Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3644

Umsetzung eines sicheren Systems zur Verwaltung und Bereitstellung von Gesundheitsdaten

Frank Steimle

Studiengang:	Informatik
Prüfer:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Betreuer:	Dr. rer. nat. Matthias Wieland
Beginn am:	1. April 2014
Beendet am:	22. Oktober 2014
CR-Nummer:	C.2.4, H.2.8, H.4.1, J.3

Kurzfassung

Im Rahmen dieser Arbeit wurden Teile des Backends für das deutsch-griechische Forschungsprojekt ECHO (Enhancing Chronic patients' Health Online) entwickelt. Das Ziel des Projekts ist es mit Hilfe von Cloud-Computing-Technologien, Data Mining und Smartphones die Situation von Patienten mit chronischen Lungenkrankheiten zu verbessern. Unter anderem soll es für den Patienten möglich sein, jeden Tag Fragen via Smartphone zu beantworten. Diese Antworten sollen bei der Früherkennung von dauerhaften Verschlimmerungen der Krankheit, auch Exazerbationen genannt, helfen. Vor dem Start des ECHO Projekts wurden die Patientendaten und Untersuchungsergebnisse mit Hilfe einer Microsoft Access Datenbank verwaltet. Um die tägliche Dateneingabe zu ermöglichen, musste die Access Datenbank in eine relationale Datenbank überführt werden. Um die Daten vor unautorisiertem Zugriff zu schützen, wurde für jeden Benutzer des Backends ein Datenbankbenutzer erstellt, der nur Zugriff auf die Views und Stored Procedures hat, die er im Rahmen seiner Rolle (Arzt, Patient oder Administrator) benötigt. Zum Zugriff auf die Datenbank wurde eine REST API implementiert, die mittels OAuth 2.0 und TLS/SSL gesichert wurde. Außerdem wurden mögliche Analysen vorgestellt, die auf den erhobenen Daten durchgeführt werden können.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1 Einleitung

Das Ziel dieser Arbeit ist die Erstellung eines Backends zur sicheren Speicherung von Gesundheitsdaten. Die Arbeit wird im Rahmen des deutsch-griechischen Forschungsprojekts *Enhancing Chronic patients' Health Online* (ECHO) durchgeführt, das vom Bundesministerium für Forschung und Bildung (BMBF) gefördert wird. Ziel des Projekts ist die Verbesserung der Gesundheit von Patienten mit chronischen Atemwegserkrankungen durch Onlinedienste. Im folgenden wird die Aufgabenstellung dieser Arbeit erläutert. Danach wird ein Einblick in den Hintergrund gegeben und die Motivation beschrieben.

1.1 Aufgabenstellung

Ziel dieser Diplomarbeit ist es ein System prototypisch zu realisieren, welches es ermöglicht Gesundheitsdaten sicher abzuspeichern. Dazu muss einerseits ein Grunddatenmodell für Patientendaten erstellt werden. Hierfür existiert bereits eine Access Datenbank, welche von Ärzten bei der Untersuchung verwendet wird. Diese muss in ein standardisiertes Datenmodell umgewandelt werden (SQL) und in einem Datenbanksystem sicher bereitgestellt werden. Es müssen zudem geeignete Sicherheitskonzepte erstellt werden, um die Daten gegen unbefugten Zugriff abzusichern, (z. B. gegen Angriffe durch SQL-Injection oder durch Verschlüsselung).

Zum Zweiten sollen in Zukunft Gesundheitsdaten täglich durch die Patienten selbst mit Hilfe von Apps (Anwendungen auf mobilen Endgeräten) erfasst werden. Dafür muss das existierende Datenmodell erweitert werden, um diese Daten aufzunehmen. Des Weiteren soll eine Schnittstelle (API) für die mobilen Endgeräte angeboten werden und Sicherheitskonzepte für den Datenaustausch zwischen Datenbank und Apps erstellt werden. Eine weitere API zum Zugriff auf die Gesundheitsdaten sollte für eine zeitgleich zu bearbeitende Diplomarbeit erstellt werden. Da diese sich die Bearbeitung dieser Diplomarbeit jedoch verzögerte, wurde diese API nicht implementiert.

1.2 Hintergrund

Chronisch obstruktive Lungenerkrankung (englisch: chronic obstructive pulmonary disease, Abkürzung: COPD) bezeichnet eine Gruppe von Krankheiten der Lunge, bei der die Atemwege verengt oder eingeengt sind. Diese Schädigung der Lunge ist oft die Folge von jahrelangem Rauchen oder anderen länger andauernden Reizungen der Lunge durch Schadstoffe in der Luft. Typische Symptome sind die sogenannten **AHA**-Symptome: **A**temnot bei körperlicher Belastung, täglicher **H**usten über längere Zeit und **A**uswurf (beim Husten hervorgebrachter Schleim aus den Atemwegen).

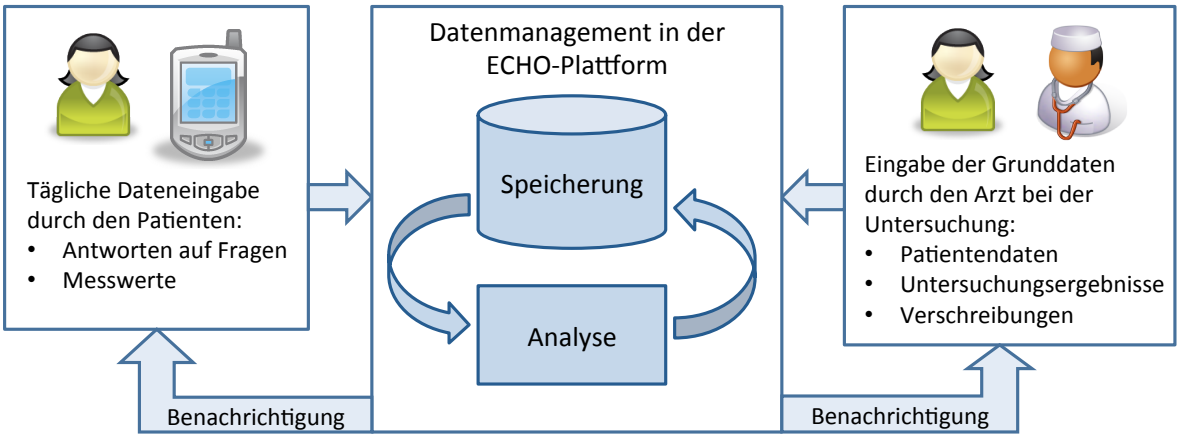

Abbildung 1.1: Datenmanagement in der ECHO Plattform [BKK+14]

Wenn sich die Symptome einer COPD bei einem Patienten in kurzer Zeit drastisch verschlimmern, spricht man von einer Exazerbation. Schwere Exazerbationen können sogar Krankenhausaufenthalte erforderlich machen. Da sich mit jeder Exazerbation der Zustand des Patienten nachhaltig verschlechtert und das Risiko für weitere Exazerbationen erhöht wird, ist es wichtig, mögliche Exazerbationen frühzeitig zu erkennen und zu vermeiden. [copa]

1.3 Motivation

Im Projekt ECHO soll ein System entwickelt werden, das Cloud-Computing-Technologien, Data Mining und Smartphones einsetzt, das Ärzten hilft, ihre Patienten besser zu überwachen und zu betreuen. Damit soll das Risiko für Exazerbationen gesenkt werden. Über die Smartphones beantworten die Patienten jeden Tag Fragen zu ihrem Zustand und geben nach Möglichkeit noch Messwerte wie Blutdruck oder Sauerstoffsättigung an, wie in Abbildung 1.1 dargestellt. Die Daten werden zur ECHO-Plattform übertragen und können dann vom Arzt eingesehen werden, der dadurch den Zustand seiner Patienten besser überwachen kann. Identifiziert die ECHO-Plattform durch die Analyse der Daten bestimmte medizinische Situationen, können sowohl der Arzt als auch der Patient benachrichtigt werden, um frühzeitig eine Verschlechterung des Zustandes des Patienten zu verhindern. Der Arzt kann über die ECHO-Plattform nicht nur den Zustand seiner Patienten überwachen, sondern auch die Patientendaten, Untersuchungsergebnisse oder Verschreibungen speichern.

1.4 Gliederung

Die Arbeit ist in folgender Weise gegliedert:
Kapitel 2 beschreibt verwandte Arbeiten und Grundlagen der Arbeit. **Kapitel 3** stellt das in der Arbeit erstellte Konzept zur Verwaltung, zum Zugriff und zur Analyse der Gesundheitsdaten vor. In

Kapitel 4 wird die Umsetzung des Konzepts beschrieben. **Kapitel 5** fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

2 Verwandte Arbeiten und Grundlagen

In diesem Kapitel werden zuerst verwandte Arbeiten vorgestellt. Es werden Methoden vorgestellt um Daten in Datenbanken abzusichern. Abschließend wird auf den Architekturstil Representational State Transfer (REST) und HTTP Authentifizierungsmechanismen, die mit REST genutzt werden können, eingegangen.

2.1 Verwandte Arbeiten

Bei der elektronischen Verarbeitung von Patientendaten spielt Sicherheit eine große Rolle. Ragib Hasan et al. befassen sich in [HWS07] mit Anforderungen an einen sicheren Speicher für Gesundheitsdaten. Dafür untersuchten sie den Health Insurance Portability and Accountability Act (HIPAA [hip]). Der HIPAA besteht aus zwei Teilen. Der erste Teil regelt den Krankenversicherungsschutz, und der zweite Teil behandelt die elektronische Patientenakte und ihre Sicherheit. Außerdem untersuchten sie noch die Occupational Safety and Health Administration Regulation [Occ], die den Umgang mit Gesundheitsdaten von Arbeitnehmern in den USA regelt, und die europäische Datenschutzrichtlinie (EU-Direktive 95/46/EG, [Eur]), die die Sicherheit im Umgang mit persönlichen Daten regelt. Aus diesen Gesetzen konnten dann die folgenden Anforderungen abgeleitet werden:

Vertraulichkeit und Zugangskontrolle: Da Gesundheitsdaten sensible Informationen darstellen, muss der Speicher ihre Vertraulichkeit garantieren, insbesondere dürfen nur autorisierte Personen Zugang erhalten. Um die Vertraulichkeit der Daten sicherzustellen sollten sie sowohl im Speicher selbst als auch während des Transports verschlüsselt werden. Falls das Speichermedium gewechselt wird, muss sichergestellt werden, dass die Daten auf dem alten Medium vertraulich bleiben.

Integrität: Das System muss die Integrität der Daten sicherstellen, das heißt es muss sie vor unautorisierten Änderungen schützen. Insbesondere vor solchen, die durch einen Nutzer des Systems erfolgen, der nicht autorisiert ist, die Daten zu ändern. Durch die Sicherheitsmaßnahmen soll es auch möglich sein, geänderte Daten zu erkennen.

Verfügbarkeit und Performance: Die Gesundheitsdaten sollen immer in einer angemessenen Zeit zur Verfügung stehen. Falls zur Erfüllung dieser Anforderung ein Index notwendig ist, muss dieser auch vertraulich sein, damit es nicht möglich ist über den Index Rückschlüsse auf die Gesundheitsdaten zu ziehen.

Logging: Jeder Zugriff auf das System sollte zuverlässig geloggt werden. Insbesondere sollte jeder Zugriff und jede Modifikation auf die Gesundheitsdaten geloggt werden. Durch die Logs sollte es möglich sein, jede Änderung nachzuvollziehen und ihren Verursacher zu ermitteln.

Vorhaltezeit und Migration: Da die Gesetzte teilweise lange Vorhaltezeiten für bestimmte Daten vorschreiben, sollte das System lange Vorhaltezeiten garantieren zu können. Dabei sollte auch auf zuverlässige und nachvollziehbare Migrationsstrategien geachtet werden, für den Fall, dass Hardware ausfällt oder ersetzt werden muss, weil sie zu alt ist.

Backup: Es sollten Backup- und Restoreoperationen zur Verfügung stehen. Dabei sollte darauf geachtet werden, dass die Backups nicht am selben Ort aufbewahrt werden wie die Gesundheitsdaten. Sonst wären die Daten im Fall eines Brandes oder anderer Naturkatastrophen verloren.

Kosten: Das System sollte kostengünstig sein, da das Einhalten von Bestimmungen wie HIPAA einen nicht zu unterschätzenden Mehraufwand bedeutet. Das kann durch den Einsatz von Standardhardware und billigem Massenspeicher erreicht werden.

In der ISO 27001 (Specification for Information Security Management [iso]) werden Verantwortlichkeit und Nachvollziehbarkeit als Bestandteile, Vertraulichkeit, Integrität und Verfügbarkeit der Daten sogar als wesentliche Bestandteile der Informationssicherheit genannt. Verantwortlichkeit bedeutet, dass es einen persönlich Verantwortlichen für jedes zu schützende Gut gibt. Nachvollziehbarkeit steht zum einen dafür, dass es nachvollziehbar sein muss, wie das System in den aktuellen Zustand gekommen ist und zum anderen, dass durch fortlaufende Prozesse sichergestellt wird, dass das System noch den Anforderungen entspricht.

Mit Cloud Computing werden die Eigenschaften Skalierbarkeit, Verfügbarkeit, Ausfallsicherheit und eine Reduktion der Kosten verbunden. Mittels Cloud Computing lässt sich unter anderem das Problem lösen, dass die Daten eines Patienten bei verschiedenen Ärzten liegen und diese erst zusammengeführt werden müssen, um eine vollständige Sicht auf den Patienten zu haben. Deshalb schlagen Dalia Sobhy et al. in [SESAE12] mit MedCloud ein HIPAA konformes System zur Verwaltung von elektronischen Patientenakten in der Cloud vor. Mittels eines von den Autoren gestellten SDKs können sogenannte Services für MedCloud geschrieben werden. Services sind sind Programme, die auf den Patientendaten arbeiten, wie zum Beispiel UpdatePatientById. Das System besteht aus drei Teilen: dem Data Storage Layer, dem Server Management Layer und dem Application Layer. Im Data Storage Layer werden die Patientendaten in einem verteilten Dateisystem gespeichert. Das Dateisystem wird durch eine Data Warehouse Erweiterung ergänzt, die den schnellen Zugriff auf die Daten ermöglicht. Das Data Warehouse wird mittels einer spaltenorientierten NoSQL Datenbank umgesetzt. Durch die Schemalosigkeit der NoSQL Datenbank ist es möglich, dass verschiedene Krankenhäuser das System nutzen, obwohl sie unterschiedliche Informationen über den Patienten speichern. Außerdem ist das System dadurch einfach zu erweitern, da Services beliebige Daten speichern können. Der Server Management Layer besteht aus einer Master-Slave-Archtitektur. Der Master nimmt dabei die Anfragen aus dem Application Layer entgegen und verwaltet das verteilte Dateisystem. Außerdem ist er dafür zuständig, die eingehenden Requests auf die Slaves zu verteilen. Die Slaves bestehen aus zwei Komponenten: dem Data Storage Manager, der die Daten auf diesem Knoten verwaltet, und dem Task Manager, der die Tasks verwaltet und ausführt, die der Knoten vom Master zugewiesen bekommt. Der Application Layer stellt die Schnittstelle zum Client dar, der über ein RESTful HTTP Interface mit der MedCloud kommunizieren kann. Der Application Layer stellt zusätzlich noch einige Funktionen des Systems zur Verfügung, wie zum Beispiel den Authenticator, der dafür zuständig ist Benutzer einzuloggen, die Service Registry, die alle an der MedCloud angemeldeten Services verwaltet, oder den Authorizer, der anhand von HIPAA konformen Regeln entscheidet, ob der Benutzer den angeforderten Service verwenden darf.

Die MedCloud ist durch die Schemalosigkeit der NoSQL Datenbank und die selbstgeschriebenen Services eine sehr felxible Lösung. Die Funktionalität des ECHO Projekts könnte wahrscheinlich auf Services abgebildet werden, aber ob die Daten wirklich sicher wären bleibt offen. Abgesehen vom Authorizer, der anhand von HIPAA konformen Regeln Zugang zu Services gibt, gibt es aber keine Details dazu, mit welchen Mitteln die Daten HIPAA konform und sicher abgespeichert werden.

Ein anderes Cloud-System aus dem Trustworthy Clouds (TClouds) Projekt schlagen Deng et al. in [DPNB11] vor. Sie erstellten eine Cloud Anwendung, um alle Aspekte der Behandlung von Depressionspatienten zu unterstützen. Sie implementieren dafür drei Komponenten: Medikamenten Management, Schlaf- und Lichtmanagement und Management der sportlichen Aktivitäten. Dabei sollen allerdings nicht nur Ärzte und Patienten auf die Daten zugreifen können, sondern auch Familienmitglieder des Patienten, Apotheker, sportliche Einrichtungen und öffentliche Stellen. Die sportlichen Einrichtungen sollen auf diesem Weg beispielsweise Trainingspläne festlegen oder überprüfen können, ob der Patient regelmäßig Sport treibt. Die öffentlichen Stellen sollen über ihren Zugang Statistiken einsehen und prüfen können, ob Richtlinien eingehalten werden.

Die Architektur ist in vier Schichten aufgeteilt: Das Data Store Layer, Back-End Layer, Middle-Tier Layer und Front-End Layer, in dem die Anwendungen für die verschiedenen Beteiligten laufen. Es gibt vier Data Stores, die gleichzeitig das System in vier Application Domains einteilen. Im Prescription Repository sind beispielsweise alle Daten über Medikamentenverschreibungen gespeichert. Außerdem gibt es noch die Repositories für elektronische Patientenakten (EHR), für die persönlichen Gesundheitsdaten (PHR) und für die sportlichen Aktivitäten. Im Middle-Tier befinden sich Anwendungen, die die Daten aus den Repositories bearbeiten und mit Anwendungen aus anderen Application Domains kommunizieren. Beispielsweise teilt die Anwendung für Verschreibungen aus der Application Domain des Arztes (EHR Domain) der Verschreibungsmanagement Anwendung aus der Prescription Domain mit, wenn ein neues Rezept ausgestellt wird. Wenn der Patient nun in eine an das System angeschlossene Apotheke geht, kann der Apotheker das verschriebene Medikament und die Dosierung aus seiner Anwendung ablesen. Anschließend zeigen Deng et al. wie diese Architektur mit Hilfe von Openstack umgesetzt werden kann.

