ulm university universität
uulm

**Universität Ulm** | 89069 Ulm | Germany

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

# Konzeption und Realisierung eines Suchmoduls für eine interdisziplinäre Datenbank für Tinnituspatienten

Bachelorarbeit an der Universität Ulm

**Vorgelegt von:**
Jan Musmann
jan.musmann@uni-ulm.de

**Gutachter:**
Prof. Dr. Manfred Reichert

**Betreuer:**
Dr. Rüdiger Pryss und Michael Stach

2017

„Konzeption und Realisierung eines Suchmoduls für eine interdisziplinäre Datenbank für Tinnituspatienten“
Fassung vom 6. Oktober 2017

# Zusammenfassung

Die Tinnitus Database ist ein internationales Projekt zur Erhebung von medizinischen Patienten- und Behandlungsdaten. Ziel der Datenbank ist es, Medizinern dazu zu verhelfen, die vielversprechendste Behandlungsmethode für einen vom Symptom Tinnitus betroffenen Patienten zu finden. Die weitere Optimierung dieses Prozesses und dem damit verbundenen Arbeitsfluss im System, für beteiligte Nutzer, ist Motivation und Anlass dieser Arbeit. Insbesondere wurde das Suchmodul und die betreffende Patientenliste der aktuellen webbasierten Benutzeroberfläche auf Performanz und Funktionalität analysiert. Dabei wurden Leistungsprobleme der Applikation festgestellt, welche zu erhöhten Wartezeiten beim Aufbau der Website führen und den Nutzer dadurch in seiner Produktivität beeinträchtigen können. Um dem entgegenzuwirken, wurden die Ursachen der identifizierten Probleme sukzessive mit entsprechenden Gegenmaßnahmen behandelt. Durch Optimierung von Laufzeitumgebung, Datenbankabfragen und Darstellungskonzept konnte eine vielfache Leistungssteigerung der betroffenen Seite, aber auch der Applikation, erreicht werden. Die momentan für den Endnutzer benötigte Ladezeit von knapp 15 s konnte auf weit weniger als 1 s reduziert werden. Zusammen mit einer Überarbeitung der Suchmaske bietet die neuentstandene Lösung eine für den Anwender robuste und schnelle Oberfläche.

# Inhaltsverzeichnis

# 1 Einleitung

Die Tinnitus Database[1] ist ein internationales Projekt zur Erhebung von medizinischen Patienten- und Behandlungsdaten. Ziel der Datenbank ist es, Medizinern dazu zu verhelfen, die vielversprechendste Behandlungsmethode für einen vom Symptom Tinnitus betroffenen Patienten zu finden. Die weitere Optimierung dieses Prozesses und dem damit verbundenen Arbeitsfluss im System, für beteiligte Nutzer, ist Motivation und Anlass dieser Arbeit. In diesem speziellen Fall wird die Performanz und Funktionalität des Suchmoduls für Patienten in der aktuellen webbasierten Benutzeroberfläche genauer untersucht.

## 1.1 Problemstellung des aktuellen Suchmoduls

Die stetig wachsende Patientenzahl, welche mit Hilfe der Tinnitus Database verwaltet und abrufbar gemacht wird, erzeugt eine hohe Datenmenge, die visualisiert werden muss. Damit der Nutzer die von ihm gewünschten Datensätze so schnell wie möglich finden kann, ist eine präzise und performante Suche von Einträgen nötig, jedoch im aktuell verwendeten System nicht gewährleistet. Das Suchmodul bietet momentan keinerlei Funktionalität zur erweiterten Filterung der Daten, was ein gezieltes Suchen erschwert. Des Weiteren benötigt das Laden und Rendern der aktuellen Patientenliste sehr viel Zeit, womit der Nutzer zum Warten gezwungen wird, um eine Suchanfrage stellen zu können. Solch ein Umstand kann sich laut mehreren Studien negativ auf die Zufriedenheit [10] und Produktivität [13] des Nutzers im System auswirken.

[1] `https://www.tinnitus-database.de`

## 1.2 Zielsetzung

Um ein benutzerfreundliches Suchmodul konzipieren und implementieren zu können, ist es zum einen die Aufgabe dieser Arbeit, die aktuellen Probleme und Fehlerquellen aufzudecken und zu analysieren. Zum anderen werden diese Ergebnisse zusammen mit Lösungsansätzen diskutiert. Im Anschluss werden direkt implementierte Lösungen und ergänzende Vorschläge präsentiert, die zukünftig von den Betreibern umgesetzt werden können.

## 1.3 Struktur der Arbeit

Im nächsten Kapitel wird zunächst der Kontext der Arbeit genauer beschrieben. Dazu gehören Begrifflichkeiten wie Tinnitus, aber auch die organisatorischen Strukturen unter welchen das ursprüngliche Projekt entstanden ist. Außerdem werden kurz die wichtigsten Strukturen des auf dem PHP-Framework Laravel basierten Projektes beleuchtet. Das folgende Kapitel zur Methodik beinhaltet eine Übersicht von Arbeiten und Anforderungen, die für mögliche Verbesserungen am System und das Thema der Arbeit relevant sind. Folgend wird hier auf mein weiteres Vorgehen, sowie die dazu verwendeten Werkzeuge und Umgebungen der Analyse eingegangen. Im vierten Kapitel werden Entscheidungen und Details zur Implementierung dargelegt. Zum Abschluss werden sowohl die daraus entstandenen Ergebnisse geschildert und diskutiert, als auch ein Ausblick auf mögliche weiterführende Arbeiten gegeben.

# 2 Hintergrund

Dieses Kapitel dient der Erläuterung einiger Begriffe und Hintergründe, die für das Verständnis und den Kontext in dem die Arbeit steht hilfreich sind. Dazu gehören zum einen das Krankheitsbild Tinnitus selbst, zum anderen aber auch die Organisationen, welche hinter der Datenbank stehen und die Forschung in diesem Gebiet voran treiben.

## 2.1 Tinnitus

Tinnitus ist eine Wahrnehmung eines Tones, ohne dass ein eigentlicher akustischer Reiz zugrunde liegt. Allein in Europa sind mehr als 70 Millionen Menschen von diesem Symptom betroffen, welches damit keine Seltenheit darstellt. Für mehr als 7 Millionen Betroffene sind die Folgen so drastisch, dass der Alltag erschwert wird und es zu Folgeleiden wie Depressionen, Angstzuständen und Schlaflosigkeit kommen kann [18].

Der Schwerpunkt der Ursachenforschung lag lange Zeit auf Veränderungen des Innenohrs oder des Gehörs. Aktuellere Erkenntnisse aus der Hirnforschung haben jedoch gezeigt, dass die Veränderung von neuronalen Aktivitäten in spezifischen Hirnarealen ebenfalls zu Tinnitus führen kann. Dies hat bereits zu neuen Entwicklungen von therapeutischen Behandlungsansätzen geführt. Trotz medizinischem Fortschritt bleibt Tinnitus eine schwer zu behandelnde Herausforderung, für die bisher keine etablierten Methoden zur Heilung verfügbar sind. Viele untersuchte Behandlungsmethoden führen bei einigen Patienten zu positiven Ergebnissen, andere Patienten sprechen auf diese wiederum nicht an. Daraus lässt sich ableiten, dass es mehrere Formen von Tinnitus geben muss. [24, 20] Genau hier setzt die Tinnitus Research Initiative mit einer Patientendatenbank an, um eine Quelle zur Profilierung von betroffenen Patienten zu bieten [22].

## 2.2 TINNET

Das Projekt „**Tin**nitus Research **Net**work“[1] oder einfach „TINNET“, wurde 2013 im Rahmen einer sogenannten *Action* [24] der „European Cooperation in Science and Technology“[2] (COST) ins Leben gerufen. COST besteht seit 1971 und ist damit das am längsten laufende europäische Rahmenprogramm zur Förderung von Kooperationen in der Wissenschaft [27].

Dieses paneuropäische Netzwerk hat es sich zum Ziel gesetzt, die bedeutendsten Kriterien, die zur Subtypisierung von Tinnitus nötig sind, zu identifizieren. Außerdem sollen diese verschiedenen Subtypen neurobiologisch weiter fundiert und nach ihrer Relevanz im Bezug auf die Wirksamkeit bei Behandlung der Patienten untersucht werden [24]. Um dieses Ziel zu erreichen ist eine standardisierte und koordinierte Erhebung von klinischen neurobiologischen und genetischen Daten der Patienten nötig. Diese Daten erleichtern dann die Entwicklung von neuen Therapien und ermöglichen eine höhere Effizienz bei bereits praktizierten Methoden. Zur Bewältigung dieser Aufgabe, wurde TINNET in fünf Arbeitsgruppen aufgeteilt. Die Tinnitus Research Initiative Database ist eine dieser Gruppen und beschäftigt sich mit der Datenverwaltung in einer zentralen Datenbank [18, 19].

## 2.3 Tinnitus Research Initiative Database

Die Tinnitus Research Initiative[3] (TRI) ist eine gemeinnützige Organisation, die es sich zur Aufgabe gemacht hat, die Entwicklung von wirkungsvollen Behandlungen gegen Tinnitus voran zu treiben [20]. Im Sommer 2008 wurde zu diesem Zweck die TRI Database gestartet. Sie ist die erste internationale Kooperation mehrerer auf Tinnitus spezialisierter Kliniken [21]. Im Rahmen der COST Action TINNET wird diese Datenbank in aktiver Zusammenarbeit der Universität Regensburg (Lehrstuhl für Psychiatrie und Psychotherapie[4]) und der Universität Ulm (Institut für Datenbanken und Informationssysteme[5]) weiterentwickelt und ausgebaut, um den Prozess der systematischen Analyse von Patienten weltweit zu vereinfachen. Dabei

[1] http://tinnet.tinnitusresearch.net
[2] http://www.cost.eu
[3] http://www.tinnitusresearch.org
[4] http://www.uni-regensburg.de/medizin/psychiatrie-psychotherapie
[5] https://www.uni-ulm.de/en/in/iui-dbis/about-dbis/home

kommen standardisierte Instrumente, wie zum Beispiel psychoakustische Messungen oder Fragebögen zum Einsatz. Die Datenbank steht jedem interessierten Wissenschaftler, der am Projekt teilnehmen möchte, zur Verfügung. Dadurch wächst sie extrem schnell und umfasst aktuell mehr als 130000 Datensätze aus 16 verschiedenen Institutionen [22].

## 2.4 Projektstruktur

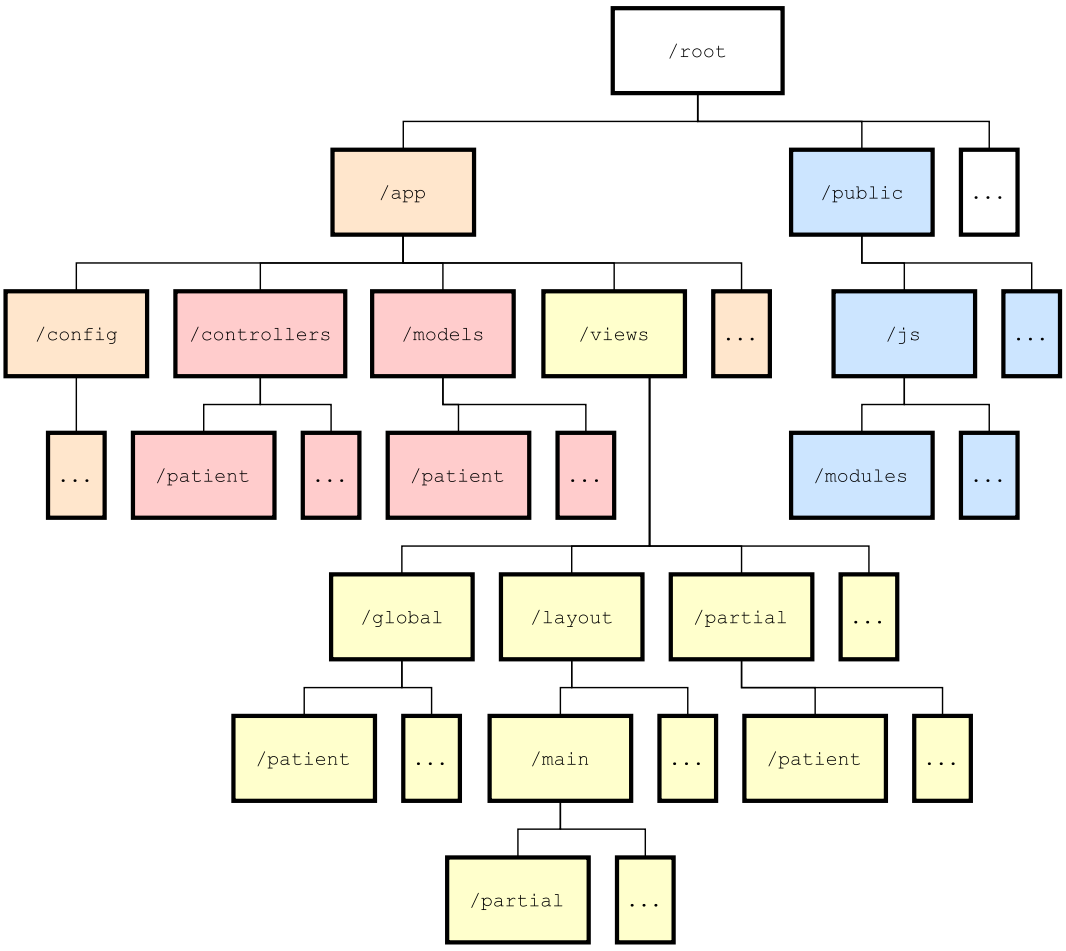



Abbildung 2.1: Relevanter Strukturauszug des Projektes

Die Projektstruktur der Applikation basiert auf Laravels Standardarchitektur. Alle für den Rahmen dieser Arbeit relevanten Strukturen sind in Abbildung 2.1 zur besseren Übersicht skizziert. Die Applikation selbst sitzt unter dem Pfad `/app` und gliedert sich in Model View

Controller (MVC) Strukturen. Die hauptsächliche Programmlogik (rot gefärbt) sitzt damit in den weiteren Unterordnern `../controllers` und `../models`. Die HTML-Templates (gelb gefärbt) des Projekts liegen vollständig im Ordner `../views`. Diese sind partiell nach Anwendungsfall organisiert und nutzen Laravels Blade-Templatesystem[6]. Alle Blades werden für die Verwendung von Laravel in reinen PHP-Code kompiliert und zwischengespeichert, bis sie erneut modifiziert werden. Der Pfad `/public` beinhaltet in weiteren Unterordnern alle nötigen statischen Ressourcen (blau gefärbt), wie Bilder, Cascading-Style-Sheets (CSS) und JavaScript-Dateien (JS), die zur Darstellung der Website herangezogen werden.

