ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften, Informatik und Psychologie
Institut für Datenbanken und Informationssysteme

Entwicklung und Umsetzung einer anforderungsgestützten API auf Grundlage einer medizinischen Datenbank

Bachelorarbeit an der Universität Ulm

Vorgelegt von:
Philipp Brieger
philipp.brieger@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert

Betreuer:
Prof. Dr. Rüdiger Pryss

2019

Fassung 6. Mai 2020

Satz: PDF-LaTeX 2$_\varepsilon$

Kurzfassung

Es besteht in der Informatik seit jeher ein großes Bedürfnis, Informationsnetzwerke aufzubauen und exklusive Daten einer größeren Anzahl an Interessenten zu erschließen. Es gilt Schnittstellen zu kreieren, die zum Einen dem technischen Anspruch gerecht werden, als auch zum Anderen in der Handhabung Effizienz über Komplexität stellen. Nicht einfach eine durchdringende Vernetzung ist das Ziel, sondern die Integration neuer Ideen und die sinnvolle Aufteilung von Arbeit und Segmenten der IT-Infrastruktur.

Die Tinnitus Datenbank der TRI (Tinnitus Research Initiative) ist eine medizinische Datenbank, die stark von einer Schnittstelle, in diesem Fall auch API genannt, profitieren würde und plant, sich dadurch kreativen Entwicklern und bisher unbeachteten Anwendungsmöglichkeiten zu öffnen. Dafür ist es notwendig, die Bedingungen der unterliegenden Datenbanktechnik zu überprüfen und Anwendungsgebiete in Betracht zu ziehen, die ohne API nicht realisierbar wären.

Der Gestaltung und Implementierung solch einer API ist die folgende Arbeit gewidmet. Erforderlich ist das Einbeziehen zeitgemäßer Techniken und eine Anpassung an die Ausgangslage, d. h. die derzeitigen Datenmodelle und eingesetzte Architektur. Letztendlich soll eine lauffähige API Wege aufzeigen, wie die zukünftige Arbeit mit der Tinnitus Datenbank aussehen könnte.

Danksagung

Ich danke hiermit allen Personen, die mich im Laufe der Arbeit unterstützt haben und mir mit Rat und Tat beiseite standen. Dieser Dank gebührt vor allem *Prof. Dr. Manfred Reichert* für die Begutachtung und *Prof. Dr. Rüdiger Pryss* für die Betreuung und tätliche Unterstützung bei allen möglichen Anliegen. Außerdem danke ich dem Institut für Datenbanken für die überaus freundliche Aufnahme. Des Weiteren sollen die kritischen Tipps und die ausdauernd hilfreichen Beiträge meiner Freundin nicht unerwähnt bleiben.

Inhaltsverzeichnis

1

Einleitung

Informationstechnologie lebt davon, Daten aufzubereiten, anzubieten und zu verarbeiten. Diese Grundstruktur ist in lokalen Rechnern, wie in kontinentüberspannenden Netzwerken vorzufinden. Spezialisierte Teile der Hard- und Software funktionieren in abgegrenzten Aufgabenbereichen und harmonieren als Gesamtarchitektur. Dabei nehmen Datenbanken die Rolle ein, Informationen zentralisiert zu speichern und zu verknüpfen. Sie sind Hilfsmittel, permanente Sammlung und oft Ausgangspunkt für größere Anwendungen. Die Tinnitus Datenbank der TRI (Tinnitus Research Initiative) ist solch eine Sammlung von detaillierten Untersuchungsergebnissen bezüglich des weitverbreiteten Phänomens Tinnitus. Sie steht in ihrem Einsatzbereich stellvertretend für weitere medizinische Datenbanken mit gleichem Anspruch an wissenschaftlichem Fortschritt und humanitärem Mehrwert.

Mit der Allgegenwart des Internets besteht die Möglichkeit, Entwickler und Ärzte in globalem Maßstab an einem Projekt, wie es die Tinnitus Datenbank ist, teilhaben zu lassen. Ziel ist es, das volle Potential der Software entfalten zu können, mit optionaler Smartphone-Unterstützung, wie auch mit komplexer Visualisierungssoftware. Schwer erfassbare Gruppen von Tinnitus Betroffenen sind somit besser ansprechbar und bereichern die Datenbasis [1].

Das fehlende Verbindungsglied zwischen Datenbank und vernetztem Anwender ist dabei die API, das *Application Programming Interface*. Die API sorgt auf der Software-Ebene für einen standardisierten, simplen und vielseitig einsetzbaren Zugriff auf die Funktionalitäten der Anwendung unter ihr. Sie überprüft Berechtigungen, arbeitet Daten ganz im Sinne einer zielgerichteten Kommunikation auf und ist das Fundament der modernen *Internet Economy* schlechthin.

1.1 Problemstellung

Die Tinnitus Datenbank ist in ihrem Funktionsumfang auf das Speichern und Abrufen von tinnitusspezifischen Untersuchungsdaten spezialisiert. Sie bietet dazu ein MariaDB Backend und ein Webseiten Frontend an. Registrierte Benutzer, insbesondere Ärzte und Patienten, können per Frontend Sitzungen anlegen, verfolgen und eine Vorauswahl standardisierter Fragebögen zur Diagnostik und Erfassung des Symptomverlaufs verwenden. Im Backend verwaltet der php-Server die Verbindung zur Webseite über frameworkabhängige Endpunkte und Sicherheitstechniken. Zugriff auf die gespeicherten Relationen und Einträge der MariaDB beschränkt sich auf Anfragen über den Server oder die lokale Datenbankinstanz. Der Ausbau und die Integration neuer Software-Komponenten ist damit nur auf Basis der bisher eingesetzten Sprache, des Codes und des Frameworks möglich.

Mit der Anbindung einer API wäre die Grundlage für eine dezentralisierte und technisch unabhängige Entwicklung geschaffen. Sie kann auf die Bedürfnisse künftiger Anwendungen zugeschnitten werden und umgeht als Schnittstelle alle Beschränkungen der bisherigen Plattform.

1.2 Zielsetzung

Die folgende Aufgabe wird sein, die Vorraussetzungen für eine passende Form der API-Implementation zu erörtern und diese Implementation in ausreichendem Umfang durchzuführen. Ausreichend meint damit die Mindestmenge an Funktionen zur Verfügung zu stellen, die notwendig ist, um sinnvolle Schlüsse zu Erfolg und Misserfolg im Vergleich zur theoretischen Ausarbeitung ziehen zu können. In der Erörterung inbegriffen ist, den Zweck des TinnitusDB-Projektes explizit hervorzuheben, ein Verständnis über das Datenbankmodell zu erlangen, sowie Referenz- und Technik-Recherche zu betreiben. Sind die Möglichkeiten der Umsetzung in Betracht gezogen worden, folgt daraus die Konzeption der API-Architektur als Lösung der angeführten Problemstellung.

1.3 Struktur der Arbeit

Nach dieser Einleitung mit Hinführung zum Thema und kurzer Aufschlüsselung des Ausgangsproblems, wird im nächsten Abschnitt näher auf die Tinnitus Datenbank eingegangen. Das Phänomen Tinnitus selbst wird vorgestellt und es wird aufgezeigt, welchen Stellenwert das Projekt hinsichtlich der Zukunft der Tinnitus Forschung besitzt. Im Anschluss kommt eine kurze Anforderungsanalyse mit Use Cases, sowie Auflistung der Software-Bedingungen. Der umfangreichste Abschnitt beschäftigt sich dann mit der Definition von APIs und welche Techniken für eine erfolgreiche Schnittstellenanbindung eingesetzt wurden und aktuell eingesetzt werden. Die Konzeption mit Überblick über die Realisation ist im vorletzten Kapitel beschrieben und wird von einer Diskussion, inwieweit die eingesetzten Mittel zum Ziel geführt haben, abgeschlossen.

2

Tinnitus Database

In den folgenden Abschnitten wird einzeln auf das Phänomen Tinnitus, die Hintergründe der Entstehung des Datenbankprojektes und genauere Details zur Technik der Datenbank eingegangen. Sie bilden die Grundlage für ein tieferes Verständnis, das für spätere Entscheidungen von Wichtigkeit ist.

2.1 Tinnitus

Der Begriff Tinnitus (H93.1 im ICD-10) bezeichnet „eine auditorische Empfindungsstörung, die Ausdruck einer veränderten Hörwahrnehmung ist“ [2] und mit Geräuschen ohne äußere Schallquelle in Verbindung gebracht wird. Die Art der Geräusche reicht von Pfeifen, Summen, über monoton-rhythmische Töne bis zu Klopfen und Pochen. Am Häufigsten wird ein andauernd hoher Ton auf einem der beiden Ohren beklagt, der bei Betroffenen Konzentrationsschwierigkeiten und Einschlafstörungen verursacht. Unterteilt wird das Phänomen zudem in einen subjektiven Tinnitus, bei dem keine Geräuschquelle festgestellt werden kann und einen objektiven Tinnitus, der durch Konvulsionen von Muskelgruppen oder Blutgefäßen entsteht, wobei Letzterer sehr selten auftritt und nur ca. 0.01% aller Tinnitus-Patienten betrifft.

Ursache für die am häufigsten auftretende Form, sind fehlerhafte Nervenaktivitäten in den auditorischen Teilen des Gehirns. Diese sind Folgeerscheinung von Hörstürzen, Knalltraumata, Tauchunfällen, (z. B. zu schnelle Dekompression) oder der Einnahme von ototoxischen Substanzen. Objektiv nachweisen lässt sich Tinnitus nur mit bildgebenden

Verfahren und der damit verbunden Darstellung abweichender neuronaler Aktivitäten. Die Theorie, den Tinnitus vor allem im Umfeld der Hörorgane, bzw. des Hörnervs, verorten zu können, ist inzwischen widerlegt.

Die Behandlung von Tinnitus erweist sich bislang als sehr schwierig. Der derzeit wichtigste Ansatz ist die Psychotherapie, bei der eine Linderung der Leiden und allgemeiner Umgang mit der Störung im Mittelpunkt steht. Für den Erfolg einer medikamentösen Behandlung gibt es bisher keine eindeutigen Belege. Auch gelten alternative Therapieformen, wie Klang- und Lasertherapien als wissenschaftlich nicht haltbar.

2.2 Das Datenbankprojekt

Die Tinnitus Datenbank entspringt einer Workgroup des TINNET Research Network und wird federführend von der Universität Ulm und der Universität Regensburg entwickelt [3]. Sie ist zudem als Projekt in der TRI (Tinnitus Research Initiative) eingegliedert [4]. Die TRI ist eine Stiftung, die es sich zur Aufgabe gemacht hat, die Entwicklung effektiver Behandlungsmethoden für alle Tinnitusformen zu fördern und damit zur Milderung der Leiden aller Betroffenen beizutragen. Finanzielle Unterstützung bekommt die Initiative als COST Action der EU [5].

Die Datenbank ist ein von der TRI 2008 gestartetes Projekt, das allen teilnehmenden Kliniken die Sammlung und den Zugriff auf einen einheitlichen Datenpool ermöglicht. Sie ist das erste Projekt seiner Art mit derzeit 19 Datenerhebungszentren in 11 Ländern. Das Ziel ist der Vergleich, die Subtypisierung und dadurch die erleichterte Suche nach neuen und effektiveren Therapieansätzen.

Als theoretische Grundlage der Diagnostik existiert ein ausführlicher Leitfaden der TRI und für die Gewährleistung einer validierbaren Datenerhebung zudem ein Consensus-Statut [6]. Die Schnittpunkte zur Datenbank sind dabei die im Consensus empfohlenen, standardisierten Fragebögen und Untersuchungsfragen.

2.3 Questionnaires & Examinations

Ziel und Mittelpunkt der Datenbank ist ein standardisierter, auswertbarer Datensatz, der Einblicke in die unterschiedlichsten Aspekte der Störung liefern kann und darüber hinaus sinnvolle Querbezüge ermöglicht. Zu diesem Zweck steht eine Sammlung etablierter wissenschaftlicher Fragebögen zur Verfügung, deren Relevanz in Kombination nachträglich bestätigt wurde [7].

Die Kategorisierung der einzelnen Bögen ist nicht ganz unproblematisch, da sich, je nach Verwendung, verschiedene, teils überschneidende, Zuordnungen ergeben haben. Am Wichtigsten, wenn auch sehr ungenau, ist der Consensus der TRI, mit einer Einteilung in *Essential* (**A**: z. B. THI, THQ), *Highly Recommended* (**B**: z. B. Tinnitus Severity) und *Might be of Interest* (**C**: z. B. MDI).

Die Datenbank-Software ist spezifischer und teilt in Visiten und Questionnaire ein und erzwingt für eine auswertbare Erhebung den speziell konzipierten TSCHQ und einen Mindestsatz mit abschließender Catamnesis durch TQ, bzw. CGI. Allerdings verbleibt die Wahl zwischen klassischen Verfahren, wie THI und moderneren, wie den neueren ESIT-SQ ohne Wertung [8].

