	none (in process)

10.4.3 Authorisierung

Kategorie	MariaDB	MySQL	SQLite	MongoDB	LiteDB
Nutzer-authorisierung	x	x		x	
Mehre Accounts	x	x		x	
Rollen	x	x		x	
Rechte-management	detailiert	detailiert		detailiert	

10.4.4 Weiteres

Kategorie	MariaDB	MySQL	SQLite	MongoDB	LiteDB
Erster Release	29.10.2009	23.05.1995	17.08.2000	11.02.2009	17.09.2016
Roadmap für die nächsten 5 Jahre	Pläne für nächsten Release	-	Nächste Version	-	-
Herkunft (Neuentwicklung, Fork)	MySQL Fork	Neuentwicklung in Schweden, mehrfach verkauft	Neuentwicklung beim US Militär, Syntax an PostgreSQL angelehnt	Neuentwicklung bei 10gen	Neuentwicklung

10.4.5 lokale Verschlüsselung unter MariaDB

- eventuelle Backupprobleme, externe Programme können verschlüsselte Logs nicht lesen (mit Ausnahme von MariaDB Backup)
- Galera gcache ist unverschlüsselt in Community Version. In Enterprise Server ist es dennoch verschlüsselt. (Wird für Cluster benötigt)
- Dateibasierte `general query log` und `slow query log` sind unverschlüsselt
- `aria log` ist unverschlüsselt
- `error log` ist unverschlüsselt
- viele Konfigurationen müssen angepasst werden
- es muss ein Verschlüsselungsplugin installiert werden
- Verschlüsselungskeys liegen auf der Platte als Konfigurationsdatei
  - Die Keys selber können verschlüsselt werden, dafür muss der Schlüssel wieder als Datei vorliegen
  - Quelle: https://mariadb.com/kb/en/file-key-management-encryption-plugin/

10.4.6 Quellen

- MySQL Lizenzmodell: https://www.mysql.com/about/legal/licensing/oem/

11 Literatur

[1] Martin R. Albrecht. *Security Analysis of Telegram (Symmetric Part)*. 2021. URL: https://mtpsym.github.io/ (besucht am 23. 10. 2021).

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

[3] 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. 10. 2021).

[4] 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. 10. 2021).

[5] 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. 10. 2021).