[6] `https://laravel.com/docs/5.4/blade`

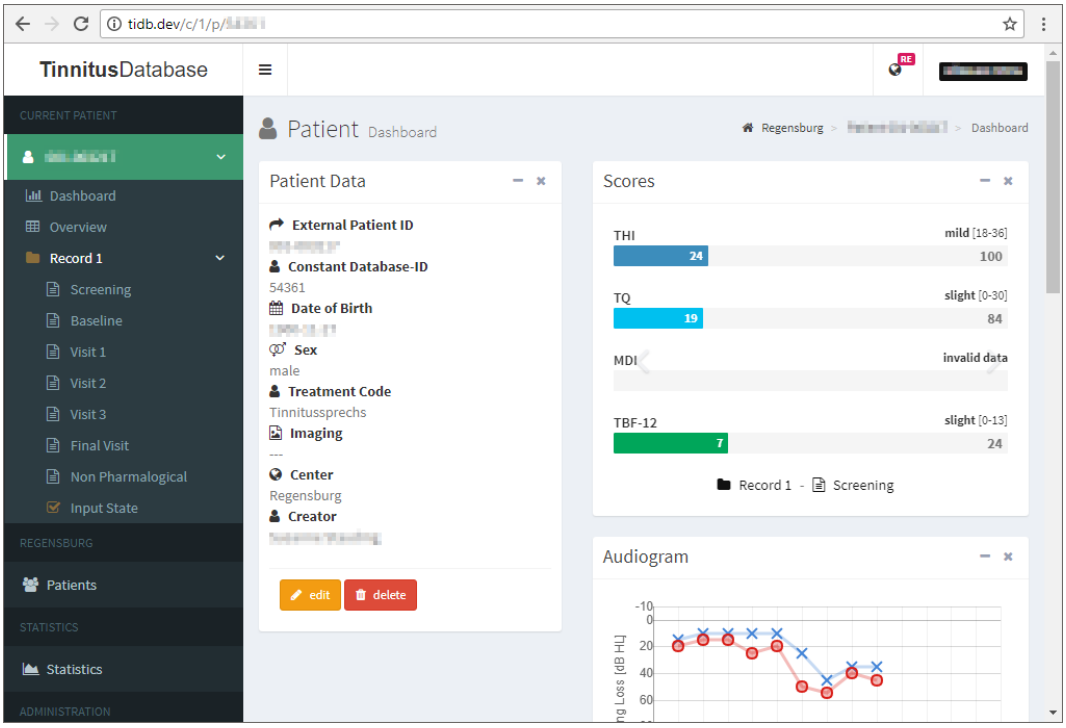



Abbildung 2.2: Screenshot der Übersicht eines Patienten

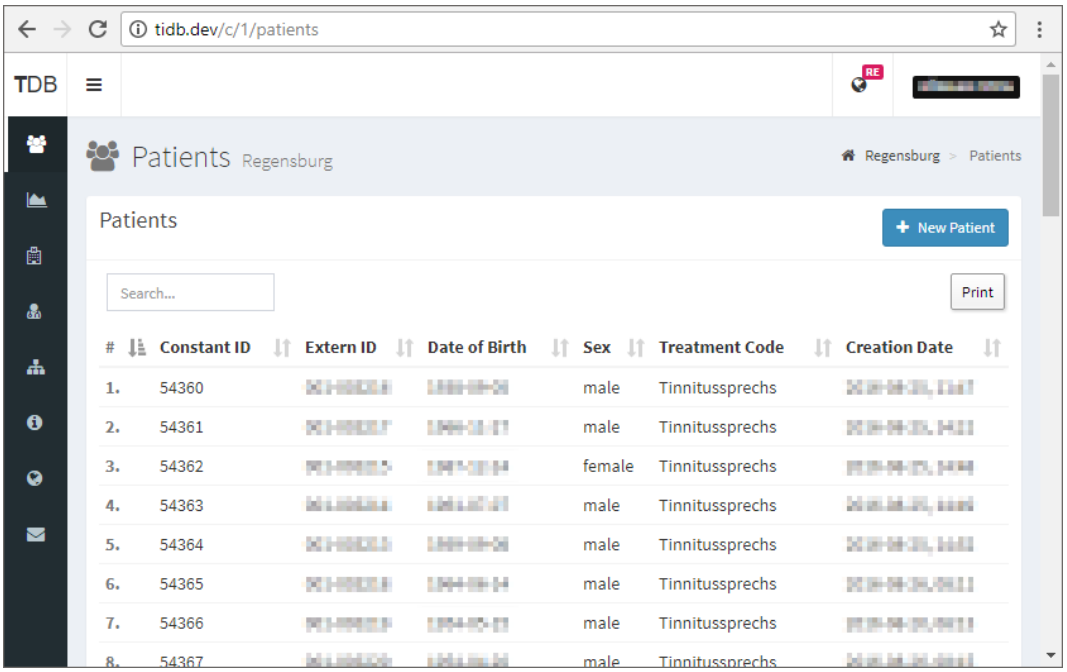



Abbildung 2.3: Screenshot der Patientenliste

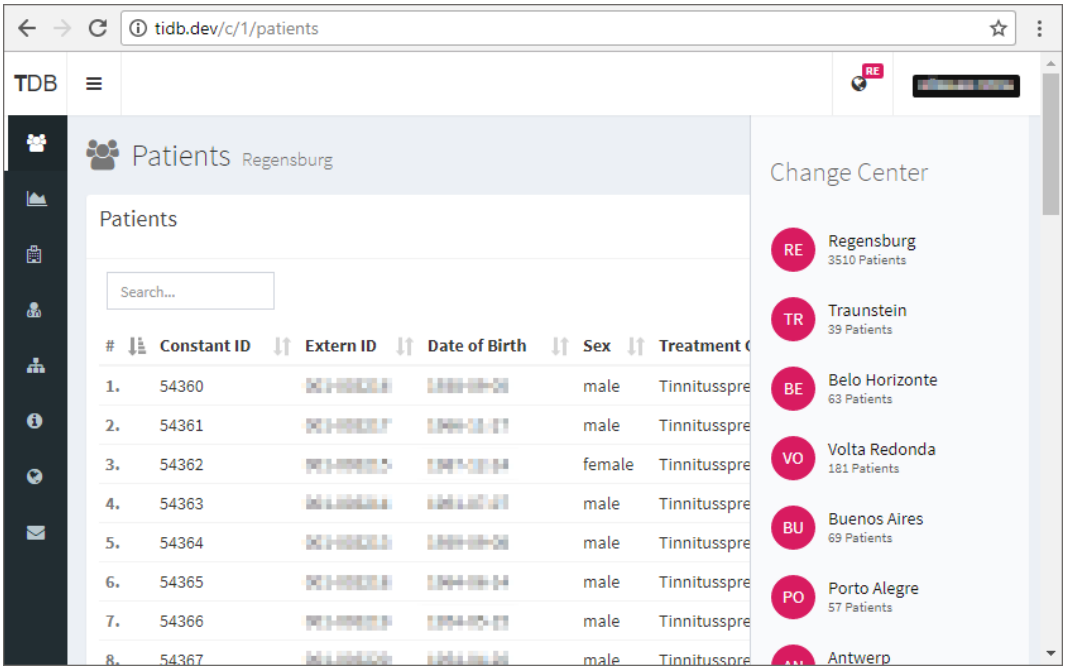



Abbildung 2.4: Screenshot der Patientenliste mit Superadmin-Sidebar (rechts)

# 3 Methodik

Das Kapitel der Methodik beschreibt zunächst einige wissenschaftliche Studien und Arbeiten, welche die Grundlage für die Anforderungen an die Patientenliste und das dort befindliche Suchmodul bilden. Dazu gehören Benutzerstudien, welche zum Beispiel die Zufriedenheit eines Nutzers in Abhängigkeit der Reaktionszeit eines Systems untersucht haben. Des Weiteren werden hier die Ergebnisse der Analyse, der Projektaufbau und die dazu verwendeten Werkzeuge dargelegt, welche ebenfalls im Kapitel der Implementierung eine wichtige Rolle spielen.

## 3.1 Verwandte Arbeiten und Grundbegriffe

### 3.1.1 Usability

Apps, Websites, nahezu jede Art von Software und sogar Hardware bestrebt es heutzutage, benutzerfreundlich zu sein. Doch was macht ein „freundliches“ System aus? Nielsen [15] hinterfragt schon den Begriff per se und kommt zum Schluss, dass die „Benutzerfreundlichkeit“ eine unangebrachte Beschreibung darstellt. Zum einen benötigt ein Benutzer kein System, welches freundlich zu ihm ist. Es ist wesentlich wichtiger, dass das System dem Nutzer nicht im Weg steht und ihn dabei unterstützt, seine Arbeit zu erledigen. Zum anderen haben verschiedene Nutzer verschiedene Bedürfnisse. So erscheint, laut Nielsen [15], ein solches System dem einen als „freundlich“ und einem anderen als lästig. Etabliert haben sich an dieser Stelle Begriffe wie *Human-Computer-Interaction* (HCI), *User-Interface-Design* (UID), *Ergonomie* und *Usability*.

Nielsen [15] unterteilt die Usability in fünf wesentliche Komponenten: *Erlernbarkeit*, *Effizienz*, *Einprägsamkeit*, *Fehlerrate* und *Zufriedenheit*. Für den Kontext dieser Arbeit sind zwei dieser Definitionen besonders hervorzuheben:

**Effizienz** Hat der Benutzer erlernt mit dem System umzugehen, muss dieses so effizient zu bedienen sein, dass ein hohes Maß an Produktivität gegeben ist. Die Effizienz lässt sich beschreiben als die Zeit, die ein Benutzer benötigt gewisse Aufgaben im System zu erledigen.

**Zufriedenheit** Um die Zufriedenheit eines Benutzers mit einem System zu gewährleisten, sollte ihm dieses subjektiv gefallen, angenehm zu bedienen und zu erlernen sein. Das System sollte dem Benutzer die Möglichkeit geben, all seine Aufgaben sicher abarbeiten zu können ohne dabei auf Widerstand, wie zum Beispiel fehlende Funktionalität, lange Ladezeiten oder andere frustrierende Ereignisse zu stoßen.

### 3.1.2 Reaktionszeit von Systemen

Die Reaktionszeit eines Computers ist definiert als die Zeit, die von einem Computer benötigt wird, um Ergebnisse auf einem Ausgabemedium darzustellen, welche von einem Benutzer durch die Initiierung einer Aktivität gefordert wurden [10]. Im Zeitalter des „World Wide Web“ (WWW) sind es nicht mehr nur einzelne Computersysteme, sondern ganze Netzwerke auf welche diese Definition zutrifft und daher eine große Rolle spielen. Das WWW ist zu einem der wichtigsten Medien zum Suchen und Empfangen von Informationen herangewachsen, sorgt aber trotz stetig steigender Software- und Hardwarekapazitäten weiterhin mit langen Ladezeiten für Frustration bei seinen Nutzern [6].

Aus diesem Grund hat Fui-Hoon Nah [6] eine Studie durchgeführt, welche untersucht, wie lange Anwender bereit sind auf das Herunterladen von Webinhalten zu warten, bevor sie den Vorgang abbrechen. Insbesondere wurde dabei der Unterschied der „tolerable waiting time“ (TWT), also die Zeit die ein Nutzer bereit ist zu warten, zwischen zwei Versuchsgruppen betrachtet. Dabei erhielt eine Gruppe während der Wartezeit mit Hilfe eines Ladeindikators Feedback vom System. Die zweite Gruppe musste darauf verzichten. Alle Teilnehmer wurden aufgefordert auf einer eigens angefertigten Website zehn Hyperlinks zu öffnen, welche zu geforderten Informationen führten. Die letzten drei Links verwiesen jeweils in eine

endlose Warteschleife. Fui-Hoon Nah [6] stellte fest, dass die Teilnehmer beim ersten nicht funktionierendem Hyperlink noch Geduld zeigten und im Durchschnitt 13 s warteten. Die Teilnehmer mit Feedback warteten im Schnitt sogar 38 s. Der Großteil der Abbrüche erfolgte im Intervall zwischen 5 – 8 s bzw. 15 – 46 s (mit Feedback). Für die folgenden zwei toten Links reduzierte sich die Geduld der Anwender drastisch, sodass in beiden Gruppen die meisten Teilnehmer schon nach 2 – 3 s aufgaben. Die TWT liegt also unter dem Strich bei ungefähr 2 s, bevor sich der Fokus des Anwenders auf andere Aktivitäten verschiebt.