Nachfolgend die vollständige Auflistung, logisch unterteilt nach Relevanz und Verwendungszweck:

Primär:

- ESIT-School Screening Questionnaire
- Tinnitus Sample Case History Questionnaire
- Tinnitus Functional Index
- Tinnitus Handicap Inventory
- Tinnitus-Beeinträchtigungs-Fragebogen (TBF-12)
- Tinnitus Questionnaire
- Tinnitus Severity

- Geräuschüberempfindlichkeits-Fragebogen
- Tinnitus Empowerment Skala
- Audiological Examination
- Clinical Global Impression

Sekundär:

- Big Five Inventory 2
- Perceived Health Competence Scale
- General Self Efficiency Scale
- WHO Quality of Life BREF
- Beck Depression Inventory
- Major Depression Inventory

Standardisierte Visiten-Abfragen:

- Comorbidity
- Concomittant
- Non Pharmalogical
- Adverse Event

2.4 Datenbankstruktur

Die aktuelle Implementation der Datenbank beruht auf den Erfordernissen einer Datenerhebung während einer Patientenuntersuchung. Eine Weboberfläche ermöglicht Dateneingabe und Benutzermanagement, während ein Server sich um Auswertung und Speicherung kümmert. Diese Programmstruktur spiegelt sich in Tabellen für Nutzer, Berechtigungen und Patienten wieder.

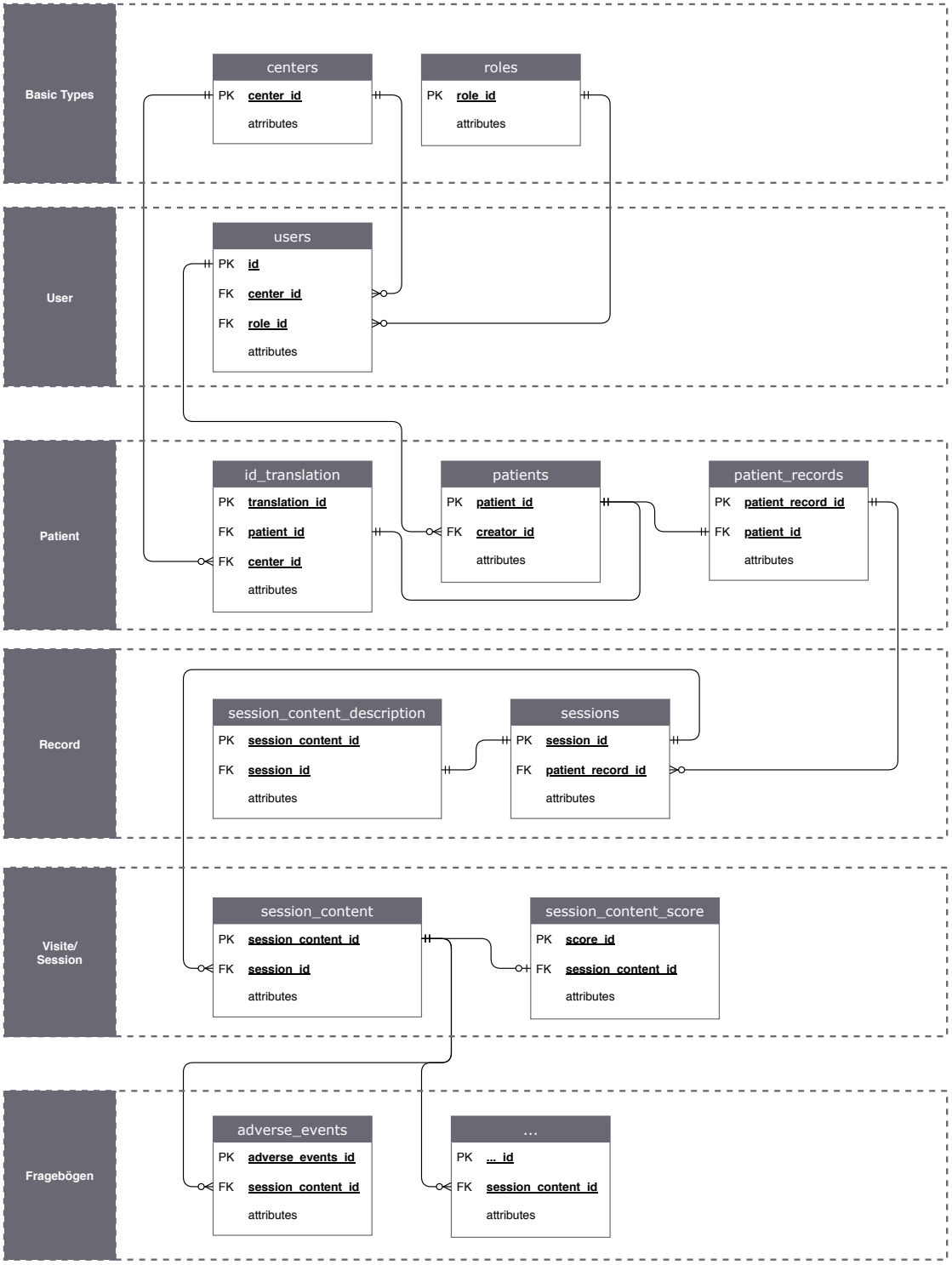

Abbildung 2.1: Relationenmodell der Datenbank

Darüber hinaus ist der Ablauf, bzw. die Einbettung der Untersuchung – innerhalb der Software *Session* genannt – wichtig. Ein kompletter Datensatz pro Patient wird als Record identifiziert, das wiederum einzelne Sitzungen mit variabler Zahl an Befragungen enthält. Im Sinne einer Hierarchie sind alle Daten den Erhebungszentren untergeordnet.

Die hier gezeigte Abbildung 2.1 enthält die wichtigsten Relationen und Entitäten. Ausgelassen wurden die separaten Module Export und Surveys sowie einzelne Tabellen, die implementierungsspezifische Eigenschaften speichern und aus dem Code auslagern.

2.5 Eingesetzte Technik

Die Architektur hinter der Weboberfläche wurde mit dem php-Framework *Laravel* (V5.5) realisiert. Laravel ist mit Stand 2019 das meistgenutzte php-Framework und wird von Stackoverflow auf Platz 10 der beliebtesten Frameworktechniken gelistet [9]. Ähnlich Angular oder Spring, arbeitet im Hintergrund ein Dependency Injection-System und das Angebot an Klassen deckt den kompletten Stack einer Web-App ab. Für das Speichern und Lesen der Daten wird *MariaDB* eingesetzt, das ehemalige MySQL.

Das Frontend ist in *ECMA-Script* geschrieben und benutzt *Babel* als Compiler, bzw. Transpiler. Die Bibliotheken für die Hauptfunktionalitäten sind *AdminLTE* und *DataTables*, wobei die Kommunikation mit dem Backend über die Laravel-eigene Scriptsprache *Blade* initiiert wird. Als Stylesheet-Language für die weitere Individualisierung, dient der CSS Ableger *SASS*.

3

Anforderungsanalyse

Die Anforderungsanalyse geht auf die möglichen Use Cases ein und versucht daraus, Kriterien für die API-Software herzuleiten sowie deren Dringlichkeit zu klassifizieren. Dieser Abschnitt kommt noch vor dem theoretischen Teil, um die Erfassung der Aufgabenstellung abzuschließen und Entscheidungen, z. B. welches Paradigma umgesetzt wurde, nachvollziehbar zu machen. Die Auflistung der Kriterien dient letztendlich als Orientierung bezüglich des Funktionsumfangs der Implementation.

3.1 Use Cases

Die Use Cases dienen als Vorbereitung der API-Spezifikation und entwerfen ein kurzes Benutzerszenario. Das Design der API orientiert sich somit an den Anforderungen dieser fiktiven Anwendungsfälle und sollte immer in Hinsicht einer späteren Nutzung begründet werden.

Software-Entwickler sind aus dieser Perspektive Nutzer ersten Grades. Sie stellen Ansprüche an Integrität, Stabilität und Simplizität und versuchen mit Hilfe der API wiederum eigene Zielgruppen anzusprechen. API-Schema Konformität wird meist gefordert und Daten, die ein konkreter Anwendungsfall voraussetzt, sollten zwingend und einfach abrufbar sein. Im Idealfall nimmt die API den Verwendungszweck schon vorweg, ohne aber weitere Möglichkeiten einzuschränken.

Use Case 1	
Attribut	**Beschreibung**
Rolle	App-Developer
Zielgruppe	Patienten & Ärzte
Anwendungsfall	Smartphone-App für Patienten zur Überwachung ihres Behandlungsstatus und Teilnahme an Befragungen, ohne die Notwendigkeit einen privilegierten User (Arzt) für den Datenbankzugriff aufsuchen zu müssen.
Bevorzugte Techniken und Entwicklungsumgebungen	Angular, Electron, Laravel, php, Typescript, JSON
Anforderungen an die API	Übersichtliches und einheitliches Datenschema, das alle Standardfälle abdeckt und schnellen, unkomplizierten Zugriff ermöglicht.
Passendes Paradigma	REST / RPC-Funktionen

Tabelle 3.1: Use Case 1

Bei den Use Cases übernimmt der erste Use Case die Rolle eines Standart Anwendungsfalles. Die Entwicklung einer Mobile-App steht bei der Verwendung einer API immer im Mittelpunkt und hat bei derzeitigen Anbietern und Entwicklern höchste Priorität [10]. Ziel ist, mit wenig Aufwand und bekannten Schemata, unkonventionelle Ideen zu testen und schnelle Ergebnisse zu erzielen. Der zweite Use Case entspricht den Vorstellungen, die man von den zukünftigen Features der Tinnitus Datenbank hat und ist konkret damit verknüpft. Die Bereitschaft, sich in neue Techniken einzuarbeiten, ist höher, da meist nur aufwendige Alternativen vorliegen und die Realisation der Ideen längerfristig geplant ist.

Use Case 2	
Attribut	**Beschreibung**
Rolle	Backend-/Frontend-Developer
Zielgruppe	Ärzte & Statistiker

Anwendungsfall	Von der Weboberfläche unabhängiges Modul zur statistischen und grafischen Auswertung der Befragungsdaten.
Bevorzugte Techniken und Entwicklungsumgebungen	Charts.js, D3, JavaScript, JSON
Anforderungen an die API	Flexible und detaillierte Queries mit geringem Overhead.
Passendes Paradigma	GraphQL / RPC-QueryLanguage

Tabelle 3.2: Use Case 2

Die weiteren Kapitel konzentrieren sich auf Use Case 2. Aufgrund der geplanten, noch notwendigen Module, profitiert das TDB-Projekt am meisten von einer leistungsfähigen Schnittstelle, die gezielt große, individuelle Datenmengen übertragen kann.

3.2 Funktionale Anforderungen

In der Anforderungsanalyse wird ein IT-Projekt von den Vorstellungen und Bedingungen der Interessenvertreter (engl. *Stakeholder*) geformt. Jeder dieser Vertreter hat berechtigte Interessen, die in Funktionale und Nichtfunktionale Anforderungen münden. Neben dieser gebräuchlichen Aufteilung gibt es ebenfalls noch DIN und ISO Modelle mit größerem Augenmerk auf Qualitätsmerkmalen, ähnlich den nichtfunktionalen Anforderungen. [11].

Funktionale Anforderungen definieren dabei die benötigten Funktionalitäten [12]. Diese können nach Interessengruppe, bzw. Abstraktionsgrad, nochmals in Organisations-, Systems- und Bausteinsanforderungen aufgeteilt werden. Da der Umfang der Datenbank-API überschaubar bleibt, wird sich hier allgemein auf Systemanforderungen beschränkt.

Name	Beschreibung	Priorität
Query-Anfrage	Die API soll ein individuelles Daten-Objekt bereitstellen, das vorher per Query angefragt wurde	Hoch
Datenbankabdeckung	Die API sollte alle Tabelleninhalte der Benutzer, Patienten, Zentren, Rollen und Fragebögen in einer verwendbaren Form anbieten	Hoch
JSON-Response	Datensatz im standardisierten JSON-Format	Hoch
Beschreibende Fehlercodes	Bei erfolgloser Anfrage wird eine Antwort in JSON mit ausreichender Exception-Beschreibung zurück geschickt	Hoch
Authentication	Die API soll mittels OAuth oder Token eine Nutzerauthentifizierung durchführen	Hoch
Authorization	Die API soll einem Nutzer bestimmte Rechte mit eigenem Zugriffsbereich innerhalb der Datenbank zuweisen können	Mittel
Cache	Ein Zwischenspeicher sollte mehrfach angefragte Ergebnisse schnell ohne Datenbankzugriff erneut anbieten	Mittel
Rate Limiting	Ein Rate Limiting-Verfahren sollte die Anfragen eines Clients pro Zeitabschnitt begrenzen	Mittel
Versionierung	Die API soll ein Versionierungsschema erhalten, das anzeigt, ob Anfragen einer älteren API-Version mit einer neueren Version kompatibel sind	Mittel
Pagination	Um mit großen Datensätzen umgehen zu können, muss ein Seiteneinteilungsschema implementiert sein	Hoch
Dokumentation	Der Schnittstelle sollte eine ausführliche Dokumentation zur Seite gestellt werde	Mittel
SDK	Es wird eine Programmbibliothek für Client-Entwicklung zur Verfügung gestellt	
Software-Tests	Die Unit-Test-Abdeckung der API sollte mindestens 80% betragen	Mittel

Tabelle 3.3: Funktionale Anforderungen

3.3 Nichtfunktionale Anforderungen

Nichtfunktionale Anforderungen verkörpern Erwartungen und Notwendigkeiten, die die Qualität und Eigenschaften der Funktionalitäten sowie der gesamten Benutzererfahrung betreffen. Sie gelten als genauso Ausschlag gebend für die Güte einer Software, wie die Funktionalen Anforderungen. Vorherrschende Definitionen sind trotz allem nicht immer eindeutig. Daher bezieht sich die beiliegende Auflistung auf keinen offiziellen Standard.