Nach einer Analyse der Sicherheitsanforderungen für die Data Stores und die Datenübertragung zwischen den Schichten kommen Deng et al. zu dem Ergebnis, dass für die Daten und deren Übertragung die Vertraulichkeit, Integrität und Verfügbarkeit der Daten notwendig ist. Als zusätzliche Sicherheitsmaßnahme schlagen sie vor, alle Daten vor dem Upload vom Patienten verschlüsseln zu lassen. Für die Verschlüsselung betrachten sie attribut-basierte Verschlüsselung und Lizenzen, allerdings kommen sie zu dem Schluss, dass beide nicht ausreichenden Schutz bieten.

Das hier vorgestellte System ist sehr umfangreich durch die vielen beteiligten Parteien. Um die in ECHO geforderte Funktionalität umzusetzen, ist keine Architektur notwendig, in der mehrere Data Stores existieren.

2.2 Datenbanksicherheit

In diesem Abschnitt werden Aspekte der Datenbanksicherheit vorgestellt. Zum einen wird auf die Möglichkeit eingegangen, mit Hilfe von Views die Zugriffskontrolle zu ergänzen. Zum anderen werden Möglichkeiten vorgestellt, mit denen sich SQL Injections verhindern lassen.

Zugriffskontrolle mit Views

Bei vielen Datenbanksystemen kann man Zugriffsrechte bis zur Tabellenebene vergeben. In manchen Fällen ist dies jedoch nicht ausreichend, beispielsweise ist es nicht ausreichend, wenn die Noten aller Studenten in einer Tabelle gespeichert sind. Um zu verhindern, dass ein Student die Noten der anderen einsehen kann, können Views zum Schutz der Daten eingesetzt werden. Dadurch muss man nicht mehr darauf vertrauen, dass die Anwendung das Prädikat in der Where Klausel richtig einsetzt, da die Verantwortung durch die Views an die Datenbank übergeht. Pro Student kann ein View angelegt werden, oder es können dynamische Views angelegt werden, die Daten in Abhängigkeit des aktuellen Benutzers anzeigen. Da die meisten Webanwendungen aber nur über eine Verbindung zur Datenbank verfügen, die immer denselben Benutzer verwendet, ist es nicht möglich dynamische Views nach diesem Muster zu verwenden. Roichman und Gudes schlagen deswegen in [RG07] vor parametrisierte Views zu verwenden. Parametrisierte Views nehmen beim Aufruf einen Parameter entgegen, zum Beispiel eine User ID, und zeigen basierend darauf Daten an. Solche Views können mit Stored Procedureds implementiert werden. Roichman und Gude erweitern parametrisierte Views dahingehend, dass nicht mehr die (erratbare) Benutzer ID als Parameter verwendet wird, sondern ein zufälliger Key. Der zufällige Key wird bei jedem Login erzeugt, dann in der Benutzertabelle in der Datenbank abgelegt und dem Benutzer mitgeteilt, der ihn bei jeder Anfrage mitschicken muss. Dadurch, dass der zufällige Key auf dem Server hinterlegt werden muss, ist diese Lösung nicht im REST Umfeld einsetzbar. Da es Node.js ermöglicht, den Datenbankbenutzer einer aktiven Verbindung zu wechseln, werden in dieser Arbeit Views verwendet, die Daten in Abhängigkeit des aktuellen Benutzers zur Verfügung stellen.

Angriffe durch SQL Injections vermeiden

Als SQL Injection bezeichnet man das Injizieren von SQL Code in die Anwendung und die anschließende Ausführung des durch den SQL Code veränderten SQL Statements durch die Datenbank. Dies funktioniert, wenn es in der Anwendung SQL Statements gibt in die vor dem Ausführen Benutzereingaben ungeprüft oder mangelhaft geprüft eingefügt werden. Das Open Web Application Security Project (OWASP) gab 2013 eine Liste der zehn größten Risiken für Webanwendungen heraus, auf der die Injection den ersten Platz belegt [OWA13].

In Listing 2.1 ist eine Anfrage dargestellt, mit deren Hilfe eine Anwendung dem Angestellten 123 sein Gehalt für einen bestimmten Monat ausgeben kann. Die SQL Anfrage wird zusammengesetzt aus einem vorgefertigten Teil und einer Eingabe des Benutzers. Wenn der Benutzer mehr als einen bestimmten Monat eingibt, zum Beispiel **01.2007 or 1 = 1**, erhält man allerdings eine Anfrage wie in

Listing 2.1 Beispiel SQL Injection Teil 1 aus [RG07]

```
strSQL = "SELECT Salary
            FROM Salary_Table
            WHERE Employee_No = 123
            AND Salary_Date = " + dateParam
```

Listing 2.2 Beispiel SQL Injection Teil 2 aus [RG07]

```
strSQL = "SELECT Salary
            FROM Salary_Table
            WHERE Employee_No = 123
            AND Salary_Date = 01.2007 or 1 = 1"
```

Listing 2.2 dargestellt. Wenn diese Anfrage von der Anwendung verarbeitet wird, ist das Ergebnis eine Auflistung aller Gehälter aller Angestellten, da das Prädikat 1=1 immer wahr ist.

Um mögliche SQL Injections zu vermeiden, schlägt die OWASP drei Optionen vor:

1. Prepared Statements: Die Anfrage wird mit Platzhaltern an die Datenbank geschicken. Dadurch kann die Datenbank zwischen der Anfrage und den Daten, mit denen die Anfrage vervollständigt wird, unterscheiden. Damit wird es einem Angreifer unmöglich gemacht, den Sinn der Anfrage zu verändern.

2. Stored Procedures: Anstatt eine Query zu verwenden, können Anfragen über Stored Procedures gemacht werden. Das sind Routinen, die vorher in der Datenbank hinterlegt wurden. Sie haben damit denselben Effekt wie Prepared Statements, da die Daten ebenfalls getrennt von der Anfrage übermittelt werden. Bei der Implementierung von Stored Procedures ist allerdings darauf zu achten, dass kein SQL Statement mit ungeprüften Parametern erstellt wird, wie in Listing 2.2.

3. Escapen sämtlicher Benutzereingaben: Jede Datenbank kennt bestimmte Zeichen um Benutzereingaben vom Rest der Anfrage zu unterscheiden. Werden diese Zeichen verwendet um Benutzereingabe zu markieren, kann die Datenbank sie von der eigentlichen Anfrage unterscheiden. Wenn der Angreifer jedoch weiß, welche Datenbank verwendet wird, kann er, wie im Beispiel dargestellt, diesen Schutz umgehen.

Zusätzlich werden noch das Least-Privilege-Prinzip und White List Input Validierung empfohlen. Das Least-Privilege-Prinzip besagt, dass den Benutzern nur die Rechte gewährt werden, die sie zur Erfüllung ihrer Aufgaben benötigen. Dadurch entsteht im Fall eines Angriffs auch ein kleinerer Schaden, als wenn der Benutzer alle Rechte der Anwendung hätte. Bei der White List Validierung wird jede Benutzereingabe, die einem bestimmten Muster folgt, auf ihre Gültigkeit geprüft. Das kann zum Beispiel durch reguläre Ausdrücke erfolgen.

Kindy and Pathan fassten in [KP11] Maßnahmen gegen SQL Injections zusammen. Allerdings sind alle bis auf Roichmans und Gudes Lösung [RG07] nur mit zusätzlicher Software umsetzbar. In dieser Arbeit wird die Empfehlung der OWASP zum Schutz vor SQL Injections umgesetzt.

2.3 REST und ROA

Representational State Transfer (REST) ist ein Architekturstil für verteilte Anwendungen. Der Begriff REST wurde zuerst von Roy Fielding in seiner Dissertation im Jahr 2000 beschrieben. Architekturen, die sich an REST orientieren, werden teilweise auch Resource-oriented Architecture (ROA) genannt. Durch den Einsatz von REST ergeben sich im Gegensatz zu anderen Architekturstilen laut Stefan Tilkov [Til11] die folgenden Vorteile:

- Der Grad der Kopplung der zu verbindenden Systeme wird minimiert, da durch die uniforme Schnittstelle alle Operationen und wie sie aufzurufen sind im voraus bekannt sind.
- Dadurch, dass die meisten REST Implementierungen Webstandards wie HTTP, URIs und XML einsetzen, wird die Interoperabilität dieser Systeme erhöht, da die Gegenseite nur diese Standards kennen muss, um mit der REST Implementierung zu kommunizieren.
- Da es bei REST nur eine Schnittstelle gibt und nicht immer wieder eine neue definiert werden muss, ist die Wiederverwendbarkeit höher.
- Performance und Skalierbarkeit können erhöht werden, wenn das System auf der Basis von HTTP und den REST Regeln entworfen wird. Dann ist es nämlich möglich, Anfragen mit Hilfe von Caches zu beantworten oder aufeinander folgende Anfragen nicht mit demselben Server zu beantworten,

Die wesentlichen Grundkonzepte um diese Vorteile zu erreichen sind laut Tilkov [Til11]:

Eindeutige Identifikation von Ressourcen: Ein wesentlicher Bestandteil von REST sind Ressourcen. Die Ressource ist ein abstraktes Konzept und bezeichnet alle Objekte, die durch die Anwendung nach außen hin sichtbar gemacht werden sollen oder Listen dieser Objekte. Ressourcen sollten dabei aber nicht nur einfach Datenbankeinträge nach außen sichtbar machen. Sie sind kein Konzept der Persistenzschicht, sondern ein nach außen sichtbares Konzept der Anwendungsschicht. Jede Ressource wird durch zwei wichtige Eigenschaften definiert: eine eindeutige ID und eine oder mehrere Repräsentationen. Zur eindeutigen Identifikation einer Ressource können zum Beispiel Uniform Resource Identifiers (URIs) verwendet werden. URIs bestehen aus fünf Bestandteilen: Scheme, Authority (bestehend aus Host und Port), Path, Query und Fragment. Dabei sind Authority, Query und Fragment optional.

scheme://authority/path?query#fragement

Verschiedene Repräsentationen: Der zweite wesentliche Bestandteil einer Ressource sind ihre Repräsentationen. Über die Repräsentationen werden die Ressourcen dem Rest der Welt zu Verfügung gestellt und über diese können sie bearbeitet werden. Über Content Negotiation kann ein Client die Ressource in einem Format anfordern, das er verarbeiten kann. Gängige Repräsentationen sind zum Beispiel XML, HTML, CSV oder JSON.

Verlinkungen: Mit *Hypermedia As The Engine Of Application State* (HATEOAS) wird das Konzept von Verlinkungen beschrieben. Antworten des Servers sollten zum einen Links zu anderen Ressourcen enthalten, mit denen die empfangene Ressource in Relation steht. Zum anderen sollte die Antwort Links auf die Aktionen enthalten, die dem Client als nächstes zur Verfügung

stehen. Wenn der Server REST korrekt umsetzt, kann ein Client allein anhand der Links durch den Server navigieren.

Standardmethoden: Hinter diesem Punkt verbirgt sich das Konzept der uniformen Schnittstelle: jede Ressource soll denselben Satz von Operationen unterstützen. Für REST Anwendungen im HTTP Umfeld sind dies die acht Operationen von HTTP: GET, POST, PUT, DELETE, HEAD, OPTIONS, TRACE und CONNECT. Im folgenden werden GET, POST, PUT und DELETE näher erläutert.

- Die wichtigste Operation ist GET. Sie dient dazu, eine Ressource, die durch eine URI identifiziert wird, in ein einer bestimmten Repräsentation abzuholen. GET wird als safe und idempotent spezifiziert. Safe bedeutet dabei nicht, dass die Operation keine Seiteneffekte auf dem Server auslösen darf, wie zum Beispiel das Erzeugen eines Logeintrags, sondern nur, dass durch GET keine Änderung der Ressource erfolgen darf. Idempotent bedeutet, dass das mehrmalige Aufrufen der Operation dasselbe Ergebnis erzeugt wie ein einmaliger Aufruf.
- POST kann in zwei Szenarien eingesetzt werden. Das erste Szenario ist das Erstellen einer neuen Ressource. Das andere ist, wenn eine Funktionalität aufgerufen werden soll, die über keine andere HTTP-Operation abgebildet werden kann, oder wenn die andere HTTP-Operationen nicht verfügbar sind. Strikt oder falsch konfigurierte Firewalls könnten zum Beispiel nur GET und POST zulassen. In diesem Fall lässt sich ein PUT beispielsweise durch einen selbstdefinierten Header emulieren (Beispiel: X-HTTP-Method-Override : PUT).
- PUT kann dazu benutzt werden, eine Ressource zu aktualisieren oder um eine neue anzulegen. Um die Operation aufzurufen, benötigt man auf jeden Fall die URI der Ressource. Wenn also eine Ressource mit einer vordefinierten URI erstellt werden soll, sollte PUT anstatt POST verwendet werden. PUT ist außerdem ebenfalls idempotent.
- Um eine Ressource zu löschen, kann DELETE verwendet werden. Dabei müssen die Daten, die durch die Ressource dargestellt werden nicht immer gelöscht werden. Es ist auch üblich, nur ein Flag zu setzen (deleted = true) und die Daten dann nicht mehr als Ressource auszuliefern. DELETE ist ebenfalls idempotent.

Statuslose Kommunikation: Jede Nachtricht soll alle Informationen enthalten, damit sie von Client und Server verstanden werden kann. Dadurch wird die Skalierbarkeit von REST Anwendungen erhöht, da aufeinander folgende Anfragen nicht vom selben Server beantwortet werden müssen oder Caches verwendet werden können, um Anfragen zu beantworten.

Für die Implementierung einer REST Anwendung im HTTP Umfeld spielen die HTTP Statuscodes eine wichtige Rolle. Es gibt über 70 HTTP Statuscodes in fünf Statusklassen. Tabelle 2.1 gibt eine Übersicht über die Statusklassen und über häufig genutzte HTTP Statuscodes und erklärt ihre Bedeutung.

Statuscode / -klasse	Beschreibung
1xx Klasse	Informationen. Anfrage erhalten, ist aber noch in Bearbeitung.
2xx Klasse	Erfolgreiche Anfrage. Anfrage erhalten und erfolgreich verarbeitet.
200 - *Accepted*	Die Anfrage wurde erfolgreich bearbeitet, und das Ergebnis wird mit dieser Antwort übertragen.
201 - *Created*	Die Anfrage wurde erfolgreich bearbeitet und eine neue Ressource wurde erstellt. Die Adresse der neuen Ressource ist im Location Header zu finden.
204 - *No Content*	Die Anfrage wurde erfolgreich bearbeitet, und die Antwort des Servers enthält keinen Inhalt.
3xx Klasse	Umleitung. Es sind weitere Schritte seitens des Clients erforderlich.
301 - *Moved Permanently*	Die angeforderte Ressource befindet sich nun in der im Location Header angegebenen Adresse. Die Adresse sollte bei allen zukünftigen Anfragen verwendet werden.
4xx Klasse	Client Fehler. Diese Codes werden zurückgegeben, wenn es wahrscheinlich ist, dass der Fehler beim Client liegt.
400 - *Bad Request*	Die Anfrage konnte vom Server nicht verarbeitet werden, da sie falsch aufgebaut ist. Der Client sollte die Anfrage nicht ohne Änderungen wiederholen.
401 - *Unauthorized*	Die Anfrage konnte nicht verarbeitet werden, weil keine oder ungültige Authentifizierungsinformationen vorhanden sind. Die Antwort des Servers wird über den WWW-Authenticate Header mitteilen, wie die Authentifizierung durchzuführen ist.
403 - *Forbidden*	Die Anfrage wurde aufgrund mangelnder Berechtigung nicht durchgeführt. Anders als beim Code 401 wird erneutes Authentifizieren nichts ändern.
404 - *Not Found*	Die Ressource wurde nicht gefunden.
5xx Klasse	Informationen. Anfrage erhalten, ist aber noch in Bearbeitung.
500 - *Internal Server Error*	Es trat ein unerwarteter Fehler auf, der den Server daran hinderte, die Anfrage zu bearbeiten.
503 - *Service Unavailable*	Der Service steht nicht zur Verfügung. Mögliche Gründe sind zum Beispiel Wartungsarbeiten oder Überlastung.

Tabelle 2.1: Auszug aus den HTTP Statuscodes

2.4 REST Authentifizierung

In diesem Abschnitt werden Mechanismen vorgestellt, die verwendet werden können um Benutzer über HTTP zu authentifizieren. Da der HTTP Authentifizierungsmechanismus leicht zu erweitern ist, gibt es viele selbstgeschriebene Lösungen. Im folgenden wird nur auf standardisierte Verfahren eingegangen, da es sicherer ist bereits implementierte Verfahren zu benutzen, anstatt neue Verfahren zu implementieren, wobei Fehler, und damit eventuell Sicherheitslücken, nicht ausgeschlossen werden können. Zuerst wird auf die in HTTP umgesetzten Authentifizierungsmöglichkeiten eingegangen, wie sie in [Til11] beschrieben sind. Anschließend wird OAuth 2.0 vorgestellt.

HTTP Authentifizierung

Wenn ein Client auf eine geschützte Ressource zugreifen will, ohne dass er Authentifizierungsinformationen mitschickt, antwortet der Server mit dem HTTP Statuscode 401. In dieser Antwort teilt der Server dem Client mit Hilfe des WWW-Authenticate Headers, mit wie er sich zu authentifizieren hat.

WWW-Authenticate: Basic realm=„Protected“

Mit dieser Antwort teilt beispielsweise der Server dem Client mit, dass er sich über das Basic Schema authentifizieren muss. Außerdem wird dem Client mit Hilfe des Parameters realm mitgeteilt, für welchen Bereich er sich authentifizieren soll. Diese Antwort wird auch Authentication Challenge gennant.