Diese Ergebnisse decken sich mit anderen Studien. So haben Hoxmeier and DiCesare [10] in einer vom WWW unabhängigen Studie festgestellt, dass die Zufriedenheit mit einem System unter Benutzern am höchsten ist, wenn dieses innerhalb von 3 s reagiert und die erwartete Antwort ausgibt. Weiterhin zeigten die Teilnehmer bis zu einer Verzögerung von 10 s Toleranz und gaben bei der zugehörigen Umfrage mäßige Zufriedenheit an. Überschritt das System diese Reaktionszeit wurde es als zu langsam oder sogar als unbrauchbar eingestuft. Nielsen [15] schreibt ähnliches über die Reaktionszeit von Systemen und gibt ebenfalls das als vom Nutzer maximal akzeptierte Limit von 10 s an. Im Rahmen dieser Zeit kann der Anwender seinen Fokus auf eine Tätigkeit aufrecht erhalten. Benötigt das System länger, fängt der Nutzer an, sich anderen Aufgaben zu widmen. In diesem Fall empfiehlt Nielsen [15] die Implementierung von Feedback-Dialogen, die Auskunft über den Zeitpunkt der Fertigstellung der Anfrage geben. Bei Reaktionszeiten von bis zu 1 s ist laut Nielsen [15] keinerlei Feedback nötig, da der Gedankenfluss des Anwenders nicht unterbrochen wird und das Gefühl erhalten bleibt, direkt mit den Daten zu arbeiten.

## 3.2 Anforderungen

Hier werden die Anforderungen definiert, die an die Patientenliste und das dort befindliche Suchmodul gestellt werden um dem Nutzer eine optimale Usability zu gewährleisten. Dabei bleibt zunächst unbeachtet, ob diese Aspekte im Rahmen dieser Arbeit umgesetzt werden können.

### 3.2.1 Patientenliste

Die Patientenliste soll mit einem vom Anwender tolerierbaren Antwortzeitverhalten heruntergeladen und dargestellt werden können. Eine Reaktionszeit von weniger als 2 Sekunden ist hierbei anzustreben. Dieses Zeitlimit wird vom Nutzer bei einfachen Datenabfragen erwartet, um als flüssige Interaktion empfunden zu werden.

### 3.2.2 Suchmodul

Das Suchmodul soll dem Anwender so schnell wie möglich zur Verfügung stehen und funktionell neben einer Volltextsuche aller relevanten Datensätze über erweitere Filterfunktionen innerhalb der dargestellten Tabelle verfügen um gezieltere und komplexere Suchanfragen stellen zu können.

## 3.3 Projektaufbau

### 3.3.1 Verwendete Werkzeuge

In diesem Abschnitt wird kurz auf die Technologien eingegangen, die dazu verwendet wurden das bereits bestehende Datenbankprojekt zu entwickeln, aber auch auf die Werkzeuge, die bei der Analyse und in der folgenden Implementierung zum Einsatz kamen. Dazu gehören die verwendete Entwicklungsumgebung, Debug- und Analysewerkzeuge, sowie die für das Projekt benötigten Laufzeitumgebungen und Frameworks.

**Laravel** [1] ist ein Open-Source-PHP-Framework zur Entwicklung von RESTful-Webservices nach der MVC-Architektur. Für die Verwaltung der Abhängigkeiten kommt Composer[2] zum Einsatz. Die Tinnitus-Database wird derzeit unter der Version 4.2.22 betrieben.

Um eine bessere Einsicht in die Web-Applikation zu erhalten und eine detaillierte Analyse durchführen zu können, verwende ich zusätzlich die PHP Debug Bar[3], welche unter einem eigenen Projekt (Laravel Debugbar 1.8[4]) für das Laravel-Framework angepasst wurde. Sie erlaubt es zum Beispiel SQL-Abfragen, das Routing und die für eine View verwendeten Partialansichten nach Ausführung zu inspizieren.

**PHP** [5] ist eine serverseitige Skriptsprache, welche mit Version 5.6 die Basis des Laravel 4.2 Frameworks bildet und daher in Produktions- sowie Entwicklungslaufzeitumgebung des Projektes genutzt wird. Die Veröffentlichung von PHP 7.0[6] im Jahr 2015 hat unter anderem Verbesserungen der Performanz mit sich gebracht, auf welche ich näher in Kapitel 4 eingehen werde.

**PHPStorm** [7] wird von JetBrains entwickelt und ist eine auf der hauseigenen IntelliJ IDEA Plattform basierende Entwicklungsumgebung. Durch ein Pluginsystem bietet sie die Möglichkeit, mit verschiedensten Frameworks, Bibliotheken oder Debug-Werkzeugen zu arbeiten. Zum Beispiel bietet die Umgebung eine direkte Kopplung an Vagrant und eine Schnittstelle zur PHP Extension Xdebug[8] für erweiterte Debugfunktionalitäten einer Applikation.

**Vagrant** [9] ist ein Verwaltungswerkzeug für virtuelle Maschinen (VM), das dafür verwendet wird, das Zielsystem, auf welchem eine Webapplikation am Ende laufen soll, möglichst genau nachzuempfinden. Für die Arbeiten an diesem Projekt verwende ich die Virtualisierungssoftware VirtualBox von Oracle mit einer durch PuPHPet[10] erstellten

[1] `https://laravel.com/docs/4.2`
[2] `https://getcomposer.org`
[3] `http://phpdebugbar.com`
[4] `https://github.com/barryvdh/laravel-debugbar/tree/1.8`
[5] `http://php.net`
[6] `http://php.net/archive/2015.php#id2015-12-03-1`
[7] `https://www.jetbrains.com/phpstorm`
[8] `https://xdebug.org`
[9] `https://www.vagrantup.com`
[10] `https://puphpet.com`

Konfiguration. Die eigens für Laravel vorgefertigte VirtualBox Homestead[11] habe ich nicht verwendet, da diese gegenüber PuPHPet weniger Konfigurationsspielraum bietet.

**JMeter** [12] ist ein Werkzeug zur Analyse von Webapplikationen, welches von der Apache Software Foundation[13] verwaltet und entwickelt wird. Die Software erlaubt es, Skripte zu schreiben, welche es ermöglichen, ein System mit simulierten Lasten auf Schwächen bezüglich der Performanz zu testen. Die Ergebnisse können dann auf verschiedenste Art visualisiert werden.

**Chrome DevTools** sind die im Browser Google Chrome[14] integrierten Entwicklerwerkzeuge zur Live-Analyse der von einem Webserver übertragenen Antworten, sowie des dargestellten HTML Inhalts und der anhängenden Ressourcen bzw. dynamischen Anteile wie JavaScript-Funktionen. Sie ermöglichen damit zum Beispiel das direkte Modifizieren des „Document Object Models“ (DOM) und das Aufzeichnen und Messen der Netzwerkübertragungen.

### 3.3.2 Setup

Das Setup beschreibt kurz, wie die Webapplikation lokal aufgesetzt wurde um Reproduzierbarkeit zu gewährleisten. Verwendet wurden dazu die im vorherigen Abschnitt 3.3.1 genannten Technologien, auf deren Konfiguration hier eingegangen wird. Im Zuge der Inbetriebnahme der Applikation, welche über ein Git Respository[15] zur Verfügung gestellt wurde, kam es zu einigen Schwierigkeiten, wie z.B. für die Log-Erstellung fehlender Code oder fehlende Ordnerstrukturen. Diese hätten sich durch eine ausführlichere Dokumentation, aber vor Allem durch ein konsequentes Benutzen von Git sowie den von Laravel bereitgestellten Funktionen und Werkzeugen vermeiden lassen können.

Um mir ein einfaches Entwickeln über mehrere Systeme hinweg zu ermöglichen, aber auch um den Installationsvorgang von den genannten Schwierigkeiten zu befreien, habe ich meinen Fork des ursprünglichen Repositories derart angepasst, dass dies mit einem Minimalaufwand möglich ist. Dazu nötig waren die Überarbeitung und Vervollständigung einer

[11] https://laravel.com/docs/4.2/homestead
[12] https://jmeter.apache.org
[13] http://www.apache.org/
[14] https://www.google.com/chrome/browser
[15] https://bitbucket.org/mstach/tinnitus-research-db

sinnvollen *.gitignore*[16], sowie das Hinzufügen benötigter *.gitkeep*[17] Dateien, um Ordnerstrukturen, die von der Applikation initial gebraucht werden, aufrecht zu erhalten. Des Weiteren wurden alle Dateien, welche automatisch erzeugt werden, aus dem Repository entfernt, um Konflikte mit beispielsweise Composer-Dependencies beim Aufsetzen zu vermeiden. Die Composerkonfiguration wurde um fehlende und neue Pakete, wie die Laravel-Debugbar, ergänzt. Für eine einheitliche und konsistente Codeformatierung zwischen verschiedenen Editoren und Entwicklern habe ich das Projekt um eine *.editorconfig*[18] erweitert, welche es erlaubt grundlegende Parameter, wie zum Beispiel Tabgröße, Zeichenkodierung und Zeilenlänge zu definieren. Diese werden dann global und ungeachtet der Voreinstellung des Editors automatisch übernommen.

**1. Installation von VirtualBox und Vagrant**
Es kann jeweils die neueste Version verwendet werden. Eine weitere Konfiguration ist nicht nötig. Unter Windows ist jedoch darauf zu achten, dass die Virtualisierungstechnologie Hyper-V deaktiviert ist, da diese nicht parallel mit VirtualBox verwendet werden kann.

**2. Konfiguration und Initialisierung der Vagrant Box**
Zur Konfiguration der VM wurde aufgrund der Einfachheit und Flexibilität PuPHPet verwendet. Die webbasierte Applikation ermöglicht es, Schritt für Schritt eine von Vagrant nutzbare Konfiguration für die VM zu erstellen und herunter zu laden. Über die enthaltene *config.yaml* lässt sich die Konfiguration im Nachhinein modifizieren und an die eigenen Umgebungs- und Projektpfade anpassen. Die für diese Arbeit verwendeten Konfigurationen sind im digitalen Anhang zu finden.

Die vorgegebene *config.yaml* kann mit PuPHPet geöffnet und mit den korrekten Pfaden zum Projektverzeichnis aktualisiert werden. Nachdem alle Konfigurationsdateien in das Ausführungsverzeichnis der Box extrahiert wurden, muss in diesem über eine Shell der Befehl `vagrant up` ausgeführt werden. Die Vagrant Box wird nun initialisiert und kann genutzt werden. Stoppen lässt sich die VM mit dem Befehl `vagrant halt`, gelöscht wird sie über `vagrant destroy` und ein SSH Tunnel lässt sich mit `vagrant ssh` aufbauen.

[16] `https://git-scm.com/docs/gitignore`
[17] Eine inoffizielle Konvention, um leere Verzeichnisse in einem Git Repository aufnehmen zu können.
[18] `http://editorconfig.org`

**3. Aufbau der Projektabhängigkeiten**

Um die mit Composer verwalteten Pakete des Projektes zu installieren, muss sich mit `vagrant ssh` auf die VM eingewählt und ins Wurzelverzeichnis des Projektes navigiert werden. Hier ist der Befehl `composer install` auszuführen. Nach Fertigstellung des Prozesses, ist die Konfiguration der VM abgeschlossen und das Projekt eingerichtet.

**4. Letzte Schritte**

Damit auf die Website in der VM zugegriffen werden kann, muss noch die Hosts-Datei des eigenen Betriebssystems angepasst und die Datenbank des Servers mit einem gültigen Datenbankdump gespeist werden. Die Hosts-Datei benötigt den folgenden Eintrag, um die in der *config.yaml* angegebenen IP-Adresse der Domain tidb.dev zuordnen zu können:

```
192.168.10.10   tidb.dev www.tidb.dev
```

Die Einspielung der Datenbank ist zum Beispiel über das schon synchronisierte Projektverzeichnis mittels `mysql` per Shell möglich.

## 3.4 Analyse

### 3.4.1 Patientenliste

Wie in Abbildung 2.3 zu sehen ist, werden die Patienten für jedes teilnehmende Zentrum tabellarisch separat gelistet. Dabei werden alle zugehörigen Datensätze auf einer Seite dargestellt. Die Tabelle enthält auf sieben Spalten verteilt Informationen zur Identifikation, Geschlecht, Geburtstag, Behandlung und Erstellung der jeweiligen Patienten. Der URL-Pfad zur Darstellung aller Regensburger Patienten lautet `<domain>/c/{id}/patients`, wobei der Parameter *{id}* den Code des Zentrums angibt. Die Ansicht wird aus zehn Blade-Templates zusammengesetzt und stellt 22 SQL-Anfragen, falls keine Queries im Cache vorliegen. Das Laden dieser Liste, mit derzeit rund 3600 Einträgen bringt eine deutlich spürbare Verzögerung mit sich, welche für den Anwender nicht mit visuellem Feedback kenntlich gemacht wird.

Für die Betrachtung der Netzwerkaktivitäten wurden die Übertragungsgeschwindigkeiten der Analysewerkzeuge auf 256 kB/s[19] gedrosselt und das Caching deaktiviert. Das genutzte Benutzerkonto besitzt die Rolle des Superadmin mit maximalen Zugriffsrechten. Dies ist notwendig, um für Produktions- und Lokalumgebung gleiche Bedingungen zu schaffen und somit vergleichbare Ergebnisse aufzeichnen zu können. Weiterhin wurden für die Reproduzierbarkeit alle Tests und Arbeiten im Rahmen dieser Arbeit auf dem selben Endgerät durchgeführt. Die zwei darauf verwendeten lokalen virtuellen Maschinen sind identisch initialisiert und unterscheiden sich lediglich in der installierten Version der PHP Laufzeitumgebung. Zum Einsatz kamen dabei die Version 5.6.30[20] und 7.0.17[21].