Leistungsanforderungen		
Name	**Beschreibung**	**Metrik**
Reaktionszeit	Bei der Anfrage sollte die Antwort der API innerhalb von höchstens 2 Sekunden erfolgen. Wird die Wartezeit überschritten, ist von einer Störung der Servertechnik auszugehen. Eine valide Antwort erfolgt auch bei Fehlern im definierten Protokoll-Schema.	Millisekunden/ Anfrage
Stabilität & Last	Die API sollte eine größere Menge gleichzeitiger Anfragen verarbeiten können. In jedem Fall mindestens 1000.	Anfragen/ Minute
Qualitätsanforderungen		
Standardisierung	Alle angebotenen GraphQL-Schnittstellen sind einheitlich bezeichnet und folgen, sofern sinnvoll, den vom Schema empfohlenen Standards.	
Simplizität & Genügsamkeit	Die Kommunikation des relationalen Datenmodells über die API erfolgt so einfach wie möglich, bei Beachtung aller notwendigen Attribute.	

Tabelle 3.4: Nicht-Funktionale Anforderungen

4

Technische Grundlagen

Das folgende Kapitel erörtert die theoretischen Grundlagen und eingesetzten Techniken, wie sie für eine API benötigt werden. Es wird kurz auf die Entwicklungsgeschichte eingegangen und maßgebende Eigenschaften genannt. Referenz und Ursprung bekannter Techniken ist oft proprietärer Natur (z. B. **GraphQL** - *Facebook*), eine wissenschaftliche Aufarbeitung unterliegender Konzepte fand aber immer schon im Vorfeld im universitären Rahmen statt. Dem Zweck dieser Arbeit angemessen, werden nur die wichtigsten Verweise vorgenommen.

Zur Visualisierung dient noch eine Serie von Diagrammen, mit Priorität auf den Komponenten des jeweiligen Unterkapitels. Die hier gezeigte Ausgangsgraphik zeigt das Komplett-Modell einer Web-Architektur. Der Umfang der (hier nicht sichtbaren) Authentifikations- und Scaling-Maßnahmen ist vom unterliegenden Server abhängig und kann, aber muss nicht, durch den API-Code gesteuert werden. Auf Darstellungen der Systeme unterhalb der OSI-Anwendungsschicht, also Transport-Protokolle und Routing, wird wegen geringer Relevanz verzichtet.

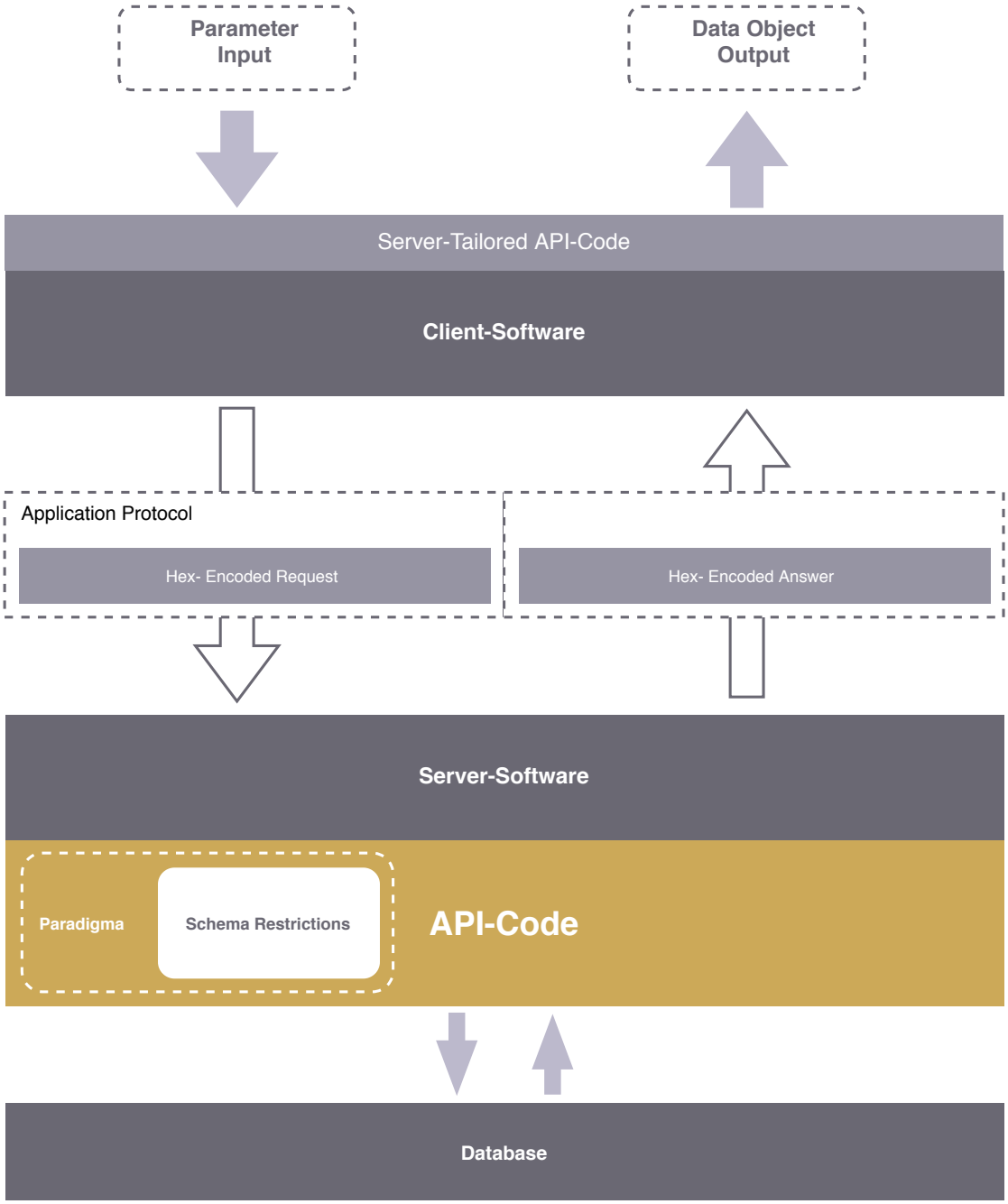

Abbildung 4.1: Übersichtsgraphik Web-Architektur

4.1 API-Paradigmen

APIs, ausgeschrieben *Application Programming Interfaces*, sind Programmteile oder definierte Protokolle, die komplexere Funktionalitäten auf Quellcode-Ebene anbieten. Ziel von APIs ist eine effektivere Programmierung, das Erzwingen von Standards und im Falle moderner Web-Schnittstellen, der extensive Austausch von Daten. Es ergibt sich dadurch eine Unterscheidung zwischen Interfaces innerhalb einer Programmiersprache oder -umgebung, in denen beliebige Funktionen im Sinne einer Blackbox abgekapselt sind und Interfaces, deren Zweck in der Bereitstellung von Daten ohne explizite Sprachenbindung besteht [13].

In der Software-Entwicklung ist vor allem Ersteres in Form von Modul-Schnittstellen seit Entwicklung der ersten Betriebssysteme verbreitet [14]. Diese werden für eine Programmiersprache auch Libraries genannt und sind meist mit Schwerpunkt auf hohe Qualitätstandards und Stabilität geschrieben. Hier sei z. B. auf die GNU C Library nach ISO C Standard verwiesen [15]. Mit Durchbruch der Internettechnologie verschob sich im gleichen Zuge die Bedeutung von APIs. Im Allgemeinen bezeichnet der Term jetzt immer eine Web-, bzw. Remote-API, die über ein Kommunikationsprotokoll Daten anbietet oder selbst annimmt. In allen weiteren Kapiteln wird sich nur noch auf Web-APIs bezogen.

Moderne APIs sind (neben Big Data) das Rückgrat nahezu der gesamten Internet Ökonomie [16]. Sie sind Schnittstelle zwischen Client und Server und kommunizieren mit einfachen HTTP-Calls oder dauerhafteren Verbindungen auf derselben technischen Basis. Das Paradigma entscheidet dabei, was für Daten in welcher Form übertragen werden. Folgend werden die bekanntesten Paradigmen kurz beschrieben.

4.1.1 Remote Procedure Calls

RPC ist eine API-Technik, bei der ausgewählte Methoden des Servers für den Client so abrufbar sein sollen, als wären sie clientseitig implementiert. Im Mittelpunkt der 'Calls' stehen Aktionen, die mit Parametern gespeist werden und ein vordefiniertes Datenobjekt als Ergebnis zurückschicken.

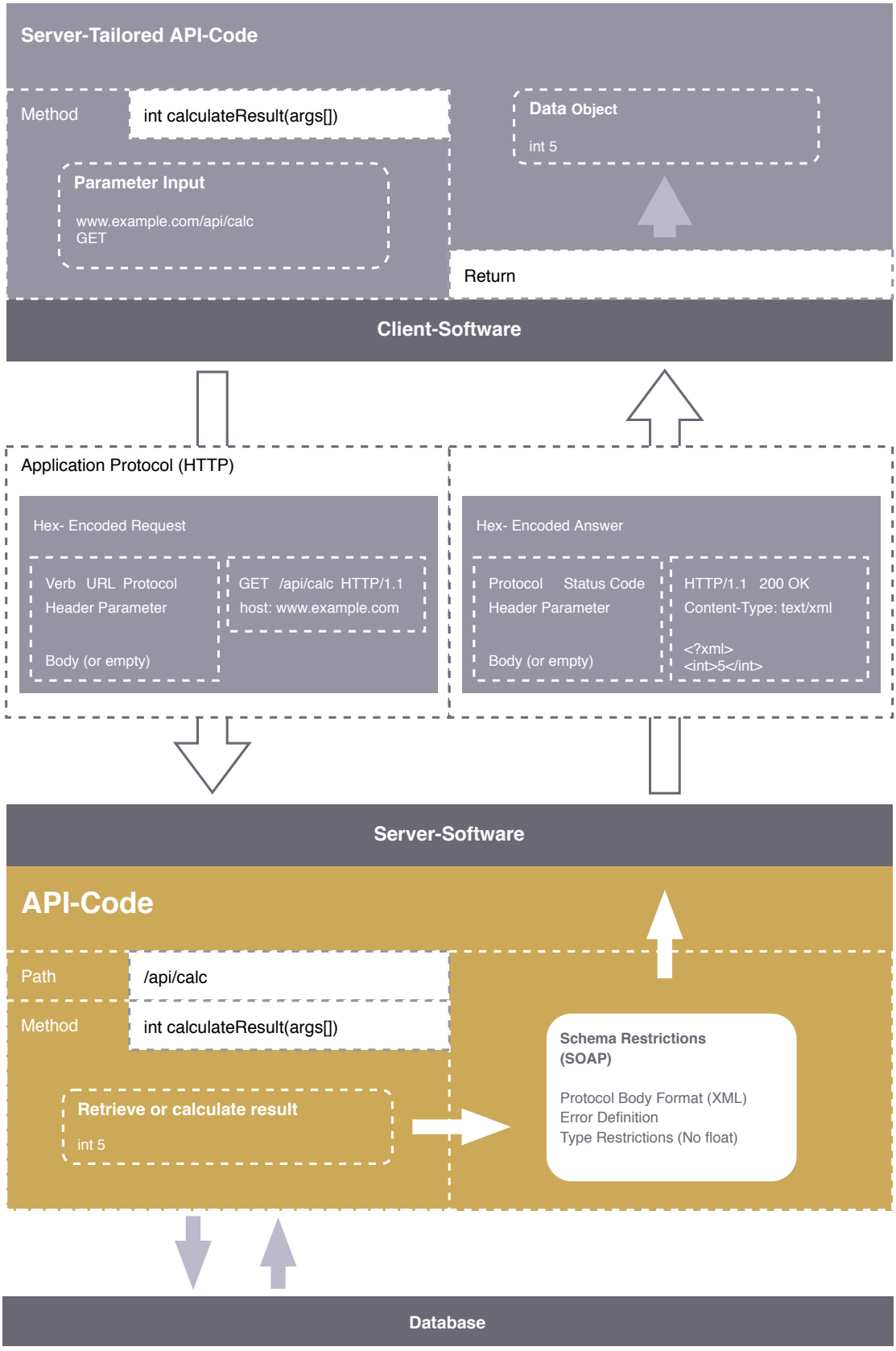

Abbildung 4.2: Komplexes Beispiel einer API Kommunikation

Als Netzwerk-Technik bekam RPC erstmalig vermehrte Aufmerksamkeit 1994 mit der Auszeichnung der Arbeit von Bruce Nelson und Andrew Birrell, die einen theoretischen Entwurf sowie eine effektive Umsetzung für das Xerox Research Network beinhaltete [17]. Damit ist es das Älteste der etablierten Paradigmen. Als ausgearbeitete Schemata bzw. Standards, sind SOAP, JSON-RPC und gRPC im Einsatz.