Die standardisierten Authentifizierungsschemata für HTTP sind Basic und Digest. Beim Basic Schema werden Benutzername und Passwort mit einem Doppelpunkt als Trennzeichen konkateniert und dann mit Base64 kodiert. Wenn zum Beispiel die Kombination username und password kodiert werden soll, sieht das so aus:

base64(„username“+„:“+„password“) = dXNlcm5hbWU6cGFzc3dvcmQ=

Der so entstandene String wird im Authorization Header unter Angabe des Schemas an den Server übertragen.

Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=

Anschließend kann der Server die Base64 Kodierung rückgängig machen und den Benutzername und das Passwort überprüfen. Falls die Informationen nicht gültig sind, wird wieder die Authentication Challenge gesendet, andernfalls wird die Anfrage verarbeitet.

Das Basic Schema hat einige Schwächen:

- Das Passwort wird praktisch im Klartext übertragen, da Base64 keine Verschlüsselung ist.

- Die Identität des Servers wird nicht sichergestellt. Der Client schickt also unter Umständen seinen Benutzernamen und sein Passwort dem Angreifer.
- Die Nachricht kann auf dem Weg vom Client zum Server verändert werden.
- Wenn ein Angreifer die Nachricht abhören kann, kann er sie ganz oder teilweise noch einmal senden.

Die genannten Schwächen können aber über die Kombination mit SSL ausgeglichen werden. Bei der Kommunikation via SSL beweist der Server mit einem Zertifikat dem Client gegenüber seine Echtheit. Außerdem ist ein Abhören der Verbindung nicht mehr möglich, da sie verschlüsselt ist.

Das Digest Schema hat sich zum Ziel gesetzt, die Schwächen von Basic umgehen. Die größte Änderung gegenüber dem Basic Schema ist, dass das Passwort nie im Klartext übertragen wird. Stattdessen wird ein Hashwert übertragen, der auch Digest genannt wird. Dieser Hashwert wird berechnet indem unter anderem eine nonce benutzt wird, die der Server dem Client in der Authentication Challenge mitteilt. Eine nonce ist eine beliebige Zeichenkette, die vom Server allerdings nur einmal erzeugt wird. Im einfachsten Fall wird der Digest nach folgender Formel berechnet, wobei h() eine Hashfunktion wie zum Beispiel SHA bezeichnet:

h(h(Benutzername + „:“ + Realm +„:“ + Passwort) + „:“ + nonce + „:“ + h(HTTP-Methode + „:“ + URI))

Der Server erhält nun den Digest, die nonce und den Benutzernamen und kann dauraus auch wieder den Digest berechnen. Wenn die beiden Digests übereinstimmen, kann der Server den Client authentifizieren. Digest hat damit gegenüber Basic wesentliche Vorteile:

- Das Passwort wird nie im Klartext übertragen und kann damit nicht mitgelesen werden.
- Durch die nonce (auch in Kombination mit einem Zeitstempel) können Replay-Attacken verhindert werden.

HTTP Digest wird trotz der Verbesserungen nicht oft genutzt. Das liegt unter anderem daran, dass die Nachrichten immer noch nicht verschlüsselt sind, Man-in-the-Middle Attacken immer noch möglich sind und unterschiedliche Digest Implementierungen in Browsern und Webservern nicht kompatibel waren.

OAuth 2.0

OAuth ist ein offenes Protokoll für die sichere Autorisierung. Mit OAuth 1.0 und OAuth 2.0 kann man einer Anwendung A das Recht einräumen, auf die Daten einer Anwendung B im Namen eines bestimmten Benutzers zuzugreifen, ohne dass dieser seine Zugangsdaten der Anwendung A zur Verfügung stellen muss. OAuth 1.0 erschien 2007 und nutzte kryptografische Signaturen um die Anfrage abzusichern. Die Tatsache, dass manche Entwickler die Signaturen falsch berechneten und damit die Sicherheit nicht mehr gewährleistet war, und die Tatsache, dass das Protokoll sich nicht leicht an andere Anwendungsfälle anpassen ließ, führten zur Entwicklung von OAuth 2.0. OAuth 2.0 kommt ohne kryptografische Signaturen aus, setzt aber TLS vorraus. Außerdem wurden in OAuth 2.0 mehrere Abläufe für verschiedene Anwendungsfälle definiert. Außerdem wird die Möglichkeit

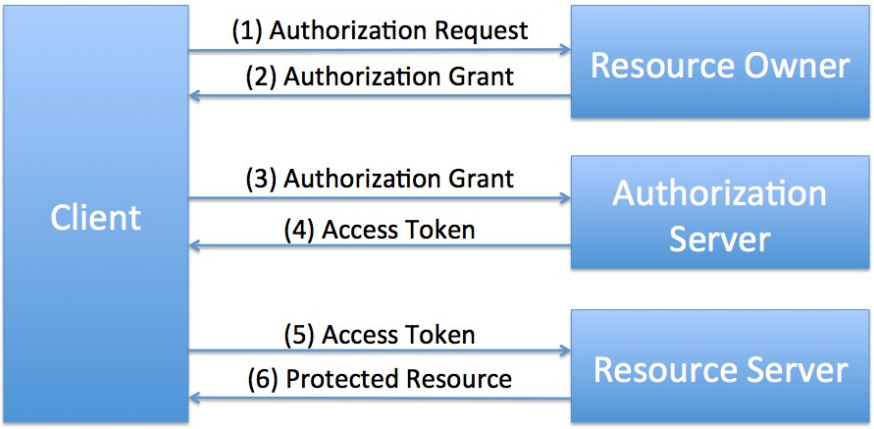

Abbildung 2.1: OAuth 2.0 Control Flow

geboten, das Protokoll um eigene Abläufe zu erweitern. Um den prinzipiellen OAuth 2.0 Ablauf zu verstehen, muss man folgende Rollen kennen:

Resource Owner: Eigentümer der Daten auf die zugegriffen werden soll.

Resource Server: Server, auf dem diese Daten liegen.

Client: Die Anwendung, die im Auftrag des Resource Owners auf besagte Daten zugreifen möchte.

Authorization Server: Server, der nach Zustimmung des Resource Owners Access Token für den Zugriff auf die geschützten Ressourcen ausstellt.

Der allgemeine Ablauf von OAuth 2.0 ist in Abbildung 2.1 dargestellt und besteht aus den folgenden Schritten [Har12]:

1. Der Client beantragt den Zugriff auf die geschützten Daten beim Resource Owner.
2. Als Antwort erhält der Client einen Authorization Grant, der zum einen aus einem Code besteht, der die Zustimmung des Resource Owners repräsentiert und zum anderen aus dem Grant Type (siehe unten)
3. Der Client sendet diesen Grant zum Authorization Server um ein Access Token zu bekommen.
4. Falls der Grant gültig ist, sendet der Authorization Server ein Access Token an den Client zurück.
5. Der Client beantragt beim Resource Server unter Verwendung des Access Token Zugriff auf die Resourcen des Resource Owner.
6. Falls das Access Token gültig ist, gewährt der Resource Server den Zugriff auf die Resourcen.

Authorization Flows

Im folgenden werden alle vier Ausprägungen des oben beschriebenen Ablaufs vorgestellt und für welche Szenarien sie sich laut [Boy12] eignen.

Der *Authorization Code Grant Flow* wird empfohlen für Webanwendungen bei denen der OAuth-Client serverseitig läuft und der Resource Owner nur über ein Webinterface mit dem Client interagiert. Mit diesem Flow lässt sich ein langfristiger Zugriff auf die geschützten Daten realisieren.

Im Gegensatz zum Authorization Code Grant Flow ist der *Implicit Grant Flow* für OAuth Clients ausgelegt, die im Browser laufen (JavaScript-Clients, Flash-Anwendungen oder Browser-Erweiterungen). Dieser Flow wird empfohlen, wenn der Zugriff nur temporär gestattet werden soll und es nicht bedenklich ist, wenn der Nutzer Zugriff auf das Access Token hat.

Der *Resource Owner Password Credentials Flow* unterscheidet sich sehr von den zwei bisher vorgestellten Flows, da er es erlaubt, die Zugangsdaten des Resource Owners direkt in ein Access Token zu tauschen, ohne den Grant-Zwischenschritt. Dieser Flow sollte deshalb auch nur von Applikationen verwendet werden, denen ein hohes Maß an Vertrauen entgegen gebracht wird, wie zum Beispiel selbst entwickelten mobilen Anwendungen. Es ist zu beachten, dass die Clientanwendung bei diesem Flow Zugriff auf die Zugangsdaten des Resource Owner bekommt.

Der *Client Credentials Flow* funktioniert ähnlich wie der Resource Owner Password Credentials Flow, allerdings ist der Client selbst der Eigentümer.

Im Rahmen dieser Arbeit wird OAuth 2.0 implementiert, da die tokenbasierte Authentifizierung verschiedene Vorteile gegenüber den anderen Verfahren bietet:

- Das Passwort muss nicht jedes Mal im Klartext übertragen werden, wie bei Basic.
- Das Passwort muss nicht unverschlüsselt gespeichert werden, damit es zur Verschlüsselung der Nachricht genutzt werden kann, wie bei Digest.
- Ein Token kann mit einer limitierten Gültigkeit ausgestellt werden. Damit geht von entwendeten Token ein kleineres Risiko aus, als von einer abgehörten Nachricht mit Basic Authentifizierung.
- Es ist möglich in dem Token Anmeldeinformationen zu speichern. Damit ist es möglich einen Benutzer zu authentifizieren, ohne immer auf die Datenbank zugreifen zu müssen.

3 Konzept

In diesem Kapitel wird zuerst die Architektur der ECHO Plattform beschrieben. Anschließend wird auf die Komponenten eingegangen, die im Rahmen dieser Arbeit entwickelt wurden. Außerdem werden Analysen vorgestellt, die mit Hilfe der erfassten Daten durchgeführt werden können.

3.1 Architektur

Die Anwendungsebene besteht, wie in Abbildung 3.1 dargestellt, aus den Smartphone-Anwendungen für Patienten und Ärzte und den jeweiligen Web-Portalen. Über Anwendungsebene können die Patienten die tägliche Dateneingabe durchführen und die Ärzte Patienten, Untersuchungsergebnisse und Verschreibungen erfassen (siehe Abbildung 1.1). Die Schnittstelle zwischen Infrastrukturebene und Anwendungsebene stellt eine RESTful API dar. Falls Analysen der Daten zeigen, dass Gesundheitsrisiken für den Patienten bestehen, kann der Patient per SMS oder E-Mail benachrichtigt werden. Ärzte werden per SMS oder E-Mail benachrichtigt, wenn ein Patient beispielsweise mehrere Tage keine Daten eingegeben hat.

Im Rahmen dieser Arbeit wurden die RESTful API, die Datenbank für die Gesundheitsdaten und die Analysekomponente neu entworfen und entwickelt. Da die Analysekomponente nicht implementiert wurde, ist sie, im Gegensatz zu den anderen beiden Komponenten, in Abbildung 3.1 rot gestrichelt umrahmt.

3.2 Gesundheitsdaten

Im folgenden wird das Datenmodell der Datenbank zur Speicherung der Gesundheitsdaten erläutert. Im darauf folgenden Abschnitt wird beschrieben, wie man unbefugten Zugriff auf diese Daten verhindern kann.

3.2.1 Datenmodell

Das in Abbildung 3.2 dargestellte Diagramm stellt das Datenmodell für die Gesundheitsdaten dar und beinhaltet zusätzlich die Benutzerverwaltung für das System. Um sicherzustellen, dass jeder Benutzer des Systems nur auf die Daten zugreifen kann, die für ihn bestimmt sind, wird jedem Benutzer eine der folgenden Rollen zugewiesen:

- **admin**: für die Accountverwaltung und administrative Aufgaben.

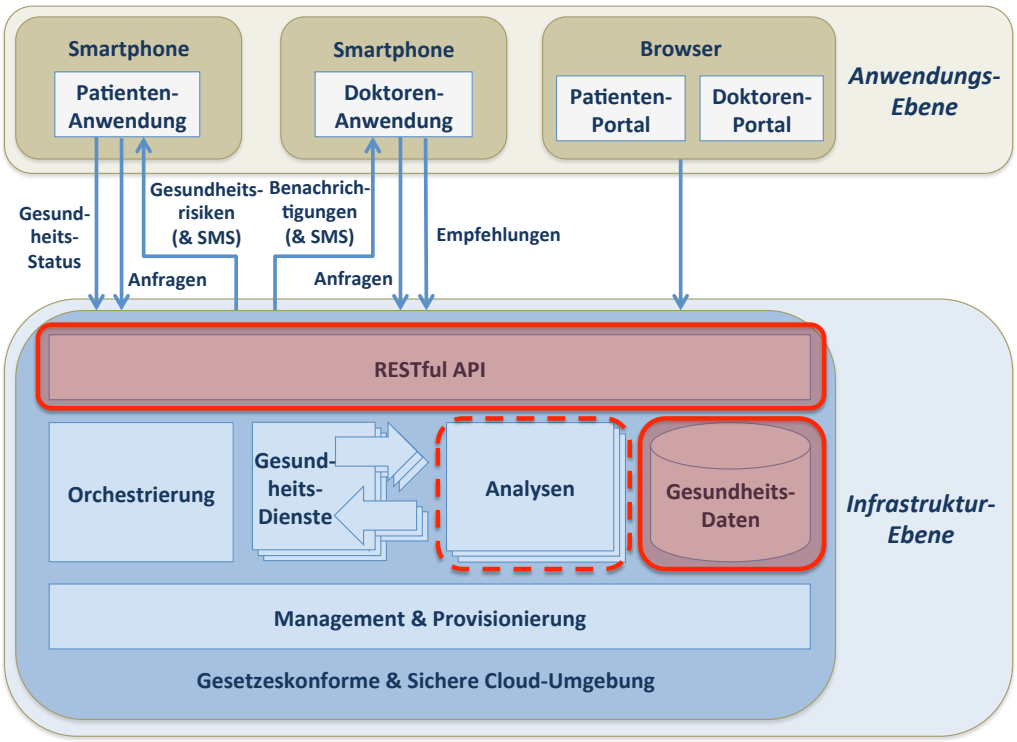

Abbildung 3.1: Architektur der ECHO Plattform. Die rot umrandeten Komponenten wurden für diese Arbeit entwickelt. Die rot gestrichelten Analysen werden als Konzept vorgestellt. [BKK+14]

- **doctor**: wird Ärzten zugewiesen, damit sie die ihnen zugewiesen Patienten überwachen können.
- **patient**: für Patienten, die Daten in das System eingeben.

In den Accounts sollen zusätzlich zur Rolle des Benutzers die Zugangsdaten für das System und die Einstellungen zu den Benachrichtigungen gespeichert werden. Die Einstellungen sollen beinhalten, ob der Benutzer benachrichtigt werden will, wie dies erfolgen soll und gegebenenfalls die Kontaktdaten zu den Benachrichtigungsmodi (SMS, Push, E-Mail).

Der Arzt soll in der Lage sein, bei jedem Besuch des Patienten Untersuchungsergebnisse und Messwerte zu speichern. Zusätzlich zu den Ergebnissen sollen auch noch die verschriebenen Medikamente und ihre Dosierung gespeichert werden. Falls der Patient durch die COPD stirbt, sollen auch die Umstände seines Todes gespeichert werden können, beispielsweise für eine spätere Analyse.