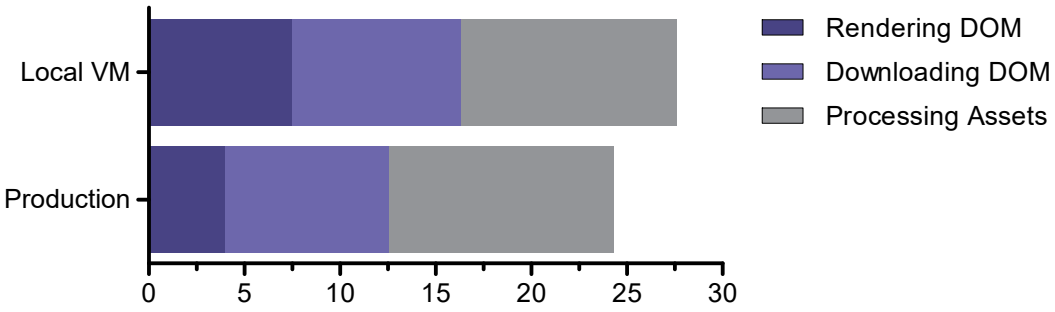



Abbildung 3.1: Netzwerkanalyse (Mittelwerte in s)

Die Daten der Netzwerkaktivitäten in der Produktionsumgebung zeigen, dass das Laden und Rendern der Seite inklusive aller Assets unter den genannten Bedingungen im Schnitt 24,2 s benötigt. Von dieser Zeit werden durchschnittlich 12,6 s dafür benötigt, um das DOM zu generieren und herunter zu laden. Ermittelt wurden die Daten zu nächst qualitativ mit Hilfe der Google Chrome DevTools und nochmals durch ein JMeter Anfrageskript (siehe [digitaler] Anhang) quantitativ bestätigt, welches die Patientenliste 20 mal nacheinander aufgerufen hat. Dabei wurde die Zeit, die die Applikation zum Generieren des DOM benötigt über die „Time To First Byte“ (TTFB), also die Zeit, welche bis zum ersten Byte der Antwort vom Server vergeht, ausgelesen. Die weitere Analyse und folgenden Arbeiten werden von einem lokalen Abbild der Website in einer VM erfolgen, was die Anwendung von benötigten

[19] Eine exakte Festlegung des Limits über alle Werkzeuge hinweg ist aufgrund von technischen Einschränkungen nicht möglich; es ist daher zu minimalen Abweichungen gekommen.
[20] `http://www.php.net/ChangeLog-5.php#5.6.30`
[21] `http://www.php.net/ChangeLog-7.php#7.0.17`

Debug-Werkzeugen ermöglicht. Die oben genannten Antwortzeiten für die lokale Umgebung betragen ca. 27,6 s und respektive 16,3 s für die Generierung und Download des DOM.

Abbildung 3.1 zeigt im Detail aufgeschlüsselt, dass von den 27,6 s gesamter Ladezeit ungefähr 7,5 s vom Server in Anspruch genommen werden, um das DOM zu generieren. Die restliche Zeit dient der Übertragung und der klientseitigen Verarbeitung aller Ressourcen. Diese bestehen, neben dem DOM, aus zehn JS-Dateien, acht CSS-Datein und fünf eingebetteten Schriftarten. Insgesamt müssen dazu 2,8 MByte transferiert werden, von welchen 2,3 MByte zur Last des DOM fallen und 0,5 MByte für die genannten Ressourcen benötigt werden.

```
1  select * from `patients`
2      inner join `id_translation`
3      on `patients`.`patient_id` = `id_translation`.`patient_id`
4      where `id_translation`.`center_id` = '1'
5      and `patients`.`deleted_at` is null
6
7  select * from `id_translation`
8      where `id_translation`.`deleted_at` is null
9      and `id_translation`.`patient_id`
10     in (/*! all patient ids */)
11
12 select * from `patient_records`
13     where `patient_records`.`deleted_at` is null
14     and `patient_records`.`patient_id`
15     in (/*! all patient ids */)
```

Listing 3.1: Auszug der Datenbankabfragen

Ein Blick auf die SQL-Anfragen, die bei der Erstellung der Patientenliste ausgeführt werden, zeigt drei relevante Queries, welche im obigen Codeauszug 3.1 zu sehen sind. Besonders auffällig sind hierbei die unteren zwei Anfragen. Sie beinhalten jeweils ein Tupel mit allen knapp 4600 Patienten-IDs und beanspruchen damit ca. 97,5% der durchschnittlichen 3,25 s Rechenzeit aller SQL-Anfragen unter PHP 5.6.

| | Query 1 | Query 2 & 3 | Alle Queries |
|---|---|---|---|
| Applikation PHP 5.6 | 0,040 | 3,170 | 3,250 |
| Applikation PHP 7.0 | 0,016 | 0,576 | 0,613 |
| Nativ | 0,003 | 0,011 | 0,036 |

Tabelle 3.1: Ausführungszeiten der Datenbankabfragen (Mittelwerte in s)

Die dazu gehörigen Applikations-Mittelwerte in Tabelle 3.1 wurden mit Hilfe der Laravel Debugbar berechnet und stellen den Mittelwert aus einem Datensatz mit zehn nacheinander erfolgten Aufrufen dar. Produktions- sowie Lokalumgebung verwenden PHP 5.6 und damit die langsamste Laufzeitumgebung laut Analyse. Der direkte Vergleich zur Version 7.0 und besonders zu den mit HeidiSQL[22] ermittelten nativen Ausführungszeiten zeigt deutlich, dass die Mächtigkeit der Queries nicht allein das Problem der schwachen Leistung bildet. Die Zeiten heben einen deutlichen Mehraufwand der SQL-Query-Verarbeitung in PHP 5.6 hervor. Eine weitere Auffälligkeit aller Queries ist das durchgehend verwendete `select *` Statement. Es werden also für jede Anfrage alle Spalten der Tabellen zurück gegeben, was einen unnötig hohen Datenverarbeitungsaufwand darstellt und die Übersichtlichkeit sowie das Profiling der Queries erschwert [28]. Falls nicht alle Daten benötigt werden, sollten immer nur die für die Produktionsumgebung relevanten Spalten der Tabelle zurückgegeben werden [12].

Die zur tabellarischen Darstellung der Patientendaten zuständige SQL-Query ist die Erste im Auszug 3.1, startend in Zeile 1. Sie wird durch ein Callback in `/app/bindings.php:154` ausgeführt und fragt, aufgrund der Rolle des Superadmin, alle existierenden Patienten des jeweiligen Zentrums ab. Die Aufgabe der beiden letzteren Queries, startend in Zeile 7 und 12 des Codeauszugs 3.1, erschließt sich nicht auf Anhieb. Sie werden ebenfalls über einen Callback aus der Datei `/app/bindings.php:87` heraus ausgeführt und sollen laut Quellcodekommentierung alle Patientendaten eines Zentrums zurückgeben. Aufgrund der extrem hohen Ausführungszeiten der Queries und der damit verbundenen gesamten Leistungsminderung des Systems habe ich deren Einfluss auf die Seite weiter untersucht. Aufgerufen werden sie ausschließlich aus dem Blade `../main-layout.blade.php:187`, wo die Daten an das dort eingebundene Blade `../sidebar-superadmin.blade.php` übergeben werden. Genutzt werden sie hier jedoch nicht. Das Blade enthält den Code für eine einblendbare Liste aller auszuwählenden Zentren, sowie deren vorliegende Patienten-

[22] `https://www.heidisql.com`

anzahl. Sie ist nur für Superadmins zugänglich und in Abbildung 2.4 zu sehen. Um sicher zu gehen, dass tatsächlich keine Verwendung dieser Daten in der Ansicht vorliegt, habe ich den Quellcode des jeweiligen DOM, mit und ohne Ausführung der Queries, direkt miteinander verglichen. Die durch Laravel erzeugten temporären vorkompilierten Templates wurden vorher jeweils gelöscht. Zwischen den Versionen konnten keine Unterschiede festgestellt werden. Dies legt die Vermutung nahe, dass dieser Funktionsaufruf mit seinen anhängenden Datenbankabfragen bei früheren Überarbeitungen des Systems ersetzt, jedoch nicht entfernt wurde.

Ein weiteres Indiz auf unsauberen Code bzw. unvollständiges Refactoring, dass diese Vermutung bekräftigt, zeigt sich nochmals in der Datei `/app/bindings.php:272`. Hier wird die zugehörige Datenbankabfrage direkt zwei Mal hintereinander ausgeführt. Des Weiteren konnte ich bei Performanztests unter PHP 7.0 bei einem Blade, welches ein Teil der Patienten-Detailansicht (Abbildung 2.2) ist, Fehler bei der Kompilierung feststellen. Diese wurden durch fehlerhafte Kommentierung im Code verursacht. Die genannten Probleme hätten sich durch Unit-Tests, die im aktuellen Projekt nicht existieren, vermeiden lassen, oder zumindest meine Analyse unterstützen und darauf basierende Vermutungen untermauern können.

### 3.4.2 Suchmodul

Das implementierte Suchmodul ist vollständig über JavaScript realisiert und filtert die Daten in der gerenderten Tabelle auf Seite des Klienten. Verwendet wird dazu die DataTables-Bibliothek[23] in der Version 1.9 – die derzeit aktuelle Version ist 1.10. Diese durchsucht die Datensätze der Patienten auf Basis des DOM. Es können daher nur die Daten gefiltert werden, welche bereits an den Browser übertragen wurden. Die Performanz der Suche ist also abhängig von der Leistung und Kompatibilität des genutzten Endgeräts. Eine asynchrone API[24] ist ebenfalls nicht eingebunden, wodurch das Modul nicht in der Lage ist, dem Nutzer bei Eingabe einer Suchanfrage dynamische Vorschläge nachzuladen. Durch die lange Antwortzeit, die das System benötigt, um das DOM zu erstellen, werden die benötigten JavaScript-Bibliotheken und Module entsprechend spät geladen und angewandt. Dies hat zur Folge, dass das Eingabefeld der Suche und weitere Funktionalitäten der DataTables-

[23] `https://legacy.datatables.net`
[24] Application Programming Interface

Bibliothek, beispielsweise die Sortierung, erst zum Ende des gesamten Ladeprozesses in das DOM gerendert werden und zur Verfügung stehen.

Für den Anwender ist es auf den ersten Blick nicht ersichtlich, welche Filterfunktionalitäten das Modul zur Verfügung stellt oder nach welchen Kriterien gesucht werden kann. Ein Dialog für eine erweiterte Suchfunktionalität oder vordefinierte Kriterien, wie zum Beispiel eine Liste aller Behandlungscodes oder eine Auswahloption des Geschlechts werden nicht angeboten.

# 4 Ausgewählte Implementierungsaspekte

In diesem Kapitel wird die im vorherigen Teil erstellte Analyse aufgegriffen, auf deren Basis Lösungen vorgeschlagen werden, um die gestellten Anforderungen an die Patientenliste und das Suchmodul erfüllen zu können. Analyseergebnisse außerhalb des Rahmens der Anforderungsliste werden ebenfalls kurz mit Verbesserungsvorschlägen erläutert. Danach werden die für das System getroffenen Entwicklungsentscheidungen erläutert und ausgewählte Vorgehensweisen zur Problemlösung genauer beschrieben.

## 4.1 Lösungsvorschläge

Die im Kapitel der Methodik beschriebenen Studien und Arbeiten, sowie natürlich die Ergebnisse der Analyse, bilden nun die Grundlage für Lösungsvorschläge, um die Zielsetzung dieser Arbeit erfüllen zu können. Im Laufe der Analyse hat sich die schlechte Performanz des vollständigen Ladeprozesses von Patientenlisten mit einer hohen Anzahl an Einträgen als besonderes Problem herauskristallisiert. Diese werden vollständig vom Server in das DOM gerendert. Weitere Manipulation der Daten findet auf Seite des Klienten über JavaScript-Module statt. Unter dem Strich ist mit diesem Konzept keine stabile Performanz auf unterschiedlichen Endgeräten zu gewährleisten, da diese stark abhängig von verwendeter Hard- und Software ist.

Dies wurde durch einen einfachen Test auf einem leistungsstarken Computer bestätigt, welcher selbst die nicht optimierte Patientenliste mit PHP 7.0 in unter 2 s vollständig generieren und laden konnte. Dieser Test steht im offensichtlichen Widerspruch zu den zuvor gewonnenen Erkenntnissen und berichteten Erfahrungen von Endnutzern des Systems. Das Performanzproblem kann demnach durch die Verwendung von moderner leistungsfähiger Hardware mit einem ansehnlichen Ergebnis gehemmt werden. Diese Lösung kommt jedoch

mit einem erheblichen Preis einher und ist damit langfristig nicht tragbar. Insbesondere lässt sich dieses Vorgehen auf Seite des Klienten mit entsprechenden Endgeräten kaum durchsetzen. Aus diesem Grund wird dieser Ansatz im Rahmen dieser Arbeit weder weiterverfolgt noch als Lösungsmöglichkeit vorgeschlagen.

### 4.1.1 Patientenliste

Das serverseitige Rendern des DOM benötigt in der lokalen VM durchschnittlich 7,5 s allein. Sogar die Produktionsumgebung nimmt dafür laut Analyse ca. 4 s in Anspruch. Diese Werte (siehe Abbildung 3.1) sind unabhängig von Netzwerkgeschwindigkeit und Leistung des Endgeräts, was sie für Endnutzer zu einer Konstanten im gesamten Ladeprozess macht. Der restliche Ladevorgang, bestehend aus Datenübertragung und lokalem Rendering, ist jedoch von den genannten beiden Faktoren abhängig und lässt sich damit nur partiell als Entwickler, hauptsächlich durch die Verringerung der zu übertragenden Datenmenge, beeinflussen.