Hinsichtlich der Verbreitung sind RPC-Interfaces inzwischen weit hinter REST zurückgefallen, da der Schwerpunkt der Replikation von Server-Methoden die Verwirklichung bestimmter Funktionalitäten auf dem Client verkompliziert und eine Standardisierung erschwert. Für einen effizienten Einsatz, insbesondere bei Wechselwirkung mehrerer angebotener Funktionen, ist es notwendig zu wissen, wie sich der Code auf der Server-Seite verhält.

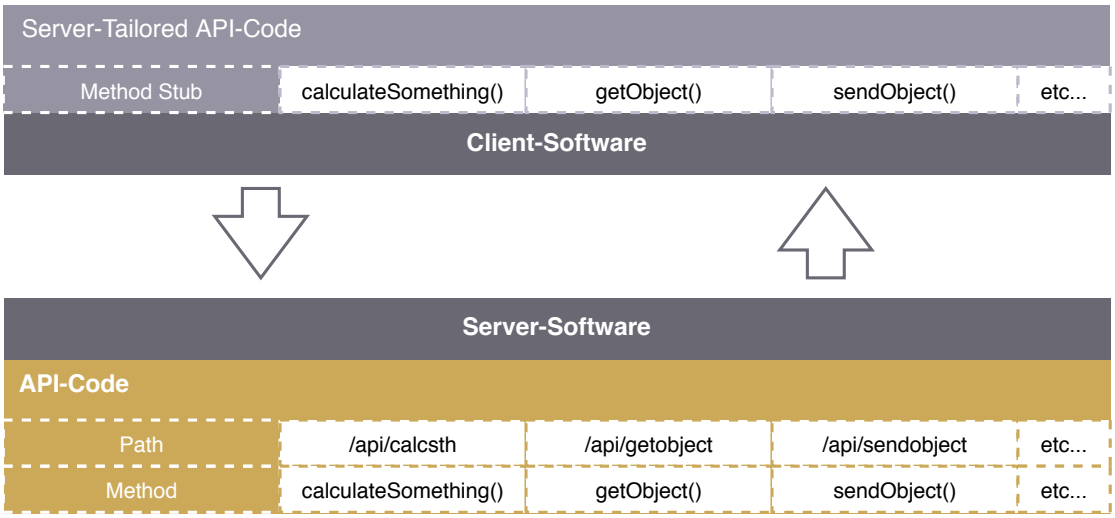

Abbildung 4.3: RPC als Konzept

4.1.2 Representational State Transfer Protocol

Liegt der Schwerpunkt beim Design einer API auf der Bereitstellung oder Manipulation von Datenobjekten und nicht mehr auf Server-Methoden sowie deren Aktionen, nennt

man dieses Prinzip REST (Representational State Transfer Protocol). Die Besonderheit von REST beschränkt sich nicht nur auf die Modellierung eines konsistenten Datenmodells, sondern beinhaltet ebenfalls Restriktionen, wie unbedingte Zustandslosigkeit, Cachebility und eingebetteter Hypermedia State (HATEOAS). In einer strengeren Definition ist der Server also gleichzeitig mitverantwortlich den Client darüber zu informieren, welche Aktionen auf das Objekt möglich und sinnvoll sind.

Die Grundlage von REST wurde 1993 von Roy Thomas Fielding gelegt, als Antwort auf die inhärenten Skalierungsprobleme des damals noch jungen World Wide Webs. Fielding spezifiziert 6 Constraints, die für die Architektur des Internets notwendig sind, um erfolgreich wachsen zu können und fasste sie später in seiner Dissertation unter dem Namen REST zusammen [18][19].

Obwohl bisher kein einheitlicher Standard exisitiert, haben sich auf REST basierende API Description Languages etabliert und werden allumfassend eingesetzt. Unter den Populärsten befinden sich OpenAPI (Swagger), OData und API Blueprint.

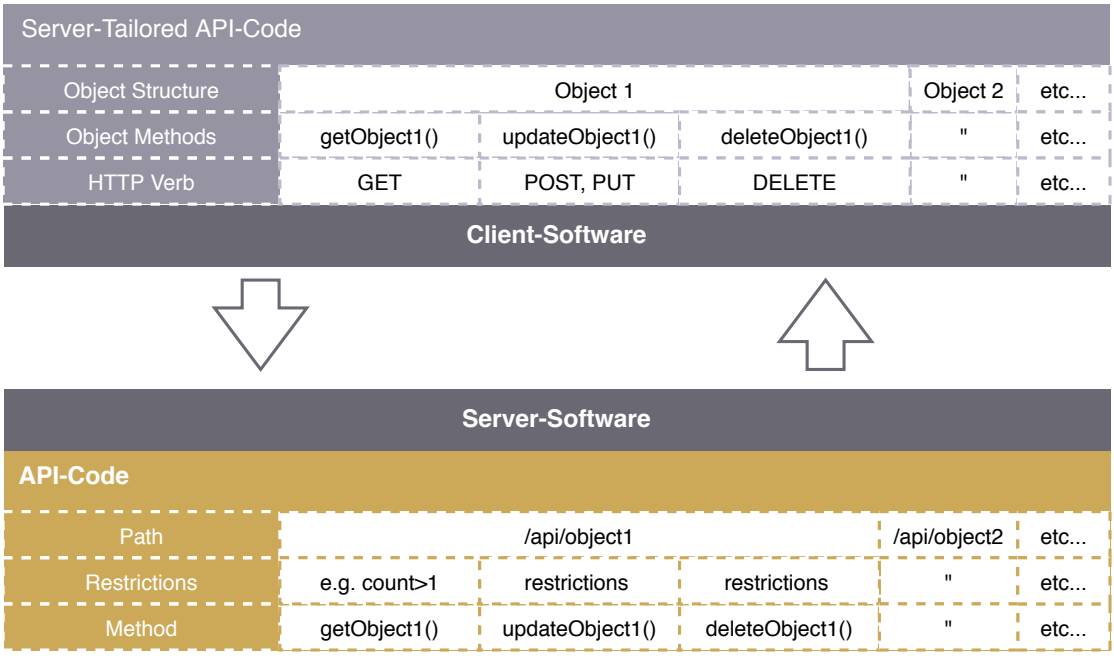

Abbildung 4.4: REST als Konzept

4.1.3 Query Languages

Unter einer Query Language versteht man eine, auf speziell strukturierte Datenobjekte zugeschnittene, komplexe Abfragesprache, die Relationen und Eigenschaften berücksichtigt und ein hochindividuelles Ergebnis als Antwort zurückschicken kann [20]. Das bedeutet die Bündelung aller Funktionen auf einen zugreifbaren Endpunkt (im Vergleich zu RPC - nur noch eine, statt mehrere Stub-Methoden), der nur über den Parameter, also die *Query* gesteuert wird.

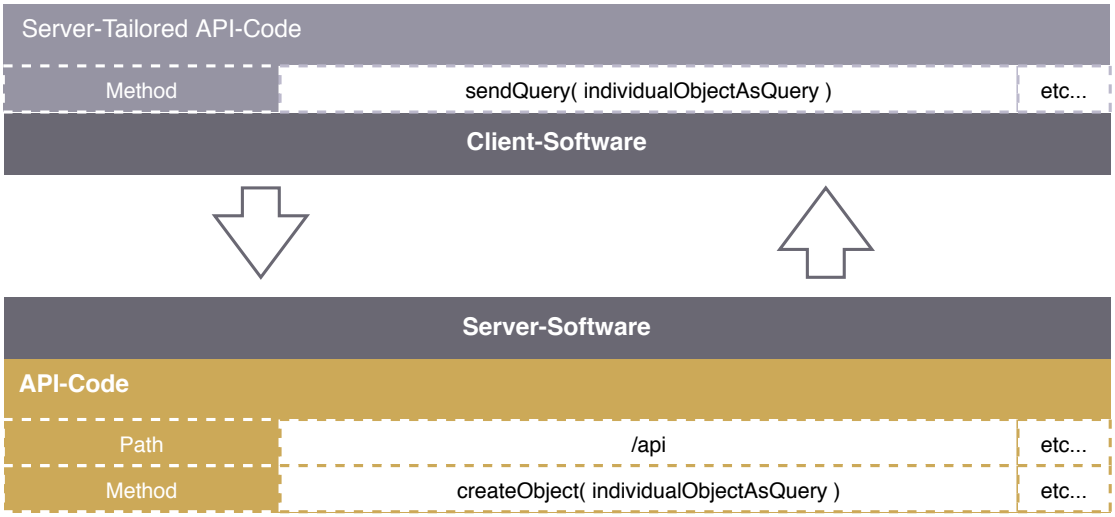

Abbildung 4.5: Query APIs als Konzept

Query Languages sind schon in den 70ern aus der Notwendigkeit entstanden, Werte aus relationalen Datenbanken zu lesen, wie zu speichern [21]. Aus diesem Grund ist SQL (Structured Query Language) einer der bekanntesten Vertreter und Datenbank-Interfaces sind die ersten APIs, die von ihren Eigenschaften modernen Web-APIs sehr nahe kommen.

Als Teil moderner Web Architektur finden Query Languages nur selten Anwendung, meist direkt in Zusammenhang mit Service-Datenbanken (z. B. InfluxDB mit InfluxQL). Die größte Ausnahme davon stellt Facebooks GraphQL-Technik dar, die 2016 die konventionelle *Facebook Query Language* ersetzt hat. QraphQL vereint die Kontrolle über

die Form und den Umfang der ausgetauschten Daten mit Konzepten aus REST und RPC, wie z. B. die Verteilung explizit definierter Datenobjekte auf verschiedene Query-Methoden und die Unterscheidung zwischen GET und POST Queries.

4.1.4 Event Driven APIs

Reichen die Möglichkeiten für eine effektive Kommunikation über normale Interface-Endpunkte nicht aus, ist also ein ressourcenintensives Polling des Servers notwendig, greift man auf Event Driven APIs zurück. Diese sind weniger ein weiteres Paradigma, als eine selbstständige Technik und eine Alternative, um die Einschränkungen konventioneller APIs zu umgehen. Dabei stehen diese nicht unabhängig zur Verfügung, sondern werden meist zusätzlich, z. B. zu einer REST-Schnittstelle, angeboten. Beispiele für diese Doppel-Interfaces sind Twitter-, Slack- und Google-API.

Bei einer Event Driven API kann man zwischen Kommunikation, basierend auf Event-Bedingungen und kontinuierlichem Streaming, unterscheiden. Technisch ist der Unterschied allerdings minimal und verschwimmt schnell, da selbst das Warten auf registrierte Events oft über Streaming-Protokolle realisiert wird. Die Umsetzung erfolgt mit WebSockets, WebHooks oder HTTP Streaming und bringt jeweils eigene Vor- und Nachteile mit sich [22].

4.2 Modeling Schemas

Will man die API genauer definieren, benutzt man dafür ein Modeling Schema, bzw. eine IDL (Interface Definition Language). Modeling Schemas stellen den Rahmen für eine maschinenlesbare Beschreibung aller Eigenschaften einer API. Darunter Fallen Endpunkte, Datenstrukturen, Datentypen, Aktionen und Fehlermeldungen. Die individuelle Ausarbeitung einer API in einer Schema-Sprache nennt man API-Definition. Das

Format ist im Allgemeinen beliebig, beschränkt sich aber in der Praxis auf JSON, XML und XAML.

Vorteile der Benutzung einer Schema-Sprache gegenüber einer 'naiven' Implementierung ergeben sich durch die automatische Generierung von API-Clients, Dokumentationen und Test-Servern. Auf die API angewiesene Entwickler sind somit nicht mehr von dem tatsächlichen Server-Code abhängig, sondern können unabhängig arbeiten und Weiterentwicklungen in der Konzeption werden schnell reflektiert. Welche Hilfsmittel zur Auswahl stehen, unterscheidet sich von Schema zu Schema, wobei das umfangreichste Toolset momentan von Swagger gestellt wird.