Um Patienten dem richtigen Schweregrad (A, B, C oder D, siehe Abbildung 3.3) zuzuordnen und damit die richtigen Medikamente verschreiben zu können, wird Anzahl der Exazerbationen im letzten Jahr und das Ergebnis des COPD Assessment Tests (CAT) benötigt. Der CAT besteht aus acht Fragen, die Aufschluss darüber geben sollen, wie stark der Patient durch die COPD in seinem täglichen Leben

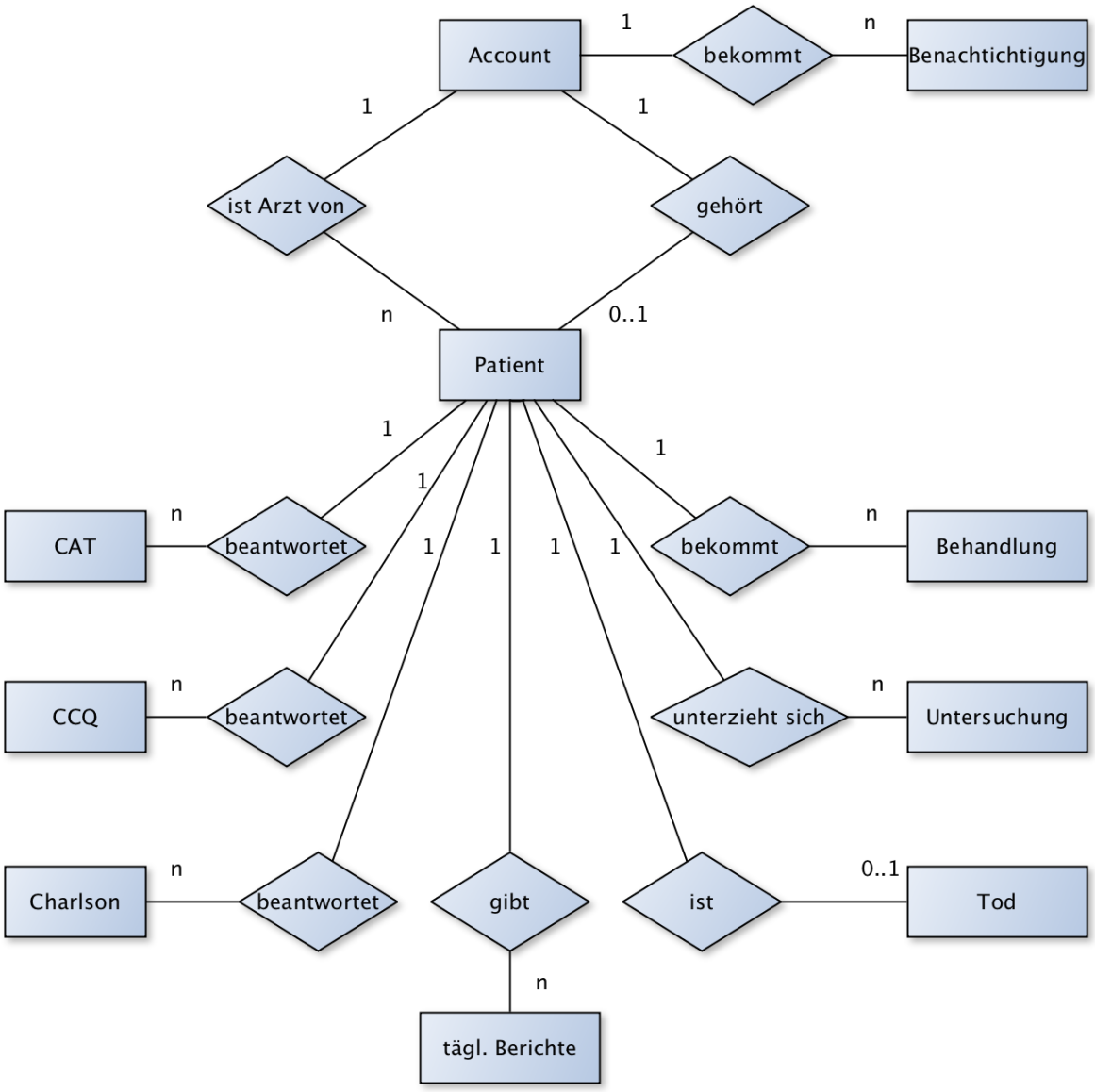

Abbildung 3.2: Grobes ER-Diagramm der Gesundheitsdaten

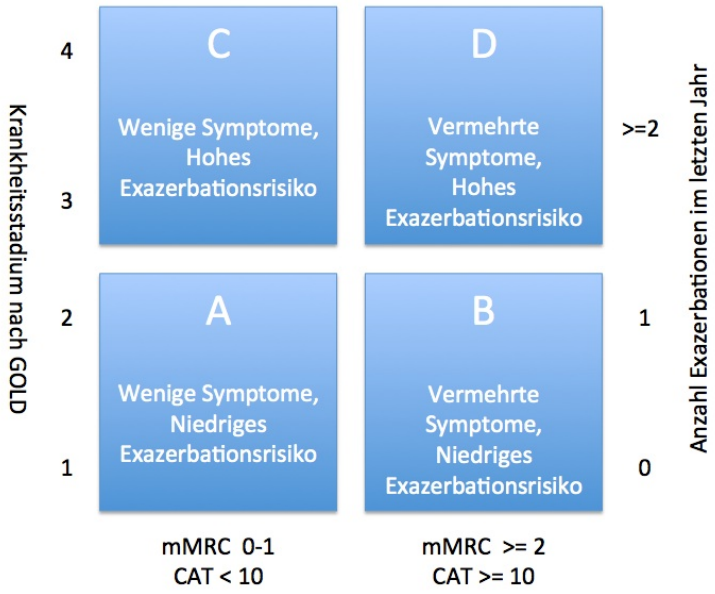

Abbildung 3.3: COPD Schweregrade: Berechnung und Bedeutung

beeinflusst wird. Bei der ersten Frage soll der Patient beispielsweise auf einer Skala von null bis fünf angeben, wie häufig er husten muss. Null bedeutet „ich huste nie“, während fünf bedeutet „ich huste ständig“. Die anderen sieben Fragen müssen ebenfalls auf einer Skala von null bis fünf beantwortet werden. Das Ergebnis des CAT ist die Summe der Ergebnisse der acht Fragen [cat].

Ein anderer Fragebogen für COPD Patienten ist der Clinical COPD Questionnaire (CCQ), welcher verwendet wird, um den klinischen Gesundheitszustand zu erfassen. Die zehn Fragen des Fragebogens haben eine feste Reihenfolge und müssen auf einer Skala von null bis sechs beantwortet werden. Sie lassen sich in drei Kategorien (Symptome, mentaler Zustand und funktionaler Zustand) einteilen. Symptome, die mit dem CCQ beobachtet werden können sind Atemnot, Husten und Auswurf. Mit den Fragen zum mentalen Zustand sollen Informationen zum Gemütszustand des Patienten gewonnen werden. Die Fragen zum funktionalen Zustand sollen die COPD-bedingten Einschränkungen beschreiben, mit denen der Patient im täglichen Leben umgehen muss. Das Ergebnis des CCQ besteht aus vier Teilen: einem Gesamtergebnis und je einem Ergebnis pro Kategorie [SJM+12]. Die Teilergebnisse werden wie folgt berechnet:

$$SymtomScore = (q1 + q2 + q5 + q6)/4$$
$$MentalScore = (q3 + q4)/2$$
$$SymtomScore = (q7 + q8 + q9 + q10)/4$$
$$TotalScore = (q1 + q2 + q3 + q4 + q5 + q6 + q7 + q8 + q9 + q10)/10$$

Der Charlson Index gehört ebenfalls zu den Skalen, die auf COPD-Patienten angewendet werden. Er beschreibt die 10-Jahres-Überlebenschance des Patienten, falls der Patient an einer anderen schweren

Krankheit außer COPD leidet. Im Charlson Index wird 22 Krankheiten eine Punktzahl zugeordnet, und die Summe der Punkte ist der Charlson Index des Patienten. Im folgenden ist aufgelistet, welche Krankheit zu welcher Punktzahl gehört:

1 Punkt: Herzinfarkt, Herzinsuffizienz, periphere arterielle Verschlusskrankheit, cerebrovaskuläre Erkrankungen, Demenz, Chronische Lungenerkrankung, Kollagenose, Ulkuskrankheit, Leichte Lebererkrankung, Diabetes mellitus (ohne Endorganschäden)

2 Punkte: Hemiplegie, Mäßig schwere und schwere Nierenerkrankung, Diabetes mellitus mit Endorganschäden, Tumorerkrankung, Leukämie, Lymphom

3 Punkte: Mäßig schwere und schwere Lebererkrankung

6 Punkte: Metastasierter solider Tumor, AIDS

In den täglichen Berichten sollen die Ergebnisse der täglichen Dateneingabe durch den Patienten gespeichert werden.

3.2.2 Sicherheit

Um die Sicherheit der Daten schon auf Datenbankebene zu schützen, sollen die folgenden Maßnahmen ergriffen werden. Es soll für jeden Benutzer des ECHO-Backends ein Benutzer in der Datenbank angelegt werden, der über das Datenbankrechtesystem nur die Rechte auf den Tabellen bekommt, die mit seiner Rolle verknüpft sind. Da MySQL keine rollenbasierte Zugangskontrolle unterstützt, müssen die Rechte einzeln vergeben werden. Für die Benutzer sollen Views erstellt werden, die nur die Daten enthalten, die ihn betreffen. Auf die unterliegenden Basistabellen soll der Benutzer dann keinen Zugriff mehr haben. Da Prepared Statements nicht im verwendeten MySQL Modul für Node.js verfügbar sind, sollen auch Maßnahmen gegen mögliche SQL Injections auf Datenbankebene ergriffen werden. Deswegen sollen alle Möglichkeiten, Daten zu ändern in Stored Procedures, die prepared Statements verwenden, gekapselt werden.

3.3 RESTful API

Die RESTful HTTP-API des Backends stellt die Schnittstelle zwischen Anwendungsebene und Infrastrukturebene dar und bietet Methoden zum Zugriff auf die Gesundheitsdaten. Da die API sämtliche Datenbankzugriffe kapselt und die einzige Möglichkeit darstellt, auf die Daten zuzugreifen, wird durch die API auch die Sicherheit der Gesundheitsdaten erhöht. Durch die uniforme und portable Schnittstelle können alle benötigten Arten von Klienten leicht mit dem Backend interagieren. Die API kann neben der Kommunikation mit den Klienten auch zur Orchestrierung verwendet werden, indem man die in diesem Kapitel beschriebenen, einfachen Dienste zu komplexeren Diensten kombiniert. Ein weiterer Vorteil einer RESTful HTTP-API ist die statuslose Kommunikation und die damit verbundene Skalierbarkeit.

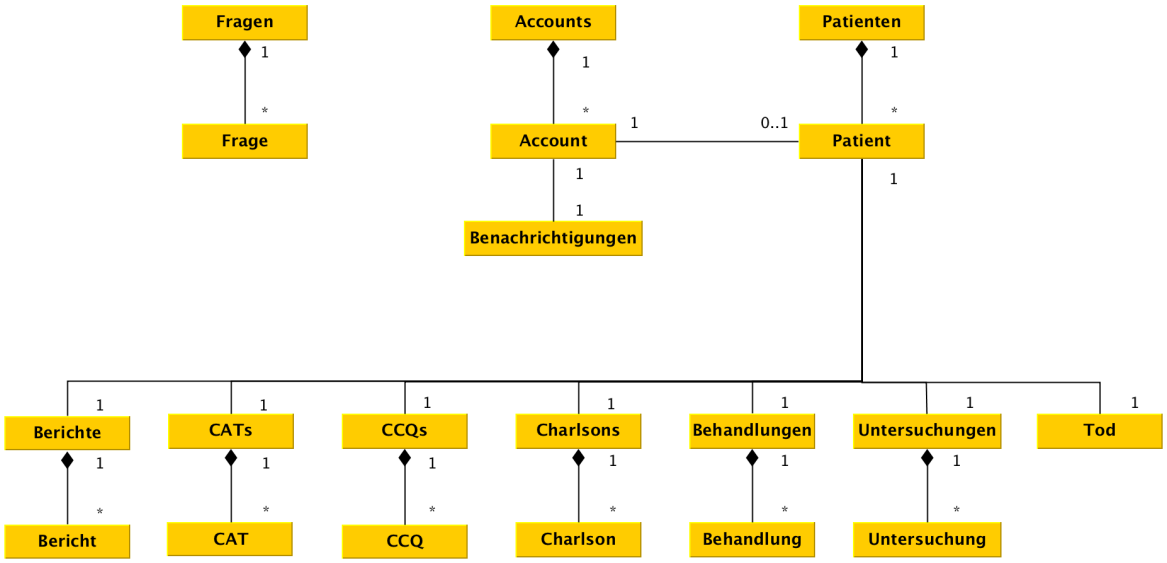

Abbildung 3.4: Ressourcen als UML-Diagramm

3.3.1 Ressourcendesign

Beim Ressourcendesign stellte sich heraus, dass die Ressource für die Patientendaten großen Einfluss auf die restlichen Ressourcen hat. Es gab die folgenden Möglichkeiten:

1. Eine „große Patientenressource“, die alle persönlichen Daten des Patienten enthält und die Ergebnisse ärztlicher Untersuchungen in Arrays mitführt.
2. Eine „kleine Patientenressource“, die alle persönlichen Daten des Patienten enthält, während alle ärztlichen Untersuchungen als eigenständige Ressourcen dargestellt werden.
3. Eine „kleine Patientenressource mit Subressourcen“, wobei die ärztlichen Untersuchungen als Subressourcen modelliert werden. Subressourcen stellen Daten in Abhängigkeit einer anderen Ressource zur Verfügung. Beispielsweise würde die Ressource /patients/1/examA nur Daten beinhalten, die mit der Ressource /patients/1 in Verbindung stehen.

Die „große Patientenressource“ hat den Nachteil, dass sie, durch den REST Architekturstil bedingt, jedes Mal komplett an den Server übertragen werden muss, wenn beispielsweise ein neuer CAT (COPD Assessment Test) ausgefüllt wurde. Das bedeutet auch, dass sie mit der Zeit immer größer wird.

Jede Ressource, die abgerufen werden kann, soll aber auch ein Dokument darstellen, das für den Anwender einen Nutzen hat. Im Fall der „kleinen Patientenressource“, hätte aber beispielsweise eine Auflistung aller CAT-Ergebnisse der Patienten keinerlei Nutzen für den Arzt.

Subressourcen sind zwar nicht bei allen Anhängern des REST Architekturstils als RESTful anerkannt, aber bei der Variante „kleine Patientenressource mit Subressourcen“ fallen beide Nachteile der anderen Varianten weg. Es ist möglich, neue ärztliche Untersuchungsdaten hinzuzufügen, ohne die komplette

Patientenressource übertragen zu müssen. Außerdem würde ein Aufruf der CAT-Subressource nur die Ergebnisse eines Patienten auflisten, anhand derer der Arzt beispielsweise den Krankheitsverlauf nachvollziehen könnte, anstatt aller.

In Abbildung 3.4 werden alle identifizierten Ressourcen und ihre Beziehungen untereinander in einem UML Klassendiagramm dargestellt. Die Ressource *Patienten* stellt die Listenressource für die Ressource *Patient* dar. Wenn die Listenressource abgerufen wird, stellt sie eine Liste aller verfügbaren Ressourcen des Typs *Patient* zur Verfügung. Eine spezifische Ressource *Patient* enthält den Namen, Kontaktdaten, demografische Angaben und Angaben zum zugehörigen Account.

Die Ressourcen *Berichte*, *CATs*, *CCQs*, *Charlsons*, *Behandlungen*, *Untersuchungen* und *Tod* sind Subressourcen der Ressource *Patient* und stellen, bis auf die Ressource *Tod*, Listenressourcen dar. Die Ressource *Tod* enthält Angaben zu den Todesumständen, wenn der zugehörige Patient bereits verstorben ist und ist nur einmal pro Patient vorhanden, da der Patient nur einmal sterben kann. Eine spezifische Ressource *Bericht* stellt das Ergebnis einer Dateneingabe an einem Tag durch den Patienten dar. Eine einzelne Ressource *CAT* ist die Repräsentation eines durchgeführten COPD Assessment Tests, *CCQ* die eines beantworteten Clinical COPD Questionnaires und *Charlson* die eines ausgefüllten Charlson Tests. Eine bestimmte Ressource *Untersuchung* stellt eine Untersuchung durch den Arzt dar, während eine bestimmte Ressource *Behandlung* eine Auflistung der verschriebenen Medikamente, die der Patient zu einem Zeitpunkt bekommt, darstellt.

Zu jeder einzelnen Ressource *Patient* gehört eine bestimmte Ressource *Account*, die die Zugangsdaten und die Einstellungen bezüglich der Benachrichtigungen enthält. Die Ressourcen des Typs *Account*, zu denen keine Ressource des Typs *Patient* gehört, sind Accounts der Rollen *doctor* und *admin*. *Accounts* ist die Listenressource zu den einzelnen Ressourcen *Account*. Zu jeder Ressource *Account* gehören Benachrichtigungen, die an diesen einen Account gingen. Diese Benachrichtigungen können über die Ressource *Benachrichtigungen* abgerufen werden.

Im Rahmen dieser Arbeit wurde anfangs ein Prototyp implementiert, der die Funktionsweise der API demonstrieren sollte. Für diesen Prototyp wurde eine Ressource angelegt, mit deren Hilfe die Fragen zu den ärztlichen Untersuchungen verwaltet werden konnten. Außerdem konnte der Prototyp mit Hilfe dieser Ressource Dialoge generieren, mit denen man die Fragen beantworten konnte. Da diese Funktion immer noch nützlich sein kann für das Browser-Frontend, ist in Abbildung 3.4 auch die Listenressource *Fragen* mit der zugehörigen Ressource *Frage* dargestellt.

Im folgenden werden alle Funktionen erklärt, die auf den Ressourcen benötigt werden. Dabei wird auch jeweils erwähnt, welche Rolle die Funktion aufrufen darf. Diese Rollen sind dieselben wie die der Accounts in den Gesundheitsdaten.

3.3.2 Funktionen der Ressource Account

Für die Listenressource *Accounts* werden folgende Funktionen benötigt:

1. eine Funktion die per HTTP GET eine Liste alle Ressourcen des Typs *Account* zurückgibt. Dabei muss Paginierung sowie das Filtern nach den Rollen *admin*, *doctor* und *patient* unterstützt werden.

2. mittels HTTP POST soll es möglich sein, eine neue Ressource des Typs *Account* zu erstellen.

Auf einer Ressource *Account* soll möglich sein:

1. mittels eines HTTP GET Requests soll eine bestimmter Ressource zurückgegeben werden.
2. per HTTP PUT Request soll eine bestimmte Ressource aktualisierbar sein.
3. über einen HTTP DELETE Request soll eine bestimmter Ressource deaktivierbar sein.

Alle Funktionen auf der Listenressource setzen vorraus, dass der Aufrufende zur Benutzergruppe *admin* gehört. Die restlichen Funktionen sollen ebenfalls nur von Benutzern der Gruppe *admin* aufrufbar sein, es sei denn die Ressource repräsentiert den eigenen Account des Aufrufenden.

3.3.3 Funktionen der Ressource Patient

Für die Listenressource *Patienten* werden folgende Funktionen benötigt:

1. eine Funktion, die per HTTP GET alle Ressourcen des Typs *Patient* zurückgibt. Dabei muss Paginierung sowie das Sortieren nach E-Mail Adresse oder Aktenzeichen unterstützt werden.
2. mittels HTTP POST soll es möglich sein, eine neue Ressource des Typs *Patient* zu erstellen.

Auf einer Ressource *Patient* soll möglich sein:

1. mittels eines HTTP GET Requests soll eine bestimmter Ressource zurückgegeben werden.
2. per HTTP PUT Request soll eine bestimmte Ressource aktualisierbar sein.
3. über einen HTTP DELETE Request soll eine bestimmte Ressource gelöscht werden können.

Alle hier aufgeführten Funktionen setzen voraus, dass der Aufrufende zur Benutzergruppe *doctor* gehört.

Funktionen der Subressourcen Berichte, CAT, CCQ, Charlson, Behandlung, Untersuchung und Tod

Für jede Listenressource der Subressourcen *Berichte, CATs, CCQs, Charlsons, Behandlung* und *Untersuchungen* sollen folgende Funktionen bereit gestellt werden:

1. eine Funktion die per HTTP GET alle Ressourcen der Subressource zurückgibt, die zu dem Patienten gehören
2. mittels HTTP POST soll es möglich sein, eine neue Ressource zu erstellen.