Die aktuelle Implementierung enthält, wie in Abschnitt 3.4.1 festgestellt, toten und duplizierten Programmcode, welcher zunächst gesucht und entfernt werden sollte. In diesem speziellen Fall kann, wie in der Analyse dokumentiert, durch das Entfernen des Codes, welcher dafür zuständig ist überflüssige SQL-Queries auszuführen, eine enorme Zeitersparnis entstehen. Die in Tabelle 3.1 gemessenen Werte geben Auskunft darüber, in welchem Rahmen sich hier Ausführungszeit beim Aufrufen der Patientenliste sparen lässt. Unter der aktuellen Laufzeitumgebung mit PHP 5.6 reduziert sich diese in der lokalen Umgebung um ca. 3,2 s. Die Applikation benötigt, nach bisheriger Messungen, also nur noch knapp 4,3 s für die identische Arbeit, welche sich noch weiter reduzieren lässt, wenn die PHP Laufzeitumgebung des Systems auf die aktuellste, von Laravel 4.2 unterstützte, Version aktualisiert wird. Details dazu sind in Abschnitt 4.1.3 zu finden.

Weitere Optimierungen bezüglich der Geschwindigkeit lassen sich über die zu generierende Datenmenge erzielen. In der Analyse wurde gezeigt, dass das DOM mit einer Tabelle von ca. 3600 Einträgen die Größe von 2,3 MByte erreicht. Diese Datei muss nicht nur vom Server generiert werden, sondern auch übertragen und schließlich auf dem Klienten verarbeitet werden. Um diese Last auf allen Ebenen reduzieren zu können, bietet sich eine Aufteilung der Datensätze an. Benötigte Daten können so bei Bedarf über eine Paginierung nachgeladen werden. Mit solch einem System lässt sich die Performanz der Applikation in Anbetracht

der stetig wachsenden Datenbank konstant halten, wodurch vor allem Netzwerkkapazität und Endgerät entlastet werden. Diese beiden variablen Faktoren sind als Entwickler und Betreiber einer Webapplikation nicht zu beeinflussen und sollten daher effizient genutzt werden. Geringe Datenmengen lassen sich schneller erzeugen und transferieren und können insbesondere auf schwächeren mobilen Geräten schneller verarbeitet und angezeigt werden. Um im Kontext dessen noch bessere Ladegeschwindigkeiten zu erreichen, kann abhängig von der Art des Endgerätes das initiale Intervall der Paginierung und damit die zu übertragende Datenmenge pro Seite angepasst werden. So werden mobile Geräte beispielsweise zunächst mit einer kleinen Anzahl von 50 und leistungsstärkere Desktop-Computer mit 200 Patienten pro Seite bedient.

Die Realisierung einer Paginierung kann auf zwei grundlegende Arten geschehen. Zum einen gibt es die Möglichkeit der serverseitigen und zum anderen der klientseitigen Implementierung. Auf Basis der klientseitigen Paginierung gibt es verschiedene Realisierungsmöglichkeiten, welche alle auf JavaScript aufbauen. Eine Variante wäre es, eine kosmetische Paginierung zu verwenden. Sie bietet schnelles subsequentes Laden folgender Daten, eignet sich jedoch nur für kleine Datensätze, da hier, wie in der aktuellen Implementierung, ein initiales Laden aller Datensätze nötig ist. Diesem Problem kann mit der Verwendung von Ajax[1] entgegengewirkt werden. Das asynchrone Nachladen der Daten über JavaScript, wäre, bezüglich der zu übertragenden Datenmenge pro Seitenaufruf, dem Aufwand einer serverseitigen Paginierung geringfügig überlegen. Es wird verhindert, dass das Endgerät die vollständige Seite neu laden und aufbauen muss. Durch die effiziente Nutzung von Caches für die statischen Ressourcen und Komprimierung der zu übertragenden Daten, beispielsweise mit gzip[2], wird dieser Vorteil gegenüber dem serverseitigen Ansatz jedoch relativiert und stellt diesbezüglich nur eine sehr geringe weitere Optimierung dar. Zu Beachten ist außerdem die Möglichkeit der Deaktivierung von JavaScript auf einem Klienten. Dies unterbindet jegliche mit JavaScript implementierte Funktionalität, aber vor allem das Ajax basierte Laden von Inhalten und macht es unmöglich Inhalte an den Nutzer zu übertragen.

Mit einem Lösungsansatz auf Seite des Klienten kann auf eine JS-Bibliothek zurückgegriffen werden, welche die benötigten Funktionen schon zur Verfügung stellt. Auf Seite des Servers ist dann lediglich eine Programmierschnittstelle einzurichten, welche dem Front-End die

[1] Asynchronous JavaScript and XML
[2] `http://www.gzip.org`

entsprechenden Daten, zum Beispiel als JSON[3], liefert. Um dies zu erzielen, kann die momentan zur Filterung und Sortierung genutzte DataTables-Bibliothek weiter verwendet werden. Sie wird offiziell als Plugin des auf Bootstrap[4] und jQuery[5] basierendem AdminLTE[6] Templates angeboten. Eine weitere Alternative ist das ähnliche Bootgrid-Plugin[7], welches auf den gleichen Technologien basiert und damit in die bestehende Umgebung eingebunden werden kann.

Weiterhin wird dem Anwender bei einer Paginierung mit einer großen Anzahl an Patienten die Möglichkeit genommen, diese auf einer Seite darzustellen. Dies bricht die momentan implementierte Funktionsweise der Datenaufbereitung für den Druck, bei der lediglich eine entsprechende CSS-Datei geladen werden muss. Um diese Funktion und die Möglichkeit einen Ausdruck der vollständigen Patientenliste weiterhin zu gewährleisten, muss ein neuer Mechanismus eingeführt werden. Dieser ist zu ermöglichen, indem der Server beispielsweise eine neue Ansicht, in einem für den Druck optimierten Layout, generiert. Aufgrund der Neugeneration der Liste, kann dem Nutzer die Möglichkeit zu weiteren Modifikationen der anzuzeigenden Daten gegeben werden. So kann dem Nutzer zum Beispiel stetig die Option geboten werden, ob nur bereits gefilterten oder alle Patienten der jeweiligen Kategorie ausgegeben werden sollen. Weitere Optionen zum Ausblenden oder besonderen Hervorheben von bestimmten Spalten der Tabelle wären ebenso ergänzende Funktionalitäten. Die Ausgabe wird dann in einem neuen Fenster oder Tab dargestellt, kann nochmals geprüft werden und schließlich über die vom Browser bereitgestellte Printfunktion gedruckt werden. Die ursprüngliche Ansicht der Liste bleibt damit separat erhalten und kann bei einem längeren Ladevorgang des Drucklayouts weiter genutzt werden.

### 4.1.2 Suchmodul

Wie auch bei der Paginierung der Patientenliste gibt es bei der Neukonzeption des Suchmoduls die Möglichkeit einer klientseitigen, vollständig auf JavaScript basierenden Implementierung. Eine solche Realisierung mit einer der in Abschnitt 4.1.1 erwähnten JS-Bibliotheken kann jedoch auf Seite des Klienten deaktiviert werden, zum Beispiel durch eine explizite

[3] http://www.json.org
[4] http://getbootstrap.com
[5] http://jquery.com
[6] https://adminlte.io
[7] http://www.jquery-bootgrid.com

Blockierung von JavaScript im Browser, und steht damit nicht mehr zur Verfügung. Einfache Filterfunktionalitäten auf dem DOM, aber auch Ajax-Anfragen zur Suche an der Server wären damit unterbunden. Diese Problematik ist mit einer klassischen Suchmaske, welche ihre Anfragen über ein HTML-Formular und HTTP-Methoden an den Server übergibt, nicht gegeben. Die Funktionalität einer solchen Maske steht unter nahezu allen Umständen zur Verfügung. Um dem Nutzer zusätzlichen Komfort zu bieten, gibt es die Möglichkeit auf eine Ajax basierte Erweiterung der Maske, welche bei Eingabe der Anfrage im Hintergrund bereits nach Ergebnissen sucht und diese dem Anwender direkt vorschlägt. Verwendet werden kann dazu beispielsweise die auf der Suggestion-Engine Bloodhound[8] basierende JS-Bibliothek typeahead.js[9]. Bei Eingabe einer einzigartigen Patienten-ID kann so dem Anwender sofort die gewünschte Referenz mit zur Quelle führendem Hyperlink angezeigt werden. Ist bei dieser Kombination von Technologien keine Ausführung von JavaScript auf dem Klienten möglich, kann der Anwender das Suchmodul trotzdem weiter verwenden und ist in der Lage Suchergebnisse zu erhalten.

Darüber hinaus gilt es, das momentane Suchmodul um seine Filterfunktion, der im DOM bestehenden Daten, zu erweitern. Durch die neue Funktionalität von expliziten Datenbankanfragen für Suchanfragen besteht die Möglichkeit, nach weiteren in der Datenbank liegenden Informationen von Patienten zu suchen bzw. zu filtern. So können Patienten danach gefiltert werden, ob sie vollständige Records, bestehende Bilder oder abgeschlossene Fragebögen in ihren Akten vorliegen haben. Auch Detailinformationen wie der familiäre Hintergrund bezüglich Tinnitusbeschwerden oder der Ort an dem der Tinnitus wahr genommen wird können als Eingrenzung der Suche dienen um präzisere Ergebnisse zu generieren. Oft gebrauchte Schlüsseloptionen sollten dabei schnell und einfach, beispielsweise über Dropdown-Menüs, zu erreichen sein. Für komplexere und tiefgreifende Suchanfragen gibt es zum einen die Möglichkeit, diese über spezielle Kommandos textuell direkt in der Suche einzugeben. Zum anderen wäre ein visueller Dialog möglich, welcher es dem Nutzer erlaubt, erweiterte Filterkriterien zu aktivieren.

[8] `https://github.com/twitter/typeahead.js/blob/master/doc/bloodhound.md`
[9] `https://twitter.github.io/typeahead.js`

### 4.1.3 Allgemeine Systemverbesserungen

#### Laufzeitumgebung

Eine weitere Leistungssteigerung der vollständigen Applikation ist durch eine Aktualisierung der PHP-Laufzeitumgebung möglich. Das Laravel-Team hat im Oktober 2016 das Update[10] Version 4.2.20 veröffentlicht, welches speziell die Kompatibilität mit PHP 7.0 im Fokus hatte. Das Tinnitus-Database-Projekt arbeitet aktuell auf Laravel Version 4.2.22, womit ein problemloser Umstieg gewährleistet sein sollte. PHP 7.0 bringt gegenüber Version 5.6 einige Vorteile mit sich. Durch eine vollständige Überarbeitung der Engine konnte der Speicherbedarf der Sprache signifikant gesenkt werden. Besonders 64-bit Systeme profitieren laut Entwickler deutlich von den Optimierungen und beanspruchen bis zu dreieinhalb Mal weniger Speicher [7]. Als Resultat erhöht sich die Performanz auf das bis zu Zweifache seines Vorgängers. Offizielle Benchmarks[11] zeigen die in der Stellungnahme zur Veröffentlichung versprochene Steigerung auch für Laravel-basierte Projekte [25, 26]. Ein weiteres Indiz zur Untermauerung findet sich in Tabelle 3.1 der Analyse. Der konkrete Vergleich beider PHP Versionen bezüglich ihrer Verarbeitungszeiten der SQL-Queries zeigt eine bis zu 5-fach höhere Leistung der neueren Version im Rahmen dieser Arbeit.

#### Exkurs: Tests

Aufgrund der während der Arbeiten an diesem Projekt festgestellten Mängel bezüglich der Qualität des Codes, aber auch dem schlichten Fakt, dass dieses Projekt von einer Vielzahl an Entwicklern unterhalten wird, ist eine Integration von dynamischen Testverfahren anzustreben. In der Webentwicklung, insbesondere auch bei der Wartung, ist es wichtig, die Qualität des Codes stetig zu überwachen und zu verbessern. *Continuous Integration* ist eine Softwareentwicklungsmethode, welche genau diese Ziele anstrebt und über periodisch automatisierte Testszenarien erlaubt, Integrationsfehler zügig aufzudecken. Dies ermöglicht es, im Team schneller kohesive Software entwickeln zu können. Die Methode führt die Entwickler durch den Prozess der Entwicklung, wobei die Richtlinien von der korrekten Unterhaltung des Repositories über die vollständige Aufrechterhaltung von Transparenz im

[10] `https://twitter.com/laravelphp/status/791302938027184128`
[11] `https://docs.google.com/spreadsheets/d/1qW0avj2eRvPVxj_5V4BBNrOP1ULK7AaXTFsxcffFxT8`

Projekt bis hin zur Ausführung von Tests reicht [5]. Um eine Automatisierung dieser Art zu erreichen, kann ein entsprechender Server wie Jenkins[12] genutzt werden. In Verbindung mit einem bereits auf PHP Projekte optimiertem Template[13] übernimmt dieser den Build-, Deployment- und Automatisierungsprozess.

Die verwendeten Testlevel sollten *Unit-*, *Integration-* und *Component-Interface-Tests* beinhalten. *Unit-Tests* spielen dabei eine der wichtigsten Rollen. Sie verifizieren die Funktionalität der kleinsten Komponenten eines Systems wie Funktionen, Methoden und Klassen. Dabei ist eine Codeabdeckung von 100% anzustreben [17], wobei in der Praxis eine qualitative Aussage der Tests bereits ab 90% Abdeckung möglich ist. Das Verifizieren von Ein- und Ausgabedaten über mehrere Units hinweg wird durch *Component-Interface-Tests* erzielt [14]. Des Weiteren prüfen *Integration-Tests* die Schnittstellenfunktionalität von Komponenten, welche aus mehreren zusammengehörigen Units bestehen. Laravel bietet mit PHPUnit[14] bereits ein integriertes Framework für *Component-Tests* an.

Die Einführung solcher Verfahren sollte künftig dazu verhelfen, überflüssigen Programmcode, Bugs und andere Fehler frühzeitig zu erkennen und es ermöglichen, schneller auf diese zu reagieren. Im Vordergrund steht jedoch die Erzeugung von konstant stabilen Programmen.

#### Exkurs: SQL-Query Optimierung

Unter der Annahme, die in der Analyse und Abschnitt 3.1 beschriebenen SQL-Queries, welche die `WHERE IN` Klausel beinhalten, hätten benötigte Daten abgefragt und damit weiterhin genutzt werden müssen, wäre eine Optimierung dieser bezüglich ihrer Performanz unumgänglich. Die vollständigen Ausführungszeiten, die in der Applikation von Queries dieser Art benötigt werden, liegen viel zu hoch und stellen vor allem im Kontext dieser Arbeit ein Performanzproblem dar. Tabelle 3.1 zeigt, dass die hauptsächlichen Leistungseinbußen nicht auf Kosten des SQL Servers gehen und damit die `WHERE IN` Klausel an sich nicht das maßgebliche Problem bildet. Aufgrund der drastischen Leistungssteigerung unter PHP 7.0 muss der Ursprung des Problems auf Ebene der Applikation zu finden sein.

[12] `http://jenkins-ci.org`
[13] `http://jenkins-php.org/index.html`
[14] `https://phpunit.de`

```
1 $patient_data = Patient::with('relation1', 'relation2')->get();
```

Listing 4.1: Beispiel von Eager Loading unter Eloquent

```
1 select * from `patients`
2
3 select * from `relations_1`
4     where `foo_id` in (`1`, `2`, `3`, `4`, ...)
5
6 select * from `relations_2`
7     where `bar_id` in (`1`, `2`, `3`, `4`, ...)
```

Listing 4.2: Erzeugte SQL-Queries durch Eager Loading

Aufgerufen werden die verantwortlichen Queries in der `/app/bindings.php` und werden dort durch das in Laravel integrierte Eloquent[15] ORM[16] gebildet. Eloquent ist eine ActiveRecord[17] Implementierung und ermöglicht es, auf objektorientierter Basis relational mit einer Datenbank zu arbeiten. Dabei wird jede Tabelle als Modell im System abgebildet, welches dann mit dieser interagieren kann. Das Erzeugen eines neuen Objektes generiert einen neuen Tabelleneintrag und das Laden eines Objektes bezieht die Daten aus der Datenbank. Die Bildung einer solchen Klausel ist über einen Aufruf der Form, wie in Auszug 4.1 exemplarisch dargestellt, möglich. Sie werden aufgrund der definierten Klassen und deren Abhängigkeiten erstellt und liefern eine, wie in Auszug 4.2 gezeigt, Folge von SQL-Aufrufen.

Im Rahmen dieses Projektes werden auf beschrieben Weise Queries gebildet, welche in Relation zu jedem aktiven Patienten in der Datenbank stehen. Es werden also Subqueries mit entsprechender Anzahl an Ergebnissen erzeugt, zu welchen Eloquent jeweils ein zugehöriges Objekt initialisiert und mit Daten befüllt. Dieser Vorgang des Ladens der Daten wird *population* oder *hydration* genannt und benötigt, für die Initialisierung von knapp 4600 Objekten (siehe Abschnitt 3.4.1), entsprechend viel Zeit. Umgehen lässt sich dieser ineffiziente Prozess mit dem Verzicht auf ActiveRecords und damit dem Eloquent ORM. Dazu könnte die entsprechenden Datenbankabfragen zum Beispiel mit dem QueryBuilder[18] erzeugt werden.

[15] `https://laravel.com/docs/4.2/eloquent`
[16] Object-relational mapping
[17] `https://de.wikipedia.org/wiki/Active_Record`
[18] `https://laravel.com/docs/4.2/queries`

Weiterhin ist anzumerken, dass die `WHERE IN` Klausel generell nicht auf derart großen Listen bzw. Subqueries ausgeführt werden sollte, da die meisten SQL Server diese Queries intern umschreiben, wobei der logische `OR` Operator genutzt wird. Dieser Prozess benötigt zusätzlich Zeit und wird noch aufwendiger, wenn sich Parameter der Query ändern. In diesem Fall muss der vollständige Ausführungsplan neu erstellt werden. Gleichzeitig sind viele Server mit einer Limitierung bezüglich der Anzahl und Komplexität der logischen Operatoren pro Query versehen, was zum schlichten Abbruch der Ausführung führen kann [3].

Noch bestehende und benötigte Abfragen, welche das Eloquent ORM utilisieren und einen potentiellen Engpass darstellen, sollten demnach überarbeitet werden, um die Applikation hinsichtlich ihrer Performanz zu optimieren.

#### Exkurs: Sicherheit

Nicht nur Leistungseinbußen durch ineffiziente oder überflüssige Datenbankabfragen können ein System schwächen, sondern auch Sicherheitsmängel. Im Rahmen der Arbeiten an diesem Projekt ist problematischer Programmcode aufgefallen, welche leicht zu unerwünschten Datenfreigaben führen kann. Alle `switch` Statements im Code der Datei `/app/bindings.php` sind nach einem Schema aufgebaut, welches durch einfache Tippfehler die Zugriffsrechte der Nutzerrollen überflüssig macht und somit sensible Daten Unbefugten zugänglich machen könnte.

```
1 switch ($access_right) {
2     case 'superadmin':
3         return DB::('table')->get();
4     case 'none':
5         return null;
6     default:
7         return DB::('table')->where('foo', '=', $bar)->get();
8 }
```

Listing 4.3: Beispiel für sicherheitsproblematisches Schema

```
switch ($access_right) {
    case 'superadmin':
        return DB::('table')->get();
    case 'user':
    case 'expert':
        return DB::('table')->where('foo', '=', $bar)->get();
    default:
        return null;
}
```

Listing 4.4: Beispiel für sicheres Schema

Wie im Beispiel 4.3 zu sehen ist, wird für den Standardfall die gewünschte Datenbankabfrage für alle Nutzerrollen des Systems ausgeführt und die Daten zurückgeben. „Gefiltert“ werden lediglich die Fälle des Superadmins und eines Nutzers ohne Rechte. Liegt hier nun ein Fehler vor, fragt das System immer Daten ab und gibt diese weiter. Um solche Lücken zu schließen, sollten die betroffenen Statements überarbeitet werden, indem wie in Beispiel 4.4 gezeigt, alle Nutzerrollen und ihre zugehörige Funktionalität explizit definiert werden. Liegt hier nun ein Fehler in der Variablen vor oder der Nutzer hat keine Rechte, wird stets der Standardfall gewählt und es werden keine Daten zurückgegeben.

## 4.2 Entwicklungsentscheidungen

Der wichtigste Aspekt der Patientenliste ist es, dem Anwender dazu zu verhelfen, gezielt bestimmte Patienten zu finden um deren Daten einzusehen, zu erweitern oder modifizieren. Zu finden sind diese über numerische Identifikatoren, welche ein manuelles Suchen in einer Liste mit bis zu 3600 Einträgen nahezu nicht ermöglichen. Funktionalität zur Erleichterung dieses Prozesses ist also unumgänglich, um den Arbeitsfluss des Anwenders nicht unnötigerweise aufzuhalten. Ein Suchmodul, welches diese Aufgabe erledigt und sowohl schnell als auch zuverlässig die gewünschten Daten liefert, ist aus einer Datenbankanwendung wie der TRI Database und besonders in diesem Anwendungsfall also nicht wegzudenken. Aus diesem Grund ist eine ständige Präsentation aller Daten mit hohen Ladezeiten im Bereich der Arbeitsumgebung überflüssig und stellt eine seitenweise Prä-

sentation nicht als Einschränkung des Systems dar. Untermauert wird dies von möglichen Leistungsverbesserungen auf Seite des Servers sowie des Klienten. Von der momentan vom Produktionssystem benötigten Zeit, die Patientenliste vollständig zu laden, fühlt sich der Anwender laut Nielsen [15] in seinem Arbeitsfluss unterbrochen. Das Gefühl, direkt mit der Applikation und den Daten zu arbeiten geht verloren. Der Fokus des Nutzers verschiebt sich auf andere Tätigkeiten und vor allem die Produktivität, aber auch die Zufriedenheit, sinkt.

Eine schnelle Präsentation der Daten, aber auch die Möglichkeit mit diesen sofort Arbeiten zu können ist demnach das anzustrebende Ziel nach welchem die Entwicklungsentscheidungen maßgeblich getroffen werden. Des Weiteren ist die Endgerät-unabhängige Verfügbarkeit der Informationen und Funktionalitäten ein wichtiges Kriterium der Entscheidungsfindung.

Für die gesamte Applikation ist eine allgemeine Leistungssteigerung des Faktors zwei mit einem Update der PHP-Laufzeitumgebung auf Version 7.0 zu erreichen. Dazu sind lediglich minimale Modifikationen (siehe Analyse 3.4.1) des Projektes nötig, da eine volle Unterstützung dieser Version durch Laravel gewährleistet ist. Weitere Reserven werden durch das Entfernen der in der Analyse aufgedeckten überflüssigen Programmcodereste, sowie durch die Optimierung bestehender Datenbankanfragen frei gegeben. Diese Maßnahmen sind mit Hinblick auf die zukünftige Entwicklung, Unterhaltung und Stabilität der Applikation verpflichtend.

Um die Geschwindigkeit der Generierung des DOM, unabhängig von der maximalen Anzahl der zu ladenden Patienteneinträge, konstant niedrig halten zu können, muss auf eine vollständige Übertragung aller Datensätze zum Klienten verzichtet werden. Eine Aufteilung der Daten in Form einer Paginierung reduziert die zu übertragende Datenmenge und ermöglicht so ein Minimum an Lade- und Übertragungszeiten. Mit dem Webentwicklungsansatz und der Idee des *Progressive Enhancement* im Hinterkopf ist auf eine JavaScript basierte Ajax-Implementierung der Paginierung zu verzichten.

*Progressive Enhancement* ist ein Ansatz, webbasierte Projekte zu entwickeln und entstand aus der allgemeinen Unzufriedenheit unter Entwicklern heraus, das in die Jahre gekommene Prinzip der *Graceful Degradation* weiter zu führen. Dieser neue Ansatz stellt den Inhalt einer Website in den Vordergrund. Die Informationen sollen für jeden Nutzer vollständig zugänglich sein. Der dazu verwendete Browser und verfügbare Technologien wie JavaScript oder CSS sollen dabei nur eine minimale Rolle spielen und keine Barriere für die Zugänglichkeit des

eigentlichen Inhalts oder der Funktionalität der Applikation darstellen. Diese Idee steht im Gegensatz zur *Graceful Degradation*, welche zwar immer die neuesten Webtechnologien und Features zur Verfügung stellt und fördert, jedoch Nutzer, denen diese Technologien nicht zur Verfügung stehen, außer acht lässt und nur mit einem Minimum an Inhalt, Funktionalität und Kompatibilität bedient [8].

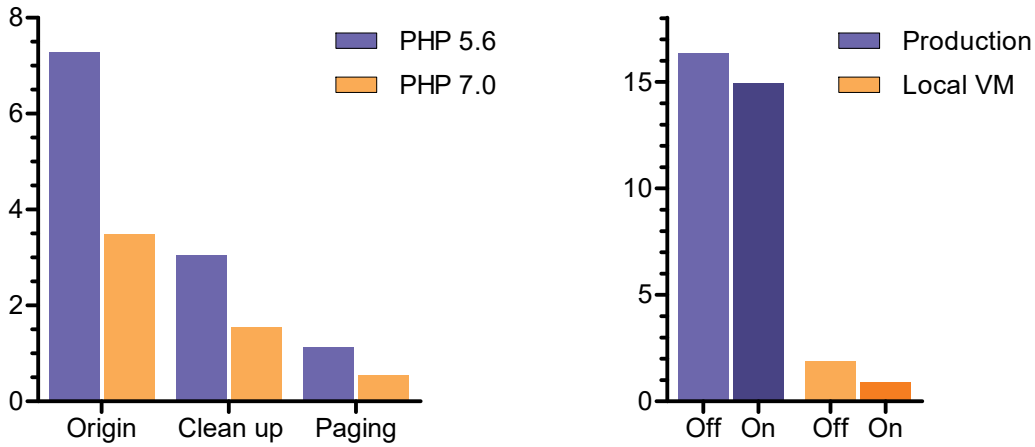
Im Kontext der Art der Anwendung als Suchdatenbank mit Fokus auf ein zuverlässiges und präzises Suchmodul rücken die wenigen Vorteile einer Ajax basierten Lösung noch weiter in den Hintergrund. So spielt, in Anbetracht der zu erreichenden Performanz der Applikation, das vollständige Laden einer neuen Seite der Paginierung gegenüber den mit Ajax lediglich nachzuladenden Kerninformationen keine Rolle mehr. Weiterhin erhöht der Verzicht auf JavaScript die Kompatibilität zu älteren Browsern und umgeht Performanzunterschiede der verwendeten JS-Engines [4].

Infolgedessen kommt für die Suchmaske ebenfalls die nicht auf Ajax basierende Lösung über ein HTML-Formular zum Einsatz. Diese Kernfunktion sollte dem Anwender unter allen Umständen zur Verfügung stehen. Jedoch bietet sich hier die Möglichkeit an, nach dem Prinzip des *Progressive Enhancement* die Usability der Maske zu erhöhen. Dies ist über eine, wie in den Lösungsvorschlägen (siehe Abschnitt 4.1.2) beschriebe Ajax basierte Livesuche möglich. Außerdem ist zu beachten, dass das Eingabefeld mindestens 35 bis 40 Zeichen darstellen kann und damit die Länge einer typischen Anfrage vollständig abdeckt [23]. Dies hilft dem Anwender, seine Eingabe im Überblick zu behalten und Fehleingaben zu vermeiden. Weitere Usability-Verbesserungen werden durch simple und leicht zugängliche Filteroptionen erreicht. Diese erweitern die Maske um die Möglichkeit zur Einschränkung auf häufig verwendete oder im Kontext zu den Daten stehende Kriterien der gestellten Suchanfrage. Erweiterte Filterfunktionalität über komplexe Befehle, die in die Maske einzugeben sind, werden von Anwendern laut Nielsen [16] nicht genutzt.

# 5 Ergebnis

Im Fokus der Arbeit stand es, dem Anwender der Applikation die Funktionalität des Suchmoduls im Kontext der Patientenliste schneller zur Verfügung zu stellen. Motivation dieses Ziels war es, dem Nutzer lästige Wartezeiten zu ersparen, welche zu Unzufriedenheit, aber auch zu Ineffizienz im Umgang mit dem System führen können.

## 5.1 Performanz




(a) JMeter TTFB (Mittelwerte in s)

(b) Chrome Ladezeiten (Mittelwerte in s)

Abbildung 5.1: Vorher/Nachher Leistungsvergleiche

Die in Abschnitt 3.4.1 der Analyse festgestellten Leistungsprobleme der Applikation und die damit einhergehenden hohen Wartezeiten, welche vom Nutzer in der aktuellen Implementierung geduldet werden müssen, konnten reduziert werden. Dazu wurden die in den

Lösungsvorschlägen 4.1 und Entwicklungsentscheidungen 4.2 erläuterten Optimierungen des Systems sukzessive angewandt.

Die Aktualisierung der Laufzeitumgebung von PHP 5.6 auf PHP 7.0 zeigt eine Leistungssteigerung der Applikation um den Faktor zwei oder mehr. Diese Steigerung deckt sich mit den Werten, die vom PHP-Entwicklerteam bei der Veröffentlichung bekannt gegeben wurden [25, 26]. In Abbildung 5.1a sind die Mittelwerte der TTFB in drei entscheidenden Phasen der Entwicklung zu sehen. Die Zeiten repräsentieren die Dauer, die die Applikation benötigt um das DOM zu generieren. Sie wurden, wie die Daten aus der Analyse, über ein JMeter-Skript ermittelt, welches den Server der lokalen VM mit 20 aufeinanderfolgenden Anfragen belastet hat. Die Gruppe „Origin“ steht für die unveränderte Applikation, welche keinerlei Optimierungen des Codes beinhaltet, die im Rahmen dieser Arbeit umgesetzt wurden. Hier ist deutlich zu erkennen, dass die in der Analyse ermittelten Werte zur Generierung des DOM von ca. 7,5 s auf ca. 3,5 s heruntergebracht werden können. Eine weitere Halbierung dieser ist durch Optimierungen des Codes möglich. Das Entfernen von ungebrauchtem Programmcode und eine Optimierung der SQL-Datenbankabfragen bringt die TTFB unter PHP 7.0 auf durchschnittlich 1,5 s und ist in der Gruppe „Clean up“ zu sehen. In diesen beiden Stufen der Optimierung wurden noch keine Modifikationen an der Funktionsweise der Applikation per se ausgeführt. Sie erzeugt immer noch die identische Ausgabe wie das Ursprungssystem, kann jedoch im Vergleich zu diesem einen rund 5-fachen Performanzsprung verzeichnen.

Durch die Einführung der Paginierung der Datensätze reduziert sich das zu generierende DOM von 2,3 MByte auf gerade einmal 80 kByte für 100 Datensätze. Zusätzlich werden die benötigten 24 Requests für alle Ressourcen der Seite auf 18 Dateien verringert. Dies wird durch den Verzicht der JavaScript basierten Front-End-Verarbeitung der Tabelle erreicht und spart weitere Daten, welche hätten geladen werden müssen. Wie die Gruppe „Paging“ im Diagramm 5.1a zeigt, bringen diese Modifikationen die TTFB der Applikation auf ein Durchschnitt von 0,5 s. Es konnte in der lokalen Umgebung also eine 15-fache Leistungssteigerung der Applikation bewältigt werden. Um diese gemessenen Werte bezüglich der realen Ladezeiten etwas besser einordnen zu können, ist in Abbildung 5.1b ein direkter Vergleich zwischen dem aktuellen Live-Produktionssystem und dem optimierten System in der lokalen VM skizziert. Die Mittelwerte stellen die vollständigen Ladezeiten der Seite mit aktiviertem „On“ und deaktiviertem „Off“ Browser-Cache dar. Für die Aufzeichnung der Daten wurden die Google Chrome DevTools verwendet und mit diesen jeweils 20 Anfragen an den Server

gestellt. Aus dem Diagramm 3.1 der Analyse ist erkenntlich, dass die Produktionsumgebung wesentlich mehr Hardware-Ressourcen zur Verfügung hat als die in dieser Arbeit verwendete Entwicklungsumgebung und ist damit klar im Vorteil. So benötigt die Produktionsumgebung gerade einmal die Hälfte der Zeit für die identische Arbeit. Im Kontext der in Abbildung 5.1b dargestellten Werte bedeutet dies, dass die minimal erreichten Ladezeiten der lokalen VM von durchschnittlich 0,87 s mit den Hardware-Ressourcen des Webservers noch weiter reduziert werden können.

## 5.2 Funktionalität

Mit der Neukonzeption der Patientenliste und der dadurch entstandenen Entwicklungsentscheidung, JavaScript überall dort zu vermeiden, wo ohne erheblichen Mehraufwand keine weiteren Ebenen als Fallback bereit gestellt werden können, ergeben sich die im Folgenden dargestellten Auswirkungen auf das Ergebnis des Suchmoduls und seine Funktionalität. Die vollständig neu entwickelte Suchmaske bedient sich dabei am klassischen HTML-Formular, welches die Suchanfragen zusammen mit den Filtereinstellungen an den Server überträgt um dort evaluiert zu werden. Auf Basis von Usability-Aspekten, aber auch aufgrund der neu gewonnen Funktionalitäten, ändert sich das User Interface der Patientenliste zu Gunsten des Anwenders. Wie in Abbildungen 5.2 zu sehen ist, wurde die alte Suchbox durch ein prägnant größeres Eingabefeld ersetzt, welches sich über die vollständige Breite des Inhaltes erstreckt. Durch diese Präsenz spiegelt sie ihre Wichtigkeit im Kontext des System wider und bietet den Vorteil, eine Anfrage mit einer Länge von 35 bis 45 Zeichen darstellen zu können. Dies führt zu mehr Übersichtlichkeit und hilft dem Anwender die Korrektheit seiner Eingabe zu prüfen. Über einen in der Suchmaske befindlichen Toggle-Button lassen sich die erweiterten Filtereinstellungen unter dieser einblenden und ermöglichen das Ergebnis weiter einzuschränken. So können hier von vornherein beispielsweise spezifische Altersklassen der Patienten ausgewählt oder zu Zwecken der Administration die aktuellsten Einträge einer bestimmten Zeitspanne gefiltert werden.

Abbildung 5.3 skizziert die zusätzliche auf Ajax basierte Livesuche, welche dem Anwender schon während seiner Eingabe Vorschläge bezüglich seiner Suchanfrage vorschlägt. Ein visuelles Hervorheben der Schlüsselwörter in den Ergebnissen unterstützt dabei die Lesbarkeit für den Anwender. Jedes Ergebnis stellt dabei einen Patienten dar zu dessen Akte

mit einer Bestätigung direkt weiter navigiert werden kann. Im Anwendungsfall einer Suche nach einem spezifischen Patienten macht diese Funktionalität die klassische Suchmaske überflüssig.

Das durch die Paginierung benötige Element zur Navigation durch die Datensätze befindet sich jeweils auf der linken Seite am Anfang und Ende der Patientenliste. Es bietet eine Repräsentation der aktuellen Position und der gesamten Anzahl an Seiten auf welche sich die Daten der jeweiligen Liste verteilen. Des Weiteren bietet es einfache Navigationselemente, die es erlauben, in beide Richtungen seitenweise zu blättern (siehe Abbildung 5.2). Das Paginierungsintervall lässt sich über ein einfaches Dropdown-Menü auswählen und befindet sich im kontextuellen Zusammenhang direkt rechts neben der Navigation zur Paginierung. In der gleichen Werkzeugleiste befinden sich auf der rechten Seite des Layouts der schon ursprünglich existierende Button zur Neuanlage eines Patienten und der neupositionierte Print-Button, welcher die Druckoptionen bereitstellt. Über einen separates Dropdown-Menü lässt sich hier beispielsweise der gewünschte inhaltliche Bereich der Liste als Druckausgabe erzeugen.

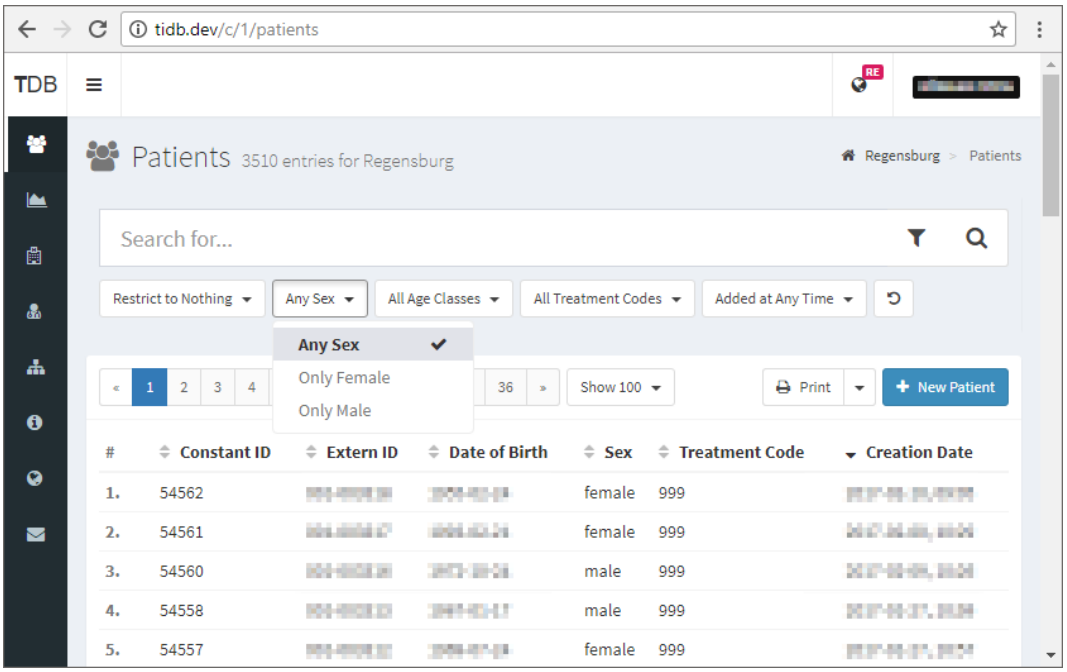



Abbildung 5.2: Neugestaltung der Patientenliste mit „ausgeklappter“ Suchmaske

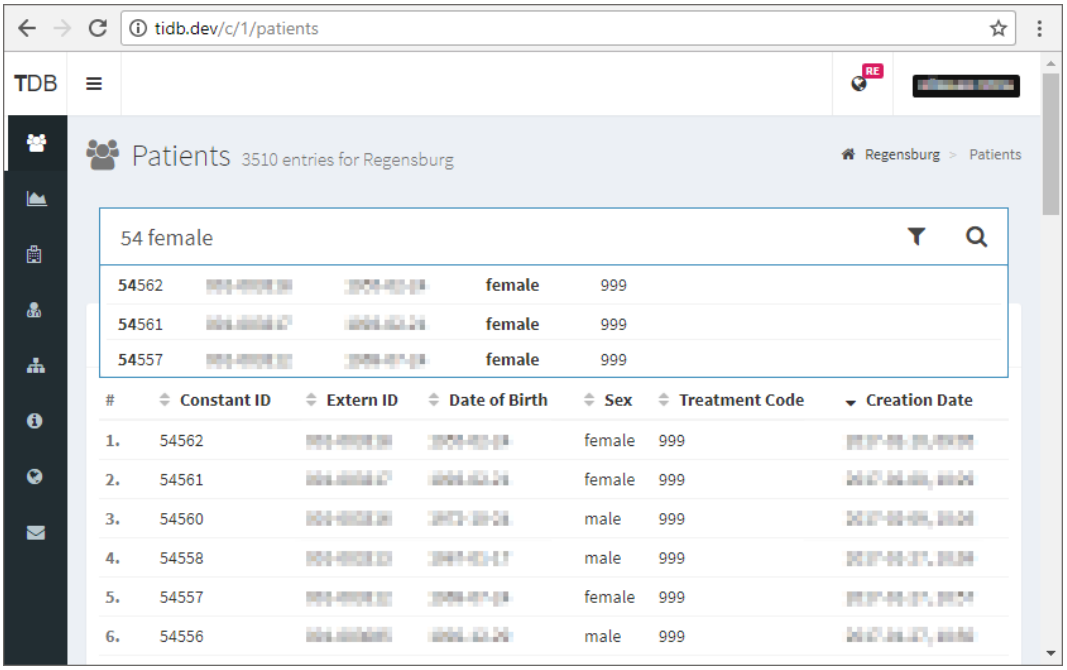



Abbildung 5.3: Neue Suchmaske mit Beispiel einer Livesuche

# 6 Diskussion und Ausblick

Durch die Umsetzung der Entwicklungsentscheidungen konnten die Anforderungen an das Antwortzeitverhalten der Applikation übertroffen werden. Die erreichten Ladenzeiten liegen weit unter dem von Nielsen [15] und Fui-Hoon Nah [6] definierten Rahmen von maximal 2 s. Praxisnahe Tests mit der lokalen Entwicklungsumgebung erreichen Werte im durchschnittlichen Bereich von 0,87 s. Hochgerechnet mit den aus dieser Arbeit entstandenen Daten bezüglich der Performanz, bedeutet dies für eine leistungsstärkere Serverumgebung Ladezeiten von ungefähr 0,6 s oder weniger. Verglichen mit den vom aktuellen Produktionssystem durchschnittlich benötigten 14,95 s (siehe Abbildung 5.1b) würde die Umsetzung der Lösungsvorschläge, welche im Kontext dieser Arbeit entstanden sind, eine für den Endnutzer bis zu 25-fache Leistungssteigerung bedeuten. Zusätzlich profitiert die gesamte Applikation durch die Aktualisierung der PHP-Laufzeitumgebung von einer Verdoppelung der Leistung und reduziert die TTFB dementsprechend. Insgesamt gewinnt auch der Server an Ressourcen und kann durch diese Ersparnis unter anderem eine höhere Anzahl an Anfragen parallel bearbeiten.

Weiterhin wurde im Rahmen der Problemstellung dieser Arbeit der Verzicht auf eine tiefgreifende JS-Implementierung der Kernfunktionalitäten entschieden. Betroffen sind also die Seite der Patientenliste und das dort integrierte Suchmodul per se. Im Anbetracht der gesamten Plattform hätte diese Entscheidung ebenso zugunsten einer JS basierten Version führen können, da für die aktuelle Implementierung die Ausführung von JS auf dem Endgerät obligatorisch ist. Aufgrund der Prinzipien der *Graceful Degradation* und vor allem des *Progressive Enhancement* (siehe Abschnitt 4.2) ist jedoch klar eine Implementierung vorzuziehen, welche diese Methoden verinnerlicht und konsequent umsetzt. Folglich ist sogar über eine Neugestaltung des Front-Ends für die gesamte Applikation nachzudenken. Eine internationale medizinische Datenbank mit Zugang sowie Ein- und Ausgabefunktionalität für Kliniker, Mediziner und bedingt Patienten sollte keinerlei Barrieren beinhalten, welche den

Zugang von Inhalt und jeglicher Kernfunktion blockieren. Vor allem dann nicht, wenn diese aus rein optischen Gründen entstehen.

Im Zuge des Setups und der Analyse wird deutlich, dass das Projekt der Tinnitus Database nicht allein durch das Beheben einzelner Leistungsschwierigkeiten von spezifischen Komponenten langzeitig stabil gehalten werden kann. Um zukünftig eine robuste und qualitativ hochwertige Plattform gewährleisten zu können, an der ein Team mit stetiger Fluktuation arbeitet, müssen Methoden etabliert werden, welche dies ermöglichen. Ein sauberes Git Repository, welches den Entwicklern erlaubt die Applikation in so wenigen Schritten wie möglich lokal aufsetzen zu können, ist dabei eine grundlegende Voraussetzung und bereits ein Teil der anzustrebenden Entwicklungsmethode *Continuous Integration*. Die periodische Automatisierung von Build-, Deployment- und Testprozessen ist hier ein weiterer Meilenstein, der die in dieser Arbeit aufgezeigten Mängel am Code schon zur Zeit des Commits hätte vermeiden können. Des Weiteren ist das Einhalten eines projektweiten allgemeingültigen Codestils eine Hilfe für Entwickler um sich effizienter mit fremdem Code auseinandersetzten zu können.

Bei der zukünftigen Projektgestaltung ist demnach neben der Weiterentwicklung der Applikation besonders eine konsequente Durchsetzung der genannten Softwareentwicklungsmethoden und Prinzipien von großer Bedeutung. Eine Aktualisierung der Laufzeitumgebung ist eine temporäre Lösung, welche jedoch nicht in der Lage ist, den Entwicklern dabei zu helfen, eine sichere und hochwertige Plattform aufzubauen und zu erhalten. Dennoch sind auch solche Prozesse stetig durchzuführen um der Serverumgebung aktuelle Sicherheitsaktualisierungen bereitstellen zu können. Laravel bietet für die momentan verwendetet Version 4.2 eine Langzeitunterstützung bezüglich Sicherheitsaktualisierungen an, leistet diese jedoch nur für drei Jahre[1], womit sie Mitte des Jahres 2017 endet. Eine Umstieg auf die aktuellste Version des Frameworks ist damit auf lange Sicht nicht zu umgehen.

Unter den aktuellen Rahmenbedingungen der Applikation und in Anbetracht der neu hinzukommenden Anbindung einer speziellen Schnittstelle für Patienten zur autonomen Durchführung von medizinischen Fragebögen ist, neben einem Refactoring der Applikation, sogar über eine vollständige Neukonzeption der Plattform nachzudenken. Eine Trennung der Datenverarbeitung vom Front-End würde Vorteile und neue Möglichkeiten hervorbringen

[1] `https://laravel.com/docs/5.4/releases#support-policy`

die Plattform weiter auszubauen. Dieser initial aufwendige Schritt einen RESTful[2] Service neu zu implementieren, würde es mehreren Entwicklerteams erlauben gleichzeitig und vor allem unabhängig voneinander an verschiedenen Projekten innerhalb der Plattform zu arbeiten. Realisiert wird dies durch eine für die Plattform standardisierte API, über welche alle Applikationen mit der selben Datenverarbeitungsbasis als Service kommunizieren. Die Konstruktion solch einer API kann beispielsweise mit einem Werkzeug wie Swagger[3] durchgeführt werden. Neben der dadurch erhöhten Skalierbarkeit, wäre mit dem neuen Architekturstil zukünftig eine einheitliche und plattformunabhängige Entwicklung von Diensten rund um die Tinnitus Database im modernen Web gegeben.

[2] https://en.wikipedia.org/wiki/Representational_state_transfer
[3] http://swagger.io

# A Anhang

Die beiliegende Digital Versatile Disc beinhaltet folgende Daten:

- Digitale Kopie dieser Bachelorarbeit
- Screenshots in voller Größe
- Quellcode der Applikation „Clean up“
- Quellcode der Applikation „Paging“
- PuPHPet Konfigurationsdatei für Vagrant VM mit PHP 5.6
- PuPHPet Konfigurationsdatei für Vagrant VM mit PHP 7.0
- Apache JMeter Skripte und Ergebnisse

Einen Zugang zum Git Repository mit dem im Rahmen dieser Arbeit modifizierten Master-Branch des original Repository[1] stelle ich gerne auf Anfrage zur Verfügung.

[1] `https://bitbucket.org/mstach/tinnitus-research-db`

# Literaturverzeichnis


[1] Burak Baltakiranoglu. Konzeption und Realisierung eines Patientenmoduls für eine multinationale und interdisziplinäre Datenbank. Masterarbeit, October 2015.

[2] Paul Böhm. Evaluation aktueller Web-Techniken am Beispiel eines interdisziplinären und internationalen Datenbank Projekts. Bachelorarbeit, 2014.

[3] L. Bushkin. Is SQL IN bad for performance?, Juni 2009. URL http://stackoverflow.com/questions/1013797/is-sql-in-bad-for-performance/1013959#1013959. Zuletzt abgerufen am 18. Mai 2017.

[4] Douglas Crockford. Actual JavaScript Engine Performance. URL http://javascript.crockford.com/performance.html. Zuletzt abgerufen am 20. Mai 2017.

[5] Martin Fowler. Continuous Integration, Mai 2016. URL https://martinfowler.com/articles/continuousIntegration.html. Zuletzt abgerufen am 20. Mai 2017.

[6] Fiona Fui-Hoon Nah. A study on tolerable waiting time: how long are Web users willing to wait? *Behaviour & Information Technology*, 2004.

[7] Sebastian Grüner. PHP reduziert Array-Speicherbedarf deutlich, Dezember 2014. URL https://www.golem.de/news/php7-php-reduziert-array-speicherbedarf-deutlich-1412-111336.html. Zuletzt abgerufen am 11. März 2017.

[8] Aaron Gustafson. Understanding Progressive Enhancement, Oktober 2008. URL https://alistapart.com/article/understandingprogressiveenhancement. Zuletzt abgerufen am 20. Mai 2017.

[9] Robin Hagenlocher. Designkonzept für eine Webanwendung zum Zugriff auf eine interdisziplinäre und multinationale Datenbank zur Erfassung Tinnitus-geschädigter Patienten. March 2015.

[10] John A Hoxmeier and Chris DiCesare. System response time and user satisfaction: An experimental study of browser-based applications. *AMCIS 2000 Proceedings*, 2000. URL http://aisel.aisnet.org/amcis2000/347.

[11] Dominik Könke. Funktionsanalyse einer interdisziplinären Datenbank zu Optimierungszwecken. Bachelorarbeit, September 2016.

[12] Dave Markle. Stack Overflow - Why is SELECT * considered harmful?, September 2010. URL http://stackoverflow.com/questions/3639861/why-is-select-considered-harmful/3639964#3639964. Zuletzt abgerufen am 16. Mai 2017.

[13] Gale L Martin and Kenneth G Corl. System response time effects on user productivity. *Behaviour & Information Technology*, 1986.

[14] Gundars Mēness. Unit Testing – The Big Picture, April 2017. URL https://gundars.me/php/unit-testing-php-big-picture. Zuletzt abgerufen am 20. Mai 2017.

[15] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., 1995. ISBN 0125184069.

[16] Jakob Nielsen. Search: Visible and Simple. Mai 2001. URL https://www.nngroup.com/articles/search-visible-and-simple. Zuletzt abgerufen am 20. Mai 2017.

[17] o.V. Gelber Grad - Code Coverage Analyse, . URL http://clean-code-developer.de/die-grade/gelber-grad/#Code_Coverage_Analyse. Zuletzt abgerufen am 20. Mai 2017.

[18] o.V. The Tinnitus Research Network TINNET, . URL http://tinnet.tinnitusresearch.net/index.php/about-tinnet/tinnet-overview. Zuletzt abgerufen am 10. April 2017.

[19] o.V. TINNET - Description of WG 2: Data management in a central database and identification of subtype candidates, . URL http://tinnet.tinnitusresearch.net/index.php/2015-10-29-10-22-16/wg-2-database. Zuletzt abgerufen am 10. April 2017.

[20] o.V. Tinnitus Reserach Initiative - About Us, . URL http://www.tinnitusresearch.net/index.php/about-tri/about-us. Zuletzt abgerufen am 10. April 2017.

[21] o.V. Tinnitus Reserach Initiative - Tinnitus Database, . URL http://www.tinnitusresearch.net/index.php/for-researchers/tinnitus-database. Zuletzt abgerufen am 10. April 2017.

[22] o.V. TinnitusDatabase - About, . URL https://www.tinnitus-database.de/welcome#about. Zuletzt abgerufen am 10. April 2017.

[23] o.V. Allow Simple Searches, . URL https://guidelines.usability.gov/guidelines/191. Zuletzt abgerufen am 20. Mai 2017.

[24] o.V. BMBS COST Action BM1306 - Better Understanding the Heterogeneity of Tinnitus to Improve and Develop New Treatments (TINNET), November 2013. URL http://www.cost.eu/COST_Actions/bmbs/BM1306. Zuletzt abgerufen am 10. April 2017.

[25] o.V. PHP 7.0.0 Released, Dezember 2015. URL http://php.net/archive/2015.php#id2015-12-03-1. Zuletzt abgerufen am 11. März 2017.

[26] o.V. PHPNG (next generation) - Performance Evaluation, Dezember 2015. URL https://wiki.php.net/phpng. Zuletzt abgerufen am 11. März 2017.

[27] o.V. About COST, September 2016. URL http://www.cost.eu/about_cost. Zuletzt abgerufen am 10. April 2017.

[28] Robert Paulson. Stack Overflow - What is the reason not to use select *?, November 2008. URL http://stackoverflow.com/questions/321299/what-is-the-reason-not-to-use-select#322216. Zuletzt abgerufen am 2. März 2017.

[29] Thomas Probst, Rüdiger C Pryss, Berthold Langguth, Myra Spiliopoulou, Michael Landgrebe, Markku Vesala, Stephen Harrison, Johannes Schobel, Manfred Reichert, Michael Stach, et al. Outpatient Tinnitus Clinic, Self-Help Web Platform, or Mobile Application to Recruit Tinnitus Study Samples? *Frontiers in Aging Neuroscience*, 2017.

[30] Irina Stenske. Entwicklung eines Design-Konzepts für eine multinationale Forschungsdatenbank zur Speicherung von longitudinalen Patientendaten. Bachelorarbeit, 2015.

[31] Norman Thiel. Konzeption und Realisierung eines generischen Statistik-Moduls zur Auswertung klinischer Longitudinaldaten am Beispiel der Internationalen Tinnitus-Datenbank. December 2016.

Name: Jan Musmann Matrikelnummer: 737651

**Erklärung**

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

Ulm, den .....

Jan Musmann