4.2.1 SOAP und WSDL

SOAP (Simple Object Access Protocol) ist ein Netzwerkprotokoll in XML, das oft in Verbindung mit WSDL, der *Web Service Description Language*, eingesetzt wird. Im Vergleich zu jüngeren Schema-Sprachen, die beide Aspekte vereinen und vereinfachen, existiert hier eine konkrete Aufteilung in einen Nachrichten-Beschreibungs-Standard (SOAP) und einen Service-Beschreibungs-Standard (WSDL). Beide Standards sind mit dem Ziel entwickelt worden, Netzwerkstrukturen und deren Kommunikation einfach und unabhängig abbilden zu können. In der Praxis werden SOAP und WSDL nur noch selten eingesetzt, da inzwischen effizientere, paradigmengebundene Sprachen diesen Platz eingenommen haben. Das als großangelegtes Service-Verzeichnis konzipierte WSDL-Repository UDDI hat z. B. 2005 mit dem Absprung von Microsoft, IBM und SAP seine wichtigsten Stützen verloren [23].

Wie bei Protokollen üblich, ist bei SOAP die XML-Nachricht in Envelope, Header und Body eingeteilt. Optional kann ein weiterer Abschnitt für Fehlermeldungen eingefügt werden. Als Generalisierung von Middleware-Lösungen innerhalb der Server Software wird ein 'path' System angeboten, mit dem weitere Service-Endpunkte definiert werden können, die sich z. B. um die Authentifizierung oder Formatierung kümmern. Der *body* kann beliebige Zeichenketten beinhalten, ist bei der Übertragung von Datenobjekten

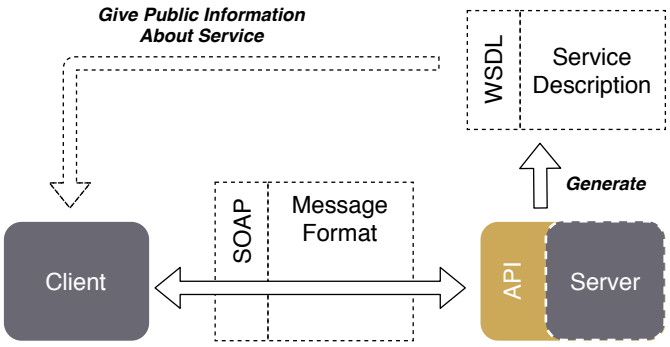

Abbildung 4.6: Rollenverteilung SOAP & WSDL

jedoch fast immer mit einem vereinbarten Encoding-Standart strukturiert. Insgesamt sind SOAP Nachrichten unabhängig von der unterliegenden Architektur einsetzbar, können also in HTTP genauso wie in FTP oder UDP übermittelt werden.

Die Schema-Sprache, mit der SOAP in Zusammenhang steht, ist WSDL. Diese Sprache in XML-Format soll potentiellen Clients einen Überblick über den Service geben und eine dynamische Anpassung an die Eigenschaften verschiedener Server ermöglichen. Der Fokus für den Service-Dienstleister ist dementsprechend eine WSDL-Beschreibung anbieten zu können, die vorher aus dem Quellcode generiert wurde. Die Verwendung von WSDL als Entwurf-Sprache rückt damit in den Hintergrund [24].

Der Aufbau eines WSDL-Dokumentes beruht auf abstrakten und konkreten Abschnitten. Definierte, abstrakte Elemente werden in konkreten Bindungen verwendet und beschreiben Typen, Interfaces und Input/Output-Operationen in einem oder beliebig vielen Service-Endpunkten. REST wird als Beschreibung ebenso unterstützt, wie RPC und der aktuelle Standard 2.0 hat die Nomenklatur zugunsten zeitgemäßerer Termini angepasst.

SOAP Example

```
Envelope
  Header
    <s:Envelope xmlns:s="...soap-envelope">
      <s:Header>
        <m:transaction xmlns:m="trans"
            s:mustUnderstand="true">
          <ID>5295</ID>
        </m:transaction>
      </s:Header>
  Body
      <s:Body>
        <n:getData xmlns:n="urn:Service">
          <symbol xsi:type="xsd:string">
              Hello
          </symbol>
        </n:getData>
      </s:Body>
    </s:Envelope>
```

WSDL Example

```
Description Envelope
  Types
    <description xmlns="http://www.w3.org/ns/wsdl">
      <types>
        <xs:schema xmlns:xs="...XMLSchema">
          <xs:element name="getData"> ... </xs:element>
        </xs:schema>
      </types>
  Interface
      <interface name="Interface">
        <operation name="Get" pattern="...in-out">
          <input messageLabel="In" element="tns:getData"/>
        </operation>
      </interface>
  Binding
      <binding name="SoapBinding" interface="tns:Interface"
          type="...soap">
        <operation ref="tns:Get" />
      </binding>
  Service
      <service name="Service" interface="tns:Interface">
        <endpoint name="SoapEndpoint"
            binding="tns:SoapBinding"
            address="http://www.example.com/soap/"/>
      </service>
    </description>
```

Abbildung 4.7: XML-Beispiel für SOAP & WSDL

4.2.2 OpenAPI

Aus dem Bedürfnis heraus, eine einfache und effektive WSDL-Alternative für das populäre REST-Paradigma zu besitzen, wurde 2010 mit Swagger die bis heute wichtigste Schema-Sprache entwickelt und veröffentlicht. Um herstellerneutral zu bleiben, entstand 2015 unter der Linux Foundation das Entscheidungsgremium *Open API Initiative* und veranlasste die Umbenennung der Sprache von Swagger zu OpenAPI [25]. Mit dem Namen Swagger wird ab diesem Zeitpunkt das Toolset bezeichnet, das für die Arbeit mit der OpenAPI-Spezifikation seit Entstehung bestimmend war. Weitere unabhängig entwickelte Tools sind z. B. *OpenAPI generator*, *ReDoc* oder *API Transformer*.

OpenAPI Dokumente werden in JSON oder YAML verfasst und XML wird nur als Alternativformat für die Datenserialisierung angeboten. Ein strenges Übertragungsprotokoll wie SOAP existiert nicht, aber das Datenmodell ist eine Untermenge der *JSON Schema*-Spezifikation. Um im Bedarfsfall dem XML-Format gerecht zu werden und Ambivalenzen auszuschließen, gibt es Attribute für die Zuweisungen von XML Namespaces und weiteren Eigenschaften [26]. Die im Swagger-Paket enthaltenen Tools zur effektiven Arbeit mit OpenAPI schließen eine UI zur Visualisierung, einen Editor und einen Codegenerator mit ein. Der Contract-First Ansatz wird bevorzugt, jedoch arbeiten Frameworks für die

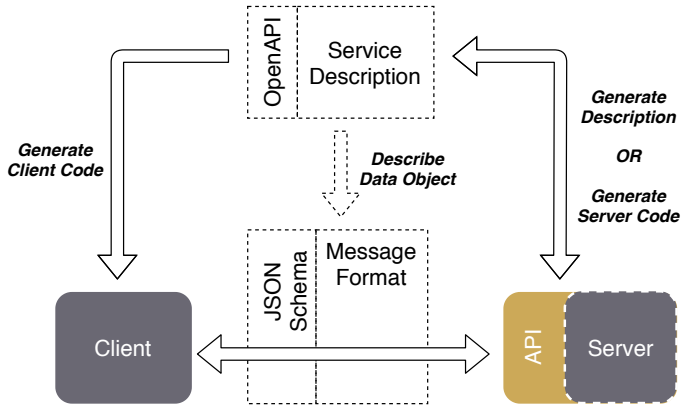

Abbildung 4.8: OpenAPI in Client-Server Architektur

einzelnen Programmiersprachen oft mit Annotationen und lassen Abstufungen bei der Erstellung des API-Dokumentes zu.

Im Gegensatz zu WSDL ist bei OpenAPI keine feste Reihenfolge in der Struktur gefordert. Notwendige Attributfelder sind **openapi** (Version), **info**, **paths**, und eine Streckung über mehrere Dateien wird unterstützt. Ab Version 3.0 gibt es außerdem ein einheitliches Komponentensystem zur Minimierung von Redundanz [27].

4.2.3 gRPC

Mit dem Erfolg von Swagger und dem REST Paradigma wurde die Bedeutung von Remote Procedure Calls für die Vernetzung von Web-Services in den Hintergrund gedrängt. Seit 2015 und der Veröffentlichung von Googles Framework **gRPC** ist diese Technik wieder mehr im Fokus. Dabei ist der Kern von gRPC Googles Protocol-Stack, bzw. Serialization-Framework *Protocol Buffers*, das mit Microsofts *Bond*, Facebooks *Apache Thrift* und Apaches hauseigenem *Avro*, konkurrierende Entsprechungen hat. Alle Stacks sind auf RPC ausgerichtet, kommen mit einer eigenen Schema-Sprache und profitieren von der festen Stack-Bindung.

OpenAPI Example

YAML Document

```yaml
openapi: 3.0.0
info:
  title: Sample API
  description: Optional description
  version: 0.1
servers:
  - url: http://api.example.com/
    description: Optional server description.
paths:
  /object:
    get:
      summary: Returns a list of objects.
      responses:
        '200':
          description: A JSON array of object objects.
          content:
            application/json:
              schema:
                type: array
                items:
                  type: string
```

Abbildung 4.9: Minimalbeispiel für OpenAPI in YAML

Die Hauptaufgabe dieser Stacks und der große Unterschied zu z. B. OpenAPI, ist der Einsatz speziell entwickelter Protokolle für eine API-Kommunikation in höchstmöglicher Geschwindigkeit. Diese Protokolle sorgen für eine Serialisierung und anschließender Binarisierung aller gesendeten Daten. Verknüpft wird dies in der Praxis mit Code-Generatoren für die einfache Anbindung von Server- oder Client-Stubs und im Falle von gRPC einer Entwicklungsumgebung mit allen notwendigen Tools für die Erstellung und Skalierung von Service-Netzwerken.

Die Besonderheit im Vergleich zu REST Schnittstellen ist diesmal nicht nur das Spiegeln der Servermethoden auf den Client, sondern das Aufgeben von frei ansprechbaren HTTP Endpunkten zur Datenabfrage. Wie der Client und der Server Informationen übertragen ist so abgekapselt, dass mit einfachen HTTP GET oder POST Anfragen keine sinnvolle Verbindung etabliert werden kann. Datenpakete werden nach ausgewählter Methode serialisiert und müssen mit dem gleichen, dem Empfänger dann bekannten, Verfahren deserialisiert werden. Für Client und Server ist die Bindung an ein Framework,

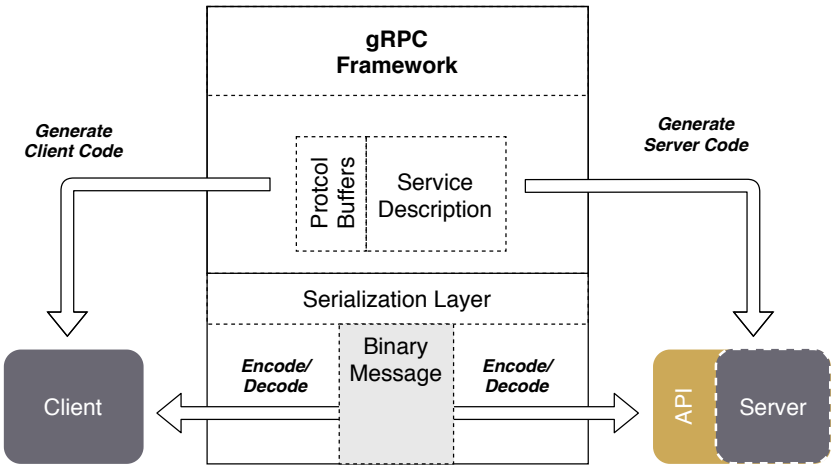

Abbildung 4.10: gRPC in Client-Server Architektur

bzw. ein Serialisierungsprotokoll also verpflichtend.

gRPC Example

Protocol Buffer Document

```
service HelloService {
    rpc getObject (ParameterObject) returns (Object);
}

message ParameterObject {
    string value = 1;
}

message Object {
    string value = 1;
}
```

Abbildung 4.11: Protocol Buffers Beispiel

4.3 Authentication und Authorization

Ein stabiles Sicherheitskonzept gehört zu den Grundpfeilern einer API. Benutzer sollen eindeutig identifizierbar sein (*Authentication*) und Berechtigungen müssen vergeben oder beschränkt werden (*Authorization*). Verschlüsselte Verbindungen bei allen API

Anfragen lösen nicht die individuellen Probleme, die eine Web Service Infrastruktur verursacht. Mehrere Clients greifen auf einen Datenpool zu und der Rechteinhaber will trotzdem die Kontrolle behalten. Letztendlich sind asymmetrische Kryptographie-Algorithmen aufwendig und ineffizient als Abschirmung des kompletten Datenverkehrs.