Auf den Ressource *Bericht, CAT, CCQ, Charlson, Behandlung* und *Untersuchungen* soll möglich sein:

1. mittels eines HTTP GET Requests soll eine bestimmte Ressource zurückgegeben werden.
2. per HTTP PUT Request soll eine bestimmte Ressource aktualisierbar sein.
3. über einen HTTP DELETE Request soll eine bestimmte Ressource gelöscht werden können.

Für die Ressource Tod sollen folgende Funktionen implementiert werden:

1. per HTTP POST Request soll die Ressource angelegt werden können.
2. mittels eines HTTP GET Requests soll die Ressource zurückgegeben werden, falls sie zuvor angelegt wurde.
3. per HTTP PUT Request soll die Ressource aktualisierbar sein.
4. über einen HTTP DELETE Request soll der Inhalt der Ressource gelöscht werden können.

Alle hier erwähnten Funktionen sollen nur verwendbar sein, wenn der Benutzer als Benutzer mit der Rolle *doctor* authentifiziert wurde. Für die Ressource *Berichte* gilt jedoch, dass sie auch von Benutzern der Rolle *patient* verwendet werden können soll, da sonst die tägliche Dateneingabe durch den Patienten nicht möglich wäre.

3.3.4 Funktionen der Ressource Benachrichtigungen

Auf der Ressource Benachrichtigungen wird eine Funktion benötigt, die nach einem HTTP GET Request alle Benachrichtigungen zur Verfügung stellt, die das Backend für den aktuell eingeloggten Benutzer generiert hat. Da die Analysen aktuell noch nicht implementiert sind, wird zusätzlich eine Funktion implementiert, die per HTTP POST eine neue Benachrichtigung erstellt. Die Funktion kann später auch dazu benutzt werden, reguläre Benachrichtigungen zu erstellen, falls Benachrichtigungstypen eingeführt werden, die nicht durch Analysen generiert werden sollen.

3.3.5 Funktionen der Ressource Fragen

Auf der Listenressource Fragen werden folgende Funktionen benötigt:

1. eine über HTTP GET ansprechbare Funktion um alle aktiven Fragen einer Kategorie abzurufen.
2. eine Funktion über die sich per HTTP POST eine neue Frage mit Antworten anlegen lässt.

Auf einer bestimmten Ressource *Frage* sollen folgende Operationen möglich sein:

1. über HTTP GET soll die einzelne Ressource *Frage* mit ihren Antworten zurückgegeben werden können.
2. über HTTP PUT soll die Ressource inklusive der zugehörigen Antworten änderbar sein, insbesondere soll der Status von aktiv auf inaktiv geändert werden können.
3. per HTTP DELETE soll die Ressource inklusiv ihrer Antworten gelöscht werden können.

Alle Funktionen der Ressourcen des Typs Frage und die der POST Funktion der Listenressource sollen nur von Benutzern des Typs admin verwendet werden können. Für die GET Funktion der Listenressource genügt es, ein authentifizierter Benutzer zu sein.

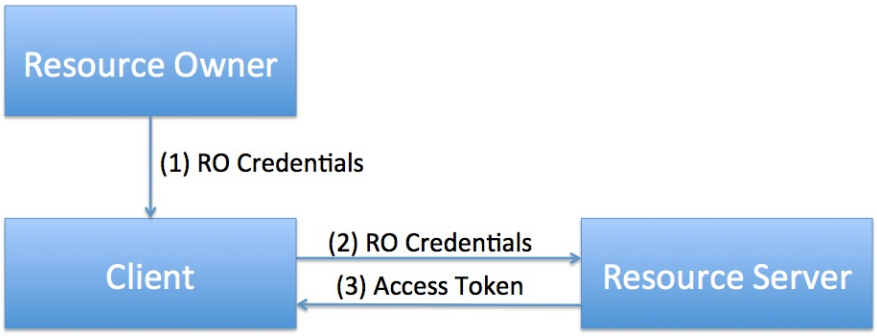

Abbildung 3.5: Resource Owner Password Credentials Flow

3.3.6 Command Ressourcen

Um mehrere Ressourcen auf einmal zu ändern oder nur teilweise zu ändern, werden Command Ressourcen eingeführt. Die folgenden Funktionen sind zusätzlich notwendig:

- Um einen neuen Patienten inklusive Account anzulegen, soll es Ärzten möglich sein mit einem POST Aufruf eine Account Ressource und eine Patienten Ressource anzulegen.
- Außerdem soll es einem Benutzer mit der Rolle *admin* ermöglicht werden, den Arzt eines Patienten zu ändern.

3.3.7 Sicherheit

Zur Authentifizierung soll OAuth 2.0 mit dem Profil *Resource Owner Password Credentials* verwendet werden. Der Ablauf ist in Abbildung 3.5 dargestellt. Der Resource Owner, der Besitzer der Daten, gibt seine Zugangsdaten auf dem Client ein und dieser sendet sie an den Resource Server. Der wiederum stellt ein Access Token aus, mit dem man die API verwenden kann. Das Token kann verwendet werden um Informationen über den Benutzer zu transportieren, sodass beispielsweise nicht bei jedem Request an die API mit Hilfe der Datenbank überprüft werden muss, ob der Benutzer berechtigt ist, die Operation auszuführen. Im Token kann auch eine Ablaufzeit gespeichert werden, nach der das Token mit Hilfe eines Refresh Tokens erneuert werden muss. Ein Refresh Token wird gleichzeitig mit dem Access Token ausgestellt und ist nur einmal einsetzbar. Mit einer begrenzten Gültigkeit eines Access Tokens lässt sich das Risiko eines Tokens, das von einen Angreifer entwendet wurde, verkleinern. Das Token und das Refresh Token müssen allerdings mit derselben Sorgfalt behandelt werden wie ein normales Passwort. Um die Übertragung der Daten zu schützen, und weil OAuth 2.0 es voraussetzt, soll SSL verwendet werden.

3.4 Analysen

In diesem Abschnitt werden einfache Analysen vorgestellt, die durch Gespräche mit Ärzten entstanden sind und im Backend implementiert werden können. Es gibt zwei Arten von Analysen: die eine wertet die täglichen Patienteneingaben anhand von Wenn-Dann-Regeln aus und die andere bestimmt die Schwere der Erkrankung anhand von Untersuchungsergebnissen und der Anzahl der Exazerbationen im letzten Jahr.

3.4.1 Tägliche Berichte

Die tägliche Dateneingabe durch Patienten wird anhand von Wenn-Dann-Regeln analysiert, um frühzeitig Exazerbationen zu erkennen und zu verhindern. Die tägliche Dateneingabe besteht aus den folgenden Fragen:

1. Hat sich Ihre Kurzatmigkeit verstärkt?
    a) Können Sie Ihre tägliche Arbeit verrichten?
    b) Können Sie sich selbst versorgen?
    c) Können Sie laufen?
2. Müssen Sie mehr husten?
3. Hat sich Ihr Auswurf verändert?
    a) Ist Ihr Auswurf gelb?
    b) Ist Ihr Auswurf grün?
    c) Ist Ihr Auswurf blutig?
4. Haben Sie Brustschmerzen?
5. Haben Sie die Dosierung der Medikamente erhöht?

Die Fragen 1a, 1b und 1c sollen nur beantwortet werden, wenn Frage 1 mit ja beantwortet wurde. Entsprechendes gilt für Frage 3. Außerdem kann der Patient, wenn er über die entsprechenden Geräte verfügt, die folgenden Messwerte eintragen:

- Sauerstoffsättigung
- Puls
- Köpertemperatur
- maximale Ausatmungsgeschwindigkeit

Anhand der folgenden Regeln können die Daten analysiert werden:

- An zwei Tagen in Folge Frage 1 mit ja beantwortet -> Arzt anrufen.
- An einem Tag Fragen 1, 2 und 3 mit ja beantwortet -> Arzt anrufen.

Stadium	FEV1 (Sollwert = 100%)	FEV1/FVC
1	>= 80%	<70%
2	50-80%	<70%
3	30-50%	<70%
4	< 30%	<70%

Tabelle 3.1: Berechnung des COPD Krankheitsstatdiums nach GOLD nach [copb]

- An einem Tag Frage 3a oder 3b mit ja beantwortet -> Arzt anrufen.
- An einem Tag Frage 3c mit ja beantwortet -> Ins Krankenhaus gehen.
- An zwei Tagen in Folge Frage 5 mit ja beantwortet -> Arzt anrufen.
- Fragen x Tage nicht beantwortet -> Erinnerung.

Wenn eine der Regeln zutrifft, soll der Patient und der Arzt des Patienten benachrichtigt werden. Benachrichtigungen werden, je nach gewähltem Modus, per SMS, E-Mail oder Push-Benachrichtigung verschickt und in der Datenbank hinterlegt.

3.4.2 Schweregrad der Krankheit

Die zweite Analyse, die anhand der vorhandenen Daten implementiert werden kann, ist die Berechnung des Schweregrads nach Kriterien der Global Initiative for Chronic Obstructive Lung Disease (GOLD) (siehe Abbildung 3.3). Der Schweregrad der COPD richtet sich nach

- der Lungenfunktion (ausgedrückt durch das Krankheitsstadium nach GOLD, siehe 3.1)
- der Anzahl der Exazerbationen im letzten Jahr
- dem Ergebnis des COPD Assessment Tests, wenn kein Ergebnis vorliegt, kann auch das Ergebnis des Modified British Medical Research Council (mMRC) Fragebogens verwendet werden.

Um das Krankheitsstadium nach GOLD zu berechnen (siehe Tabelle 3.1) benötigt man die folgenden zwei Werte:

- die größtmögliche Menge an Luft, die Sie innerhalb von einer Sekunde ausatmen können (FEV1, Forced Expiratory Volume in 1 second)
- die Luftmenge, die Sie nach tiefem Einatmen mit maximaler Geschwindigkeit insgesamt wieder ausatmen können (FVC, forced vital capacity)

Mit dem Stadium, der Anzahl der Exazerbationen im letzten Jahr und dem Ergebnis des CAT (ersatzweise der mMRC-Wert) lässt sich die Schwere anhand der Tabelle 3.2 berechnen. Dabei gilt allerdings, dass zuerst das Stadium und die Anzahl der Exazerbationen betrachtet werden und auf den höheren Schweregrad der CAT-Wert (oder der mMRC-Wert) angewendet wird.

Schweregrad	Stadium	Exazerbationen	CAT oder mMRC
A	1 oder 2	weniger als zwei im letzten Jahr	CAT < 10 oder mMRC 0-1
B	1 oder 2	weniger als zwei im letzten Jahr	CAT >= 10 oder mMRC >= 2
C	3 oder 4	mind. zwei im letzten Jahr	CAT < 10 oder mMRC 0-1
D	3 oder 4	mind. zwei im letzten Jahr	CAT >= 10 oder mMRC >=2

Tabelle 3.2: Berechnung des COPD Schweregrads nach [copb]

4 Implementierung

Bisher wurden die Patienten in einer Microsoft Access Datenbank verwaltet. Diese Datenbank war aber nur von einem Arzt und nur lokal verwendbar. Da das ECHO-Backend von vielen Benutzern gleichzeitig benutzbar sein soll, ergeben sich folgende Anforderungen an die Implementierung: Mehrbenutzerfähigkeit, Skalierbarkeit, Remote Access und Sicherheit der Patientendaten. Darum wurde ein Relationales Datenbanksystem und eine geeignete skalierbare und performante Programmiersprache für Netzwerkanwendungen verwendet.

4.1 Gesundheitsdaten

Als Datenbanksystem wurde MySQL vorgegeben. MySQL ist ein Open-Source-Datenbank und kann dadurch auch als insofern sicher angesehen werden, als dass es keine eingebauten Hintertüren hat. Außerdem existiert für MySQL ein vordefinierter TOSCA Nodetype. Mit Hilfe von TOSCA könnte eine MySQL Datenbank beispielsweise in einer sicheren und gesetzeskonformen Cloud deployed werden.

4.1.1 MySQL Workbench

Die MySQL Workbench ist ein visuelles Werkzeug für Datenbankarchitekten und Entwickler. Mit Hilfe der Workbench wurden das Schema für die Gesundheitsdaten umgesetzt und die Stored Procedures implementiert.

Durch Nutzung der Workbench während des Modellierens müssen keine langen SQL Skripte geschrieben werden. Anstatt dessen kann in einer GUI einfach eine neue Tabelle angelegt und mit Spalten versehen werden. Pro Spalte kann man den Datentyp festlegen und ob die Spalte Teils des Primärschlüssels ist, NULL Werte erlaubt, nur eindeutige Werte erlaubt oder was der Standardwert sein soll. Pro Tabelle lassen sich Indizes, Fremdschlüssel und Trigger definieren. Bei der Definition von Triggern wird man unterstützt, indem man nur den Auslösezeitpunkt auswählen muss, und die Workbench stellt dann bereits das Codegrundgerüst zur Verfügung. Abbildung 4.1 zeigt die Bearbeitung der Accountstabelle des ECHO Schemas.

Wurde die Datenbank modelliert kann, man auch ER-Diagramme erstellen lassen. Die ER-Diagramme können per Drag und Drop verändert werden und die Datenbank auch nur teilweise darstellen. Außerdem kann der Editor zum Modellieren und Ändern des Schemas verwendet werden.

Über die MySQL Workbench kann man sich zu einem beliebigen MySQL Server verbinden und dort das erstellte Schema anlegen lassen. Es ist auch möglich, nur Teile des Schemas anlegen zu lassen

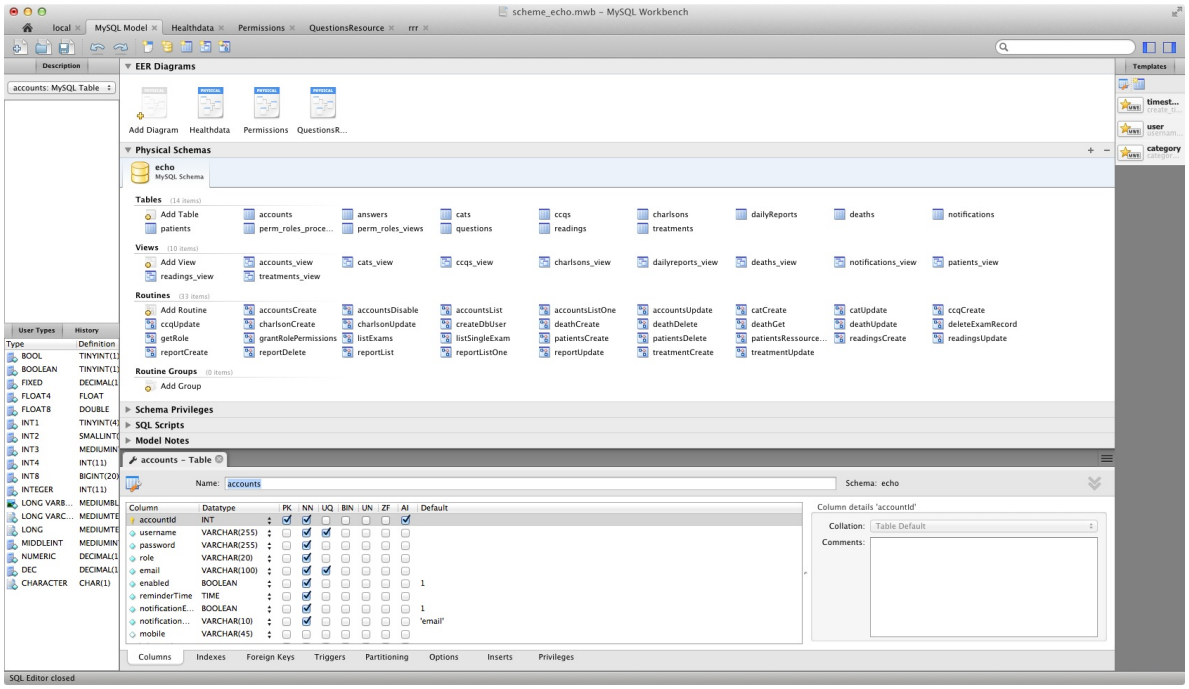

Abbildung 4.1: Screenshot MySQL Workbench

oder vor dem Anlegen das generierte SQL Skript anzupassen. Sollte ein der Workbench erstelltes Schema auf einem Server erstellt und dann nachträglich dort geändert worden sein, gibt es auch die Möglichkeit, diese Änderungen von der Workbench in das lokale Schema übernehmen zu lassen. Dabei kann man auch auswählen ob man alle Änderungen übernehmen will.

4.1.2 Access Control in MySQL

Die Rechtevergabe in MySQL erfolgt über GRANT und REVOKE Statements. Mit einem Statement lassen sich Rechte an einen bestimmten Benutzer vergeben. Der Benutzer wird dabei in der Form user@host angegeben. Dadurch können einem Benutzer unterschiedliche Rechte zugewiesen werden, je nach dem von welchem Host er sich mit der Datenbank verbindet.

Bei Stored Procedures und Views kann über das DEFINER Attribut der Ersteller des Objekts angegeben werden. Falls keiner angegeben wird, wird der Benutzer, der das Objekt erstellt als DEFINER eingetragen. Zusätzlich kann über das SQL SECURITY Attribut festgelegt werden, in welchem Kontext die Routine oder der View ausgeführt werden soll. Die möglichen Werte hierbei sind: DEFINER und DEFINER. Falls DEFINER angegeben wird, wird die Routine oder der View im Kontext des Erstellers ausgeführt. Der ausführende Benutzer muss dabei lediglich das Recht haben, die Routine auszuführen oder den View auszulesen. Während des Ausführens der Routine oder des Views werden aber die Rechte des Erstellers verwendet. Je mehr Rechte der Ersteller hat, desto mächtigere Operationen können von dem Objekt verwendet werden, unabhängig davon welche Rechte der Aufrufende hat. Wenn

Typ	Bedeutung
0	Sie müssen ihren täglichen Bericht ausfüllen!
1	Rufen Sie Ihren Arzt an!
2	Gehen Sie ins Krankenhaus!
3	Ihr Patient %name% soll Sie anrufen!
4	Ihr Patient %name% sollte ins Krankenhaus!
5	Ihr Patient %name% hat seit 2 Tagen keinen Bericht ausgefüllt!
6	Ihr Patient %name% hat seit 10 Tagen keinen Bericht ausgefüllt!