Die folgenden Unterkapitel gehen auf Konzepte und Techniken ein, die mit API-Sicherheit assoziiert werden. Sie erläutern den Ablauf zwischen allen Beteiligten und geben Einblick in aktuelle Standards.

4.3.1 HTTP Basic Authentication

Die grundlegendste Form der Authentifizierung von Web-Zugriffen ist *HTTP Basic Authentication*. Sie beschreibt den Vorgang der Identifizierung eines Benutzers mit Benutzername und Passwort durch den Server. Die Daten werden durch eine HTML-Form eingegeben und als URL-encoded Key-Value Parameter verschickt. Die dabei verwendete *Base64* Codierung trägt nicht zur Sicherheit bei und muss deshalb von einem kryptographischen Standard (aktuell **TLS 1.8**) unterstützt werden. Ein vollständig eingesetzter TLS Standard sichert gleichzeitig die Identität und die Datenübertragung [28].

Als alleinige Autorisierungsmethode einer API ist HTTP Basic Authentication nicht mehr vertretbar. Da der Nutzer der Daten (Client-App) und der Rechteinhaber oft verschiedene Entitäten darstellen, müsste für eine Autorisierung das Password dem Nutzer zur Verfügung gestellt werden. Daraus resultierende Sicherheitslücken sind bei APIs schwer unter Kontrolle zu bekommen. In Sicherheitsprotokollen wie **OAuth** ist HTTP Standard Authentication deshalb nur eine Teilvoraussetzung im gesamten Prozess.

4.3.2 OAuth

OAuth ist ein HTTP Sicherheitsprotokoll zur Autorisierung von Software durch den Rechteinhaber, ohne Kompromittierung von sicherheitsrelevanten Daten, also Passwort und

Benutzername [29]. Es ist aus der Notwendigkeit enstanden, zum Einen professionelle Blaupausen für sichere Autorisierungsprozesse zur Verfügung zu haben und zum Anderen mit einem anerkannten Standard die Implementation zu erleichtern. Sichere, aber zueinander inkompatible Zwischenlösungen, wie Googles **AuthSub** und Yahoo!'s **BBAuth** sollen damit überbrückt werden. Die neueste und seit Jahren empfohlene Revision OAuth 2.0 spezifiziert *Authorization Flows* und überlässt die Verwendung von verschlüsselten Signaturen dem Benutzer.

Der gebräuchlichste Authorization Flow hat als feste Rollen den *Client*, den *Resource Owner*, den *Resource Server* und den *Authorization Server*. Sollen dem Client vom Resource Owner Zugriffsrechte für eine API übertragen werden, übernimmt der Authorization Server (der gleichzeitig auch Resource Server sein kann) die Überprüfung und sendet einen *Access Token* zurück. Dieser wird dem Client übergeben und ermöglicht ihm jetzt die Kommunikation mit dem Resource Server [30].

Um bei Missbrauch den Schaden gering zu halten, besitzt jeder Token ein Ablaufdatum. Sind die Zeiten kurz gehalten, werden langlebige *Refresh Token* eingesetzt, um ohne erneuten Einbezug des Resource Owners dem Client selbst die Möglichkeit zu geben, seinen Zugang zu erneuern.

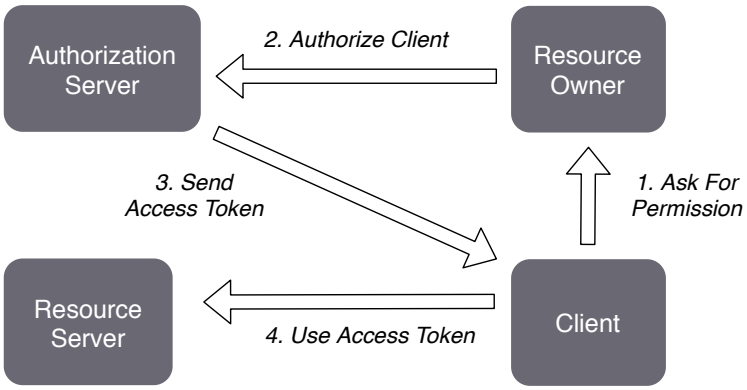

Abbildung 4.12: OAuth 2.0 Authorization Flow Beispiel

4.3.3 JSON Web Token

Eine besondere Form des Access Tokens im OAuth 2.0 Standard ist der *JSON Web Token* (JWT). Es ist ein in der JavaScript Object Notation übermittelter Token, der leicht zu parsen ist und beliebige Informationen veränderungssicher codiert. Er besteht aus Header, Payload und einer, aus genau diesen beiden Elementen verschlüsselten Signatur. Header und Payload sind immer lesbar und sollten keine sensiblen Daten enthalten. Die Signatur verhindert aber Manipulationen, da sie beim Einlösen des Tokens immer mit den offenen Daten abgeglichen wird. Die kryptographischen Algorithmen dafür, sind klassisch entweder HMAC oder RSA [31].

4.4 Scaling

Wird eine API nicht für einen einzigen, klar formulierten Anwendungsfall entwickelt, müssen Maßnahmen für ein kontrolliertes Wachstum der API in die Architekturplanung einfließen. Die Anzahl der Nutzer kann den Server überlasten oder Datenobjekte zusätzlicher Endpunkte sind nicht mehr mit der gesamten API kompatibel. Von der Anzahl an Hilfsmitteln zur Optimierung des Web Service wird hier nur eine Auswahl vorgestellt. Nicht zu unterschätzen ist ebenfalls der Einsatz von Metriken und Benchmarks zur Lokalisierung des Bottlenecks. [22]

4.4.1 Rate Limiting

Bei einem Server, der von außen angesprochen werden kann, muss man immer damit rechnen, dass die Anzahl der Anfragen die verarbeitbare Menge überschreitet und das System verlangsamt oder ausschaltet. Bei APIs ist das selten ein bewusster Angriff, sondern kann Ergebnis von hohen Benutzerzahlen oder ineffizienter Clientprogrammierung sein. Um nicht allen Nutzern, unabhängig von ihrem Einfluß, die gleichen Beschränkungen auferlegen zu müssen, werden spezielle *Rate Limiting* Techniken eingesetzt, die

gezielt den Datenverkehr verwalten. API Rate Limiting setzt für die Entscheidung, wer wann eine Anfrage beantwortet bekommt, eine *Policy* fest, also ein einfaches Regelwerk für eine faire Ressourcenverteilung. Die Umsetzung bindet dadurch direkt den Benutzer und sein individuelles Anfrageverhalten. Realisiert wird dies z. B. durch Token oder Zeitfenster.

4.4.2 Pagination

Pagination bezeichnet die Einteilung von großen Datensets, die von einer Schnittstelle angeboten werden, in kleinere „Seiten“. Es ist der Mittelweg zwischen der Ausgabe eines einzelnen Datenobjektes über z. B. ID und der Ausgabe aller Objekte. Diese Einteilung wird notwendig, sobald eine Datenbank genug Daten gesammelt hat, um eine Verarbeitung und den Transport nicht mehr gewährleisten zu können. Einfluss hierauf haben der Arbeitsspeicher des Servers, des Clients und die Netzwerkverbindung.

Mit Pagination bekommt der Benutzer der Schnittstelle Kontrolle über die Zusammenstellung des Datenteilsets. Entweder er gibt den Offset und das Limit der angezeigten Objekte an oder er navigiert mit einem „Cursor“. Beide Methoden haben Vor- und Nachteile und entspringen Performance-Überlegungen bezüglich der notwendigen Datenbank-Queries.

4.4.3 Caching

Eine verbreitete und einfache Technik die Performance des Servers zu erhöhen, ist *Caching*. Beim Caching werden Daten im Arbeitsspeicher gehalten, die besonders oft abgefragt werden. Entscheidend ist bei Schnittstellen ein Verwalten des Speichers mit Anpassung an das Benutzerverhalten. Unterliegende Server-Software deckt alle Anfragen schon mit eigenen Lösungen ab und ist auf die gängigsten Szenarien ausgelegt. Query-Sprachen wie GraphQL verzichten komplett auf Caching, da die Kosten den Nutzen überwiegt.

5

Konzeption

Der Konzeptionsteil stellt den letzten Schritt vor der Implementation dar. Es werden auf Basis der zusammengetragenen Informationen wichtige Entscheidungen getroffen und Alternativen gegeneinander abgewogen. Durch einen konkreten Architekturvorschlag sollen spätere Interdependenzen und Unkompatibilitäten so früh wie möglich sichtbar gemacht werden. Als Hilfestellung verwenden Konzeptionen in der Softwaretechnik UML Klassen-, Sequenz- und Ablaufdiagramme.

Der Entwurf einer Schnittstelle konzentriert sich auf Paradigma, Datensicherheit und nach Bedarf auf Skalierungsmethoden. Alle drei Themen werden in der genannten Reihenfolge behandelt. Wo Diagramme und Skizzen den Platzbedarf einer Seite überschreiten, wird nur ein Ausschnitt gezeigt. Dieser Abschnitt soll keine Dokumentation ersetzen.

5.1 GraphQL

2012 hat Facebook durch vermehrte Probleme mit ihrer REST API und ihrer hauseigenen SQL Sprache die Entwicklung von GraphQL beschlossen. GraphQL sollte die massenhafte Übertragung von nicht benötigten Daten eindämmen, die Performance aller Web- und Mobile-Apps insgesamt verbessern [32]. Die Lösung war eine Kombination aus Query-Sprache und REST-Konventionen, beschrieben im kompakten JSON-Format. Mit GraphQL gestaltet der Client sein individuelles Datenobjekt und ist nicht mehr komplett von den Objekten der REST Endpunkte abhängig. Der zwangsweise anfallende

Overhead kann fast vollständig vermieden werden. Durch die größere Beanspruchung des Servers bei der Bearbeitung der Queries, wird der performancekritische Abschnitt allerdings verlagert und nicht umgangen. Dies hat trotzdem den Vorteil, dass Web Service-Anbieter mit z. B. schnellerer Hardware einen größeren Einfluss ausüben können.

5.1.1 Motivation

Als Paradigma, bzw. Technik, stehen REST, RPC und GraphQL zur Auswahl. RPC ist dabei ausgeschlossen, da es seine Stärken bei fester Bindung von Client und Server ausspielt. Die Entscheidung zwischen REST und GraphQL ist weniger eindeutig und braucht eine weitere Erläuterung. Faktoren, die für REST sprechen, sind die geordneten Endpunkte für jedes Datenobjekt, die effektiveren Queries auf die Datenbank und mit der Cachebility eine geringere Belastung des Servers im Allgemeinen. REST gilt als erprobte Technik für eine schnelle und zielführende Entwicklung. GraphQL lagert den Datenoverhead, der bei komplexen REST-Anwendungen auftaucht, bewusst auf die Kapazitäten des Servers aus. Damit wird der Traffic reduziert und insb. Mobile-Apps profitieren von einer gezielten Datenabfrage.

Die Entscheidung für GraphQL folgt hieraus nicht aus eindeutigen Vorteilen, sondern die Hinführung auf ein Use Case. Diese Use Cases werden später nicht fest durchgeführt, sondern gelten als Richtung, die durch die Gestaltung der Schnittstelle bewusst angesteuert werden kann. Diese Arbeit entscheidet sich einerseits aus experimentellem Charakter für GraphQL und andererseits, um Entwicklungsideen mit Bedarf an dynamischen Datenobjekten zu unterstützen.

5.1.2 GraphQL Type Mapping

Wie die meisten Query Languages, besitzt GraphQL nur einen Endpunkt für Anfragen. Das GraphQL Datenobjekt, also der *Type*, wird vom Server aus der Anfragen-Query ge-

parst. Aus welchen Datenbanktabellen und Attributen sich die Types zusammensetzen, soll bewusst ausgewählt werden, um eine klare Struktur zu kommunizieren. Hierfür wird als Orientierung die Relationengraphik aus Abschnitt 2.4 verwendet. Ignoriert werden Tabellen ohne zwingende Notwendigkeit (z. b. *base_types*). Ausgelassen werden alle Fragebögen und Questionnaires, da der Umfang dieser Tabellen den Rahmen dieser Arbeit übersteigt und deren Implementation repräsentativ keinen Mehrwert bringt.