Tabelle 4.1: Benachrichtigungsarten

SQL SECURITY INVOKER angegeben wurde, können damit nur Operationen ausgeführt werden für die der Aufrufende Rechte besitzt.

4.1.3 Datenmodell

In Abbildung 4.2 wird das vollständige Datenmodell der Gesundheitsdaten dargestellt. In der Tabelle **accounts** hat jeder Benutzer des Systems einen Eintrag. Dort werden E-Mail Adresse und Passwort, die als Zugangsdaten zum System dienen, gespeichert. Falls das Flag enabled auf false steht, bedeutet dies, dass der Account inaktiv ist und der Benutzer sich nicht anmelden kann. Außerdem wird die Rolle des Benutzers (entweder *admin, doctor* oder *patient*) festgelegt. Zusätzlich zu diesen Daten werden die Einstellungen zu den Benachrichtigungen, die das System versenden kann, erfasst. Es wird gespeichert ob und wie Benachrichtigungen verschickt werden sollen. Mögliche Benachrichtigungsmodi sind SMS, Push-Notification oder E-Mail, dargestellt durch die Werte sms, push und email. Um die Benachrichtigung per SMS zu unterstützen muss auch noch die Mobiltelefonnummer gespeichert werden. Via Trigger wird vor Einfügen und Aktualiseren der Daten geprüft, ob die Rolle und der Benachrichtigungsmodus einen gültigen Wert haben.

Die Benachrichtigungen, die jeder Benutzer des Systems erhalten kann, werden in der Tabelle **notifications** gespeichert. Neben dem Zeitpunkt der Erstellung wird bei jeder Benachrichtigung der Typ der Benachrichtigung gespeichert (siehe Tabelle 4.1). Die Benachrichtigungen der Typen 0, 1 und 2 sind für Patienten vorgesehen, der Rest für Ärzte. Der Platzhalter %name% kann aus der patientId abgeleitet werden.

In der Tabelle **patients** sind Details zu jedem Patient hinterlegt. Der dazugehörige Account wird in accountId hinterlegt und der Account des Arztes in doctorId. Bei diesen Feldern wird auch überprüft, ob die Rolle des angegebenen Accounts patient bzw. doctor ist. Zu jedem Patient gehört eine eindeutige Versicherungsnummer und eine Akten-ID. Außerdem werden allgemeine Daten über den Patient gespeichert, wie sein voller Name, Geschlecht, Geburtsdatum und Adresse. Der zweite Name des Patienten sowie seine Festnetznummer sind optional.

Die Tabelle **cats** speichert alle COPD Assessment Tests und die Tabelle **cqqs** alle COPD Clinical Quetionnaires. Die Ergebnisse der Fragebögen werden aus den einzelnen Antworten von einem

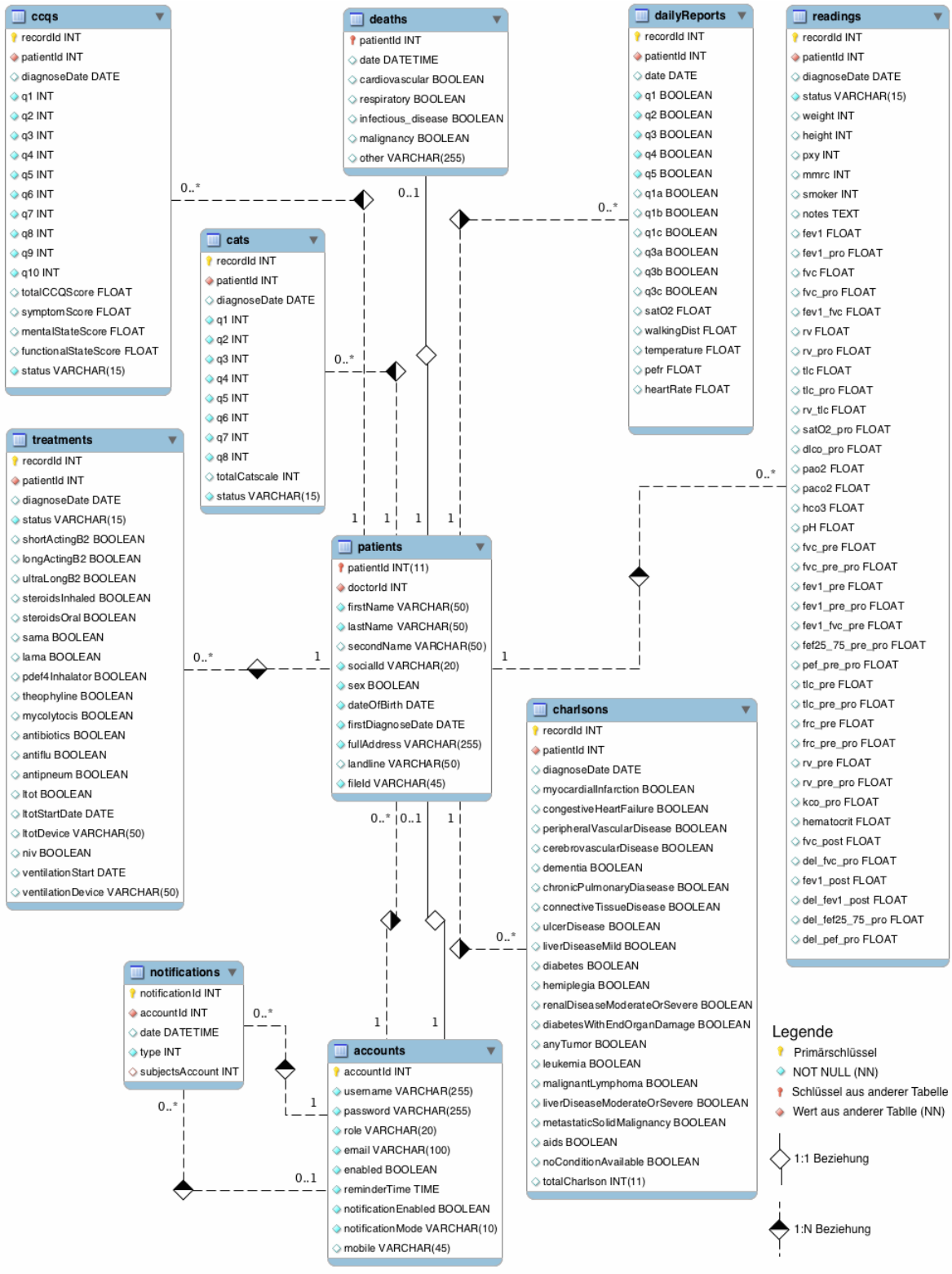

Abbildung 4.2: ER-Diagramm der Gesundheitsdaten

Trigger berechnet. Der Trigger prüft zusätzlich noch ob der Status mit baseline oder exacerbation angegeben wurde und setzt das diagnoseDate auf das aktuelle Datum, falls keines angegeben wurde.

Mit welchen Mitteln durchgeführt wird, wird in **treatments** gespeichert. Ein Trigger prüft die Werte für status (baseline oder exacerbation), ltotDevice (cpap oder bipap) und ventilationDevice (concetrator, cylinder, liquid) und setzt das Datum auf das aktuelle Datum falls kein Datum angegeben wird.

In der Tabelle **charlsons** werden einzelne Charlson Tests gespeichert. Das Ergebnis (totalCharlson) wird durch einen Trigger berechnet, der ebenfalls überprüft ob mindestens eine Bedingung zutrifft (entweder eine Krankheit oder noConditionAvailable) und das Datum auf das aktuelle Datum setzt, falls keines angegeben wurde.

Die Tabelle **readings** erfasst alle Werte, die bei einer Untersuchung erfasst werden können. Ein Trigger prüft den Wert für status (baseline oder exacerbation) und setzt das Datum.

Die tägliche Dateneingabe des Patienten wird in **dailyReport** gespeichert. Neben den Antworten auf die ja/nein Fragen werden noch die Sauerstoffsättigung im Blut, die am Tag zurückgelegte Wegstrecke, die Körpertemperatur, die maximale Ausatmungsgeschwindigkeit und der Puls gespeichert.

Falls ein Patient verstirbt, wird seine Todesursache in **deaths** festgehalten. Falls die vorgesehenen Ursachen nicht zutreffen, kann im Feld other ein Freitext gespeichert werden. Diese Tabelle ist die einzige, bei der es pro Patient maximal einen Eintrag geben kann.

4.1.4 Implementierte Views

Um sicherzustellen, dass die Benutzer des Systems nur die Daten angezeigt bekommen, die sie sehen dürfen, wurden Views implementiert. Die Views werden in diesem Abschnitt vorgestellt.

Anstatt für jeden Benutzer eine Reihe von Views zu erstellen, kann pro Tabelle auch nur ein View erstellt werden, der die Rechte des aktuell angemeldeten Benutzers auswertet. Dazu werden drei Hilfsfunktionen benötigt: user(), die Bestandteil von MySQL ist, und die selbstgeschriebene Funktion getRole(). Beim Aufruf gibt user() den Benutzer mit dem man sich am Datenbanksystem angemeldet hat zurück. Diese Funktion ist nicht zu verwechseln mit current_user(), die den Benutzer zurückgibt, mit dem sich der Client authenfiziert hat. Ein Beispiel verdeutlicht den Unterschied: Sei userA der Benutzer, mit dem man am Datenbanksystem angemeldet ist und der EXECUTE Rechte auf der Stored Procedure f() hat. f() wurde von userB erstellt und verwendet die SQL SECURITY DEFINER Klausel. Wenn user() in f() aufgerufen wird, wird userA zurückgegeben, während ein Aufruf von current_user() userB zurückgibt.

Die Funktion getRole() (siehe Listing 4.1) gibt die ECHO-Rolle des aktuell angemeldeten Datenbankbenutzers zurück. Dies funktioniert, da dem Datenbankbenutzer die Account-ID aus der Account-Tabelle als Benutzername zugewiesen wurde (siehe Abschnitt Benutzerverwaltung).

In Listing 4.2 ist dargestellt, wie die Funktionen genutzt werden können um nur die Datensätze anzuzeigen, für die der aktuelle Benutzer berechtigt ist. Der View **accounts_view** stellt sicher, dass nur Benutzer mit der Rolle *admin* auf alle Accounts Zugriff haben und andernfalls nur der eigene Account angezeigt wird. Wenn getRole() die Rolle *admin* zurückgibt, werden keine Einschränkungen gemacht. Andernfalls zeigt der View nur den Datensatz an, der zum aktuell angemeldeten Datenbankbenutzer

Listing 4.1 Stored Function: getRole()

```
CREATE DEFINER=`echo_db_usr`@`localhost` FUNCTION `getRole`() RETURNS char(10)
BEGIN
      SELECT role into @ret from accounts where accountId = substring_index(user(), '@', 1);
      RETURN @ret;
END
```

Listing 4.2 Auf der Tabelle Accounts definierter View

```
CREATE VIEW `accounts_view` AS
   SELECT *
   FROM `accounts`
   WHERE
      (CASE
         WHEN (getRole() = 'admin') THEN (1 = 1)
         ELSE (`accounts`.`accountId` = substring_index(user(), '@', 1))
      END)
```

gehört. Der View **patients_view** ist ähnlich aufgebaut, nur dass nicht über accountId eingeschränkt wird, sondern über doctorId. Allerdings wird für diesen View die Patiententabelle mit der Accounttabelle gejoint, um die zusätzlichen Informationen (Mobiltelefonnummer und E-Mail-Adresse) für die Ressource Patient bereitzustellen.

Neben den bereits vorgestellten Views wurden noch die folgenden erstellt: **cats_view**, **ccq_view**, **charlson_view**, **deaths_view**, **readings_view**, **dailyreports_view** und **treatments_view**. Diese Views sollen nur Benutzern der Rolle *admin* oder *doctor* zur Verfügung stehen. Sie sind alle gleich aufgebaut und sollen nur die Datensätze anzeigen, die zu Patienten des aktuellen Benutzers gehören oder alle falls der Benutzer die Rolle *admin* hat. In Listing 4.3 ist einer davon beispielhaft dargestellt. In diesem View wird in der WHERE Klausel ebenfalls die Rolle überprüft. Wenn die Rolle des Benutzers admin ist, wird nicht eingeschränkt. Andernfalls werden nur die patientIds angezeigt, die dem aktuellen Benutzer zugeordnet sind.

Der View **notifications_view** zeigt die Berechtigungen des aktuellen Benutzers an. Zusätzlich wird aus dem Typ der Benachrichtigung (siehe Tabelle 4.1) und der ID des Benutzers, über den berichtet wird, ein String gebildet, der die Benachrichtigung lesbar macht.

Listing 4.3 Auf der Tabelle CATs definierter View

```
CREATE VIEW `cats_view` AS
   SELECT *
   FROM `cats`
   WHERE
      (CASE
         WHEN (getRole() = 'admin') THEN (1 = 1)
         ELSE `cats`.`patientId` in (
                  SELECT `patients`.`patientId`
            FROM `patients`
            WHERE (`patients`.`doctorId` = substring_index(user(), '@', 1)))
      END)
```

4.1.5 Implementierte Stored Procedures

In diesem Abschnitt werden die Stored Procedures beschrieben, die im Rahmen dieser Arbeit geschrieben wurden. Die Stored Procedures dienen primär dazu den schreibenden Zugriff für den REST-Service zu kapseln. Alle im folgenden vorgestellten Stored Procedures haben dafür den Benutzer echo_db_adm als Definer zugewiesen bekommen und werden mit dessen Rechten ausgeführt. Das bedeutet, dass der Benutzer echo_db_adm auf allen benutzten Views und Tabellen Lese- und Schreibrechte benötigt, aber dadurch allen anderen Datenbankbenutzern für den Zugriff auf die Daten, die Rechte zum Ausführen der Stored Procedures genügen.

Die Methode **accountsCreate()** dient dazu einen neuen Account anzulegen. Falls ein Benutzer der Rolle *doctor* einen Account anlegen will, wird überprüft ob der neue Account die Rolle patient hat. Wenn nicht wird ein Fehler ausgegeben, da Benutzer der Rolle *doctor* nur Accounts der Rolle *patient* anlegen dürfen und keine der Rollen *admin* oder *doctor*. Anschließend wird der Account über ein prepared Insert Statement erstellt. Außerdem wird ein Datenbankbenutzer angelegt, wobei die ID des angelegten Backendnutzers als Benutzername verwendet wird. Als Passwort wird eine Konkatenation aus der ID des neues Benutzers und des Parameters pwdPrefix verwendet. Dieses Präfix sollte für alle Benutzer gleich sein. Durch ihn soll es zum einen für Nutzer, die Zugriff auf das System haben, schwerer werden das Passwort des Benutzers zu erraten. Zum anderen kann dadurch das Passwort in der REST API „errechnet“ werden, wenn es nötig ist, sich mit einem anderen Benutzer zur Datenbank zu verbinden. Um dem neu angelegten Datenbankbenutzer alle erforderlichen Rechte zuzuteilen, muss zusätzlich die Funktion grantRolePermissions() aufgerufen werden (siehe Abschnitt Benutzerverwaltung).

Mit der Methode **accountsUpdate()** lässt sich ein Account bearbeiten. Die Rolle des Patienten und das enabled-Flag lassen sich durch diese Methode nicht ändern. Um zu überprüfen, ob der Benutzer den angegebenen Account ändern darf, wird überprüft, ob der zu ändernde Account in accounts_view enthalten ist. Falls er das nicht ist, wird mit einem Fehler abgebrochen. Wenn kein Fehler geworfen wird, wird der angegebene Account aktualisiert. Dabei wird noch unterschieden, ob das Passwort als leerer String übergeben wurde. Falls ein leerer String übergeben wurde, wird das Passwort nicht geändert. Abschließend wird der Account über ein prepared Statement aktualisiert und die Anzahl der betroffenen Datensätze zurückgegeben. Da es nicht vorgesehen ist, dass ein Account zu löschen, sondern nur zu deaktivieren, ändert die Methode **accountsDisable()** das *enabled* Flag, nachdem überprüft wurde, ob der Benutzer die Rolle *admin* hat.

Um einen neuen Patient anzulegen kann die Methode **patientsCreate()** verwendet werden, die ein prepared Statement verwendet, um die Daten einzufügen. Über die Methode **patientsRessourceUpdate()** sollen alle Daten bearbeitet werden können, die durch die Ressource *Patient* des REST Services dargestellt werden. Das sind zum einen alle Daten der Patiententabelle und die E-Mail Adresse und die Mobiltelefonnummer, die in der Accounttabelle gespeichert werden. Zuerst wird über den patients_view überprüft, ob der aktuelle Benutzer berechtigt ist, den Patienten zu bearbeiten. Dann werden in einer Transaktion beide Tabellen aktualisiert. Wenn ein Patient gelöscht werden soll, kann dies per **patientsDelete()** getan werden.

Für die restlichen Tabellen wurden folgende Stored Procedures erstellt, um Daten zu erstellen: **catCreate(), ccqCreate(), charlsonCreate(), deathCreate(), reportCreate(), readingsCreate()** und

Abbildung 4.3: ER-Diagramm der Berechtigungstabellen

treatmentCreate(). Alle diese Methoden prüfen mit Hilfe der patients_view, ob der aktuelle Benutzer Zugriff auf den betreffenden Patient hat. Wenn der Benutzer die Rechte hat, werden die Daten über ein prepared Statement eingefügt. Die Methoden **catUpdate(), ccqUpdate(), charlsonUpdate(), deathUpdate(), reportUpdate(), readingsUpdate()** und **treatmentUpdate()** funktionieren ähnlich und dienen dazu einen Datensatz zu bearbeiten. Mit der Methode **deleteExamRecord()** lässt sich ein Datensatz aus den Tabellen cats, ccqs, charlsons, reports, readings und treatments löschen. Dazu muss der Name der Tabelle übergeben werden und der Aufrufende muss die Rolle *admin* haben oder die Rolle *doctor* und der Arzt des betreffenden Patienten sein.