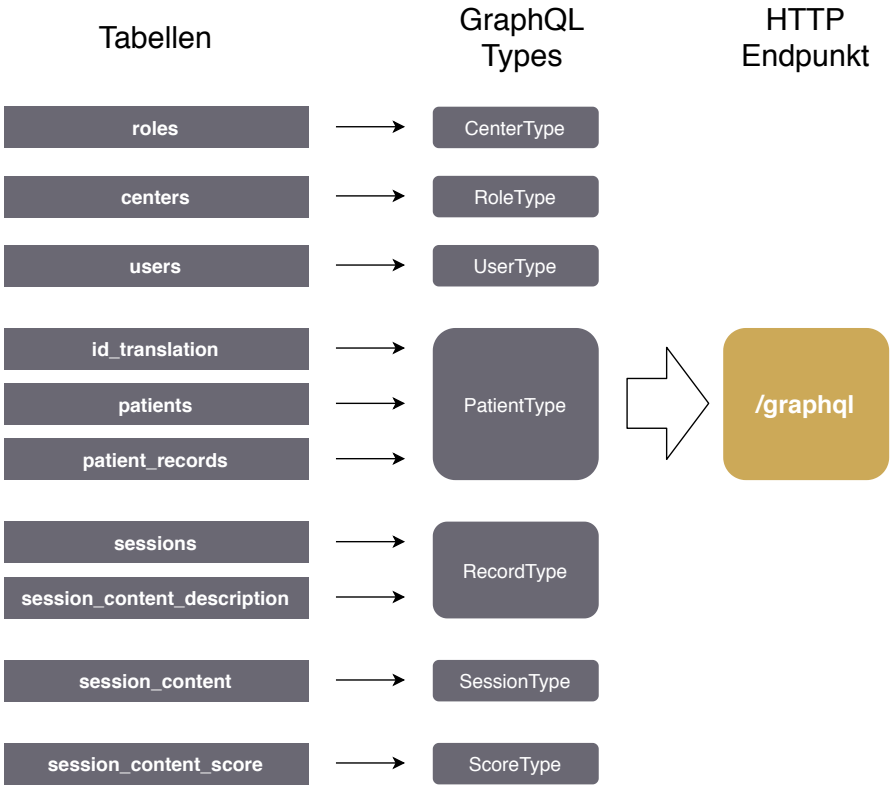

Abbildung 5.1: Mapping der Tabellen auf die GraphQL Types

5.1.3 GraphQL Types

GraphQL hat mit der *GraphQL Schema Language* eine eigene IDL, die hier für ein Beispiel benutzt wird. Die Sprache verwendet eine JSON ähnliche Grammatik mit

verschachtelten Key-Value Einträgen. Die konkreten Typen, wie String, werden *Scalar* genannt und stehen, je nach Framework, in unterschiedlichem Ausmaß, aber immer sehr begrenzt, zur Verfügung. Komplexere Scalars müssen als GraphQL Types selbst erstellt werden.

Die Gestaltung der Types für die Tinnitus DB Schnittstelle erfolgt einheitlich und nur mit den notwendigen Constraints. Nullable sind damit alle Werte außer der ID und notwendige Angaben des Benutzers, wie Passwort und E-Mail. Direkte Relationen werden komplett abgebildet und erlauben ein vollständiges Durchwandern der Objektstruktur. Inwieweit dies negative Auswirkungen auf die Query-Performance hat, wird später näher beleuchtet, da die Limitierung der Query-Komplexität stark von der verwendeten GraphQL Implementation abhängt.

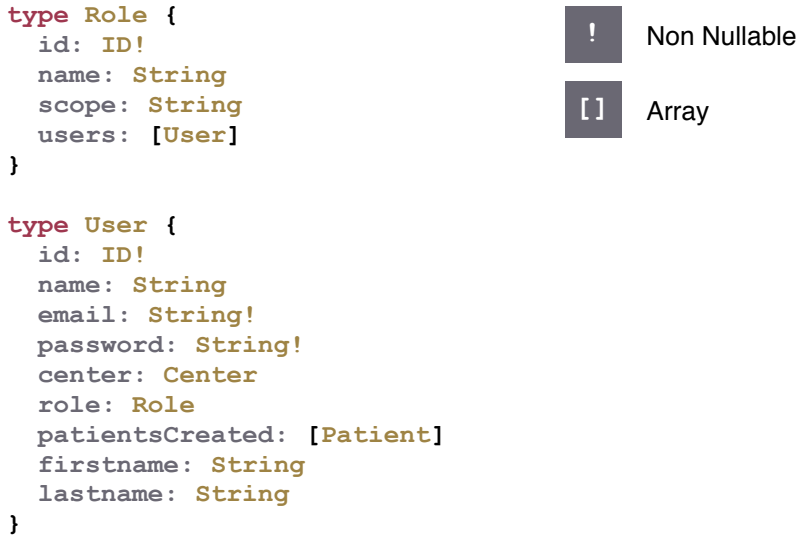

Abbildung 5.2: Role und User als GraphQL Schema

5.2 Sicherheit

Sicherheitskonzepte für APIs werden bei Einbezug aller Faktoren sehr komplex. Es wird sich hier auf Registrierung und Autorisierung von Clients ohne Verifizierung beschränkt. Der Abflauf ähnelt dem Authorization Flow der OAuth 2.0 Spezifikation und gebraucht Access und Refresh Token. Der Refresh Token dient als Passwort des Clients und würde in der Praxis nur mit asymmetrischer Verschlüsselung übermittelt werden. Er hat eine längere Lebensdauer als normale Access Tokens und kann nur vom Rechteinhaber wieder erneuert werden. Des Weiteren fällt Resource- und Authorization-Server zusammen, was für die Implementation von Bedeutung ist, da der Token, der von dem einen Server ausgegeben wird, für den anderen Server nicht unverständlich sein darf.

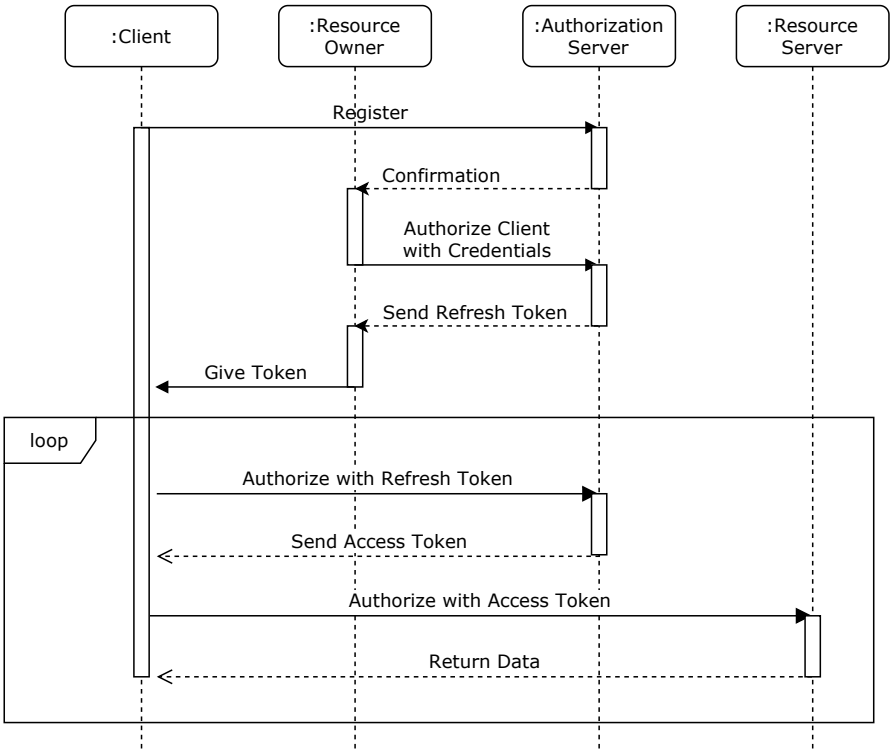

Abbildung 5.3: UML Sequenzdiagramm des Autorisierungsprozesses

5.3 Pagination

Wie bei REST werden bei GraphQL nicht nur einzelne Objekte, sondern bei Bedarf Objektsammlungen geschickt. Das Problem des effizienten Navigierens durch große Datenmengen ist damit bei GraphQL in gleichem Maße vorhanden. Die hier vorgestellte Paginierung geht einen Mittelweg zwischen optimalen Page-Ergebnissen und intuitiver Anwendung.

Es werden dafür folgende Regeln definiert:

1. jedes erfolgreiche Abfrageergebnis ist eine Seite (keine angehängten Metainformationen)
2. zum Definieren der Seite steht ein Parameter „Seite aus Seitenmenge“ und „Objektlimit pro Seite“ zur Verfügung
3. beide Parameter sind optional, die Angabe der Seitenmenge bei Angabe der Objektlimitierung ist aber logisch notwendig
4. es wird ein MAX an Objekten pro Seite festgelegt
5. Metainformationen werden in eigenem GraphQL Type angeboten

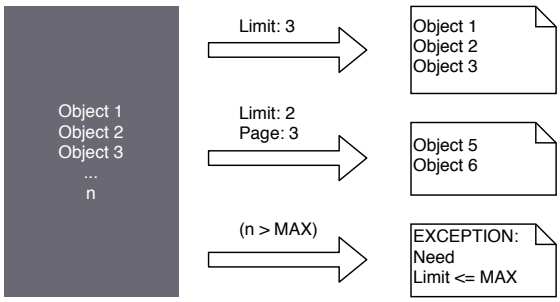

Abbildung 5.4: Beispiel der Paginierung

6

Implementierung

In diesem Kapitel wird näher auf die Umsetzung und deren Beschränkungen eingegangen. Konzepte und Konzeptionen sind auf ein Praxisbeispiel übertragen und geben Aufschluss über komplexere Zusammenhänge und das Zusammenspiel mit der vorliegenden TinnitusDB-Software. Es werden die benutzten Bibliotheken gelistet, individuelle Klassen vorgestellt und kleine Teile des Programmcodes zur Visualisierung genutzt. Für erschöpfende Klassenlistings und detaillierte Erläuterungen (z. B. zur Projekteinrichtung) wird allerdings auf die Dokumentation verwiesen.

6.1 Laravel GraphQL

Die angedachte Schnittstelle wird direkt in das vorhandene Laravel Backend eingebettet. Das bedeutet, der Code muss mit den bisherigen Funktionen harmonieren, API Endpunkte stehen direkt neben den Frontend Routes und die verwendete GraphQL Bibliothek muss mit der Laravel Version des Backends kompatibel sein. Man ist stärker an die Bedingungen gebunden, kann allerdings von den Strukturen profitieren.

Zur Auswahl für Laravel GraphQL stehen mehrere Bibliotheken bzw. Frameworks, von denen die Neuste, *Lighthouse*, ein Vorgehen mithilfe von Schemas ermöglicht. Aus Kompatibilitätsgründen wird hier eine leichtgewichtigere Lösung (Rebing), basierend auf GraphQL-php von *Webonyx* eingesetzt. Es steht ein separater */graphql* Endpunkt zur Annahme der Queries zur Verfügung und es sind die gebräuchlichsten Scalars vorimplementiert. Die Nähe zu Laravel erlaubt zudem Middleware und Eloquent Unterstützung.

Der Weg, den eine Anfrage geht, ist identisch mit Standard Laravel Routen. Statt das Ergebnis einer Datenbank-Query direkt zurückzuschicken, findet allerdings ein Mapping statt. Im Code muss sichergestellt werden, dass die Laravel Array Map aus der Datenbank eine Obermenge des individuell definierten GraphQL Types ist. Dieses Mapping findet nochmals bei Verschachtelungen statt. Erst, wenn alle Types der Anfrage zugeordnet wurden (sie können dabei auch leer sein), wird das Ergebnis verschickt.

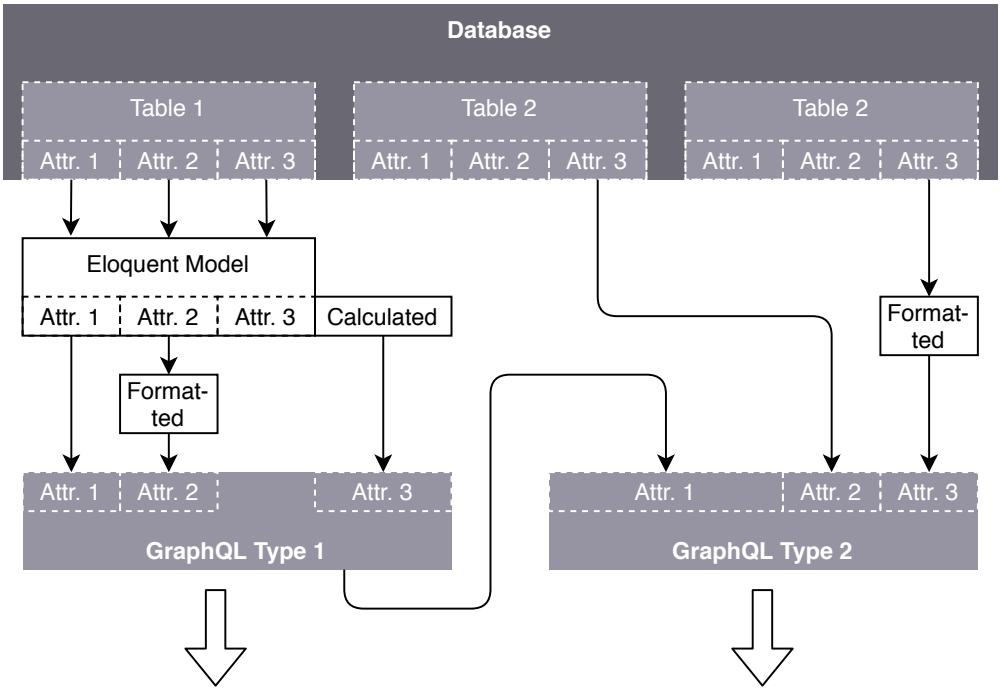

Abbildung 6.1: Komplexes Beispiel des internen Type Mapping

6.2 Allgemeine Architektur

Es gibt für alle GraphQL Types Mutationen und Queries. Queries besitzen als Parameter **id**, **limit** und **page**, die Mutationen alle Parameter für ein vollständiges Type Mapping. Mutationen, deren Zweck es ist Datensätze zu löschen, können rekursiv oder

einfach angesteuert werden. Die Verantwortung bezügl. Datenkonsistenz liegt beim API Benutzer.

Datenbankqueries werden individuell geschrieben und benutzen nur in Ausnahmen Laravels Eloquent Model. Durch die starke Abweichung von den vordefinierten Model-Klassen ist das Einbeziehen von Eloquent nicht von Vorteil. Die Queries direkt gegen die Datenbank können auf Eloquent ausgelagert werden, um Redundanz im Code zu vermeiden, sind aufgrund starker Überschneidungen aber Teil des GraphQL Codes.

Eine weitere Besonderheit der Queries ist noch, dass diese erst an die Datenbank geschickt werden, wenn sie mit allen notwendigen Informationen versehen wurden, also kurz vor dem *Response*. Sequenzielle, gestreckte Datenbankabgfragen oder Laravel-internes Querying würden die Leistung stark beeinträchtigen. Ein genaues Beispiel ist in Abbildung 6.2 vorzufinden.

```
public function resolve($root, $args, $context, ResolveInfo $resolveInfo, Closure $getSelectFields)
{

    $patients = DB::table('patients')->join('id_translation', 'patients.patient_id', '=',
        'id_translation.patient_id')->join('patient_records', 'patients.patient_id', '=',
        'patient_records.patient_id');

    //// Return with ID or continue

    if(isset($args['id'])) {
        return $patients->where('patients.patient_id', $args['id'])->get();
    }

    //////////////
    // Pagination
    // 1. Retrieve instance of Pager class
    // 2. Set page and site limit if submitted

    $pager = resolve('TDB\GraphQL\Services\PageMaker');

    if(isset($args['page'])) {
        $pager->setPage($args['page']);
    }
    if(isset($args['limit'])) {
        $pager->setLimit($args['limit']);
    }

    // 3. Make Page (Retrieve from Database
    //    as last step for minimal data overhead)

    $page = $pager->makePage($patients)->getPage();

    return $page;
}
```

Abbildung 6.2: Code der Patientenquery mit Ablaufoptimierung

6.3 Registrierung und Autorisierung

Der Ablauf der Registrierung und Token-Verteilung ist konform zur Konzeption in Abbildung 5.3. Als Access Token wird JWT eingesetzt, der Refresh Token ist ein, mit der Datenbank abgleichbarer, String. Für diesen String, der als Passwortersatz des Client gilt, müssen in der Praxis verstärkte Sicherheitsstandards gelten. JSON Web Tokens für den Refresh einzusetzen, ist in der Theorie machbar, in Laravel durch die offizielle Implementation aber direkt an ein Model geknüpft und beschränkt. Die Ausgabe von zwei Token für verschiedene Zwecke ist nur möglich, wenn ein Client zwei Konten registriert. Für die Clients existiert in der Datenbank eine separate Tabelle. Die Rechteinhaber werden durch das User-System des vorhandenen TinnitusDB Backend abgebildet und deren Funktionen integriert.

6.4 Pagination

Für die Paginierung wurde eine eigene Klasse geschrieben, die aus den Parametern die notwendigen Eckdaten berechnet und diese als Offset der MariaDB übergibt. Es ist damit eine *Offset-Based* Pagination mit gezieltem Ansteuern beliebiger Datenteilsets. Individuelle GraphQL Fehlermeldungen zeigen ein zu geringes oder zu großes Limit an. Die Anwendung der Paginierung findet in jeder Query-Methode und jeder Verschachtelung statt. Das bedeutet, dass die Relationen von Teilsetergebnissen ebenfalls in derselben Anfrage mit denselben Parametern beschränkt werden können.

7

Fazit und Ausblick

7.1 Zusammenfassung

Das Ziel dieser Arbeit war es, eine API für eine medizinische Datenbank auf ausführlicher theoretischer Basis zu entwickeln. Individuelle Interessen an der Datenbank sollten genauso mit in die Überlegungen einfließen, wie technische Konventionen und aktuelle Software-Entwicklungen. Es sollte ein Gesamtüberblick über die Realisation einer Schnittstelle geschaffen werden, der sich leicht auf andere Fallbeispiele übertragen lässt.

Parallel zu dieser Ausarbeitung wurde eine API für das Laravel-Backend geschrieben, die in GraphQL Syntax und über einen HTTP Endpunkt Datenbankabfragen beanwortet und alle hinterlegten Modelle, die für ein Beispielsystem notwendig sind, abdeckt. In den Funktionen inbegriffen sind Registrierung, Autorisierung und Paginierung. Der Einsatz von Software, basierend auf der Tinnitus Datenbank auf Smartphones oder Rechnern, wäre somit machbar. Die noch nicht abgedeckten Tabellen der Datenbank können mit geringem Aufwand eingebunden werden.

7.2 Ausblick

Da es mit dem vorliegenden Anwendungsfall schwierig ist, die Überlegenheit von GraphQL über REST auszuweisen, kann nicht von einer uneingeschränkten Empfehlung für diesen Ansatz gesprochen werden. Trotzdem ist durch die Arbeit gezeigt, dass eine

Lösung mit GraphQL stabil sowie kompakt ist und eine außergewöhnlich praktische Abfragesprache auf der Seite des Client anzubieten hat.

Unabhängig von der später getroffenen Entscheidung, profitiert die Tinnitus Datenbank stark von einer Schnittstelle. Diese Arbeit soll bei Argumenten im Zuge der Zukunftsplanung als Hilfe dienen. Es wartet der Einsatz von Visualisierungs-, Statistik- oder einfach nur Supervising-Software, die ohne API nicht umzusetzen wären.

Literaturverzeichnis

[1] Probst, T., Pryss, R.C., Langguth, B., Spiliopoulou, M., Landgrebe, M., Vesala, M., Harrison, S., Schobel, J., Reichert, M., Stach, M., Schlee, W.: Outpatient tinnitus clinic, self-help web platform, or mobile application to recruit tinnitus study samples? Frontiers in Aging Neuroscience **9** (2017) 113

[2] Probst R., Grevers G., Iro H: Hals-Nasen-Ohren-Heilkunde, 3. Aufl. Thieme (2008) S.233.

[3] TINNET: Description of WG 2: Data management in a central database and identification of subtype candidates. `https://tinnet.tinnitusresearch.net/index.php/2015-10-29-10-22-16/wg-2-database` (2014) Letzter Aufruf: 17.11.2019 12:08.

[4] Tinnitus Research Initiative Foundation: Tinnitus Database. `https://tinnitusresearch.net/index.php/for-researchers/tinnitus-database` (2019) Letzter Aufruf: 17.11.2019 11:24.

[5] COST: BM1306. `https://www.cost.eu/actions/BM1306/#tabs|Name:overview` (2014) Letzter Aufruf: 17.11.2019 16:27.

[6] Tinnitus Research Initiative Foundation: CONSENSUS FOR PATIENT ASSESSMENT AND OUTCOME MEASUREMENTS. `https://tinnitusresearch.net/index.php/for-researchers/resources` (2006) Letzter Aufruf: 17.11.2019 17:11.

[7] Zeman, F., Koller, M., Schecklmann, M. et al.: Tinnitus assessment by means of standardized self-report questionnaires: Psychometric properties of the Tinnitus Questionnaire (TQ), the Tinnitus Handicap Inventory (THI), and their short versions in an international and multi-lingual sample. In: Health Qual Life Outcomes 10, 128). (2012)

[8] Genitsaridi, E., Partyka, M., Gallus, S., Lopez-Escamez, J.A., Schecklmann, M., Mielczarek, M., Trpchevska, N., Santacruz, J.L., Schoisswohl, S., Riha, C., Lourenco, M., Biswas, R., Liyanage, N., Cederroth, C.R., Perez-Carpena, P., Devos, J., Fuller, T., Edvall, N.K., Hellberg, M.P., D'Antonio, A., Gerevini, S., Sereda, M., Rein, A., Kypraios, T., Hoare, D.J., Londero, A., Pryss, R., Schlee, W., Hall, D.A.: Standardised profiling for tinnitus research: The european school for interdisciplinary tinnitus research screening questionnaire (esit-sq). Hearing Research **377** (2019) 353 – 359

[9] Stackoverflow: Developer Survey Results 2019. `https://insights.stackoverflow.com/survey/2019#technology` (2019) Letzter Aufruf: 18.12.2019 9:52.

[10] Hunter, L. Kirsten: Irresistible APIs. Manning Publications Co. (2017) Page 9.

[11] DIN-Normenausschuss Informationstechnik und Anwendungen (NIA): ISO/IEC 25000. `https://www.din.de/de/mitwirken/normenausschuesse/nia/normen/wdc-beuth:din21:204260933` (2014) Letzter Aufruf: 16.04.2020 17:12.

[12] Vogel, O., Arnold, I., Chughtai, A., Ihler, E., Kehrer, T., Mehlig, U., Zdun, U.: Software-Architektur: Grundlagen - Konzepte - Praxis. Spektrum Akademischer Verlag (2008)

[13] Feng Qiu: A Recommendation System for Web API Services (2015) Master Thesis, Ph.D, Southeast University (China.

[14] Alan Frye: A Bit of API History. `https://www.benefitfocus.com/blogs/design-engineering/bit-api-history` (2015) Letzter Aufruf: 09.01.2020 17:06.

[15] Free Software Foundation, Inc.: The GNU C Library - 1.2.1 ISO C. `https://www.gnu.org/software/libc/manual/html_mono/libc.html#ISO-C` (2002) Letzter Aufruf: 08.01.2020 18:49.

[16] Prabath Siriwardena: Advanced API Security: OAuth 2.0 and Beyond. Apress (2019) Chapter 1. APIs Rule! - API Economy.

[17] Birrell, A.D., Nelson, B.J.: Implementing remote procedure calls. ACM Trans. Comput. Syst. **2** (1984) 39–59

[18] Fielding, R.T., Taylor, R.N.: Architectural Styles and the Design of Network-Based Software Architectures. PhD thesis (2000) AAI9980887.

[19] Mark Massé: REST API: Design Rulebook. O'Reilly (2012) Chapter 1. Introduction.

[20] Mark Edward Soper: CompTIA IT Fundamentals+ FC0-U61 Cert Guide, First Edition. Pearson IT Certification (2018) Part 4, Chapter 22, Query Languages.

[21] Larry Rockoff: The Language of SQL. Pearson (2017) S. 2ff, What Is SQL?

[22] Brenda Jin, Saurabh Sahni, Amir Shevat: Designing Web APIs. O'Reilly (2018) S.19ff Event Driven APIs, S.81ff Scaling APIs.

[23] Paul Krill: Microsoft, IBM, SAP discontinue UDDI registry effort. `https://www.infoworld.com/article/2673442/microsoft--ibm--sap-discontinue-uddi-registry-effort.html` (2005) Letzter Aufruf: 30.03.2020 13:37.

[24] Doug Tidwell, James Snell, Pavel Kulchenko: Programming Web Services with SOAP. O'Reilly (2001) S.21ff & S.79ff.

[25] Dennis Kieselhorst: Einführung in Swagger: Mehr als nur Schnittstellenbeschreibung. `https://entwickler.de/online/web/openapi-swagger-579827368.html` (2018) Letzter Aufruf: 06.04.2020 11:38.

[26] SmartBear Software: Data Models (Schemas). `https://swagger.io/docs/specification/data-models/` (2019) Letzter Aufruf: 07.04.2020 12:29.

[27] OpenAPI Initiative: OpenAPI Specification. `https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.3.md` (2020) Letzter Aufruf: 09.04.2020 12:29.

[28] Topley, K.: Java Web Services in a Nutshell. In a Nutshell (o'Reilly) Series. O'Reilly (2003)

[29] Biehl, M.: OAuth: Getting Started in Web-API Security. API University Series. Createspace Independent Pub (2014)

[30] Boyd, R.: Getting Started with OAuth 2.0. Oreilly and Associate Series. O'Reilly Media, Incorporated (2012)

[31] Auth0®: Introduction to JSON Web Tokens. `https://jwt.io/introduction/` (2020) Letzter Aufruf: 18.04.2020 20:23.

[32] Porcello, E., Banks, A.: Learning GraphQL: Declarative Data Fetching for Modern Web Apps. O'Reilly Media (2018)

A

Quelltexte

Abbildungsverzeichnis

Tabellenverzeichnis

Name: Philipp Brieger Matrikelnummer: 844997

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Philipp Brieger