Für den lesenden Zugriff auf die Tabellen der ärztlichen Untersuchungen wurden Stored Procedures implementiert, damit unterschieden werden kann, ob der Arzt keinen Zugriff auf die Daten des angeforderten Patienten hat, oder noch keine Daten zu diesem Patienten existieren. **listExam()** und **listSingleExam()** dienen dem Zugriff auf die Tabellen cats, ccqs, charlsons, readings und treatments, wobei man die gewünschte Tabelle als Parameter übergibt. Außerdem nehmen sie auch Parameter entgegen, die Pagnierung unterstützen. Da für die Tabelle death Paginierung keinen Sinn macht, wurde die Methode **deathGet()** implementiert. Diese Stored Procedures dürfen nur von Benutzern der Rolle *doctor* aufgerufen werden. Da Patienten und Ärzte lesenden Zugriff auf die täglichen Berichte benötigen, wurden die Methode **reportList()** implementiert, die ebenfalls Paginierung unterstützt. Für den Zugriff auf ein einzelnen Bericht wurde **reportListOne()** erstellt.

Für den OAuth Funktion, die die Token erstellt, wurde eine Funktion **login()** geschrieben, die Benutzername und Passwort entgegen nimmt. Diese Funktion prüft über ein prepared Statement, ob die Informationen gültig sind und gibt, wenn sie gültig sind, den entsprechenden Datensatz des Benutzers zurück aus dem dann das Token erstellt werden kann.

4.1.6 Rechtevergabe

Um das Rechtemanagementsystem der Datenbank zu nutzen, wird pro Benutzer im ECHO-System ein Datenbankbenutzer erstellt. Dem Datenbankbenutzer werden dann anhand der ECHO-Rolle Rechte auf den Tabellen und den gespeicherten Prozeduren zugewiesen, die er im Rahmen seiner Rolle benötigt. Sobald über die REST API ein Benutzer angelegt wird, erstellt die zuständige Stored Procedure einen Datenbankbenutzer, wobei die accountId als Benutzername verwendet wird. Um die Sicherheit zu erhöhen wird, der Datenbankaccount auf localhost beschränkt.

Im Anschluss kann mit der Routine grantRolePermissions() der Datenbankbenutzer mit den benötigten Rechten ausgestattet werden. Da MySQL keine Rollen unterstützt, müssen die Rechte mittels mehrerer GRANT Statements vergeben werden. Die Routine verwendet die Tabellen perm_roles_views und perm_roles_procedures, um die Statements zu erzeugen. In den Tabellen ist hinterlegt, welche Rolle

Zugriff auf welche Views bzw. Stored Procedures hat (siehe Abbildung 4.3). Die zusätzliche Routine grantRolePermissions() ist notwendig, da es nicht erlaubt ist, in einer Funktion A mittels GRANT Statements Rechte für die Funktion A zu vergeben.

Damit sichergestellt ist, dass Benutzer neue Benutzer erstellen können, wird nachdem ein Datenbankbenutzer mit der Rolle admin oder doctor erstellt wurde, dem Account ebenfalls erlaubt, die Funktion grantRolePermissions() auszuführen.

4.1.7 Schemaexport aus Microsoft Access

Vor dem Start des ECHO Projekts wurden die Patientendaten in einer Accessdatenbank verwaltet. Die Access Datenbank war mit einer grafischen Oberfläche versehen über die Daten eingegeben werden konnten. Die grafische Oberfläche konnte mit Hilfe von Visual Basic programmiert werden. Durch Visual Basic wurde beispielsweise gesteuert, wann die Daten in der Oberfläche gespeichert werden, oder wann Berechnungen durchgeführt werden. Der Nachteil dieser Lösung war, dass es nur lokal verwendbar war und nicht mehrbenutzerfähig. Außerdem wäre die Integration in ein System, über das die Patienten zuverlässig selbst Daten eintragen können, nicht möglich gewesen. Um im ECHO Backend dieselben Daten verwalten zu können und die selbe Funktionalität zu haben, wurde mit Hilfe der Accessdatenbank ein Datenbankschema für eine MySQL Datenbank erstellt.

Die Tabellen und Spalten wurden dabei von Hand extrahiert und mit MySQL Workbench wurde daraus ein Schema erstellt. Nach Rücksprache mit den am Projekt beteiligten Ärzten wurden die Tabellen Basic (Patientendaten), CCQWeek, Charlson, Catscale und Medication, mit Änderungen, übernommen.

4.2 RESTful API

Für die Implementierung der RESTful HTTP-API wurde Node.js gewählt Node.js ist ein JavaScript-Framework zur Entwicklung von skalierbaren serverseitigen Webanwendungen. Anders als klassische Webserver, die pro eingehendem Request einen Thread starten, nutzt node.js nur einen einzigen Thread zur Bearbeitung der Anfragen. Damit dieser Thread nicht blockiert, muss er seine Arbeit wenn möglich delegieren. Dabei profitiert node.js davon, dass bei den meisten Anfragen externe Ressourcen wie Datenbanken oder Dateisysteme involviert sind. Während die Threads klassischer Webserver viel Zeit mit Warten verbringen, wenn externe Ressourcen involviert sind, nimmt der node.js Thread die Anfrage entgegen, bearbeitet sie bis zu dem Punkt an dem mit der externen Ressource interagiert werden soll, startet die Interaktion und legt die Anfrage beiseite, bis eine Antwort der Ressource vorliegt. Dann kümmert er sich um die nächste Anfrage bis zu deren erster Interaktion mit einer Ressource. Wenn eine Antwort einer Ressource vorliegt, wird der Serverthread über eine Callbackfunktion informiert und die Anfrage wird weiterbearbeitet. [Rod12]

Da sich JSON aufgrund der JavaScript-Basis von node.js nativ verarbeiten lässt und node.js die Entwicklung von HTTP-basierten Webdiensten einfach macht, empfiehlt sich node.js als Basis für JSON-basierte REST-Dienste. [Rod14]

Zur Dokumentation der RESTful API wurde Swagger eingesetzt. Swagger ist eine Spezifikation und ein Framework zum Beschreiben, Erstellen, Konsumieren und Visualisieren von RESTful Webservices.

4.2.1 Tokenbasierte Authentifizierung

Um die API zu verwenden, muss man sich authentifizieren. Dafür wird ein Token benötigt, das mit dem Authorization Header wie folgt übergeben wird:

Authorization: Bearer <access_token>

Bearer bezeichnet dabei den Typ des verwendeten Tokens und das Authentifikationschema. Wenn der Typ als Bearer angegeben wird, ist das Token für den Anwender einfach nur ein String, der keine für den Client erkennbare Information enthält. Ist das Token nicht vorhanden oder fehlerhaft, wird der Request mit dem HTTP Statuscode 401 abgewiesen.

Um ein Access Token zu erhalten, muss ein Request per HTTP POST an den Token-Endpoint geschickt werden. In der Prototypimplementierung ist der Token Endpoint /login. Dieser Endpoint implementiert das OAuth 2.0 Protokoll mit dem Grantflow Ressource Owner Password Credentials. Dieser Flow tauscht die korrekte Kombination aus Benutzername und Passwort aus gegen ein Access Token (siehe Abbildung 3.5). Das OAuth 2.0 Protokoll schreibt vor, dass der Request Body wie folgt aussehen muss, wenn ein Token ausgestellt werden soll:

grant_type=password&username=<Benutzername>&password=<Passwort>

Der Parameter grant_type gibt dabei den Grantflow an, damit der OAuth 2.0 Server erkennen kann, nach welchem Flow verfahren werden soll. Der Wert des Parameters („password“) steht hier für den Ressource Owner Password Credentials Flow.

Für die Implementierung des OAuth 2.0 Servers wurde das Node.js Modul oauth2orize verwendet. Wenn der OAuth 2.0 Server einen Request erhält, der wie oben beschrieben aufgebaut ist, werden Benutzername und Passwort mit Hilfe der Datenbank validiert. Wenn die Kombination aus Benutzername und Passwort stimmt, wird ein Token ausgestellt. In diesem Token werden die Account ID, die Rolle des Benutzers, der Timestamp, an dem das Token erstellt wurde und der Timestamp, an dem das Token abläuft, kodiert. Um das Token zu erstellen, wird das Node.js Modul jsonwebtoken verwendet. Ein mit diesem Modul erstelltes Token ist ein JSON Web Token (JWT). Ein JSON Web Token ermöglicht es, JSON Objekte signiert auszutauschen. Es besteht aus drei Teilen: einem Header, den Informationen, die es beinhaltet und der Signatur, mit der man sicherstellen kann, dass die Informationen nicht geändert wurden.

Die HTTP Endpoints der REST API werden mit dem Modul passport.js geschützt. Jedes Mal wenn ein Request ein Token enthält, wird dieses Token anhand seiner Signatur geprüft. Wenn das Token geändert wurde oder abgelaufen ist, wird die weitere Verarbeitung des Requests abgebrochen.

Bei jedem Login wird neben einem Access Token ein Refresh Token ausgestellt. Wenn ein Access Token abgelaufen ist, kann das Refresh Token verwendet werden, um ein neues Access Token zu

Listing 4.4 Node.js mit Express: Hello World

```
var express = require('express');
var app = express();

app.use(function(req, res, next){
  console.log('Time: %d', Date.now());
  next();
});

app.get('/', function(req, res, next){
  res.send('Hello World');
});

app.listen(3000);
```

bekommen ohne Benutzername und Passwort erneut eingeben zu müssen. Das Refresh Token wird nach der Erstellung in einer HashMap gespeichert und nach dem Einsatz des Tokens wieder aus der HashMap gelöscht. Jedes Refresh Token ist damit nur einmal einsetzbar. Falls ein Refresh Token in ein Access Token getauscht werden soll, muss eine HTTP Nachricht mit folgendem Body übermittelt werden:

grant_type=refresh_token&refresh_token=<Refresh Token>

4.2.2 Implementierung der API Funktionen

Zur Implementierung des REST Service wurde das Express-Framework verwendet, das die Entwicklung von Webanwendungen vereinfachen soll. Eine mit dem Express-Framework geschriebene Anwendung ist im Grunde genommen ein Stack von nacheinander aufgerufenen Funktionen, sogenannter Middleware. Diese Middleware-Funktionen bekommen als Parameter den HTTP Request (req), die HTTP Response (res) und einen Verweis auf die nächste Funktion im Middlewarestack (next) übergeben. In jeder Middleware-Funktion kann beliebiger Code ausgeführt werden.

In Listing 4.4 ist ein Beispiel für eine einfache Express-Anwendung dargestellt. In diesem Fall besteht der Stack aus zwei Middlewarefunktionen. Eine Middlewarefunktion kann entweder über die Funktion app.use() oder app.VERB() eingebunden werden, wobei VERB hier für die HTTP Methoden steht. Mit use() kann dabei eine beliebige Funktion in den Stack eingebunden werden, und mit einem HTTP Verb kann ein HTTP Endpunkt erzeugt werden. Dabei kann eine Middleware auch nur für einen bestimmten Pfad aktiv sein. Die erste Middleware gibt auf der Konsole die aktuelle Zeit aus und ruft dann mittels next() die nächste Funktion des Stacks auf. Da für diese Funktion kein expliziter Pfad angegeben wurde, würde sie auch aufgerufen werden, wenn eine Funktion für einen beliebigen anderen Pfad definiert würde. Die nächste Funktion im Stack sendet dann „Hello World“ an den Client. Mit res.send() kann nicht nur am Ende des Stacks eine Antwort gesendet werden, sondern auch an einer beliebigen Stelle im Stack die Bearbeitung abgebrochen werden. Ein Beispiel wäre eine Middleware zur Überprüfung der Zugangsberechtigung. Ist der Client berechtigt kann mittels next() die nächste Funktion im Stack aufgerufen werden. Ist er nicht berechtigt, kann per res.send() die Verarbeitung des Requests abgebrochen und ein Fehler zurückgegeben werden.

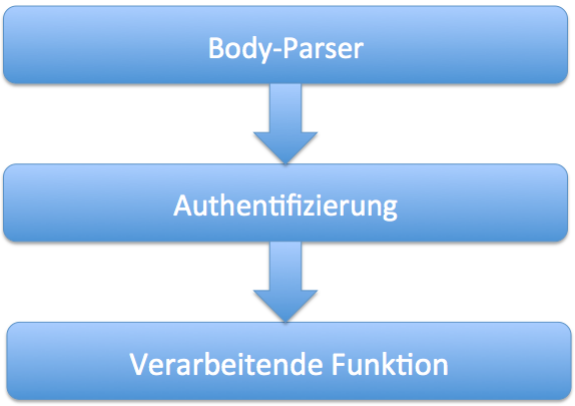

Abbildung 4.4: Middlewarestack des REST Services

Der Middlewarestack, der für den REST Service implementiert wurde, besteht aus drei Funktionen (siehe Abbildung 4.4). Der Body-Parser ist die erste Funktion von der ein Request verarbeitet wird. Da ein JSON Objekt nur als String vom Client zum Server übertragen werden kann, muss der String wieder in ein JSON Objekt umgewandelt werden. Falls der Body die Stringrepräsentation eines validen JSON Objekts enthält, wird ein JSON Objekt erstellt. Dieses Objekt kann dann über die Variable req.body angesprochen werden. In der nächsten Middlewarefunktion werden die Authentifizierungsinformationen im Authorization Header überprüft. Enthält er ein valides Token, wird die Funktion, die den HTTP Request schließlich verarbeiten soll, aufgerufen. Wenn das Token nicht valide ist oder der Authorization Header leer ist, wird der HTTP Statuscode 401 zurückgegeben. Die Informationen aus dem Token werden in der Variable req.user gespeichert. Eigentlich sollte die Autorisierung auch eine Funktion im Stack sein. Da es aber kein Node.js Autorisierungsmodul gibt, das mit Subressourcen umgehen kann, wurde diese Funktionalität in die verarbeitende Funktion ausgelagert.

Die Verarbeitende Funktion im Middlewarestack stellt immer einen HTTP Endpoint dar. Diese Funktionen sind immer nach dem folgenden Muster aufgebaut:

1. Anhand der Rolle wird überprüft, ob der Benutzer berechtigt ist diese Funktion auszuführen.
2. Eine Verbindung zur Datenbank wird aufgebaut.
3. Aus der Variable req.user wird der aktuelle Benutzer ausgelesen. Die Datenbank wird angewiesen, diesen Benutzer für die Verbindung zu nutzen. Das Passwort für den Datenbankbenutzer kann aus dem für alle Nutzer gültigen Präfix und der ID des Benutzers berechnet werden.
4. Die SQL Anweisung wird aus dem Body des HTTP Requests (req.body), den Query-Parametern (req.query) und den Platzhaltern in der URI (req.params) erzeugt und abgeschickt.
5. Das Ergebnis der SQL Anweisung wird als JSON Objekt zurückgegeben. Anschließend wird das JSON Objekt verarbeitet und dem Client eine Antwort geschickt.

Der Übergang zwischen den Schritten stellt immer einen asynchronen Aufruf dar, wie er in Node.js häufig verwendet wird.

Listing 4.5 Links in JSON mit Hypertext Application Language

```
"_links": {
  "self": { "href": "<Link zur Ressource>"}
  "first": { "href": "<Link zur ersten Seite der Listenressource>"}
  "prev": { "href": "<Link zur vorherigen Seite der Listenressource>"}
  "next": { "href": "<Link zur naechsten Seite der Listenressource>"}
}
```

4.2.3 Repräsentation

Der Body des HTTP Requests beinhaltet die Informationen, die an den Server gesendet wird und der Body der HTTP Response die Informationen, die an den Client zurückgegeben werden. Der Internet Media Type der bei Request und Response genutzt werden wird, ist „application/hal+json“. Der Body wird jeweils die JSON Daten enthalten und zusätzliche Links die mit Hypertext Application Language beschrieben werden. Die Zeichenkodierung wird UTF-8 sein. Es ist möglich, das System in Zukunft um weitere Repräsentationen zu ergänzen. Beispielsweise könnte eine Unterstützung für XML oder PDF hinzugefügt werden.

Um das REST-Konzept Hypermedia as the engine of application state umzusetzen, werden die Ressourcen mit Links verknüpft. Um die Links in JSON darzustellen, wird Hypertext Application Language genutzt. In Listing 4.5 ist ein Beispiel dargestellt. Das Beispiel stellt die Links eines Ausschnitts einer Listenressource dar. Dabei wird durch die Links ein Link auf die Ressource selbst beschrieben, ein Link auf die erste Seite der Listenressource und Links für die nächste und vorherige Seite der Listenressource.

4.2.4 Swagger

Zu Beginn der Arbeit wurde parallel zur Entwicklung des REST Services ein Browser-Prototyp entwickelt. Dieser Prototyp sollte dazu dienen, die Möglichkeiten des REST Services darzustellen. Es war allerdings sehr aufwändig ,den Prototyp um neue Funktionen des Services zu erweitern und ebenso aufwändig, den Prototyp an Änderungen der API anzupassen. Deswegen wurde nach einer Möglichkeit gesucht, die Funktionen des REST Service zu testen, ohne nach einer Änderung der Funktionen eine Anwendung anpassen zu müssen. Swagger UI bietet die Möglichkeit, alle Funktionen eines REST Services mit Beschreibungen versehen darzustellen und zu testen.

Swagger UI ist Teil des Swagger Projekts. Das Ziel des Swagger Projekts ist es einen Standard für ein sprachunabhängiges Interface für REST APIs zu definieren. Dieses Interface soll es Menschen und Computern ermöglichen alle Funktionen einer REST API zu finden und zu verstehen, ohne dass sie Zugriff auf eine Dokumentation oder den Source Code haben. Eine API, die die Swagger-Spezifikation implementiert, stellt Daten im JSON Format zur Verfügung, die die Struktur der REST API beschreiben. Diese können entweder manuell erzeugt und durch den Server bereitgestellt werden, oder aus dem Source Code heraus erzeugt werden. Swagger UI verwendet diese Daten um die Funktionen der API anzuzeigen und sie anzusprechen. Neben Informationen über die Struktur des Services lassen sich auch Models definieren, die beschreiben, wie die Daten aussehen, die von einer bestimmten Funktion verarbeitet oder generiert werden. Durch ein Model lässt sich beispielsweise beschreiben, welche

Datentypen die Felder haben oder welche Einschränkungen zusätzlich gelten. Ein Model kann auch zur verbalen Beschreibung der Felder verwendet werden.

Eine Instanz von Swagger UI kann theoretisch mit jeder REST API interagieren, die die Swagger Spezifikation umsetzt. Um eine swaggerunterstützte REST API mit Swagger UI zu erkunden, muss man auf der Swagger UI Seite die Adresse der Swagger Informationen der REST API eintragen.

In Abbildung 4.5 ist ein Teil der Swagger-UI-Darstellung der ECHO REST API dargestellt[1]. Zu sehen sind die ausgeklappten Ressourcen accounts und patients und ihre Operationen mit zugehörigen Beschreibungen. Wenn der Benutzer nun auf eine der Operationen klickt, wird diese auch ausgeklappt und der Benutzer kann sie testen ohne einen Client implementieren zu müssen. Wenn eine Operation aufgeklappt wird, wird auch sichtbar, welche Daten als Eingabe erwartet werden. Über Swagger UI ist es möglich, einer REST API Eingabedaten als Bestandteil der URL (Query), als HTTP Header, als Teil des HTTP Bodys oder als Teil des Pfads (im Screenshot beispielsweise durch {id} dargestellt) zu übergeben. Wenn für die Eingabedaten ein Model hinterlegt ist, wird das Model auch angezeigt. Daraus lässt sich dann ablesen, welche Datentypen oder sonstige Einschränkungen für die Felder des Models gelten. Das Model kann auch angezeigt werden, wenn die Funktion Daten zurückliefert. Durch ein vollständiges Model kann der Benutzer nachvollziehen, was die Daten aussagen, die die Funktion zurückgibt.

Um mit Node.js und dem Express-Framework einen Service zu schreiben, der Swagger unterstützt, kann swagger-node-express benutzt werden. Wie in Listing 4.6 dargestellt, müssen die Funktionen dann nicht mehr bei Express registriert werden, sondern, mit zusätzlichen Metainformation, bei der Swagger Implementierung. Als Beispiel wurden hier Metainformationen einer Funktion genommen, die auch in Abbildung 4.5 zu sehen ist.

Außer für Node.js stehen Implementierungen für viele andere Programmiersprachen, wie zum Beispiel Java, PHP oder Python, zur Verfügung. Außerdem gibt es Tools, die aus der Swagger Spezifikation eines Services Client- und Serverstubs in verschiedenen Progammiersprachen erstellen.

[1]Der Prototyp des ECHO Backends und eine passende Swagger-UI-Version sind unter `http://echo.informatik.uni-stuttgart.de` erreichbar.

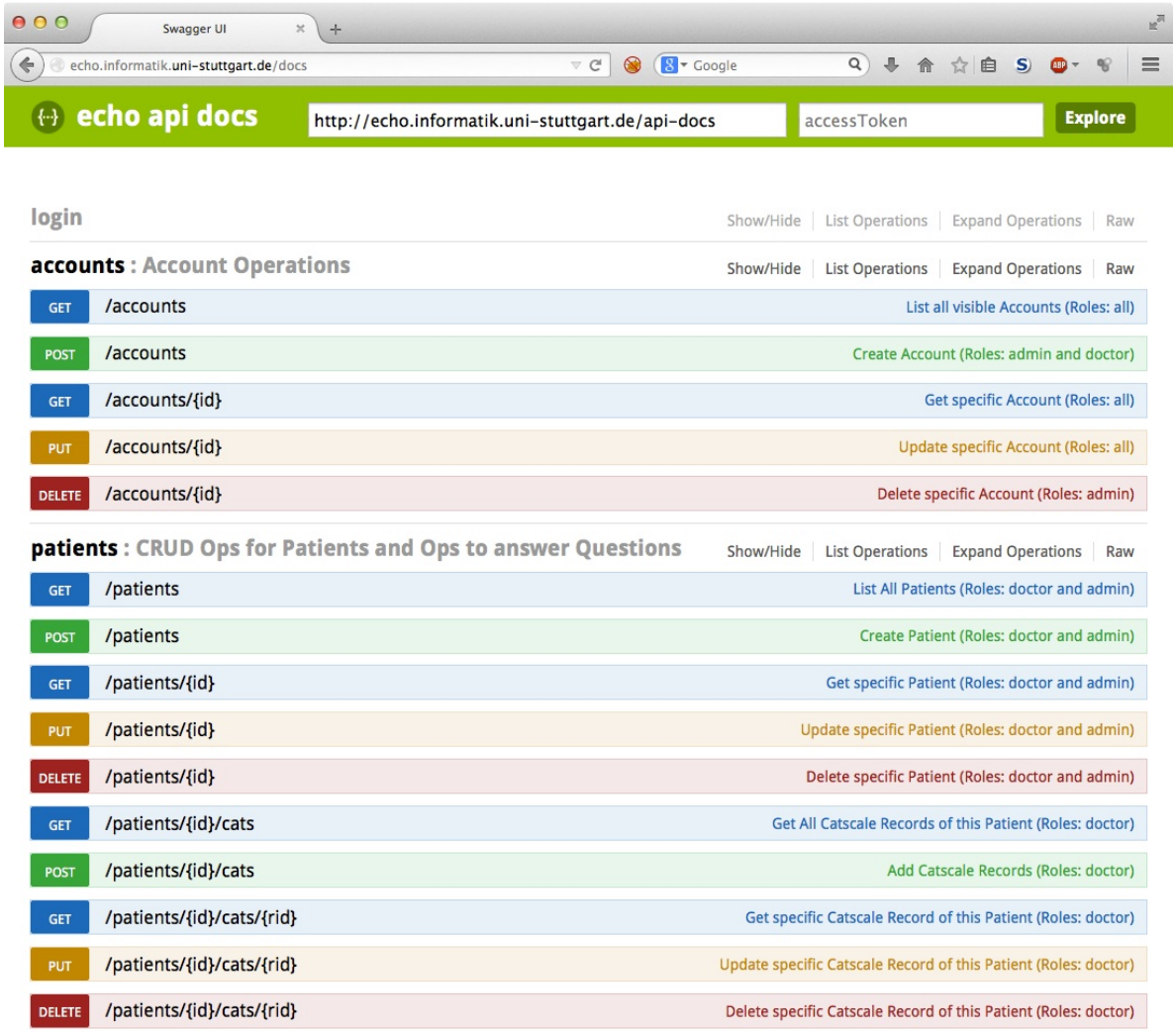

Abbildung 4.5: Screenshot der Swagger-UI-Darstellung der ECHO API

Listing 4.6 Node.js mit Express: Swagger Integration

```
// Load module dependencies.
var express = require("express") , swagger = require("swagger-node-express");
// Create the application.
var app = express();
// Couple the application to the Swagger module.
swagger.setAppHandler(app);

var findById = {
  'spec': {
            summary : "Get specific Account (Roles: all)",
            path : "/accounts/{id}",
            method: "GET",
            type : "Account",
            nickname : "accountsFindById",
            parameters : [swagger.pathParam("id", "ID of the Account", "string")],
            responseMessages : [swagger.errors.notFound('id')]
  },
  'action': findById()
};

swagger.addGet(findById);

app.listen(3000);
```

5 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurden Teile des Backends für das deutsch-griechische Forschungsprojekt ECHO (Enhancing Chronic patients' Health Online) entwickelt. Das Ziel des Projekts ist es mit Hilfe von Cloud-Computing-Technologien, Data Mining und Smartphones die Situation von Patienten mit chronischen Lungenkrankheiten zu verbessern. Unter anderem soll es für den Patienten möglich sein, jeden Tag Fragen via Smartphone zu beantworten. Diese Antworten sollen bei der Früherkennung von dauerhaften Verschlimmerungen der Krankheit, auch Exazerbationen genannt, helfen.

Vor dem Start des ECHO Projekts wurden die Patientendaten und Untersuchungsergebnisse mit Hilfe einer Microsoft Access Datenbank verwaltet.
Die Access Datenbank war nur lokal und nur von einem Arzt verwendbar. Außerdem war kein Remote Access auf die Access Datenbank möglich, der aber nötig gewesen wäre für die tägliche Dateneingabe der Patienten. Aus diesem Grund war es nötig, die Access Datenbank in eine relationale Datenbank zu überführen. Dazu wurde aus der Access Datenbank von Hand ein SQL-Schema erzeugt. Anschließend wurden alle Funktionen der Access Datenbank untersucht und die relevanten Funktionen in Triggern umgesetzt. Nach einem Gespräch mit den am Projekt beteiligten Ärzten wurde das Schema angepasst und nur die benötigten Tabellen übernommen.

Um die Datensicherheit zu gewährleisten wurden Views verwendet und Rollen eingeführt. Die eingeführten Rollen sind:

1. admin, für administrative Aufgaben
2. doctor, für Ärzte, die über das System ihre Patienten überwachen
3. patient, für Patienten

Für jeden Benutzer des Backends wurde zusätzlich ein Datenbankbenutzer angelegt, um das Rechtesystem der Datenbank nutzen zu können. Der Benutzername für die Datenbank wird dabei aus der Account ID im ECHO Backend abgeleitet. Der Benutzername kann so benutzt werden um dynamische Views zu erzeugen, die Patientendaten in Abhängigkeit davon anzeigen, welcher Arzt eingeloggt ist. Dadurch kommt man mit einem View pro Tabelle aus.

Um die Daten gegen SQL Injections abzusichern, wurden Stored Procedures, die Prepared Statements verwenden, für den schreibenden Zugriff auf die Gesundheitsdaten implementiert.

Die Kommunikation zwischen dem Backend und den Smartphones beziehungsweise den Browseranwendungen wird durch eine RESTful API ermöglicht. Die Ressourcen der API werden über URIs adressiert und unterstützen JSON als Repräsentation. Damit das REST Konzept der Verlinkungen umgesetzt werden kann, wurden die JSON Daten mit Hypertext Application Language erweitert, um die Beziehungen der Ressourcen untereinander zu modellieren. Um die Sicherheit der Daten während des Transports zu gewährleisten, wird TLS eingesetzt.

Für die Authentifizierung verwendet die RESTful API OAuth 2.0 mit dem Authorization Flow Resource Owner Password Credentials. Dabei wird gegen eine gültige Kombination aus Benutzername und Passwort ein Access Token zurückgegeben, mit dem die API dann verwendet werden kann. Das Token ist ein JSON Webtoken (JWT), bei dem der Inhalt Base64 kodiert ist, aber eine Änderung unmöglich ist, da der Inhalt signiert wurde. In dem Token sind die Rolle und die Account ID des Benutzers gespeichert, für den das Token ausgestellt wurde. Außerdem enthält das Token den Zeitpunkt an dem es erstellt wurde und den an dem es abläuft. Wenn es abläuft, kann es nicht mehr eingesetzt werden. Dann kann das Refresh Token eingesetzt werden, um ein neues Access Token zu erhalten. Der Einsatz eines dieses Tokens hat den Vorteil, dass das Passwort des Benutzers nicht gespeichert werden muss, sondern nur das Access Token und dass die RESTful API anhand des Tokens entscheiden kann, ob der Benutzer berechtigt ist, die Funktion zu verwenden oder nicht.

Ausblick

Im Rahmen dieser Arbeit wurde ein Konzept entwickelt, um die Daten des Backends zu schützen. Es wurde kein Schutz auf Dateisystemebene realisiert, um die Daten zum Beispiel im Fall eines Diebstahls der Festplatte zu schützen oder falls ein Angreifer Zugriff auf das Dateisystem erlangt. Dies könnte man durch eine Verschlüsselung der Datenbankdateien erreichen.

Die Analysekomponente, die die in Kapitel 3.4 definierten Analysen durchführt, kann aufbauend auf dieser Arbeit implementiert werden. Das sind zum einen die Analyse zur Auswertung der täglichen Dateneingabe durch den Patienten, die der frühzeitigen Erkennung und Verhinderung einer Exazerbation dient und die Analyse zur Berechnung des COPD Schweregrads, der unter anderem Rückschlüsse auf das Exazerbationsrisiko erlaubt.

Das ECHO Backend könnte auch noch um die Fähigkeit zum Autoscaling erweitert werden. Des Weiteren kann das Verhalten des Systems unter Last untersucht und anschließend optimiert werden.

Außerdem könnten noch Installationsskripte zum automatischen Deployment im Krankenhaus oder in der Cloud geschrieben werden. Um das Backend in der Cloud sicher zu deployen, könnte zusätzlich noch ein TOSCA Cloud Service Archive (CSAR) erstellt werden.

Literaturverzeichnis

[BKK+14] M. Bitsaki, C. Koutras, G. Koutras, F. Leymann, B. Mitschang, C. Nikolaou, N. Siafakas, S. Strauch, N. Tzanakis, M. Wieland. An Integrated mHealth Solution for Enhancing Patients' Health Online. In *Proceedings of the 6th European Conference of the International Federation for Medical and Biological Engineering (MBEC'14)*, S. 1–4. International Federation for Medical and Biological Engineering (IFMBE), 2014. URL http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2014-61&engl=. (Zitiert auf den Seiten 10 und 26)

[Boy12] R. Boyd. *Getting Started with OAuth 2.0.* O'Reilly Media, 2012. (Zitiert auf Seite 24)

[cat] COPD Assessment Test. URL http://www.catestonline.org/english/index_German.htm. (Zitiert auf Seite 28)

[copa] Chronisch obstruktive Lungenerkrankung (COPD). URL http://www.copd-aktuell.de/. (Zitiert auf Seite 10)

[copb] COPD: Diagnose. URL www.onmeda.de/krankheiten/copd-diagnose-3112-5.html. (Zitiert auf den Seiten 36 und 37)

[DPNB11] M. Deng, M. Petkovic, M. Nalin, I. Baroni. A Home Healthcare System in the Cloud–Addressing Security and Privacy Challenges. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, S. 549–556. IEEE, 2011. (Zitiert auf Seite 15)

[Eur] Europäisches Parlament. Richtlinie zum Schutz natürlicher Personen bei der Verarbeitung personenbezogener Daten und zum freien Datenverkehr. URL http://eur-lex.europa.eu/legal-content/DE/TXT/?uri=CELEX:31995L0046. (Zitiert auf Seite 13)

[Har12] D. Hardt. The OAuth 2.0 Authorization Framework. RFC 6749, RFC Editor, Fremont, CA, USA, 2012. URL http://www.rfc-editor.org/rfc/rfc6749.txt. (Zitiert auf Seite 23)

[hip] Health Insurance Portability and Accountability Act. URL http://www.hhs.gov/ocr/privacy/index.html. (Zitiert auf Seite 13)

[HWS07] R. Hasan, M. Winslett, R. Sion. Requirements of Secure Storage Systems for Healthcare Records. In W. Jonker, M. Petkovic, Herausgeber, *Secure Data Management*, Band 4721 von *Lecture Notes in Computer Science*, S. 174–180. Springer Berlin Heidelberg, 2007. doi:10.1007/978-3-540-75248-6_12. URL http://dx.doi.org/10.1007/978-3-540-75248-6_12. (Zitiert auf Seite 13)

[iso] ISO 27001: An Introduction To Information, Network and Internet Security. URL http://security.practitioner.com/introduction/. (Zitiert auf Seite 14)

[KP11] D. A. Kindy, A.-S. K. Pathan. A survey on SQL injection: Vulnerabilities, attacks, and prevention techniques. 2011. (Zitiert auf Seite 17)

[Occ] Occupational Safety & Health Administration. Access to employee exposure and medical records. URL https://www.osha.gov/pls/oshaweb/owadisp.show_document?p_table=STANDARDS&p_id=10027. (Zitiert auf Seite 13)

[OWA13] OWASP Foundation. OWASP Top 10 - 2013. Technischer Bericht, OWASP Foundation, 2013. URL https://www.owasp.org/index.php/Top_10_2013-Top_10. (Zitiert auf Seite 16)

[RG07] A. Roichman, E. Gudes. Fine-grained access control to web databases. In *Proceedings of the 12th ACM symposium on Access control models and technologies*, S. 31–40. ACM, 2007. (Zitiert auf den Seiten 16 und 17)

[Rod14] G. Roden. 2x Nein, 4x Ja: Szenarien für Node.js, 14. URL http://www.heise.de/developer/artikel/2x-Nein-4x-Ja-Szenarien-fuer-Node-js-2111050.html. (Zitiert auf Seite 47)

[Rod12] G. Roden. *Node.js & Co.* dpunkt.Verlag, 2012. (Zitiert auf Seite 47)

[SESAE12] D. Sobhy, Y. El-Sonbaty, M. Abou Elnasr. MedCloud: healthcare cloud computing system. In *Internet Technology And Secured Transactions, 2012 International Conference for*, S. 161–166. IEEE, 2012. (Zitiert auf Seite 14)

[SJM+12] J. Sundh, Janson, Montgomery, Stallberg, K. Lisspers. Clinical COPD Questionnaire score (CCQ) and mortality. *International Journal of Chronic Obstructive Pulmonary Disease*, S. 833+, 2012. doi:10.2147/copd.s38119. URL http://dx.doi.org/10.2147/copd.s38119. (Zitiert auf Seite 28)

[Til11] S. Tilkov. *REST und HTTP.* dpunkt.Verlag, 2011. (Zitiert auf den Seiten 18 und 21)

Alle URLs wurden zuletzt am 20. 10. 2014 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift