HOCHSCHULE MITTWEIDA
University of Applied Sciences

BACHELORARBEIT

Herr
Tobias Kallauke

Ein Mixed-Method-Konzept zur vergleichenden Evaluation der Benutzbarkeit innovativer ERP-Frontends am Beispiel einer App-basierten Klausurnotenerfassung

2020

Fakultät **CB**

BACHELORARBEIT

Ein Mixed-Method-Konzept zur vergleichenden Evaluation der Benutzbarkeit innovativer ERP-Frontends am Beispiel einer App-basierten Klausurnotenerfassung

Autor:
Tobias Kallauke

Studiengang:
Angewandte Informatik

Seminargruppe:
IF17wS-B

Erstprüfer:
Prof. Dr. rer. nat. Marc Ritter

Zweitprüfer:
Dipl.-Inf. Holger Langner

Mittweida, Dezember 2020

I. Inhaltsverzeichnis

II. Abbildungsverzeichnis

III. Quellcodeverzeichnis

1 Einführung

Enterprise-Resource-Planning (ERP) Systeme sind für die unternehmensweite Konsolidierung der Daten zur Unterstützung sämtlicher in einem Unternehmen ablaufender Geschäftsprozesse zuständig. Über die gemeinsame Datenbasis sollen verschiedene Abläufe und Ressourcen geplant und verwaltet werden [22]. Der Anwendungsfall dieser Arbeit ist die Erfassung von physischen Dokumenten und der Transfer der Daten in ERP-Systeme am Beispiel der Klausurergebniserfassung und -meldung. Im bestehenden System der Hochschule erfolgt dies durch das Eintragen der Ergebnisse in einem Webfrontend und ist als Teil des Dozentenportals umgesetzt. Zur Ergebnismeldung einer Klausur muss hier die erreichte Leistung für jeden an der Klausur teilnehmenden Studenten eingetragen werden.

In einem vorangegangenen Semester wurde im Rahmen eines kooperativen Praxismoduls ein Prototyp für ein App-basiertes System entwickelt, mit dem dieser Prozess durch Verwendung von automatischer Texterfassung vereinfacht werden kann.

Ziel dieser Arbeit ist die Entwicklung eines Konzepts und einer Erstimplementation zur Benutzerdatenerfassung, um verschiedene Digitalisierungsprozesse messen zu können. Die durch die Benutzerdatenerfassung gesammelten objektiven Interaktionsdaten sollen weitere Daten, wie z. B. eine Benutzerakzeptanzstudie bei einer vergleichenden Evaluation von Prozessen zur Dokumentenerfassung hinsichtlich der Benutzbarkeit ergänzen und unterstützen.

1.1 Motivation

Das Übertragen von physischen Dokumenten in ein ERP-System zur digitalen Verwaltung der Informationen benötigt die volle Aufmerksamkeit des Anwenders. Im Allgemeinen können ERP-Systeme Fehler bei der Übertragung der Informationen nicht erkennen. Die Eingaben sind die einzige Möglichkeit wie das System auf die Daten zum ersten Mal zugreifen kann. Deshalb müssen Fehler vom Benutzer selbst erkannt werden, um dem System keine falschen Informationen zu liefern.

Gleichzeitig ist die manuelle Digitalisierung von Dokumenten stark repetitiv und monoton. Die kognitive Last wird vor allem dadurch verursacht, dass der Anwender Daten aus einem physischen Formular lesen und diese in ein digitales Formular übertragen muss. Die doppelte visuelle Suche, nach den zu übertragenden Angaben auf dem physischen Dokument, und anschließend die Suche nach dem zugehörigen digitalen Eingabefeld, wiederholt sich bereits mehrfach für jede Angabe, selbst wenn

nur ein Dokument zu bearbeiten wäre. Zugleich darf der Anwender keine Fehler beim Übernehmen der Daten machen und muss seine Angaben kontrollieren. Im Extremfall entspricht dies dem doppelten Ausführen der visuellen Suche, einmal zur Übertragung und anschließend zum Überprüfen der Daten.

Die Grundidee des im Praxismodul entstandenen Prototypen war die Entlastung des Nutzers bei diesen repetitiven, kognitiv belastenden Tätigkeiten. Statt die Daten per Hand zu übertragen, kann der Schritt automatisiert werden. Durch das Auslesen der Daten durch Texterkennung soll das Digitalisieren der Daten automatisch erfolgen. Der Nutzer muss nur noch die Ergebnisse der Texterkennung kontrollieren und eventuelle Fehler korrigieren. Für das Übertragen der Daten und der damit verbundenen visuellen Suche ist er nicht mehr zuständig. Tätigkeiten, auf die dieser Ansatz angewendet werden könnte, sind zum Beispiel das Digitalisieren von Anträgen, Inventarzetteln und Krankmeldungen. Das Beispiel, das in dieser Arbeit verwendet wird, ist ausschließlich die Notenmeldung von Klausurergebnissen von vielen Studenten.

Erstrebenswert wäre die Möglichkeit, solche Ansätze messen und aufzeichnen zu können. Mit einem solchen System könnte man die Effektivität und Benutzbarkeit von verschiedenen Dokumentenerfassungsprozessen anhand von quantitativen Daten testen und erfassen.

Um dieses Ziel zu erfüllen, soll ein wiederverwendbares Konzept entstehen. Dieses soll künftigen Entwicklern erste Anhaltspunkte liefern, wie ein solches System prinzipiell aussehen müsste und welche Erkenntnisse man aus diesen Daten erhalten kann.

1.2 Arbeitsschritte

In einer ersten Arbeitsphase wurde das in Kapitel 4 beschriebene Konzept zur Aufzeichnung von Benutzerinteraktionsdaten entworfen.

Um die Datenerfassung als Erstimplementation umzusetzen, musste zuerst das im Praxismodul entwickelte System erweitert werden. Der Stand der bereits existierenden Texterkennungsfunktionen ist in Kapitel 2 erklärt. Das Konzept, wie die Notenmeldung integriert werden soll, ist Inhalt des Kapitel 3. Außerdem wird das System um eine Nachbildung des Notenmeldungsprozesses des Dozentenportals erweitert, da dieses nicht für Testzwecke verwendbar ist. Die benötigte Datenerfassung dort zu implementieren ist nicht möglich. Um den Notenmeldungsprozess, wie er im Dozentenportal stattfindet, trotzdem zu messen, wird dieser in einer eigenen Webanwendung als Komponente des Systems nachgebildet.

Nach der Erfüllung dieser Voraussetzungen wurde das Datenerfassungskonzept als Erstimplementation in der Nachbildung und der App umgesetzt. Wie diese Erstimplementation sowie die Voraussetzungen realisiert wurden, ist Gegenstand von Kapitel 5. Kapitel 6 beschreibt eine Demonstration des Datenerfassungkonzeptes mit der Erstimplementierung in der Nachbildung sowie der App anhand eines aufgesetzten Durchlaufs der Notenmeldung mit den beiden Plattformen.

2 Grundlagen

Dieses Kapitel befasst sich mit dem Aufbau des im Praxismodul entstandenen Systems. Die Architektur ist nach dem Representational State Transfer (kurz REST) Paradigma aufgebaut.

2.1 Representational State Transfer

Representational State Transfer ist ein Softwarearchitekturstil für den Aufbau von verteilten, netzwerkbasierten Anwendungen und definiert Bedingungen an die sich eine Anwendung halten muss. Die Bedingungen dienen zur Definition eines einheitlichen Stils zur Kommunikation zwischen Geräten im World Wide Web.

Die grundlegende Bedingung an eine REST-Architektur ist, dass sie nach dem Client-Server-Modell aufgebaut sein muss. Dies bedeutet, dass die Komponenten der Architektur in Client und Server unterteilt werden. Ein Client ist für das Stellen von Anfragen, zum Abrufen von Informationen oder Ausführen von Funktionen, an einen oder mehrere Server zuständig. Ein Server wartet auf Anfragen von Clients, bearbeiten diese, und antworten mit den gewünschten Informationen.

Jede Anfrage muss alle Informationen zur Verarbeitung enthalten. Es werden keine Zustandsinformationen zwischen Anfragen gespeichert. Kommunikation unter dieser Bedingung gilt als zustandslos.

Komponenten in einer REST Architektur sollen mehrschichtig aufgebaut sein. Dabei können Komponenten nur mit den direkt angrenzenden Schichten kommunizieren. Für Clients kann somit eine Schnittstelle angeboten werden. Die Kommunikation mit weiteren Schichten wird von dieser umgesetzt und die Komplexität der weiteren Schichten bleiben für Clients verborgen.

Die wichtigste Bedingung des Architekturstils ist die Festlegung von Eigenschaften, die alle Schnittstellen erfüllen müssen, um eine einheitliche Kommunikation zwischen Komponenten zu ermöglichen.

REST abstrahiert alle Informationen in Form von Ressourcen. Eine Ressource kann z. B. ein Dokument, ein Bild oder eine Sammlung weiterer Ressourcen sein. Jede Ressource muss über eine eindeutige Adresse erreichbar sein. Bei REST-konformen Webanwendungen erfolgt dies meist über den Uniform Resource Identifier (URI) [2] Standard.

Anfragen und Antworten enthalten nicht die Ressource selbst, sondern eine Repräsentation der benötigten Ressource in verschiedenen Formaten, wie HTML [23], JSON [4] oder XML [24]. Eine Anwendung kann die Repräsentation in einem gewählten Format in der Anfrage anfordern.

Jede Nachricht muss selbstbeschreibend sein, also alle Informationen zur Verarbeitung enthalten. Es sollten keine externen Informationen oder eine weitere Nachricht zur Verarbeitung benötigt werden.

Die letzte definierte Eigenschaft wird „hypermedia as the engine of application state“ (kurz HATEOAS) genannt. Alle Antworten des Servers sollten enthalten, welche weiteren Anfragen ein Client ausführen kann, wie z. B. die URI von weiteren Ressourcen. Über diese Links soll ein Client die Möglichkeit haben, alle Ressourcen abzurufen.

Caching und Code-on-Demand sind zwei weitere von REST beschriebene Bedingungen, die aber für das System dieser Arbeit nicht relevant sind.

Die Kommunikation zwischen Komponenten einer REST-basierten Webanwendung erfolgt über das Hypertext Transfer Protocol (kurz HTTP). Der Aufbau von HTTP Nachrichten ist in Abbildung 2.1 dargestellt. Eine HTTP-Anfrage besteht aus Methode, Route, Protokollversion, dem Nachrichtenkopf (Header) und einem optionalen Nachrichtenrumpf (Body). Eine Antwort ist ähnlich aufgebaut. Statt Methode und Route beginnt eine Antwort mit Protokollversion und Statuscode. Der Kopf einer Nachricht besteht aus beliebigen `name: wert` Paaren. Der HTTP Standard enthält bereits Definitionen für viele Paare, wie z. B. das Format der Daten im Nachrichtenrumpf mit `Content-Type`.

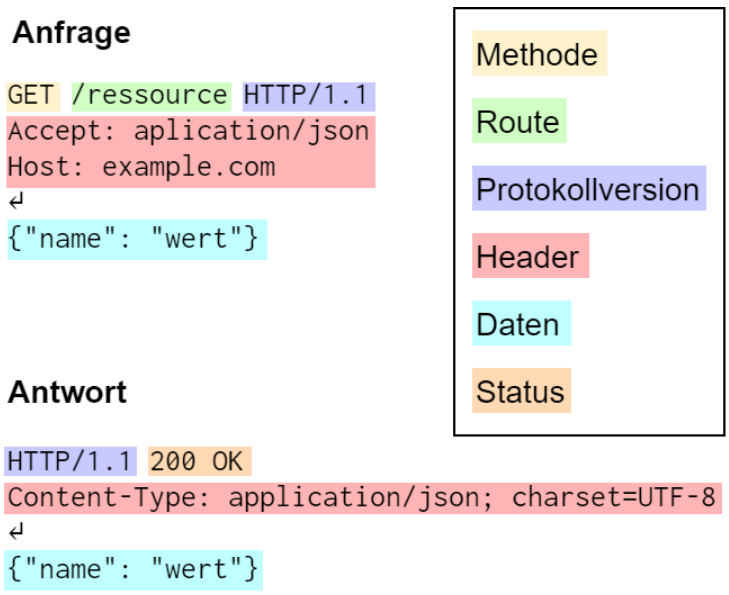


Abbildung 2.1: Aufbau von HTTP Nachrichten

Von REST-basierten Webanwendungen wird die Methode zum Angeben der auszuführenden Funktion benutzt. Die Route gibt an, auf welche Ressource die Funktion anzuwenden ist. Der HTTP Standard definiert die Methoden GET, POST, HEAD, PUT, PATCH, DELETE, OPTIONS und TRACE. Die am häufigsten benutzten Methoden sind die, die das CRUD-Muster (Create, Read, Update, Delete) umsetzen:

Methode	Funktion
GET	Abrufen der Ressource bzw. Menge von Ressourcen
POST, PUT	Anlegen einer neuen Ressourceninstanz oder Verändern einer bereits existierenden Ressource
POST	Abbildung von weiteren Operationen die von keiner anderen Methode abgedeckt werden
PATCH	Teil der angegebenen Ressource ändern. Bei POST bzw. PUT muss die gesamte Ressource angegeben werden, inklusive Eigenschaften die unverändert bleiben. Bei PATCH werden nur die zu verändernden Eigenschaften gesendet

2.2 Stand der Vorarbeiten

Ziel des im Praxismodul entstandenen Systems war die Entwicklung und Implementierung eines Konzeptes zum automatischen Erkennen und Extrahieren von Angaben auf Dokumenten, speziell Angaben auf Klausuren wie Name, Seminargruppe, Matrikelnummer und Note. Somit sollte die Verwendung von Texterkennung zur Automatisierung der Klausurnotenmeldung evaluiert werden. Das entstandene System, auf dem diese Arbeit aufbaut, ist in diesem Abschnitt beschrieben.

Die Architektur ist nach dem REST Prinzip aufgebaut und besteht aus vier Schichten: Frontend, API, Datenbank und Texterkennung und ist in Abbildung 2.2 dargestellt.

Frontend-Anwendungen sind für die Präsentation der Daten, und Interaktion mit diesen, zuständig. Im Praxismodul wurde eine iOS App zum Erstellen von Vorlagen und Scannen von Klausuren entwickelt.

Das Backend des Systems ist eine REST API, die für das Bereitstellen und Speichern der Vorlagen zuständig ist. Die persistente Datenspeicherung wird durch die Verwendung von PostgreSQL, einem objektrelationalen Datenbankmanagementsystem realisiert. Außerdem stellt die API eine einheitliche Schnittstelle zum Aufrufen von verschiedenen Texterkennungsanwendungen bereit.

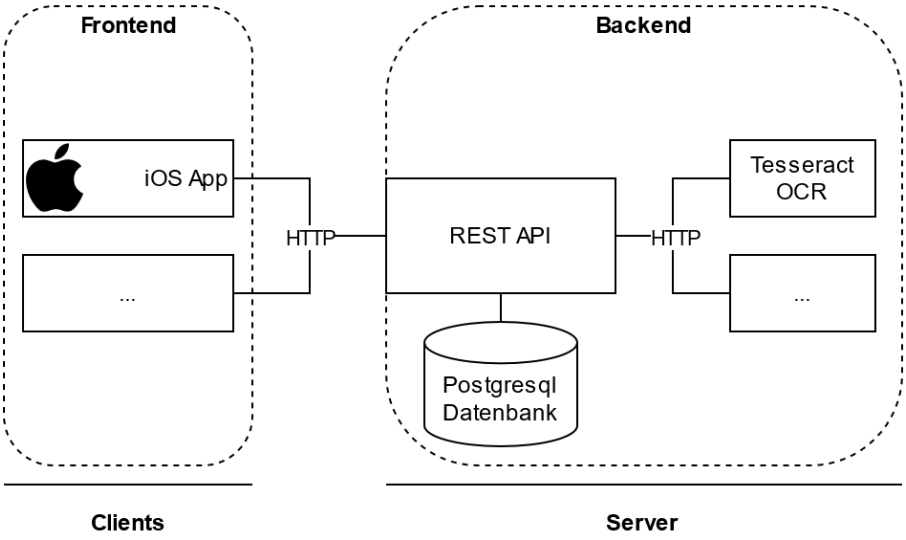


Abbildung 2.2: Architekturdiagramm

Über die Texterkennungsanwendungen können verschiedene Ansätze zur Erkennung von Angaben auf Bildern umgesetzt werden. Jede Anwendung muss einen Webserver implementieren, welcher von Backend aufgerufen wird. Jede Anfrage enthält das zu verarbeitende Bild und Informationen zur Vorlage mit welcher die Erkennung abgearbeitet werden soll. Die Antwort der Anfrage soll die erkannten Werte enthalten. Als Prototyp zur Demonstration der Schnittstelle ist der Tesseract OCR Bibliothek als Webserver im Praxisprojekt implementiert worden.

Die Architektur ist auf Erweiterbarkeit ausgelegt. Neue Frontend-Anwendungen müssen nur die API zum Abrufen und Modifizieren von Daten benutzen. Neben der Kommunikation mit der API ist nur das Anzeigen und Interagieren mit den Daten für einen Benutzer zu implementieren. Dasselbe gilt für das Hinzufügen neuer Texterkennungsansätze. Neue Anwendungen müssen das Anfrage- und Antwortformat der Erkennungsschnittstelle implementieren. Beide Schichten sind vom Backend vollkommen unabhängig. Die einzigen Voraussetzungen an Anwendungen sind, dass sie mit dem JSON Format umgehen können und das HTTP Protokoll implementieren.

Zum Ausführen der Texterkennung muss dem System erst bekannt sein, wo sich auf einem Dokument zu scannende Angaben befinden. Dafür kann ein Benutzer Vorlagen erstellen. Anhand eines Bildes des Dokuments markiert ein Benutzer alle Bereiche, in denen die Texterkennung Angaben extrahieren soll. Außerdem wird in jedem Bereich das erwartete Datenformat, wie Seminargruppe oder Note, angegeben.

Abbildung 2.3 zeigt ein Beispiel einer Vorlage anhand eines Klausurdeckblatts mit markierten Bereichen.

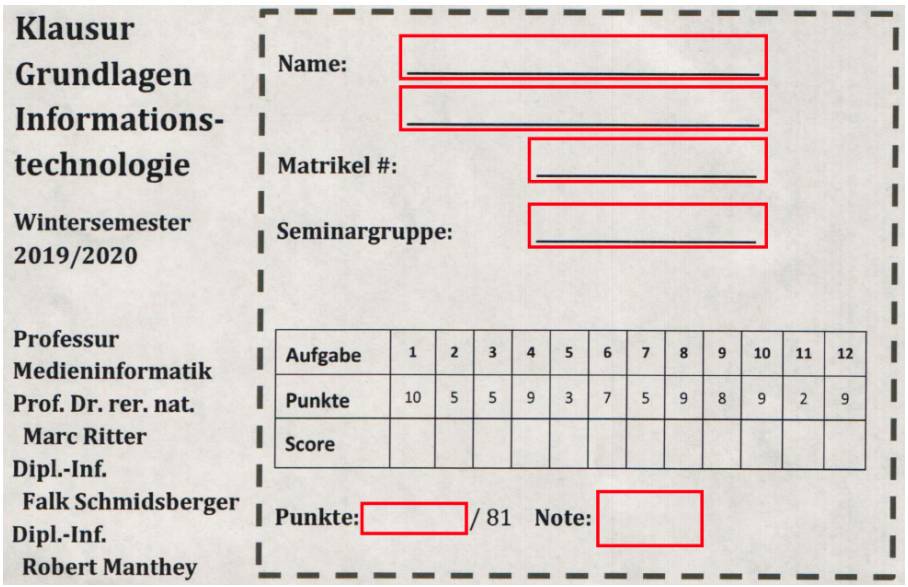


Abbildung 2.3: Beispiel einer Vorlage anhand eines Klausurdeckblatts mit markierten Bereichen

Die Funktionalität und Umsetzung der REST API sowie der Texterkennungsanwendungen ist im Praktikumsbericht des Autors beschrieben [10]. Die Funktionen und Entwicklung der iOS App ist im Praktikumsbericht von Hannes Steiner detailliert dargestellt [18].

3 Anforderungen und Konzept des Prozesses zur Notenmeldung

Die Zielstellung ist die Messung der Effektivität und Benutzbarkeit der Klausurnotenerfassung mit dem App-basierten Prozess. Dafür wird die Verwendung der App durch einen praxisnahen Vergleich mit dem bestehenden Hochschulsystem gemessen.

Für den Vergleich ist die Klausurnotenerfassung für den App-basierten Ansatz umzusetzen. Daher ist der Ablauf einer Noteneintragung mit dem vorhandenen System im Folgenden beschrieben und die vom Anwender auszuführenden Schritte im Detail dargestellt.

Außerdem erlaubt das bestehende Webfrontend der Hochschule keine Möglichkeit zur Datenaufzeichnung. Deshalb wurde entschieden, dieses nachzubilden. Die Nachbildung soll alle für eine Noteneintragung relevanten Funktionen beinhalten, sodass der gesamte Notenmeldungsprozess aufgezeichnet werden kann.

Im Hochschulsystem existieren folgende grundlegende Strukturelemente: Studenten, Seminargruppen, Module, Kurse und Lehrveranstaltungen. In Abbildung 3.1 sind diese und deren Beziehungen dargestellt. Ein Student gehört zu mindestens einer Seminargruppe. Im Modulhandbuch ist festgelegt, welche Module für jedes Semester für einen Studiengang, und somit für welche Seminargruppen, vorgesehen sind. Ein Kurs besteht aus einem oder mehreren Modulen und ein Student kann sich in einen Kurs einschreiben, wenn laut Modulplan eines der Module im aktuellen Semester für eine seiner Seminargruppen verfügbar ist. Die Lehrveranstaltungen sind einem Kurs zugeordnete Termine wie z. B. Vorlesungen, Seminare und Prüfungen.

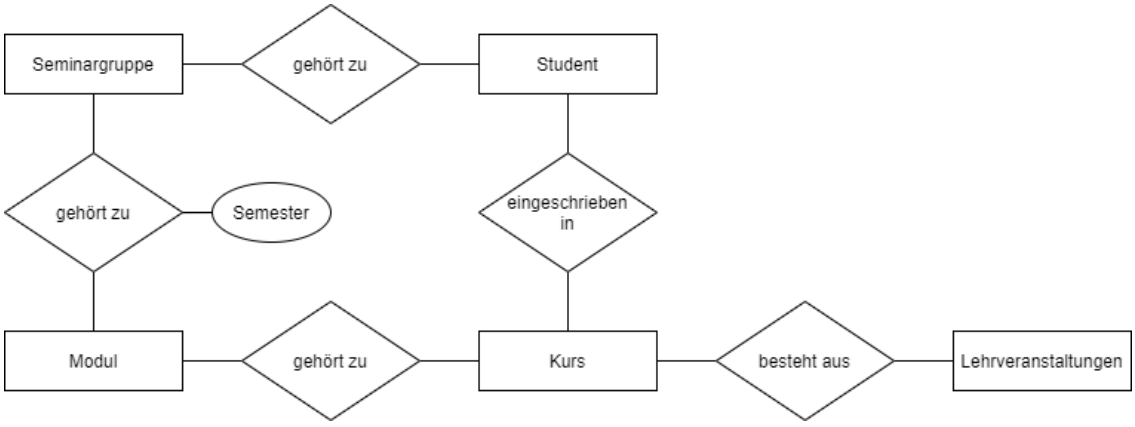


Abbildung 3.1: Business Model der Struktur des Hochschulsystems

3.1 Ablauf der Noteneintragung im Dozentenportal

Das Eintragen einer Note für einen Prüfer besteht aus der Auswahl des zur Klausur gehörenden Kurses, dem Suchen des Studenten in der Teilnehmerliste, die alle in den Kurs eingeschriebenen Studenten beinhaltet, und dem Eintragen des Klausurergebnisses. Falls ein an der Klausur teilnehmender Student nicht in den Kurs eingeschrieben ist, muss die Einschreibung manuell durchgeführt werden. Im Folgenden ist beschrieben, wie dies mit dem bestehenden System der Hochschule auszuführen ist.

Nach erfolgreicher Anmeldung im Intranet der Hochschule mit Benutzername und Passwort hat der Prüfer Zugriff auf das Dozentenportal. Unter „Meine Kurse“ findet dieser eine Liste an Kursen, in denen er als Dozent oder Prüfer eingetragen ist. Ein Screenshot dieser Liste ist in Abbildung 3.2 abgebildet.

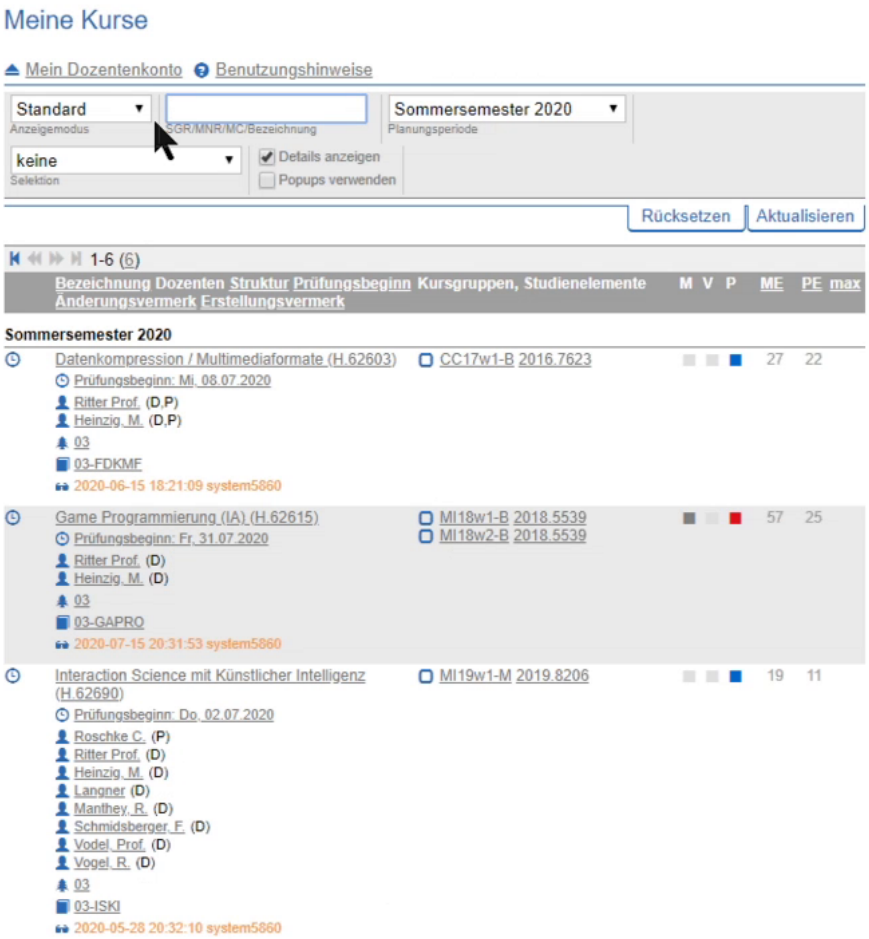


Abbildung 3.2: Kursliste

Durch Auswahl des gewünschten Kurses wechselt die Ansicht zu einer Detailansicht. Hier werden generelle Informationen über den Kurs angezeigt, wie z. B. die eingetragenen Dozenten, Termine und teilnehmende Kursgruppen. Unter dem Abschnitt „Status“ befinden sich der Status zur Modul- und Prüfungseinschreibung sowie die Anzahl an Einschreibungen in das Modul und die Prüfung. Der Abschnitt „Workflow“ dient zur Eintragung von Erst- und Zeitprüfer sowie zur Autorisierung und Freigabe der gemeldeten Noten. Abbildung 3.3 zeigt ein Beispiel der Detailansicht für einen gewählten Kurs.

Abbildung 3.3: Kursansicht

Weitere Informationen und Funktionalitäten können über die sieben Reiter angezeigt werden. Der Reiter „Zusammenfassung“ ist nach dem Wechsel zu dieser Seite ausgewählt und zeigt die oben genannten Inhalte. Unter „Veranstaltungen“ sind die zum Kurs gehörenden Termine für Lehrveranstaltungen und Prüfungen zu finden. Der Reiter „Teilnehmer“ enthält eine Liste aller in den Kurs eingeschriebenen Studenten. Das manuelle Einschreiben von eventuell fehlenden Studenten ist im Reiter „Einschreibung“ möglich.

Das Eintragen der Ergebnisse erfolgt unter dem Reiter „Leistungen“. In jeder Zeile der Leistungstabelle sind Seminargruppe, Modulnummer, Matrikelnummer, voller Name, Prüfungsversuch und Prüfungsdatum für jeden in das Modul eingeschriebenen Studenten abgebildet. Das Eintragen des Prüfungsstatus und der Note erfolgt über die beiden Felder auf der rechten Seite in jeder Zeile. Die Sortierung der Tabelleneinträge kann durch Klicken auf die Spaltenüberschriften Seminargruppe, Modulnummer, Matrikelnummer und Studentenname geändert werden. Abbildung 3.4 bildet ein Screenshot der Ansicht in diesem Reiter ab.

Die Eingabe des Prüfungsstatus eines Teilnehmers erfolgt über die Auswahl des Status in einem Dropdown-Menü. Der Standardstatus ist „unbekannt“. Die Auswahlmöglichkeiten sind:

- unbekannt
- anwesend
- bestanden
- nicht bestanden
- unentschuldigt
- Täuschung
- Fristablauf
- entschuldigt
- krank
- nicht zugelassen

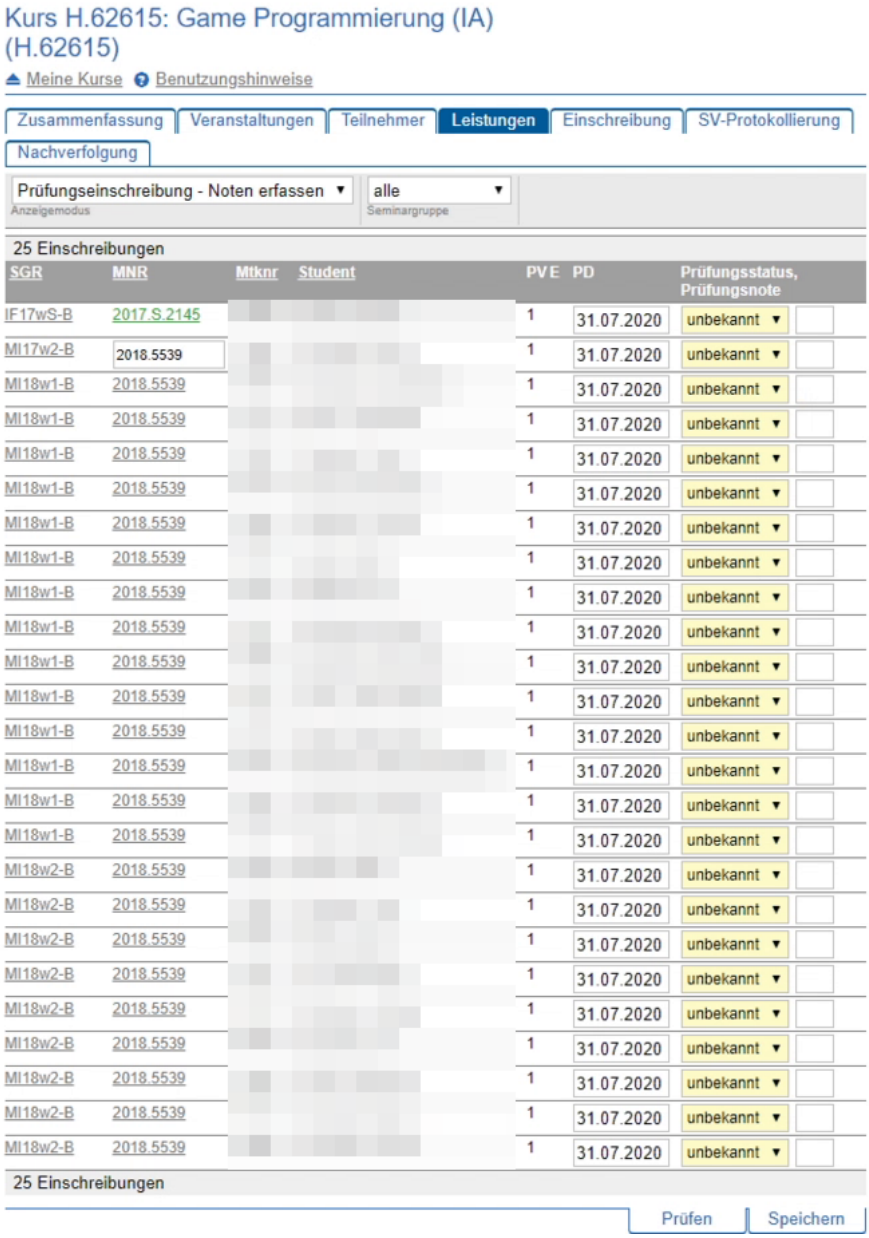


Abbildung 3.4: Leistungstabelle

Das Noteneingabefeld akzeptiert alle Werte zwischen 1,0 und 5,0. Noten mit Kommastelle müssen mit Komma geschrieben werden. Ein Punkt als Dezimaltrennzeichen wird nicht akzeptiert.

Die Validierung und Speicherung der Eingaben erfolgt mit den dafür zuständigen Knöpfen unter der Tabelle. Durch die Validierung werden alle Eingaben geprüft und Notenfelder deren Werte nicht dem akzeptierten Format entsprechen, werden als fehlerhaft markiert. Außerdem wird in Zeilen in dem der Status „nicht bestanden“, „unentschuldigt“, „Täuschung“ oder „Fristablauf“ entspricht automatisch der Wert „5,0“ in das Noteneingabefeld eingetragen. Abbildung 3.5 zeigt die Leistungstabelle nach der Ausführung der Validierung. Nach dem eventuellen Korrigieren von fehlerhaften Eingaben können diese gespeichert werden.

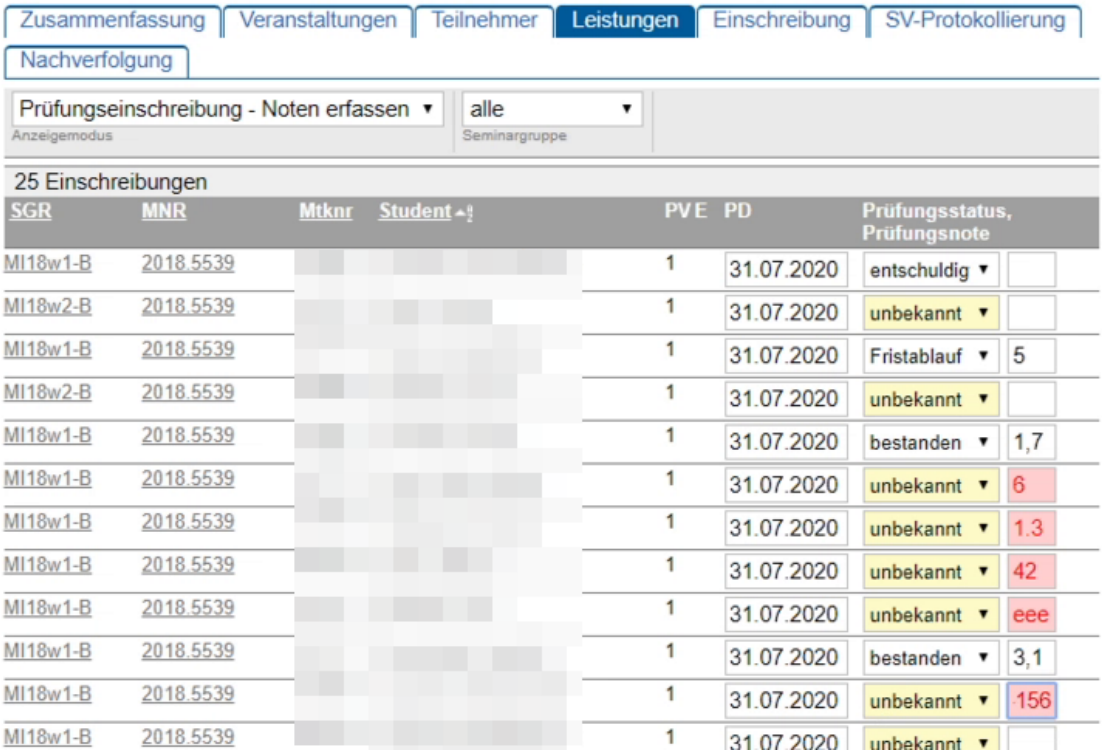


Abbildung 3.5: Validierung der Eingaben

3.2 Ausgeleitete Anforderungen für die Erweiterung des Systems

Aus den oben genannten Anforderungen werden im Folgenden die ausgeleiteten Anforderungen an die Erweiterung des Systems um die Klausurnotenerfassung sowie das Konzept der Umsetzung im Webfrontend und iOS App beschrieben.

Da für diese Arbeit nur die Notenmeldung relevant ist, werden nicht alle Elemente des Hochschulsystems benötigt. Stattdessen sind ausgewählte Anforderungen reduziert in dieser Erweiterung umgesetzt.

Nicht abgebildet werden Veranstaltungen und weitere nicht mit der Notenmeldung zusammenhängende Funktionen des Dozentenportals. Dies gilt auch für Module und Kurse. Stattdessen werden vereinfacht Klausuren abgebildet, in denen Studenten eingeschrieben sind und Ergebnisse gemeldet werden.

Bis auf diese Ergebnisse sind alle abgebildeten Daten statisch und somit nicht veränderbar. Für die Demonstration des Systems besteht die Datenbank aus einer vordefinierten Datenmenge, welche die benötigten Klausuren, Studenten und Einschreibungen enthält. Außerdem werden keine Zugriffskontrollen oder Berechtigungen implementiert. Jeder angemeldete Benutzer kann auf alle Klausuren zugreifen und Ergebnisse verändern.

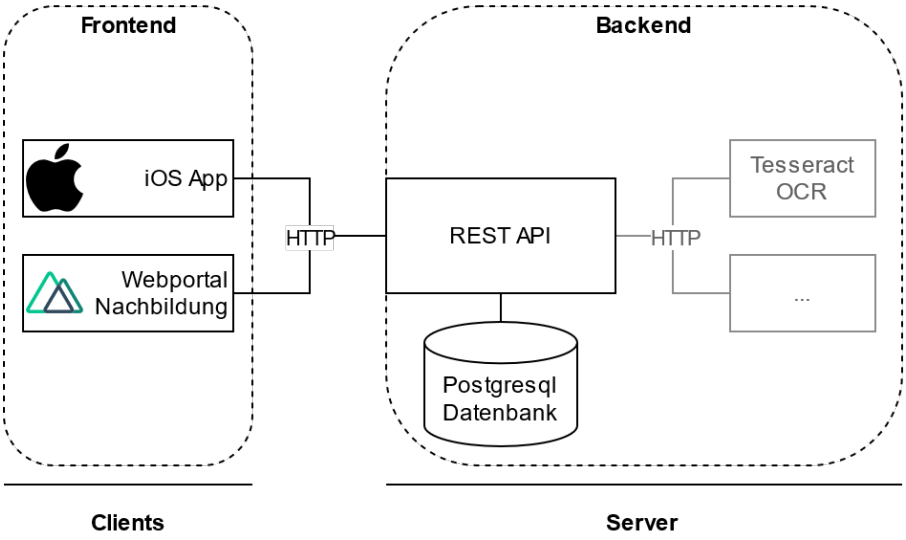


Abbildung 3.6: Architekturdiagramm

Im Vergleich zum Stand der Projektarchitektur aus dem Praxismodul, wie in Kapitel 2 beschrieben, wurden die einzelnen Komponenten wie folgt erweitert.

Das Backend ist nun zusätzlich zuständig für die Bereitstellung von Klausuren, Studenten und Einschreibungen sowie für die Verwaltung von Ergebnissen. Für die iOS App wurde ein an den automatisierten Ansatz angepasster Notenmeldungsprozess entworfen und implementiert. Die Nachbildung des Dozentenportals ist als neue Komponente in dieser Architektur für die Präsentation und Interaktion der aus dem Backend abgerufenen Daten, genau wie die App, verantwortlich.

Die erweiterte Architektur ist in Abbildung 3.6 dargestellt.

3.3 Notenmeldungsprozess der iOS App

Die Erweiterung um die Notenmeldungsfunktionalitäten erlaubt es einem Prüfer durch das in Abschnitt 2.2 beschriebene Vorlagensystem eine Klausur zu scannen. Durch die automatisch von der Texterkennung extrahierten Werte kann die Klausur einem Studenten zugeordnet und das Ergebnis abgesendet werden.

Der Entwurf und die Implementation der App-basierten Notenmeldung wurde von Hannes Steiner umgesetzt. Wie ein Prüfer mit der iOS App Noten meldet, ist in diesem Abschnitt beschrieben. Die Beschreibung nimmt an, dass bereits eine Vorlage für die zu scannenden Klausuren erstellt wurde. Das Erstellen und Benutzen von Vorlagen ist im Praktikumsbericht von Hannes Steiner dargestellt [18].

Nach der Anmeldung mit E-Mail und Passwort hat der Prüfer Zugriff auf die Liste aller Vorlagen. Zu jeder Vorlage gehört genau eine Klausur. Da die Texterkennung die Vorlagen zur automatischen Erfassung der Angaben benötigt, erfolgt die Auswahl der Klausur in der App über die Vorlagen. Diese Liste ist in Abbildung 3.7 abgebildet.

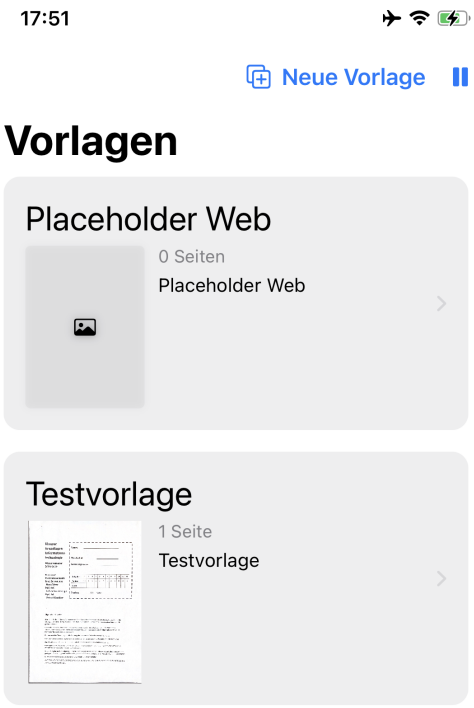


Abbildung 3.7: Klausurenliste

Durch Auswahl einer Vorlage kann der Prüfer wie in der Abbildung 3.8 gezeigten Ansicht sehen, wie viele Klausuren bereits bewertet wurden. Mit „Überprüfen“ kann in die in Abbildung 3.9 zu sehende Ansicht gewechselt werden, in der alle in die Klausur eingeschriebenen Studenten und deren Ergebnis einsehbar sind. Über den „Scannen“ Knopf in der Vorlagenansicht wird die Kamera zum Scannen der Klausur aktiviert.

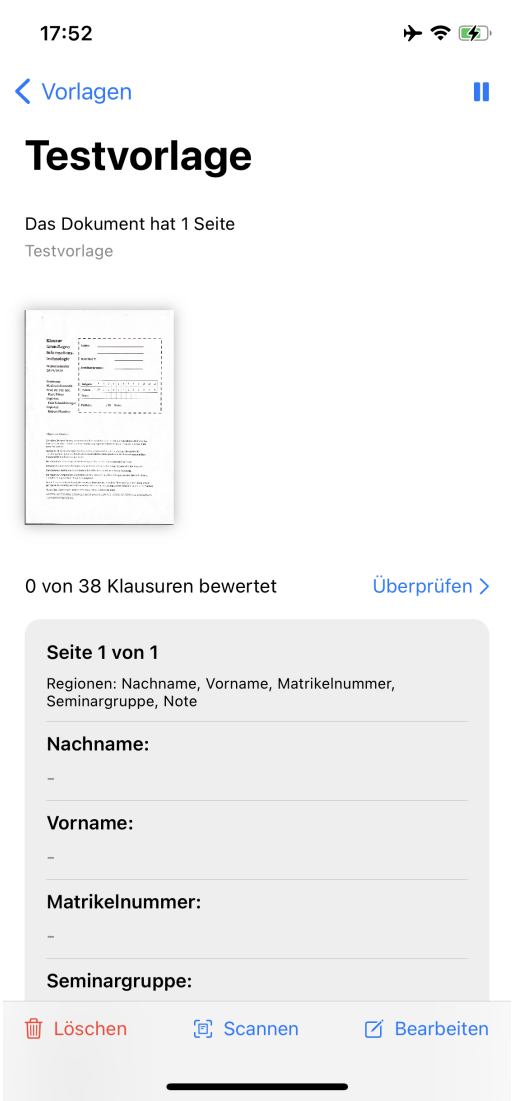


Abbildung 3.8: Klausuransicht

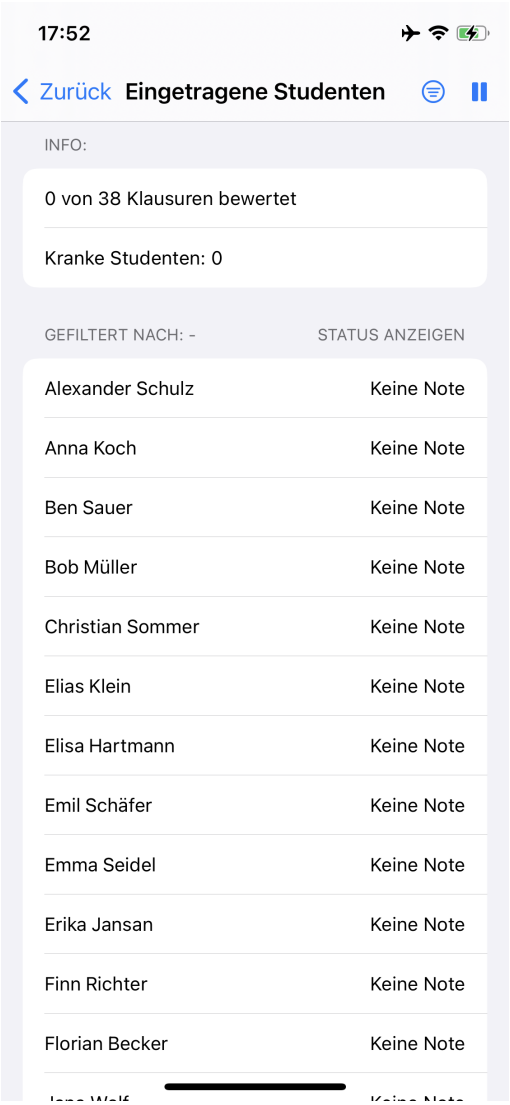


Abbildung 3.9: Studenten-/Ergebnisliste

Abbildung 3.10 zeigt die Ansicht der Texterkennungsergebnisse, nachdem eine Klausur gescannt wurde. Hier kann sich der Prüfer die durch die Texterkennung ausgelesenen Werte ansehen und eventuell korrigieren. Über den „Ergebnisse abgleichen“ Knopf kann das Ergebnis einem Studenten zugeordnet und dann gespeichert werden, wie in Abbildung 3.11 zu sehen ist. Dieser Schritt ist der entscheidende Unterschied im Vergleich zum Web-basierten Ablauf. Statt den Studenten selbst in der Leistungstabelle zu suchen, wird die Zuordnung des Ergebnisses zu einem Studenten bereits automatisch vorgeschlagen und muss nur bei Fehlern korrigiert werden.

In Abbildung 3.12 ist die Benutzung als Ablaufdiagramm dargestellt.

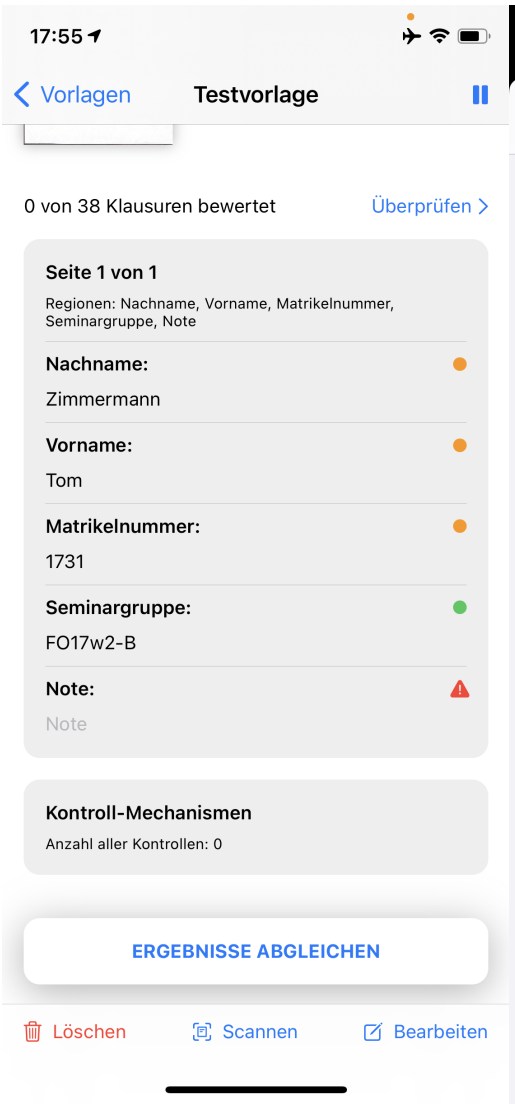


Abbildung 3.10: Texterkennungsergebnis

Abbildung 3.11: Ergebniszuordnung

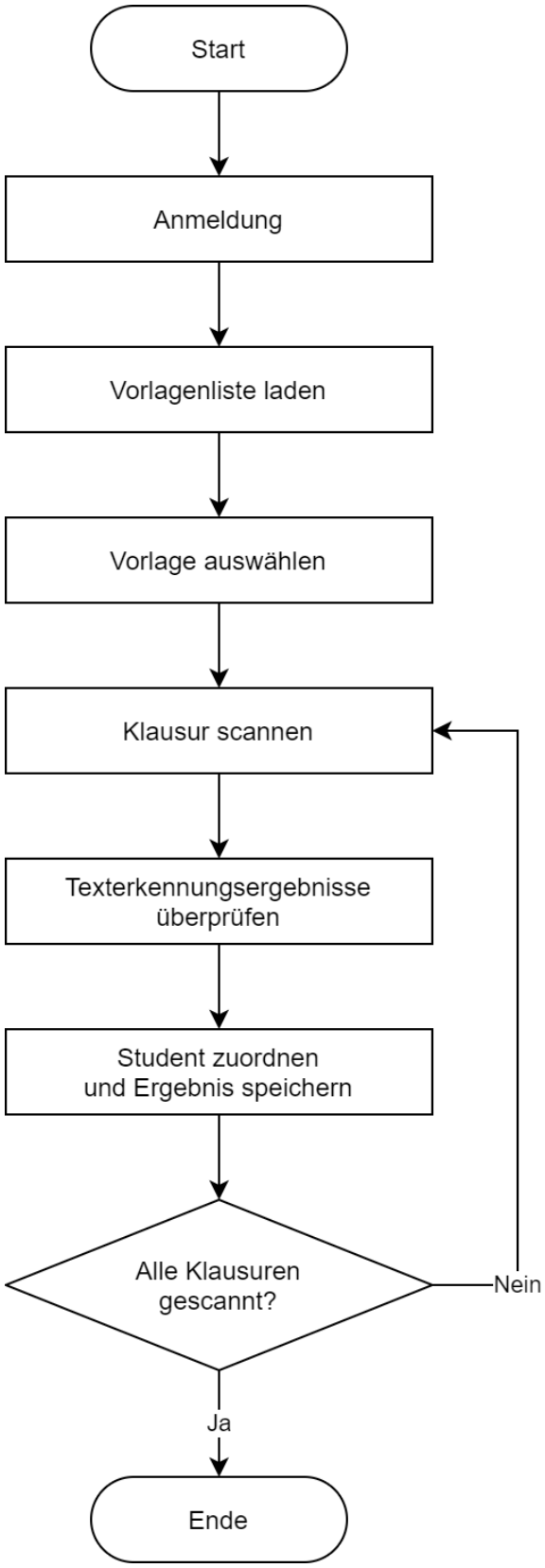


Abbildung 3.12: Ablaufdiagramm zur Notenmeldung mit der App

3.4 Nachbildung des Dozentenportals

Das in Kapitel 3 beschriebene Dozentenportal ist nicht für Testzwecke verfügbar. Um die Aufzeichnung der Benutzerinteraktionsdaten für die Web-basierten Notenmeldung zu ermöglichen, muss das Portal der Hochschule nachgebildet werden. Dieser Abschnitt beschreibt wie die Notenmeldung mit den oben genannten Vereinfachungen in der Nachbildung abläuft.

Der Prüfer beginnt, ähnlich wie im Dozentenportal, mit der Auswahl der Klausur. Die Klausurenliste zeigt alle verfügbaren Klausuren. Eine Anmeldung, um auf die Klausuren zuzugreifen, wird in der Nachbildung nicht benötigt.

In der Klausurdetailansicht sind nur zwei der sieben Reiter umgesetzt. „Veranstaltungen“ sind für die Notenmeldung nicht relevant und werden nicht abgebildet. Gleiches gilt für die für Einschreibungen und Teilnehmer verantwortlichen Reiter, da diese nicht veränderbar sind. Unter „Zusammenfassung“ sind die wenigen vom System abgebildeten Informationen über die Klausur verfügbar. Der Workflow Abschnitt ist hier im Vergleich zum Original nicht umgesetzt, da der Fokus der Nachbildung auf der Leistungstabelle liegt. Diese ist im zugehörigen Reiter zu finden und in Abbildung 3.13 dargestellt.

Wie im Original enthält die Tabelle die Seminargruppe, Matrikelnummer, Name, Prüfungsversuch, Prüfungsdatum sowie Eingabefelder für den Prüfungsstatus und -note für jeden Studenten. Die Sortierung ist ebenfalls nach Seminargruppe, Matrikelnummer und Name möglich.

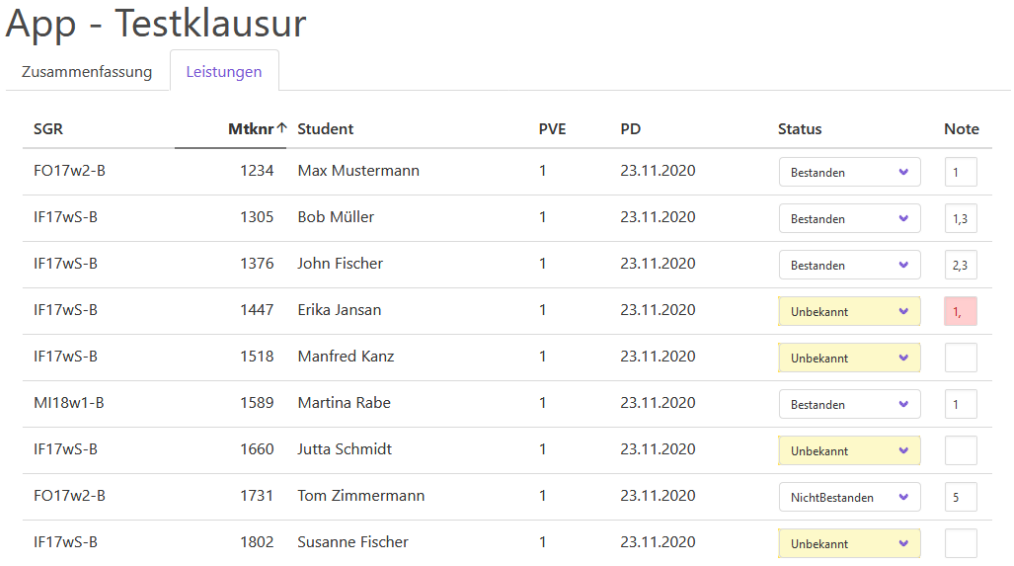


Abbildung 3.13: Ergebnistabelle

Das Prüfen und Speichern ist identisch umgesetzt. Durch den „Prüfen“ Knopf werden die Eingaben validiert, Fehler gekennzeichnet und eventuell Werte vervollständigt. Wenn alle Eingaben korrekt sind, können diese mit dem „Speichern“ Knopf abgesendet werden.

In Abbildung 3.14 ist der Notenmeldungsprozess als Ablaufdiagramm dargestellt.

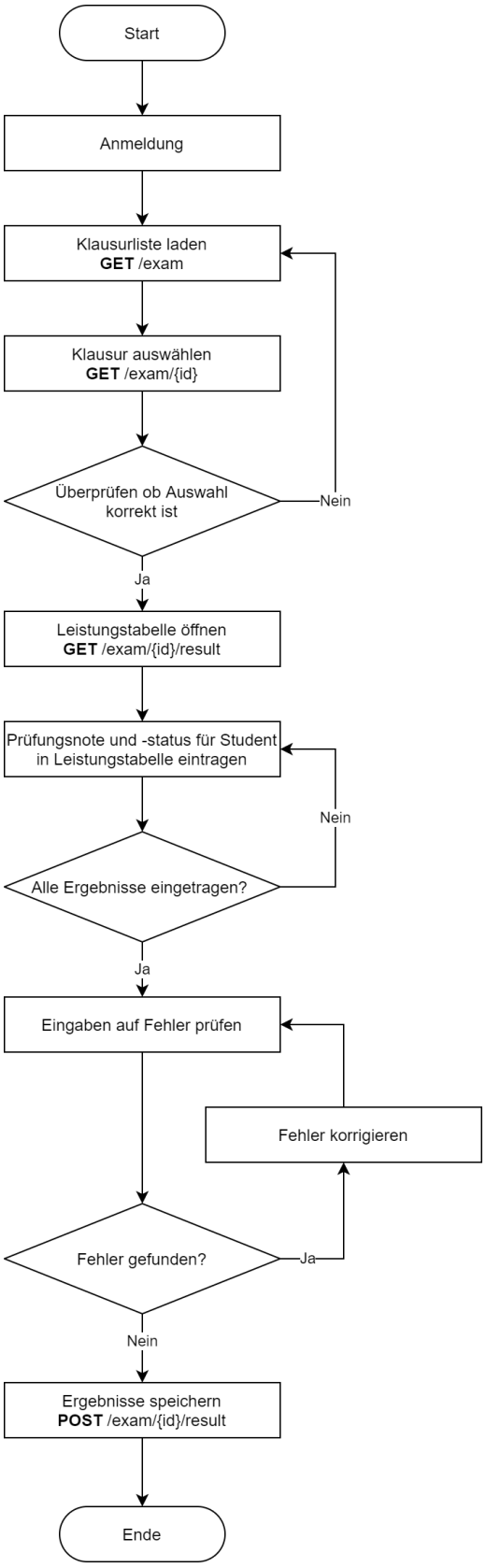


Abbildung 3.14: Ablaufdiagramm zur Notenmeldung mit Nachbildung

3.5 Erweiterung des Backends

Zur Umsetzung der Notenmeldung in der App und der Nachbildung muss das Backend erweitert werden, um die Implementation des Notenmeldungsprozesses in Clients zu ermöglichen. Es ist für das Bereitstellen und Speichern der zusätzlich benötigten Ressourcen zuständig. Diese bestehen aus Klausuren, Studenten und deren Seminargruppe, Einschreibungen und Prüfungsnoten sowie Prüfungsstatus der Studenten.

In Abbildung 3.15 ist die erweiterte Datenbankstruktur als UML-Diagramm dargestellt. Die grau markierten Entitäten Vorlage, Seite und Region sind Teil des im Praxismodul entwickelten Vorlagensystems zur automatischen Texterkennung in Kapitel 2 beschrieben.

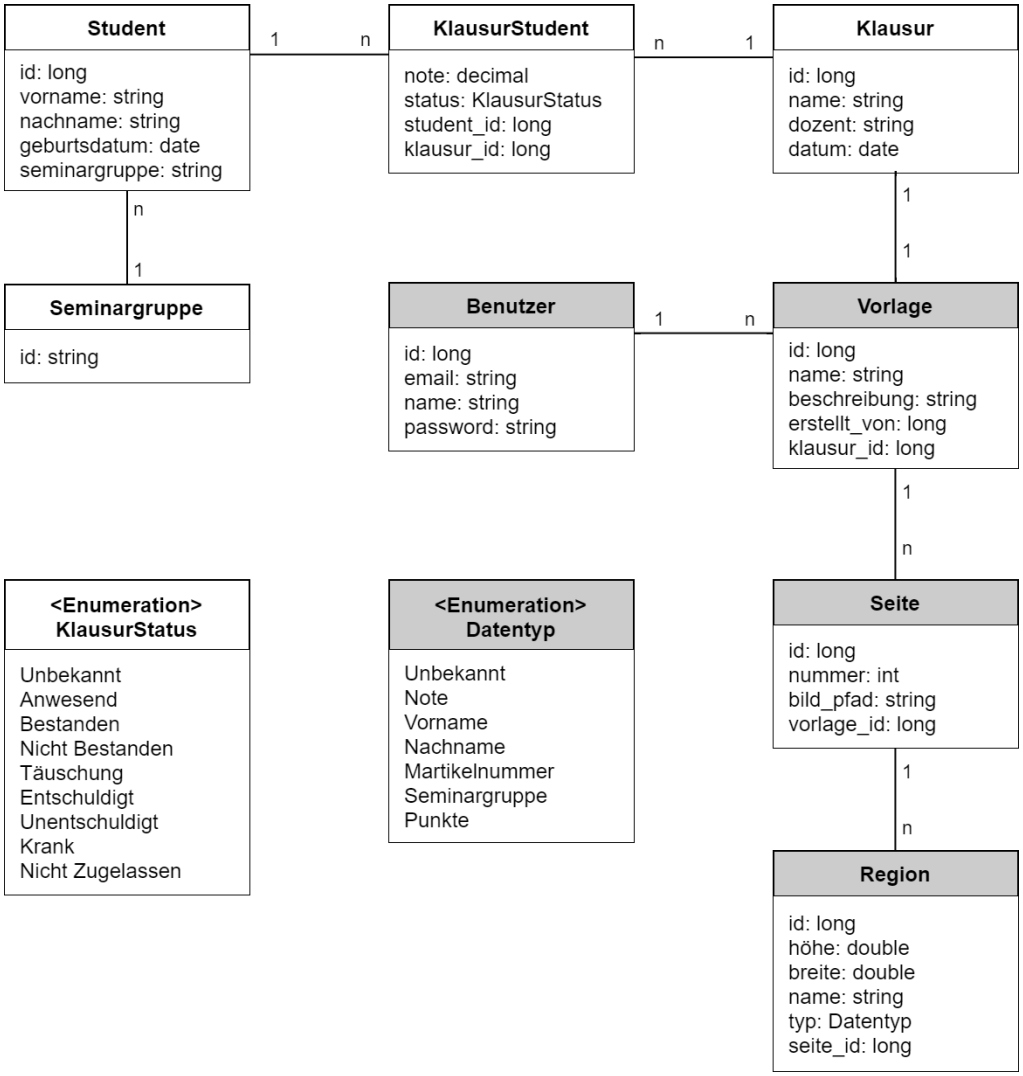


Abbildung 3.15: UML-Diagramm der Datenbankentitäten

Einschreibungen und Ergebnisse von Studenten sind zusammen in der „KlausurStudent“ Entität abgebildet, da für jeden in einer Klausur eingeschriebenen Studenten auch ein zugehöriges Ergebnis existiert.

Die zusätzlichen REST Endpunkte die vom Backend zum Abrufen und Verändern der Daten zur Verfügung stehen, sind im Folgenden beschrieben. In den Routen ist *{id}* ein dynamischer Parameter für den Primärschlüssel der gewünschten Klausur. Datumsangaben in Anfragen und Antworten der API sind im nach ISO 8601 definierten Format repräsentiert.

Das Abrufen der Klausuren erfolgt über den */exam* Endpunkt. Über */exam/{id}* können die Informationen einer einzelnen Klausur abgerufen werden.

Alle in eine Klausur eingeschriebenen Studenten, kombiniert mit deren Ergebnissen, können über den */exam/{id}/result* Endpunkt abgerufen werden. Das Verändern des Ergebnisses eines Studenten erfolgt per POST Anfrage an denselben Endpunkt. Die Anfrage enthält die Matrikelnummer des Studenten dessen Ergebnis verändert werden soll, sowie die gewünschte Note und den Prüfungsstatus.

Eine Übersicht dieser Endpunkte mit Anfrage- und Antwortformat ist in Abbildung 3.16 abgebildet.

Es werden keine Endpunkte zum Erstellen oder Bearbeiten von Studenten, Seminargruppen oder Klausuren implementiert. Stattdessen werden für einen Demonstrationsdurchlauf definierte Daten in die Datenbank vorgeladen.

Klausuren auflisten
GET /exam
Antwortformat

```
[
  {
    "id": 0,
    "name": "string",
    "owner": "string",
    "date": "2019-08-24T14:15:22Z",
    "templateId": 0
  }
]
```

Klausur abrufen
GET /exam/{id}
Antwortformat

```
{
  "id": 0,
  "name": "string",
  "owner": "string",
  "date": "2019-08-24T14:15:22Z",
  "templateId": 0
}
```

Studenten und zugehörige Ergebnisse einer Klausur abrufen
GET /exam/{id}/result
Antwortformat

```
[
  {
    "id": 0,
    "firstname": "string",
    "lastname": "string",
    "birthday": "2019-08-24T14:15:22Z",
    "seminarGroup": "string",
    "grade": 0,
    "points": 0,
    "status": "string"
  }
]
```

Klausurergebnis eines Studenten bearbeiten
POST /exam/{id}/result
Anfrageformat

```
{
  "studentId": 0,
  "grade": 0,
  "status": "string"
}
```

Abbildung 3.16: Übersicht der API Endpunkte des Backends

4 Konzept zur Datenerfassung

Dieses Kapitel beschreibt die Anforderungen und das Konzept für die Entwicklung eines Systems zur Datensammlung. Mit den Daten soll der Vergleich der Benutzbarkeit von verschiedenen Prozessen möglich sein. Speziell in dieser Arbeit handelt es sich um den Vergleich des normalen Notenmeldungsprozesses des Dozentenportals der Hochschule mit dem automatisierten App-basierten Ansatz.

Die Datenerfassung soll generisch sein, um sie auf verschiedenen Plattformen und für verschiedene Prozesse umzusetzen. Der aufzuzeichnende Prozess muss detailliert genug repräsentiert werden, um die Analyse des Prozesses anhand der Daten zu ermöglichen. In den Daten soll die Benutzung des zu messenden Systems von einzelnen Nutzern sichtbar sein.

Ein erster Ansatz, ein existierendes Web-Analytics Framework zu benutzen, würde diese Anforderungen nicht erfüllen. Die durch Frameworks dieser Art gesammelten Daten, wie z. B. welcher Anteil an Besuchern einer Seite das Anlegen eines Kontos abschließen, sind zu stark generalisiert und würden Prozesse nicht genug abbilden, um eine Analyse zu ermöglichen.

Da ein Analytics Framework die geforderten Daten nicht liefern würde, ist für die Realisierung der Datenerfassung ein eigenes System konzeptioniert und implementiert. Das Konzept dieses Systems ist im Folgenden beschrieben.

Die Aufzeichnung eines Prozesses besteht aus verschiedenen Ereignissen, über die der Ablauf eines Prozesses dargestellt wird. Die Ereignisse werden in Sitzungen gesammelt. Eine Sitzung enthält alle Ereignisse, die zwischen Beginn und Ende der Sitzung auftreten. Ein Ereignis wird relativ zum Beginn der Sitzung gespeichert, sodass sich das in Abbildung 4.1 dargestellte Format ergibt.

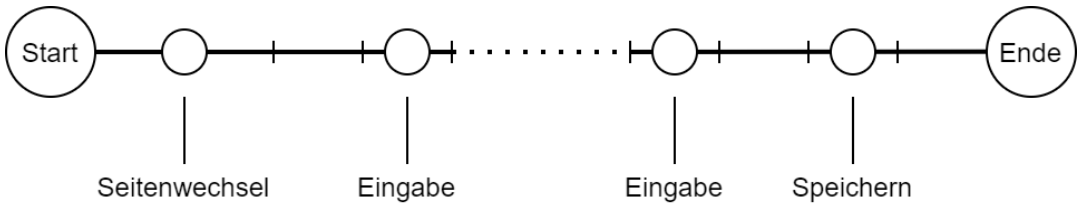


Abbildung 4.1: Schematische Darstellung einer Aufzeichnungssitzung

Eine Sitzung besteht aus dem Namen der Plattform, wie z. B. „web“ oder „app“, dem Zeitpunkt, wann die Sitzung angefangen und geendet hat, dem Benutzer, von dem die Durchführung des Prozesses gemessen wurde, und den Ereignissen, die aufgezeichnet wurden. Das Ereignisformat ist stark generalisiert, um so viele verschiedene Ereignisse wie möglich aufzeichnen zu können. Sie bestehen aus einem Namen des Ereignisses, wie „Eingabe“ oder „Scroll“, der vergangenen Zeit zum Anfang der Sitzung, der Dauer des Ereignisses und einem variablen Datenfeld. Dieses kann beliebige Name-Wert-Paare enthalten, um weitere Informationen über ein Ereignis abzubilden. Die beiden Zeitangaben werden in Millisekunden angegeben. In Abbildung 4.2 ist die Datenstruktur als UML-Diagramm dargestellt.

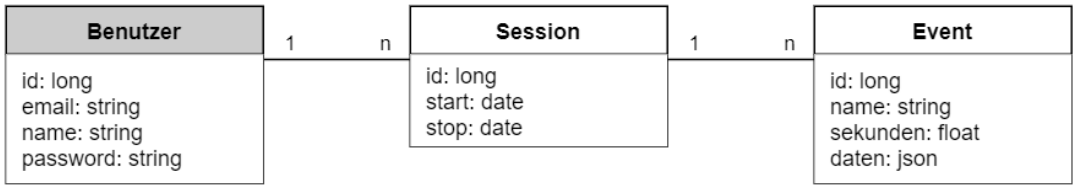


Abbildung 4.2: UML-Diagramm der Datenbankentitäten

Zur späteren Weiterverarbeitung und Analyse der Daten stellt das Backend Endpunkte zum Abrufen der Sitzungen bereit. Die Daten sind im JSON Format serialisiert und somit in den meisten Programmiersprachen ohne viel Aufwand verarbeitbar. Die Nachbildung benutzt diese Endpunkte, um eine Ansicht zum Einsehen der gesammelten Daten für den Benutzer bereitzustellen.

4.1 Ereignisse und Ablauf der Nachbildung

Für die Aufzeichnung der Benutzung der Nachbildung sind drei verschiedene Ereignisarten vorgesehen: Eingabeereignisse, um die Veränderung von Prüfungsstatus und Note aufzuzeichnen. Die Eingabeereignisse enthalten den alten und neuen Wert. Die Noteneingabe enthält außerdem die Dauer, wie viel Zeit der Benutzer für die Änderung benötigt hat. Die Benutzung der „Prüfen“ oder „Speichern“ Knöpfe werden über eigene Ereignisse repräsentiert.

Außerdem wird das Scrollen der Seite als Ereignis aufgezeichnet. Somit soll der Aufwand des Suchens eines Studenten in der Leistungstabelle gemessen werden. Das Scrollereignis ist zum Teil aggregiert. Mehrere Scrollbewegungen werden zu einem Ereignis zusammengefasst, solange die Richtung nicht wechselt und die einzelnen Bewegungen in kurzer Zeit hintereinander auftreten. Ein Scrollereignis enthält die Dauer in Millisekunden und die Weite in Pixel der zusammengefassten Bewegungen.

In Abbildung 4.3 sind die Strukturen der drei Ereignisarten dargestellt.

```
{
  "name": "SCROLL",
  "time": 1872,
  "duration": 584,
  "data": {
    "distance": 202
  }
}
```


(a) Scrollereignis


```
{
  "name": "INPUT",
  "time": 10291,
  "duration": 2658,
  "data": {
    "type": "NOTE",
    "newValue": "2,3",
    "oldValue": "2,0",
    "studentId": 1376
  }
}
```


(b) Eingabeereignis


```
{
  "name": "BUTTON",
  "time": 15867,
  "duration": 0,
  "data": {
    "type": "PRÜFEN"
  }
}
```


(c) Knopfereignis

Abbildung 4.3: Struktur der Ereignisse der Nachbildung

Es werden keine Navigationsereignisse aufgezeichnet, da die Hauptarbeitslast der Noteneintragung in der Leistungstabelle stattfindet. Andere Schritte, wie die Auswahl des richtigen Kurses, die im Dozentenportal stattfinden, sind in der Nachbildung nicht enthalten.

Ein angemeldeter Nutzer kann die Aufzeichnung über den zugehörigen Knopf in der linken Navigationsleiste starten. Nach Durchlauf der zu messenden Schritte kann die Sitzung über denselben Knopf beendet werden.

Zur Einsicht der aufgezeichneten Daten enthält die Nachbildung zusätzlich zwei Seiten zum Anzeigen der Sitzungen. Abbildung 4.4 zeigt die Listenansicht, die alle Sitzungen enthält. Diese ist über den „Interaktionsdaten“ Link in der Navigationsleiste zu erreichen. Durch Auswahl einer Sitzung wird in die in Abbildung 4.5 dargestellte Detailansicht gewechselt. Hier sind neben den Sitzungsinformationen auch alle Ereignisse der gewählten Sitzung angezeigt.

ID	User	Platform	Start	Ende
10	Test	iOS	07.11.2020 13:44:03	ⓘ
13	Test	iOS	07.11.2020 13:47:51	07.11.2020 13:54:26
15	Test	iOS	08.11.2020 21:52:09	08.11.2020 21:55:24
17	Test	web	16.11.2020 19:31:38	16.11.2020 19:35:23
19	Test	web	16.11.2020 19:52:07	16.11.2020 20:02:48

Abbildung 4.4: Sitzungsliste

Session 13

Test
Start: 07.11.2020 13:47:51
Ende: 07.11.2020 13:54:26

Events

Timestamp ↑	Länge	Typ	Daten
1.367 s	0 s	NAVIGATION	{ "to": "TemplateDetailScreen" }
4.568 s	0 s	NAVIGATION	{ "to": "ScannerScreen" }
4.582 s	0 s	NAVIGATION	{ "to": "TemplateView" }
11.523 s	0 s	NAVIGATION	{ "to": "TemplateDetailScreen" }
26.735 s	0 s	NAVIGATION	{ "to": "ScannerScreen" }
33.795 s	0 s	NAVIGATION	{ "to": "TemplateDetailScreen" }
42.243 s	0 s	INPUT	{ "to": "2", "from": "", "type": "Note" }
42.654 s	0 s	INPUT	{ "to": "2,", "from": "2", "type": "Note" }
42.860 s	0 s	INPUT	{ "to": "2,5", "from": "2,", "type": "Note" }

Abbildung 4.5: Detailansicht einer Sitzung

4.2 Events und Ablauf der iOS App

Die Benutzung der iOS App kann hauptsächlich durch das Wechseln zwischen Ansichten gemessen werden. Jeder Schritt des Notenmeldungsablaufs wird in einer eigenen Ansicht bearbeitet. Deshalb werden die Navigation zwischen Ansichten als Ereignis aufgezeichnet. Die Navigationsereignisse enthalten den Namen der Ansicht, zu der gewechselt wurde. Außerdem werden wie in der Nachbildung Eingaben aufgezeichnet. Die in der App erfolgenden Eingaben sind die eventuellen Korrekturen der Texterkennungsergebnisse und enthalten den alten und neuen Wert jeder Eingabe.

In Abbildung 4.6 ist die Struktur der beiden Ereignisse dargestellt.


```
{
  "name": "NAVIGATION",
  "time": 1367,
  "duration": 0,
  "data": {
    "to": "TemplateDetailScreen"
  }
}
```


(a) Navigationsereignis


```
{
  "name": "INPUT",
  "time": 42243,
  "duration": 0,
  "data": {
    "to": "2,3",
    "from": "2,",
    "type": "Note"
  }
}
```


(b) Eingabeereignis

Abbildung 4.6: Struktur der Ereignisse der iOS App

4.3 Erweiterung der REST API

Wie für die Umsetzung der Notenmeldung auch wird das Backend um die Verwaltung und Speicherung der Sitzungen und Ereignisse erweitert. Dafür werden mindestens Endpunkte zum Erstellen und Beenden einer Sitzung, zum Speichern von Ereignissen und zum Abrufen von Sitzungen und Ereignissen benötigt. Endpunkte zum Löschen oder Verändern von Sitzungen und Ereignissen werden für diese Erstimplementation der Datenerfassung nicht umgesetzt.

Die vier Endpunkte sind im Folgenden beschrieben. `{id}` ist ein dynamischer Parameter und mit dem Primärschlüssel der Sitzung, auf der die Funktion des Endpunktes ausgeführt werden soll, zu ersetzen.

Das Erstellen und Starten einer Sitzung erfolgt über eine `POST` Anfrage an die `/session` Endpunkt. Die Anfrage muss den Namen der Plattform, für die die Sitzung erstellt werden soll, sowie den Anfangszeitstempel der Sitzung enthalten. Die Antwort auf die Anfrage enthält die Id der durch den Aufruf erstellten Sitzung.

Das Beenden einer Sitzung erfolgt ebenfalls über eine `POST` Anfrage den `/session/{id}` Endpunkt. Die Anfrage muss den Zeitstempel zum Ende der Sitzung enthalten.

Mit `POST` Anfrage an den `/session/{id}/event` Endpunkt kann ein Ereignis gespeichert werden. `id` ist hierbei der Primärschlüssel der Sitzung, welcher das Ereignis zugeordnet werden soll. Einer bereits beendeten Sitzung können keine weiteren Ereignisse hinzugefügt werden und eine solche Anfrage würde mit dem HTTP Fehlercode 409 (Konflikt) beantwortet werden. Der Endpunkt ist nur zum Erstellen von einzelnen Ereignissen vorgesehen und muss für jedes zu speichernde Ereignis aufgerufen werden. Ereignisse können direkt, wenn diese auftreten an die API zum Speichern gesendet werden, oder erst von der Anwendung gespeichert und später, wie z. B. beim Beenden der Aufzeichnung durch den Benutzer, abgesendet werden.

Das Abrufen der Sitzungen und Ereignisse ist über GET Anfragen an die `/session` und `/session/{id}` Endpunkte möglich. `/session` antwortet mit einer Liste aller Sitzungen. Der Detailanfrageendpunkt antwortet mit denselben Informationen wie in der Liste, aber nur für eine Sitzung. Außerdem enthält die Antwort alle zu dieser Sitzung gehörenden Ereignisse.

Eine Übersicht der Endpunkte ist in Tabelle 4.1 zu finden. Die Anfrage- und Antwortformate sind in Abbildung 4.7 bis Abbildung 4.12 dargestellt

Funktion	Methode	Route	Anfrageformat	Antwortformat
Sitzungen auflisten	GET	/session		Abbildung 4.11
Sitzung und Ereignisse abrufen	GET	/session/{id}		Abbildung 4.11 und Abbildung 4.12
Sitzung starten	POST	/session	Abbildung 4.7	Abbildung 4.8
Sitzung beenden	POST	/session/{id}	Abbildung 4.9	
Ereignis erstellen	POST	/session/{id}/event	Abbildung 4.10	

Tabelle 4.1: Übersicht aller Sitzungsendpunkte der API

```
{
  "platform": "string",
  "startedAt": "2019-08-24T14:15:22Z"
}
```

Abbildung 4.7: Anfrageschema zum Erstellen einer Sitzung

```
{
  "id": 0
}
```

Abbildung 4.8: Antwortschema zum Erstellen einer Sitzung

```
{
  "stoppedAt": "2019-08-24T14:15:22Z"
}
```

Abbildung 4.9: Anfrageschema zum Beenden einer Sitzung

```
{
  "name": "string",
  "time": 0,
  "duration": 0,
  "data": {
    "name": "wert",
    "name2": "wert2"
  }
}
```

Abbildung 4.10: Anfrageschema zum Erstellen eines Ereignisses

```
[
  {
    "id": 0,
    "platform": "string",
    "startedAt": "2019-08-24T14:15:22Z",
    "stoppedAt": "2019-08-24T14:15:22Z",
    "user": {
      "id": 0,
      "email": "string",
      "username": "string"
    }
  }
]
```

Abbildung 4.11: Antwortschema der Sitzungsliste

```
{
  "events": [
    {
      "name": "string",
      "time": 0,
      "duration": 0,
      "data": {
        "property1": null,
        "property2": null
      }
    }
  ]
}
```

Abbildung 4.12: Ereignisliste einer Sitzung

5 Umsetzung

Dieses Kapitel beschreibt die Umsetzung der in Kapitel 3 und Kapitel 4 aufgestellten Konzepte. Die Realisierung des Notenmeldungsprozesses und Implementierung der Messung von Benutzerinteraktion ist nach den Systemkomponenten geteilt beschrieben. Als Erstes folgt die Umsetzung der Nachbildung des Dozentenportals als Webanwendung. Anschließend ist die Erweiterung des Backends um die Verwaltung von Klausurergebnissen und Interaktionsdaten erklärt.

5.1 Nachbildung der Klausurnotenmeldung mit Nuxt.js

Für die Realisierung der Nachbildung wird das Javascript Framework Nuxt.js [5] verwendet. Nuxt ist ein auf Vue.js [26] aufbauendes Web-Framework zum Entwickeln von Single-page applications (SPAs).

Single-page applications sind Webanwendungen, die den Seiteninhalt dynamisch verändern. Im Vergleich zu einer klassischen Webseite ist der entscheidende Unterschied hier, dass bei Veränderungen, wie dem Wechsel zu einer anderen Seite oder dem Absenden von Formularen, die Seite vom Server nicht neu geladen werden muss, sondern nur die benötigen Teile der Seite aktualisiert werden.

Vue.js implementiert das SPA Prinzip als Javascript Framework zum Entwickeln von Benutzeroberflächen und Webanwendungen. Es ist auch möglich, nur einzelne Elemente einer Webseite in Vue umzusetzen, ohne dass die gesamte Seite eine Single-page application sein muss.

Nuxt.js wurde für die Nachbildung gewählt, da SPAs sehr gut in die von der Projektarchitektur benutzte Client-Server Aufteilung passen. Eine Nuxt.js Anwendung ist ein eigenständiger Client in dieser Aufteilung. Eine traditionelle Webseite müsste einen Webserver implementieren, der die benötigten Seiten bereitstellt, und die Kommunikation zwischen Browser und Client umsetzt.

Vue und Nuxt haben sich in sowohl diesem Projekt als auch in der Vergangenheit als gute Frameworks zur Realisierung von Webanwendungen mit interaktiven Elementen bewiesen. Anwendungen die nur Daten anzeigen sollen sind mit den Frameworks zwar möglich, aber durch die Nachteile, wie höhere Ladezeit durch Dateigröße und Komplexität, sind sie für statische Seiten nicht zu empfehlen.

Für Seiten mit Interaktivität bringt die Verwendung von Frontend-Frameworks viele Vorteile. Einzelne Seiten einer Webseite ohne Frameworks müssen einerseits vom Server generiert werden, und werden dann erst durch Ausführung von Javascript Code auf Browserseite interaktiv. Frameworks wie Vue haben diese Teilung zwischen der Präsentation der Daten mit HTML plus CSS und der Interaktivität in Javascript nicht. Stattdessen ist die Präsentation der Daten ebenfalls Aufgabe des Javascript Codes.

Eine Vue-Anwendung besteht aus mindestens einer Komponente. Diese können beliebig geschachtelt werden, um eine vollständige Benutzeroberfläche mit wiederverwendbaren Elementen zu erstellen. Die beiden wichtigsten Teile einer Komponente sind die Vorlage und die Daten. Die Vorlage definiert, wie eine Komponente angezeigt wird und wird durch die Daten manipuliert. Die Verknüpfung zwischen Vorlage und Daten erfolgt über verschiedene Möglichkeiten, wie z. B. das Einbinden eines Datenwertes mit Verwendung des dafür bestimmten Syntax: Ein Bezeichner in doppelt geschweiften Klammern wird als Javascript-Anweisung ausgewertet. Wenn dieser Bezeichner auch in den Daten der Komponente existiert, wird dieser Wert eingebunden. Weitere Möglichkeiten sind über Direktiven umgesetzt, wie z. B. die Anzeige eines Elementes unter festgelegten Bedingungen mit *v-if* oder dem Anzeigen einer Liste mit *v-for*. Alle Verknüpfungen sind in Vue reaktiv. Dies bedeutet, dass wenn sich ein Wert verändert, die Template ebenfalls aktualisiert wird. Quellcode 5.1 zeigt ein einfaches Beispiel einer Komponente im Single File Component (SFC) Format. Das SFC Format kombiniert die Definition einer Komponente und der zugehörigen Vorlage sowie CSS Anweisungen in einer Datei.

```
<template>
  <p>Hallo {{ name }}</p>
</template>

<script>
export default {
  data () {
    return {
      name: 'Max'
    }
  }
}
</script>

<style>
p {
  font-size: large;
}
</style>
```

Quellcode 5.1: Beispiel einer Komponente im Single File Component Format

Nuxt.js erweitert die Funktionalitäten von Vue, um die Entwicklung von Single-Page Applications zu vereinfachen. Dies erreicht Nuxt durch die Verwendung einer einheitlichen Projektstruktur, der Einführung von Komponenten als Seiten sowie der automatischen Generation von Routen und weiteren Erweiterungen der Vue Funktionalitäten.

Neben dem Einsatz als SPA unterstützt Nuxt außerdem 2 weitere Methoden: Wenn Server-side-rendering aktiviert ist, wird eine Seite bei der ersten Anfrage zuerst wie eine klassische Seite vom Server aufgebaut und an Client zum Anzeigen gesendet. Anschließend wird die Interaktivität der single-page application nachgeladen. Dies hat den Vorteil das bei der ersten Anfrage eines Clients der Nutzer nicht auf das vollständige Herunterladen der Anwendung warten muss, sondern direkt eine statische Version der gewünschten Seite angezeigt wird. Die vollständige Anwendung ist aber trotzdem nach dem Herunterladen einsatzbereit. Die andere Verwendungsart die Nuxt unterstützt, ist das vollständige Generieren einer statischen Version der Anwendung.

Vuex [27] ist die offizielle State Management Bibliothek für Vue und dient als Implementation eines zentralen Datenspeichers auf den alle Komponenten der Anwendung zugreifen können. Daten in Vuex fließen nur immer in eine Richtung. Das bedeutet, dass Komponenten die Daten im aktuellen Zustand lesen können, aber nicht direkt verändern. Zum Verändern müssen Aktionen ausgelöst werden. Diese sind für das Abrufen der Daten aus anderen Quellen, wie z. B. einer API oder dem Berechnen von Veränderungen zuständig. Um den Zustand zu verändern, müssen von Aktionen Mutationen ausgelöst werden. Die Mutationen definieren die Modifizierung des Zustandes anhand von festgelegten Inputs. Dieser Datenfluss ist in Abbildung 5.1 dargestellt.

In der Nachbildung wird Vuex für das Verwalten der Anmeldeinformationen, der Klausurergebnisse und der Aufzeichnung der Interaktionsdaten verwendet.

Für das Design der Nachbildung wird Bulma [20] als CSS-Framework verwendet. Das Framework enthält eine Sammlung von CSS-Klassen zum Definieren des Layouts von Seiten und Elementen sowie zur Darstellung von ausgewählten Elementen wie z. B. Knöpfen, Tabellen und Eingabefeldern. Die in Bulma enthaltenen Elemente sind allerdings nicht interaktiv. Dafür wird zusätzlich Buefy [21] benutzt. Buefy stellt eine Sammlung von verschiedenen Komponenten für Vue.js bereit, die auf dem Design von Bulma basieren. Die von der Nachbildung benutzen Buefy Komponenten sind z. B. Tabellen mit Sortierung nach vom Benutzer gewählten Spalten.

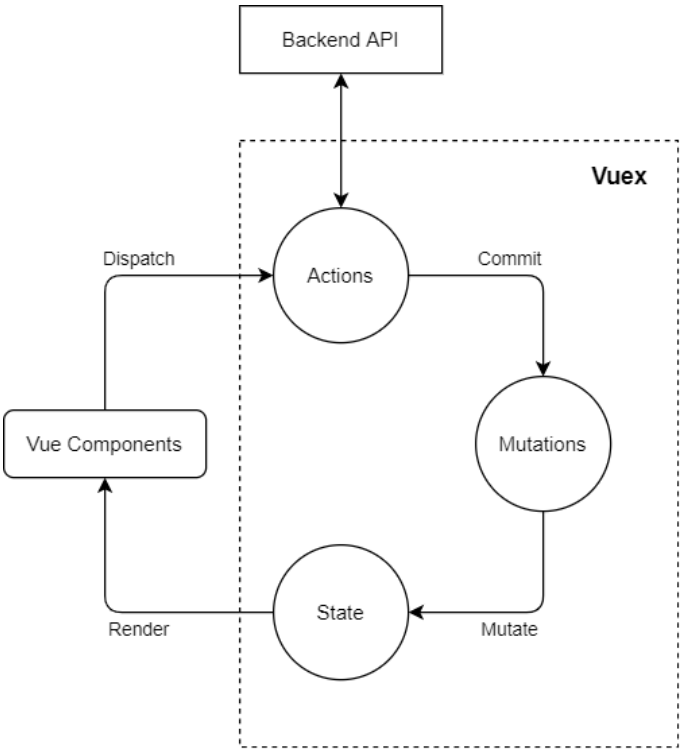


Abbildung 5.1: Schematische Darstellung der Vuex Datenarchitektur

Für die Kommunikation mit der REST API wird die Axios [28] Bibliothek verwendet und stellt einen einheitlichen, erweiterten HTTP [7] Client gegenüber den standardmäßig in Browsern enthaltenen Javascript Funktionen bereit. Eine der von der Nachbildung benutzten Funktionalitäten der Bibliothek ist die Möglichkeit Anfragen vor dem Absenden abzufangen und zu modifizieren. Quellcode 5.2 zeigt den benutzten Interceptor zum automatischen Anfügen des JSON Web Tokens an jede Anfrage, wenn der Nutzer angemeldet ist. JSON Web Tokens (JWT) [9] werden von der API benutzt, um angemeldete Nutzer zu autorisieren.

```
axios.interceptors.request.use((request: AxiosRequestConfig) => {
  // vxm ist Teil des Vuex Stores
  const jwt = vxm.user.jwt
  if (jwt !== null) {
    request.headers.Authorization = `Bearer ${jwt}`
  }
  return request
})
```

Quellcode 5.2: Axios Interceptor zum Anhängen des Autorizierungstokens an jede Anfrage

Nuxt.js und alle benutzen Bibliotheken unterstützen offiziell Typescript [13] als Alternative zu Javascript. Typescript ist eine Erweiterung der Javascript Syntax mit statischer Typisierung. Im Vergleich zu Javascripts schwacher, dynamischer Typisierung hat Typescript den Vorteil, dass der Programmcode bereits während des Kompilierens auf Fehler geprüft werden kann und durch die Angabe von Typen besser dokumentiert ist. Quellcode 5.3 zeigt ein minimales Beispiel der Fehlerprüfung. Der Code ist sowohl valider Javascript als auch Typescript Code, aber im Gegensatz zu Javascript wird die fehlende Eigenschaft `name` von dem Objekt `benutzer` bereits während des Kompilierens erkannt. In Javascript ist dies kein Fehler, sondern `benutzer.name` würde den Wert `undefined` zurückgeben. Das Existieren von Eigenschaften eines Objekts muss in Javascript zusätzlich vom Programmierer geprüft werden.

```
const benutzer = { vorname: "Max", nachname: "Mustermann" }

console.log(benutzer.name)
// Fehler: Eigenschaft 'name'
// existiert nicht auf Typ '{ vorname: string; nachname: string }'
```

Quellcode 5.3: Beispiel der Typescript Fehlerprüfung

Typescript ist eine Erweiterung des Javascript Syntax. Somit ist auch jeder Javascript Quellcode valider Typescript Quellcode, auch wenn dieser eventuell Typenfehler enthält oder die Angabe von Typeninformationen benötigt. Die Verwendung von Javascript Bibliotheken mit voller Unterstützung für die Typprüfung ist durch die Angabe von Typinformationen für Objekte, Klassen und Funktionen der Bibliothek in extra Dateien, den Declaration Files, möglich. Alle von diesem Projekt benutzen Javascript Bibliotheken stellen diese offiziell bereit, um die Verwendung dieser in Typescript ohne weiteren Aufwand zu ermöglichen.

Typescript Quellcode wird durch den Typescript Compiler, oder anderen Webentwicklungswerkzeugen die Typescript unterstützen, in normalen Javascript Code transpiliert.

Neben den offiziellen Typangaben von Vue für die Definition von Komponenten wird außerdem ein klassenbasierter Ansatz über die `vue-class-component` [25] Bibliothek angeboten. Die Definition der Beispielkomponente aus Quellcode 5.1 mit diesem Ansatz ist in Quellcode 5.4 dargestellt. `vue-property-decorator` [1] erweitert die durch den klassenbasierten Ansatz unterstützten Vue-Funktionalitäten, wie der Definition von Props als Klasseneigenschaft statt der normalen Definition als Objekt.

```
@Component
export default class BeispielKomponente extends Vue {
  name = "Max"
}
```

Quellcode 5.4: Beispielkomponente aus Quellcode 5.1 mit klassenbasiertem Ansatz

Für die Definition von Vuex Stores stellt die `vuex-class-component` [14] Bibliothek ebenfalls einen klassenbasierten Ansatz bereit. Dieser Ansatz hat den Vorteil, dass Zugriffe auf Stores, durch die Art wie diese normalerweise definiert werden, nicht von Typescript geprüft werden können. Der Ansatz der Bibliothek erlaubt einen vollständig durch Typescript prüfbaren Zugriff. Quellcode 5.5 zeigt den von der Nachbildung verwendeten Store für die Verwaltung der Anmeldeinformationen in Javascript ohne die Verwendung von Bibliotheken. Quellcode 5.6 enthält denselben Store in Typescript unter der Verwendung der Bibliothek.

```
const userStore = {
  namespace: true,
  state: {
    jwt: null,
    email: null
  },
  mutations: {
    setJwt(state, jwt) {
      state.jwt = jwt
      // Benutzerinformationen (id, name, email) aus jwt auslesen
      state.user = JwtDecode(jwt)
    }
  },
  getters: {
    jwt: state => {
     return state.jwt
    },
    user: state => {
     return state.user
    }
  }
}
```

Quellcode 5.5: Vuex User Store in Javascript

```
const module = createModule({
  namespaced: 'user',
  strict: false,
  target: 'nuxt',
})

export class UserStore extends module {
  private jwt: string | null = null
  private user: IUser | null = null

  @mutation setJwt(jwt: string) {
    this.jwt = jwt
    this.user = JwtDecode(jwt)
  }

  get jwt() {
    return this._jwt
  }

  get user() {
    return this._user
  }
}
```

Quellcode 5.6: Vuex User Store in Typescript mit `vuex-class-component` Bibliothek

5.1.1 Seiten und Routing

In Vue müssen alle Routen und die auf der Route anzuzeigende Komponente manuell konfiguriert werden. Nuxt generiert diese Konfiguration automatisch, basierend auf der Struktur der Ordner und Dateien im `pages` Ordner des Projekts. Alle Seiten der Nachbildung, die dazu gehörenden Dateipfade im `pages` Ordner und die von Nuxt generierten Routen, sind in der nachfolgenden Tabelle aufgelistet.

Dateipfad	HTTP Route	Seiteninhalt
index.vue	/	Startseite
login.vue	/login	Anmeldeformular
exam/index.vue	/exam	Klausurliste
exam/_id.vue	/exam/:id	Klausuransicht
session/index.vue	/session	Liste der aufgezeichneten Nutzerinteraktionssitzungen
session/_id.vue	/session/:id	Detailansicht einer Interaktionssitzung mit aufgezeichneten Events

Tabelle 5.1: Seiten und Routen der Nuxt-Webseite

Dynamische Parameter werden durch einen Unterstrich im Dateinamen gekennzeichnet und als `:name` in den Routen repräsentiert. Die Validierung der Parameter ist Aufgabe der Komponente. Ohne Validierungslogik würden alle Werte der dynamischen Route zugeordnet, vorausgesetzt es existieren keine anderen Routen, auf die die Eingabe zugeordnet werden kann. Quellcode 5.7 zeigt die Validierung an einem Beispiel. Eine Route wäre für diese Seite nur gültig, wenn der Parameter `id` ausschließlich aus Ziffern besteht.

```
@Component({
  validate({ params }) {
    // params ist ein Dictionary aus Paramtername -> Wert Paaren

    return /^\d+$/.test(params.id)
  },
})
export default class BeispielSeite extends Vue {}
```

Quellcode 5.7: Beispiel einer Seite mit Validierung der dynamischen Parameter

5.1.2 Two Way Data Binding mit Vuex

Eine der Grundsätze von Vue ist, dass reaktiv gebundene Dateninformationen nur in eine Richtung fließen können. Für das Beispiel eines Textfelds wie `<input v-bind:value="wert" />` ist der gezeigte Text zwar von `wert` abhängig, aber Änderungen des Textes durch den Benutzer werden nicht in `wert` reflektiert. Der bidirektionale Fluss von Daten wird mit Vue durch die Verwendung von Events realisiert. Für Textfelder kann das Standard Javascript Event `input` benutzt werden, wie z. B. `<input v-bind:value="wert" v-on:input="wert=$event.target.value" />`.
Für dieses Muster stellt Vue die `v-model` Direktive bereit, welche das Muster kombiniert mit der Behandlung von Randfällen umsetzt.

Die Implementierung der Struktur der Leistungstabelle ist einfach nur mit dem Two-Way-Databinding Muster von `v-model` nicht möglich, sondern etwas komplexer. Dies hat zwei Gründe: Die Daten müssen an einem Ort zentral in der Anwendung verwaltet werden, um für die Validierung und das Speichern veränderbar und abrufbar zu sein. Außerdem müssen sie von zwei Richtungen aus modifizierbar sein. Dies ist zum einen die Eingabe des Benutzers, welche in die Ergebnisliste überführt werden. Zum anderen die Validierungslogik, welche eventuell Werte in der Ergebnisliste ändert, und diese Änderungen in den Eingabefeldern gezeigt werden müssen, um für den Benutzer sichtbar zu sein. In Abbildung 5.2 ist dieses Problem schematisch dargestellt.

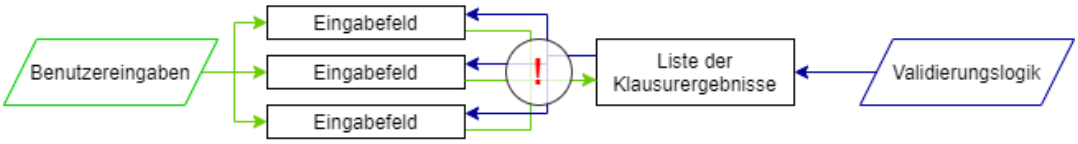


Abbildung 5.2: Schematische Darstellung des Problems des Datenflusses in der Leistungstabelle

Für die Abwicklung des Datenflusses existieren zwei verschiedene Möglichkeiten. Einerseits könnte eine Komponente als benötigter Datenspeicher dienen. Diese würde über reaktive Verbindungen die Werte an die untergeordneten Eingabefelder binden. Eingaben in diesen Feldern müssten dann über das Event-System von Vue an die Datenkomponente zurückgegeben und dort behandelt werden. Die alternative Variante benutzt Vuex als Datenspeicher und alle Eingabefelder greifen direkt auf den Speicher zu. Dieser Ansatz wurde für die Nachbildung verwendet. Wie Two-Way-Databinding mit Vuex möglich ist, wird anhand einer vereinfachten Version des benutzten Vuex Stores im Folgenden beschrieben und ist in Abbildung 5.3 dargestellt.

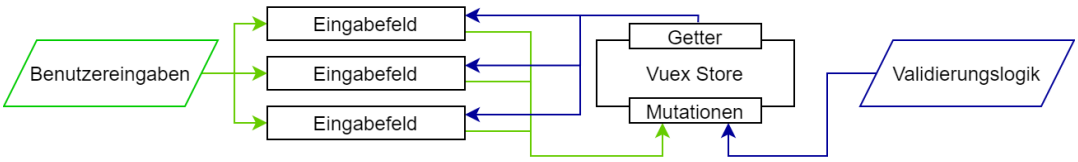


Abbildung 5.3: Schematische Darstellung des Datenflusses mit Vuex

Der für dieses Beispiel relevante Zustand des für die Klausurergebnisse zuständigen Stores besteht aus einer Liste aller Ergebnisse. Jedes Element beinhaltet dabei die Matrikelnummer des Studenten, zu dem dieses Element gehört, die Note und dem Prüfungszustand.

Für Note und Prüfungszustand ist jeweils eine Mutation zum Verändern des Wertes, und ein Getter zum Abrufen des Wertes definiert. Die Mutationen benötigen die Matrikelnummer des Studenten und die neue Note bzw. den neuen Prüfungsstatus des Studenten als Inputs.

Getter in Vuex Stores können selbst keine Inputs akzeptieren. Um trotzdem die Note oder den Status eines Studenten abrufen zu können, geben die Getter eine eigene Funktion zurück, die durch Angabe der Matrikelnummer den gewünschten Wert zurückgibt. Zu beachten ist außerdem, dass das Aufrufen der Getter in Funktionen des Stores nicht über `this` möglich ist, sondern auch wie Zugriffe von Komponenten über das `vxm` Objekt erfolgen.

Da die Ergebnisse in einer Liste gespeichert werden, aber der Zugriff auf die Ergebnisse immer durch die Angabe der Matrikelnummer eines Studenten erfolgt, wird der Zugriff auf die Ergebnisse durch ein Dictionary verbessert. Statt bei jedem Abrufen eines Ergebnisses das Element in der Liste zu suchen, dient das Dictionary als Zwischenspeicher. Nach dem Laden der Ergebnisse in den Store wird dieser befüllt und enthält die Informationen, welcher Index in der Liste zu jeder Matrikelnummer gehört. Wie die Getter für Note und Status auch, gibt der Getter `entry` über eine Funktion anhand der Matrikelnummer, durch Verwendung des Dictionary, das zugehörige Element der Liste zurück.

Quellcode 5.8 zeigt den vereinfachten Quellcode des hier beschriebenen Stores, und Quellcode 5.9 beinhaltet ein vereinfachtes Beispiel, wie eine Komponente diesen Store benutzt.

```
export class ExamStore extends module {
  private _results: ResultEntry[] = []
  private _idCache: { [id: number]: number } = {}

  get entry() {
    return (id: number) => {
      const index = this._idCache[id]
      return this._results[index]
    }
  }

  @mutation setGradeById(p: { id: number; grade: string }) {
    vxm.exam.entry(p.id).grade = p.grade
  }

  get getGradeById() {
    return (id: number) => {
      return vxm.exam.entry(id).grade
    }
  }

  @mutation setStatusById(p: { id: number; status: ExamStatus }) {
    vxm.exam.entry(p.id).status = p.status
  }

  get getStatusById() {
    return (id: number) => {
      return vxm.exam.entry(id).status
    }
  }
}
```

Quellcode 5.8: Vereinfachter Quellcode des Stores zuständig für die Verwaltung der Klausurergebnisse

```
<template>
    <input v-model="value">
</template>

<script lang="ts">
@Component
export default calss GradeInput extends Vue {
    @Prop({ required: true }) id!: number

    get value() {
        return vxm.exam.getGradeById(this.id)
    }

    set value(value: string) {
        vxm.exam.setGradeById({ id: this.id, grade: value })
    }
}
</script>
```

Quellcode 5.9: Vereinfachter Quellcode der Komponente zuständig für die Noteneingabe

5.1.3 Prüfen und Speichern der Eingaben

Die Funktionen zum Prüfen und Speichern der Eingaben sind ebenfalls in dem im Unterabschnitt 5.1.2 beschriebenen Store implementiert und werden in diesem Abschnitt beschrieben.

Der Quellcode in 5.10 beinhaltet nur die für die Validierung relevanten Teile des Stores. Der Zustand besteht für dieses Beispiel aus der oben bereits genannten Ergebnisliste und einem Dictionary zur Zuordnung, ob die Eingaben für einen Studenten gültig sind.

Die Validierung ist in zwei Aktionen und einer Mutation zum Modifizieren der Gültigkeitszuordnung implementiert. Wenn der Benutzer das Prüfen der Eingaben über den dazu gehörenden Knopf auslöst, wird zuerst die `fill` Aktion ausgeführt. Diese prüft für jede Zeile der Eingabe, ob die Werte automatisch vervollständigt werden können.

Die Vervollständigung wird durchgeführt, wenn der Prüfungsstatus „Täuschung“, „Fristablauf“, „Unentschuldigt“ oder „Nicht Bestanden“ entspricht. In diesen Fällen wird in das Notenfeld eine „5“ eingetragen. Außerdem wird, wie im Hochschulsystem auch, der Status „Anwesend“ immer auf „Unbekannt“ gesetzt.

Der zweite Schritt der Validierung ist das Prüfen auf nicht gültige Eingabe und die Markierung dieser. Dies ist in der Aktion `checkValid` implementiert. Die Funktion iteriert über alle Ergebniseingaben und prüft, ob die Noteneingabe dem akzeptierten Format entspricht. Akzeptiert wird die Eingabe, wenn diese entweder noch leer ist, oder dem durch den regulären Ausdruck `^[1-5](?:,[0-9])?` beschriebenen Notenformat entspricht.

Der Ausdruck akzeptiert alle Noten im Bereich von 1 bis 5, entweder ohne oder mit genau einer Nachkommastelle. Das Dezimaltrennzeichen muss ein Komma sein. Ein Punkt wird nicht akzeptiert. Außerdem gilt die Eingabe als fehlerhaft, wenn der Status „Bestanden“ entspricht, aber die Note entweder nicht angegeben ist oder einer 5 mit oder ohne Nachkommastelle entspricht.

```
export class ExamStore extends module {
  private _results: ResultEntry[] = []
  private _idCache: { [id: number]: number } = {}

  @action fill() {
    const failingStatus = [
      ExamStatus.Täuschung,
      ExamStatus.Fristablauf,
      ExamStatus.Unentschuldigt,
      ExamStatus.NichtBestanden,
    ];

    this._results.forEach((e) => {
      if (failingStatus.includes(e.status))
        vxm.exam.setGradeById({ id: e.id, grade: "5" });
      else if (e.status === ExamStatus.Anwesend)
        vxm.exam.setStatusById({ id: e.id, status: ExamStatus.Unbekannt });
    });

    return Promise.resolve();
  }

  @action checkValid() {
    let allValid = true;

    this._results.forEach((e) => {
      let valid = true;
      if (
        !(e.grade === "" || /^[1-5](?:,[0-9])?$/.test(e.grade)) ||
        (e.status === ExamStatus.Bestanden &&
          (e.grade === "" || e.grade.startsWith("5")))
      ) {
        valid = false;
        allValid = false;
      }
      vxm.exam.setValid({ id: e.id, valid });
    });

    return Promise.resolve(allValid);
  }

  @mutation setValid(p: { id: number; valid: boolean }) {
    Vue.set(this._valid, p.id, p.valid);
  }

  get isGradeValid() {
    return (id: number) => {
      if (this._valid[id] !== undefined) return this._valid[id]

      return true
    }
  }
}
```

Quellcode 5.10: Validierungslogik für die Ergebniseingaben

In den Eingabefeldern wird die Gültigkeit der Eingaben über eine Klassenbindung von Vue dargestellt, wie z. B. <input v-bind:class="{ 'is-invalid': !valid }" />. Wenn der Wert von `valid` `false` entspricht, wird dem Eingabeelement die Klasse `is-invalid` hinzugefügt und über die dazu gehörende CSS Anweisung für den Nutzer als fehlerhaft markiert. `valid` ist dabei an den `isGradeValid` Getter des Stores gebunden und aktualisiert sich reaktiv, wenn die `checkValid` Aktion den Validierungsstatus verändert.

```
@action async submit() {
  // Function assumes inputs are valid, only call when checkValid() passes!
  await Promise.all(
    this._results.map((e) => {
      const grade = e.grade === '' ? null : Number(e.grade.replace(',', '.'))

      if (Number.isNaN(grade))
        throw new Error(
          `Grade conversion for id ${e.id} failed! Value = ${e.grade}`
        )

      const dto: ExamResultEditDTO = {
        studentId: e.id,
        status: e.status,
        grade,
      }

      return ExamRepo.editResult(this._exam!.id, dto)
    })
  )
}
```

Quellcode 5.11: Aktion zum Speichern und Senden der Daten an die API

Die Aktion die vom Benutzer durch das Speichern der Eingaben ausgelöst wird, ist in Quellcode 5.11 angegeben. Die Funktion wird beim Drücken des Knopfes nur aufgerufen, wenn vorher durch `checkValid` geprüft wurde, ob alle Eingaben auch gültig und bereit zum Absenden sind. Die `fill` Aktion wird nicht aufgerufen, da sonst Eingaben verändert und abgesendet werden, die der Nutzer nicht sehen und überprüfen kann.

Die Funktion iteriert wieder über alle Ergebnisse in der Liste und baut für jeden Eintrag ein Datentransferobjekt zum Senden an den Endpunkt der API zum Verändern von Ergebnissen. Der Quellcode, der die Kommunikation zur API behandelt ist in diesem Projekt in Repositories abgelegt. Das Repository `ExamRepo` beinhaltet Funktionen zum Auflisten und Abrufen von Klausuren, dem Laden von in einer Klausur eingeschriebenen Studenten und deren Ergebnissen sowie das Verändern eines Ergebnisses.

Jede Funktion ist für das Aufrufen des zugehörigen Endpunktes der API, der eventuellen Transformation der von der API gesendeten Daten und der Behandlung von eventuell auftretenden Fehlern der Anfrage verantwortlich.

Quellcode 5.12 zeigt den Quellcode der `editResult` Funktion. `$axios` ist eine globale Variable und enthält die HTTP Client Instanz der Axios Bibliothek.

```
async editResult(examId: number, dto: ExamResultEditDTO) {
  const res = await $axios.post(`/api/exam/${examId}/result`, dto)

  if (res.status === 200) return ResultEditStatus.Ok
  if (res.status === 400) return ResultEditStatus.NotEnrolled
  if (res.status === 404) return ResultEditStatus.NotFound
  throw new Error(`Unkown edit result response code ${res.status}`)
}
```

Quellcode 5.12: Funktion zum Aufrufen des „Ergebnis bearbeiten“ Entpunktes der API

5.1.4 Implementierung der Interaktionsdatenaufzeichnung

Jede Art von Ereignissen ist als eigene Klasse implementiert, aber die unterschiedlichen Ereignisse müssen alle in einem einheitlichen Format an die API gesendet werden. Abbildung 5.4 zeigt die Klassenstruktur als UML Diagramm.

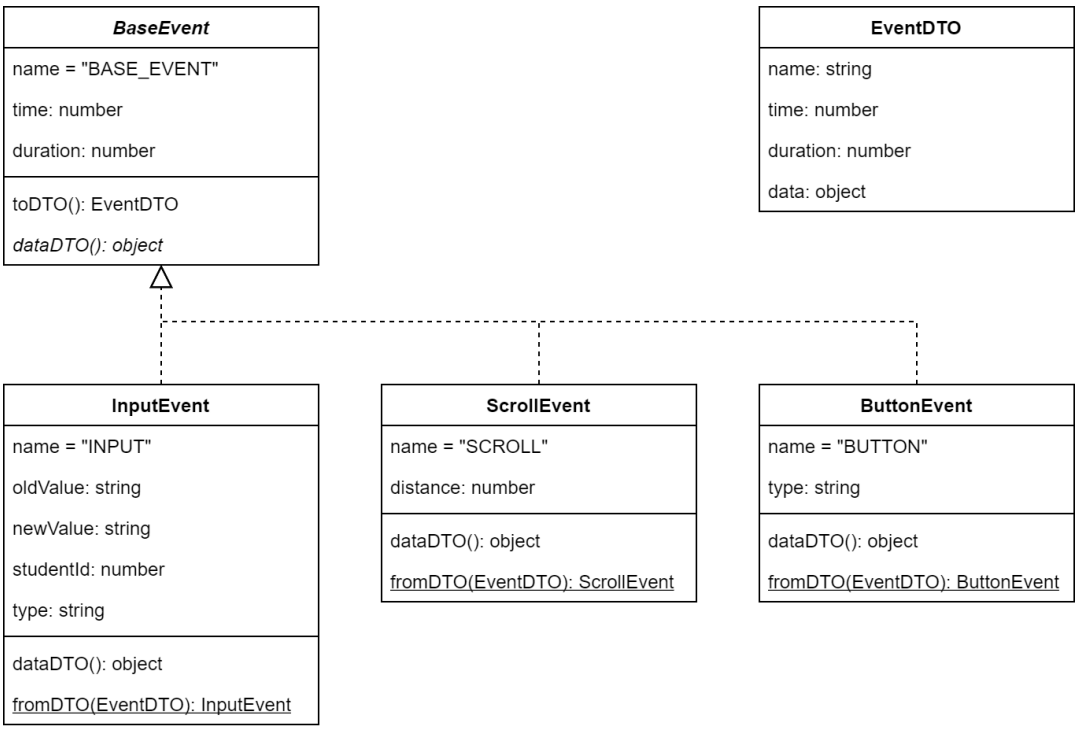


Abbildung 5.4: Struktur der Ereignisklassen als UML-Diagram

`BaseEvent` ist eine abstrakte Klasse die alle Eigenschaften enthält, welche alle Ereignisarten gemeinsam haben: Zeitstempel, Dauer und Name. Außerdem enthält die Klasse die Methode `toDTO` zum Umwandeln des Ereignisses in das von der API verlangte einheitliche Format. Zur Implementation einer Ereignisart muss somit nur von dieser Basisklasse geerbt werden, und der Name sowie die `dataDTO` Funktion überschrieben werden. Die Funktion soll alle weiteren benötigten Daten des Ereignisses als Objekt zurückgeben. Der Name wird zur Unterscheidung der verschiedenen Ereignisse im generischen Format benutzt. Außerdem ist es dadurch möglich, das generische Objekt wieder in eine Instanz der Ereignisklasse zurück umzuwandeln.

Der Quellcode der Basisklasse ist in Quellcode 5.13 dargestellt.

```
export abstract class BaseEvent {
  name = 'BASE_EVENT'
  time: number
  duration: number

  constructor(time: number, duration = 0) {
    this.time = time
    this.duration = duration
  }

  toDTO(): EventDTO {
    return {
      name: this.name,
      time: this.time,
      duration: this.duration,
      data: this.dataDTO(),
    }
  }

  abstract dataDTO(): object
}
```

Quellcode 5.13: Abstrakte Basisklasse für alle Ereignisse

Gespeichert werden die Ereignisse während einer Sitzung, wie die Klausurergebnisse auch, in einem Vuex Store. Solange eine Aufzeichnungssitzung aktiv ist, werden alle auftretenden Ereignisse in einer Liste gespeichert. Außerdem enthält der Store die Startzeit der Aufzeichnung, um den Zeitstempel von Ereignissen berechnen zu können, sowie die Id der Sitzung, um die Ereignisse bei Beendigung der Aufzeichnung der Sitzung hinzuzufügen.

ButtonEvent ist das einfachste Ereignis und wird ausgelöst, wenn entweder der Knopf zum Prüfen oder zum Speichern der Klausurergebnisse vom Prüfer benutzt wird.

Das Ereignis enthält als extra Daten den Namen des Knopfes, also „PRÜFEN“ oder „SPEICHERN“.

Veränderungen von Klausurergebniseingaben werden vom InputEvent repräsentiert. Das Ereignis enthält den alten sowie neuen Wert der Eingabe, und die Matrikelnummer des Studenten, dessen Status oder Note verändert wurde. Bei Auswahl eines anderen Prüfungsstatus wird sofort ein Ereignis ausgelöst, das dieses repräsentiert. Noteneingaben werden dahingehend zuerst aggregiert. Statt jede Eingabe oder jedes Löschen eines Zeichens als eigenes Ereignis darzustellen, wird versucht, die Veränderung von einer Note zu einer anderen mit einem einzelnen Eingabeereignis darzustellen. Ein Noteneingabeereignis wird erst ausgelöst, wenn entweder 750 ms seit der letzten Eingabeaktivität vergangen sind, oder das Eingabefeld nicht mehr ausgewählt ist.

Scrollbewegungen werden ebenfalls aggregiert. Mehrere Bewegungen werden in einem Ereignis zusammengefasst, welches die gesamte Scrollweite in Pixel als Daten enthält. Ein Scrollereignis wird erst ausgelöst, wenn entweder 500 ms seit der letzten Bewegung vergangen sind, oder die aktuelle Scrollbewegung nicht in dieselbe Richtung wie die letzte Bewegung scrollt.

5.2 Erweiterung des ASP.NET Backends

Das Backend des bereits im Praxisprojekt entstandenen Systems ist mit .NET Core, Version 3.1, umgesetzt. .NET Core ist der plattformübergreifende, quelloffene Nachfolger des .NET Frameworks und unterstützt neben Windows auch Linux und macOS. .NET erlaubt die Programmierung in mehreren Sprachen, wie Visual Basic oder F#. Für dieses Projekt wurde allerdings ausschließlich C# verwendet.

ASP.NET Core [11] ist das offizielle Framework zur Entwicklung von Webanwendungen mit der .NET Plattform und unterstützt die Entwicklung von normalen Webseiten, REST APIs und interaktiven Webanwendungen, geschrieben in C# statt Javascript, mit Blazor.

5.2.1 Datenbankzugriff mit Entity Framework Core

Für die persistente Datenspeicherung wird PostgreSQL [16], ein SQL-Standard konformer, relationaler Datenbankserver, benutzt. Die Kommunikation mit der PostgreSQL Datenbank wird von der Entity Framework Core Bibliothek [12] realisiert. Die Bibliothek ist ein auf objektrelationale Abbildung basierendes Framework zum Zugriff auf relationale Tabellen über objektorientierte Klassen. Dies hat den Vorteil, das alle Zugriffe direkt durch C# Strukturen stattfinden, statt manuell geschriebene SQL Anfragen auszuführen und die zurückgegebenen Daten in objektorientierte Klassen zu übersetzen.

Alle Entitäten sind als C# Klasse definiert und eventuell mit Informationen, die nicht automatisch ermittelt werden können, annotiert. Quellcode 5.14 zeigt die Klausur-Klasse als Beispiel. Der Datenkontext ist der Konfigurations- und Zugangspunkt von Entity Framework Core zur Benutzung im Programmcode. Im Kontext werden alle Klassen registiert, konfiguriert und später werden alle Interaktionen mit der Datenbank über den Kontext abgewickelt.

Die Codezeile `context.Exams.FirstOrDefault(e => e.Id == 1);` würde somit die Klausur mit dem Primärschlüssel `1` in der Datenbank suchen und als Instanz der Klausur-Klasse zurückgeben. Die von der Bibliothek generierten Anfrage an die Datenbank für diese Codezeile ist `SELECT e.id, e.date, e.name, e.owner, e.template_id FROM exams AS e WHERE e.id = 1 LIMIT 1`.

```
public class Exam
{
    public long Id { get; set; }
    [Required(AllowEmptyStrings = false)]
    public string Name { get; set; }
    [Required(AllowEmptyStrings = false)]
    public string Owner { get; set; }
    [Required]
    public DateTime Date { get; set; }
    public long TemplateId { get; set; }
    public Template Template { get; set; }
    public List<ExamStudent> ExamStudents { get; set; }
}
```

Quellcode 5.14: Objektorientierte Abbildung der relationalen Klausurtabelle

5.2.2 Datentransferobjekte und AutoMapper

Datentransferobjekte (kurz DTOs) [8] werden von der API zum Beschreiben der Daten benutzt, die an die Clients gesendet oder von diesen empfangen werden. DTOs dienen zur Transformation der Datenbankmodelle in an Clients angepasste Datenstrukturen. Quellcode 5.15 zeigt das Datentransferobjekt für das Klausurdatenbankmodell aus 5.14. Das DTO ist die einfachste Anwendung dieses Musters und stellt einen Filter für das Datenbankmodell dar. Alle Eigenschaften die das Modell hat, aber das DTO nicht, sind für Clients nicht sichtbar. Weitere Transformationen wären z. B. die Erweiterung um weitere abgeleitete Information wie Anzahl der Einschreibungen oder das Aufnehmen von weiteren verbundenen Modellen.

```
public class ExamDTO
{
    public long Id { get; set; }
    public string Name { get; set; }
    public string Owner { get; set; }
    public DateTime Date { get; set; }
    public long TemplateId { get; set; }
}
```

Quellcode 5.15: Klausur Datentransferobjekt

Zur Automatisierung der Transformationen wird die AutoMapper [3] Bibliothek verwendet. Statt die Transformationslogik zwischen DTOs und Datenbankmodellen für alle benötigten Kombinationen zu schreiben, können die benötigten Transformationen einmal konfiguriert werden und später durch Aufrufen der Bibliothek ausgeführt werden.

Quellcode 5.16 zeigt die Konfiguration von AutoMapper anhand eines Beispiels. `Student` ist das Datenbankmodell und `StudentDTO` das zugehörige DTO. Für alle Elemente des DTOs müssen die Werte nur aus dem Modell kopiert werden, bis aus `SeminarGroup`, welches den Namen der Seminargruppe enthalten soll. Am Datenbankmodell ist `SeminarGroupId` der Fremdschlüssel zur Seminargruppe. Da der Name der Seminargruppe auch gleichzeitig der Primärschlüssel der Tabelle ist, muss nur der Wert des Fremdschlüssels kopiert werden.

```
public class Student
{
    [DatabaseGenerated(DatabaseGeneratedOption.None)]
    public long Id { get; set; }
    [Required(AllowEmptyStrings = false)]
    public string Firstname { get; set; }
    [Required(AllowEmptyStrings = false)]
    public string Lastname { get; set; }
    [Required]
    public DateTime Birthday { get; set; }
    public string SeminarGroupId { get; set; }
    public SeminarGroup SeminarGroup { get; set; }
    public List<ExamStudent> ExamStudents { get; set; }
}
```

(a) Datenbankmodelle für Student und Seminargruppe

```
public class StudentDTO
{
    public long Id { get; set; }
    public string Firstname { get; set; }
    public string Lastname { get; set; }
    public DateTime Birthday { get; set; }
    public string SeminarGroup { get; set; }
}
```

(b) Datentransferobjekt für einen Studenten

```
CreateMap<Student, ExamStudentDTO>()
    .ForMember(
        s => s.SeminarGroup,
        opt => opt.MapFrom(src => src.SeminarGroupId)
    );
```

(c) AutoMapper Konfiguration

Quellcode 5.16: Quellcodebeispiel für AutoMapper

Als Beispiel wie ein Controller in ASP.NET Core umgesetzt ist, enthält Quellcode 5.17 einen Ausschnitt des Klausur-Controllers mit zwei Endpunkten: Klausuren auslisten und eine Klausur abrufen. `_context` ist der bereits beschriebene Datenbankkontext und `_mapper` ist eine AutoMapper-Instanz.

Die Kommentare und `ProducesResponseType` Attribute an den Funktionen dienen zum automatischen Generieren der API Dokumentation im Swagger Format. Mithilfe der NSwag Bibliothek ist unter der Route `/swagger` eine für Entwickler lesbare Dokumentation aller Endpunkte verfügbar.

```
[ApiController]
[Route("[controller]")]
[Produces("application/json")]
[OpenApiTag("Exam", Description = "")]
public class ExamController : ControllerBase
{
        private readonly Context _context;
        private readonly IMapper _mapper;

        public ExamController(Context context, IMapper mapper)
        {
                _context = context;
                _mapper = mapper;
        }

        /// <summary>
        /// List Exams
        /// </summary>
        /// <response code="200">All Exams</response>
        [HttpGet]
        [ProducesResponseType(typeof(List<ExamDTO>), 200)]
        public async Task<IActionResult> List()
        {
                var exams = await _context.Exams.AsNoTracking().ToListAsync();
                var dto = _mapper.Map<List<ExamDTO>>(exams);
                return Ok(dto);
        }

        /// <summary>
        /// Get Exam
        /// </summary>
        /// <response code="200">Exam</response>
        /// /// <response code="404">Exam not found</response>
        [HttpGet("{id}")]
        [ProducesResponseType(typeof(ExamDTO), 200)]
        [ProducesResponseType(404)]
        public async Task<IActionResult> Get(long id)
        {
                var e = await _context.Exams.AsNoTracking()
                    .FirstOrDefaultAsync(e => e.Id == id);

                if (e == null)
                        return NotFound();

                var dto = _mapper.Map<ExamDTO>(e);
                return Ok(dto);
        }
}
```

Quellcode 5.17: Volles Beispiel eines ASP.NET Core Controllers

6 Demonstration

Die Aufzeichnung der Web- und App-basierten Notenmeldungsprozesse werden anhand eines Versuches in diesem Kapitel demonstriert. Der Versuch soll zeigen, dass alle relevanten Ereignisse erfasst werden und später für eine Auswertung zur Verfügung stehen. Dafür wurde die Interaktionsdatenerfassung unter Praxisbedingungen getestet.

6.1 Demonstration der Nachbildung

Für den Testdurchlauf des Web-basierten Prozesses mit der Nachbildung des Dozentenportals wurde 38 fiktive Studenten in eine Klausur eingeschrieben und Klausurdeckblätter mit deren Matrikelnummern, Namen und Seminargruppen handschriftlich von verschiedenen Personen ausgefüllt. Außerdem wurde die erreichte Note bereits eingetragen, als wenn ein Prüfer die Klausur bereits kontrolliert hat. Die benutzten Testdaten sind in Abbildung A.1 zu finden.

Für den Versuch übertrug eine Person die Noten aller dieser Klausurdeckblätter in die Leistungstabelle der Nachbildung, wie es in Kapitel 3 beschrieben ist. Der Versuchsperson wurde keine Dokumentation zur Benutzung des Systems bereitgestellt und musste sich selbst mit der Verwendung vertraut machen.

Die Versuchsperson hat für die Eintragung aller 38 Ergebnisse in diesem Testdurchlauf 10 Minuten und 41 Sekunden benötigt. Die Aufzeichnungsdaten stammen nicht von einem Blindversuch. Der Benutzer hatte den Eintragungsprozess bereits vorher kurz ausprobiert.

Durch die gesammelten Interaktionsdaten kann der Zeitaufwand der einzelnen Schritte berechnet werden. Eingabeereignisse repräsentieren insgesamt 1 Minute und 43 Sekunden (16 %) der Aufzeichnung, und Scrollereignisse 1 Minute und 13 Sekunden (12 %). Die Eingabeereignisse bestehen aus dem Eintragen der Noten, sowie der Korrektur von Fehlern. Die restlichen 72 % der Sitzung sind Aktivitäten, die nicht von der Datenaufzeichnung erfasst werden können. Dazu zählen das Durchblättern und Lesen der Klausuren sowie das Suchen der Studenten in der Leistungstabelle.

Die durchschnittliche Zeit, die für die Eintragung des Ergebnisses eines Studenten benötigt wird, lässt sich durch die Zeit zwischen Eingaben schätzen. Der Durchschnitt der Zeit zwischen Eingabeereignissen von ausschließlich neuen Eingaben, also keine Korrekturen, liegt bei 15 Sekunden.

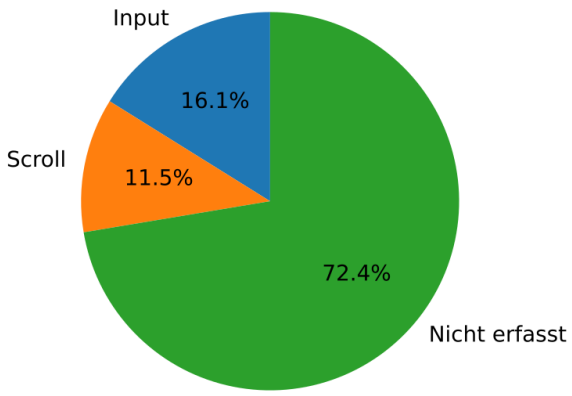


Abbildung 6.1: Kreisdiagramm zur Aufteilung der Gesamtdauer der Ereignisse

Es wurden 49 Noteneingabeereignisse erfasst. 37 von diesen sind Eintragungen in ein vorher leeres Feld, also die erste Eintragung der Note. Für einen Studenten wurde keine Note eingetragen. Dieser war zwar eingeschrieben, hat an der Klausur aber nicht teilgenommen. Die anderen 12 Ereignisse sind Korrekturen der vorherigen Eingaben.

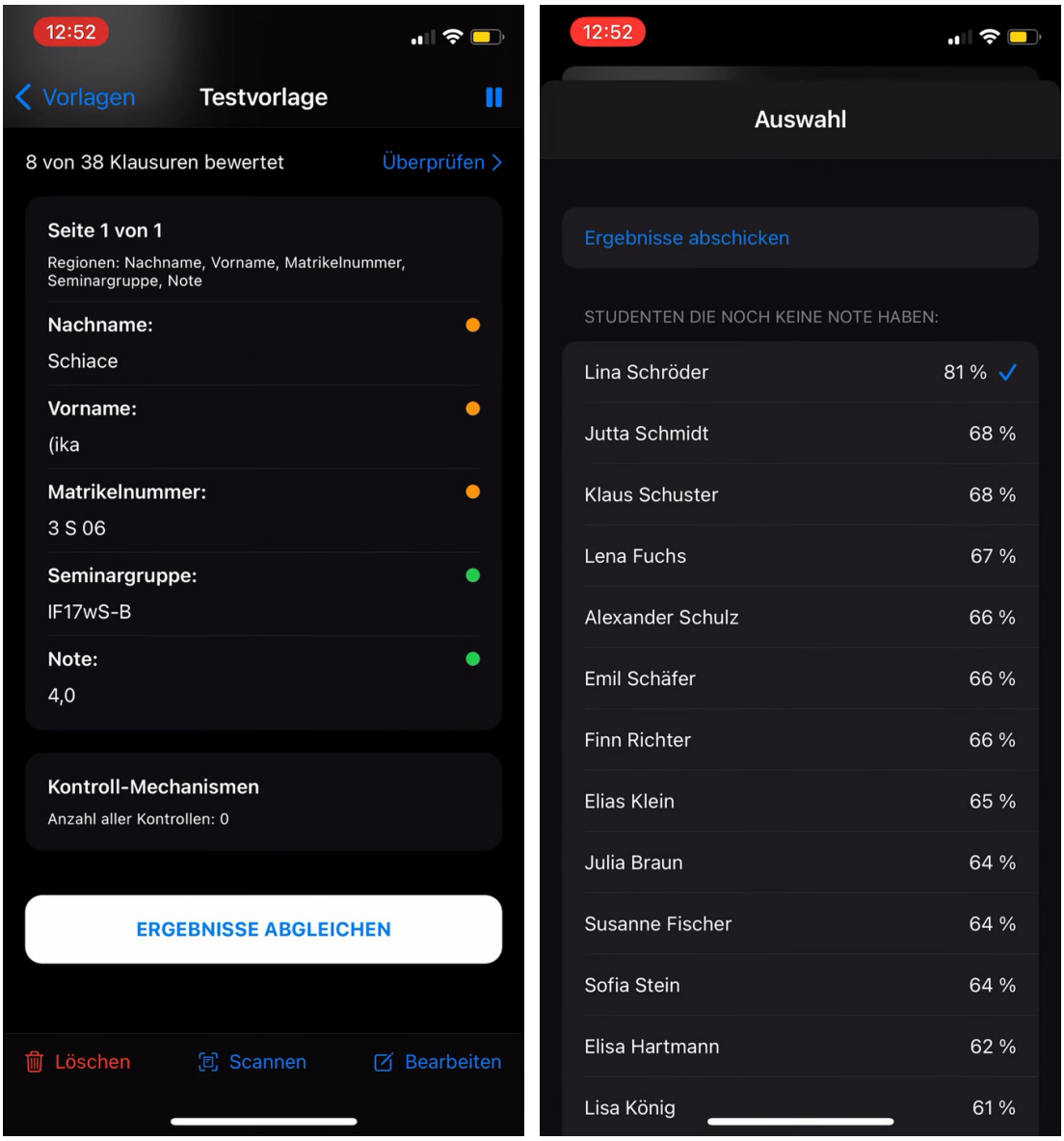
Ungültige Zwischennoten, wie „4,1“, werden nicht als Fehler markiert und müssen später von Hand korrigiert werden. Die meisten Korrekturen sind das Ändern von Punkt zu Komma. Bei einem Ergebnis ist dieser Fehler nicht aufgefallen, und die Note „1,8“ statt „1,7“ wurde gespeichert.

6.2 Demonstration der iOS App

Für den Testdurchlauf des App-basierten Notenmeldungsprozesses wurden 13 formlose Klausurdeckblätter per Hand ausgefüllt. Die Versuchsdaten sind dieselben wie sie auch im Versuch mit der Nachbildung benutzt wurden. Da die einzelnen Scans der Klausuren unabhängig voneinander sind, ist die reduzierte Anzahl zur Demonstration ausreichend.

Während des Versuches ist aufgefallen, dass eine Korrektur der Daten, selbst bei geringer Genauigkeit der Texterkennung, nicht unbedingt notwendig ist. Die Auswahl eines eingeschriebenen Studenten ist durch die Sortierung nach Konfidenz auch bei schlechten Ergebnissen ohne Korrektur möglich. Wenn die eingetragene Note richtig erkannt wurde, kann ohne weitere Eingabe das Ergebnis gespeichert werden.

Abbildung 6.2 zeigt dies an einem Beispiel. Die Note und Seminargruppe wurde korrekt erkannt, aber Vorname, Nachname und Seminargruppe nicht. Trotzdem wird durch die Sortierung nach Konfidenz der richtige Student ausgewählt. Der Prüfer musste für diese Klausur weder die Note noch die Auswahl des Studenten korrigieren und konnte dieses Klausurergebnis absenden.



(a) Texterkennungsergebnis (b) Studentenauswahl

Abbildung 6.2: Scannen einer Klausur im Testdurchlauf

Von den 13 bearbeiteten Klausuren musste die erkannte Note bei 7 korrigiert werden. Für das Melden aller Ergebnisse hat der Prüfer 6 Minuten und 25 Sekunden benötigt. Die Zeit für die Bearbeitung einer Klausur liegt zwischen 22 und 37 Sekunden. Der Durchschnitt beträgt 29 Sekunden.

In Abbildung 6.3 sind die einzelnen Schritte, also das Einscannen der Klausur und die Korrektur sowie das Absenden der Eingaben, als Zeitstrahl visualisiert. Der Mittelwert für die Dauer des Einscannens der Klausuren liegt bei 8 Sekunden, und für das Kontrollieren der Texterkennungsergebnisse, inklusive eventueller Korrekturen, bei 20 Sekunden. Nur die Zeit zwischen dem Absenden der Eingaben und dem Anfang eines neuen Scans ist nicht aus den Daten abzuleiten, da das Absenden selbst nicht als Ereignis repräsentiert ist. Diese Zeit ist in der Abbildung und im Durchschnitt der Korrektur zugeordnet.

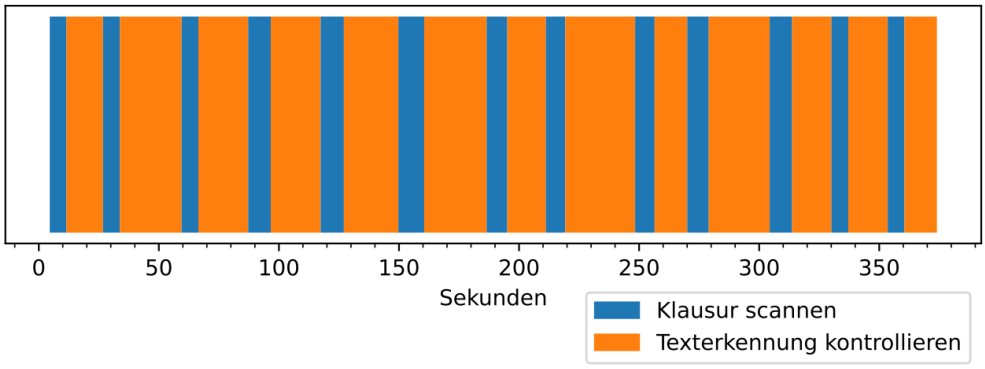


Abbildung 6.3: Aufteilung der Aktivitäten im Testdurchlauf der App

In den Abbildungen 6.4 und 6.5 ist je ein Ausschnitt der gesammelten Interaktionsdaten pro Testdurchlauf mit einmal der Nachbildung ein einmal der iOS App gezeigt.

Timestamp ↑	Länge	Typ	Daten
18.454 s	2.655 s	INPUT	{ "type": "NOTE", "newValue": "4.0", "oldValue": "", "studentId": 2867 }
19.920 s	0 s	INPUT	{ "type": "STATUS", "newValue": "Bestanden", "oldValue": "Unbekannt", "studentId": 2867 }
27.011 s	1.541 s	INPUT	{ "type": "NOTE", "newValue": "1,3", "oldValue": "", "studentId": 2441 }
28.599 s	0 s	INPUT	{ "type": "STATUS", "newValue": "Bestanden", "oldValue": "Unbekannt", "studentId": 2441 }
34.401 s	0.65 s	SCROLL	{ "distance": 815 }
34.402 s	0.651 s	SCROLL	{ "distance": 815 }
34.971 s	0.166 s	SCROLL	{ "distance": 9 }
34.972 s	0.166 s	SCROLL	{ "distance": 9 }
35.389 s	0.318 s	SCROLL	{ "distance": -305 }
35.389 s	0.317 s	SCROLL	{ "distance": -305 }
36.123 s	0.233 s	SCROLL	{ "distance": 305 }
36.123 s	0.233 s	SCROLL	{ "distance": 305 }
39.733 s	1.649 s	INPUT	{ "type": "NOTE", "newValue": "2.7", "oldValue": "", "studentId": 2370 }
42.512 s	4.42 s	INPUT	{ "type": "NOTE", "newValue": "2,7", "oldValue": "2.7", "studentId": 2370 }

Abbildung 6.4: Ausschnitt der aufgezeichneten Ereignisse des Web-basierten Testdurchlaufs

Timestamp ↑	Länge	Typ	Daten
1.367 s	0 s	NAVIGATION	{ "to": "TemplateDetailScreen" }
4.568 s	0 s	NAVIGATION	{ "to": "ScannerScreen" }
4.582 s	0 s	NAVIGATION	{ "to": "TemplateView" }
11.523 s	0 s	NAVIGATION	{ "to": "TemplateDetailScreen" }
26.735 s	0 s	NAVIGATION	{ "to": "ScannerScreen" }
33.795 s	0 s	NAVIGATION	{ "to": "TemplateDetailScreen" }
42.243 s	0 s	INPUT	{ "to": "2", "from": "", "type": "Note" }
42.654 s	0 s	INPUT	{ "to": "2,", "from": "2", "type": "Note" }
42.860 s	0 s	INPUT	{ "to": "2,5", "from": "2,", "type": "Note" }
59.469 s	0 s	NAVIGATION	{ "to": "ScannerScreen" }
66.465 s	0 s	NAVIGATION	{ "to": "TemplateDetailScreen" }
72.929 s	0 s	INPUT	{ "to": "5", "from": "", "type": "Note" }
73.043 s	0 s	INPUT	{ "to": "5,", "from": "5", "type": "Note" }
73.245 s	0 s	INPUT	{ "to": "5,0", "from": "5,", "type": "Note" }

Abbildung 6.5: Ausschnitt der aufgezeichneten Ereignisse des App-basierten Testdurchlaufs

7 Fazit

Im vorangegangenen Praxismodul ist ein Ansatz zur automatisierten Digitalisierung von Dokumenten als Prototyp entwickelt worden. Ziel war es, die Verwendung von Texterkennung zur Vereinfachung der Digitalisierung zu evaluieren. Daraus hat sich die Frage ergeben, wie man die Veränderung der Effektivität des neuen verglichen mit dem bereits existierenden Frontend herausfinden würde. Dafür wird eine Methode benötigt, welches das Messen der beiden Prozesse ermöglicht. Mit dieser Arbeit wurde ein Konzept aufgestellt, welches den Ablauf dieser Ansätze aufzeichnen kann. Das Konzept ist nicht nur auf die Notenmeldung ausgerichtet, sondern generisch ausgelegt, um auch für weitere Prozesse umsetzbar zu sein.

Um eine Erstimplementation dieses Konzepts zu realisieren, musste zuerst der zu messende Notenmeldungsprozess implementiert werden. Dafür wurde die im Praxismodul entstandene REST-API sowie die iOS App erweitert. Außerdem wurde die Notenmeldungsfunktion des Dozentenportals in einer eigenen Webanwendung nachgebildet, um auch diesen Ablauf messen zu können. Anschließend wurde das Datenerfassungskonzept in der iOS App und in der Nachbildung implementiert.

Mit dem System konnte die Notenerfassungprozesse unter Praxisbedingungen durchgängig und vollständig erfasst werden. Insgesamt entfielen im Demonstrationsversuch ungefähr 25 % der Aktivitäten auf solche, die direkt als Benutzerereignisse in der Webanwendung aufgezeichnet wurden. Die verbliebenen Aktionen können mit der Anwendung als einzige Datenquelle nicht erfasst werden. Durch Aufzeichnung der Navigationen und Eingaben in der iOS App ist der Ablauf der Notenmeldung auch hier lückenlos abgebildet. Die einzelnen Schritte, also das Scannen der Klausur sowie das Korrigieren und Absenden der Eingaben, sind erkennbar.

8 Ausblick

Aus der Entwicklung und der Demonstration haben sich weitere Ansatzpunkte und Herausforderungen für die weiterführende Nutzung und Erweiterung des Systems ergeben.

Die 75 % der unbekannten Aktivitäten in der Nachbildung des Dozentenportals könnten durch die Verwendung von weiteren Datenquellen erfasst werden. Mit einer Kamera, eventuell sogar mit Eyetracking-Funktion, könnte auch Aktivitäten außerhalb der Anwendung aufgezeichnet werden. Eine Kamera würde messen können, ob der Prüfer gerade mit der Anwendung interagiert oder mit einem anderen Schritt, wie dem Lesen einer Klausur, beschäftigt ist. Eyetracking könnte außerdem den Aufwand der visuellen Suche nach dem richtigen Eingabefeld darstellen.

Durch eine Erweiterung der Aufzeichnung der App, sodass auch die automatischen Schritte des Prozesses in den Daten erhalten sind, könnte die Genauigkeit der Texterkennung und der Studentenzuordnung gemessen werden. Kombiniert mit den Korrekturen des Benutzers können Fehlerquellen gefunden werden. Da der kognitive Aufwand hauptsächlich aus dem Prüfen der Texterkennung und der Korrektur von Fehlern besteht, ist er von der Genauigkeit der Texterkennung abhängig. Über die Aufzeichnung der automatischen Schritte kann sie anhand der gesammelten Daten verbessert werden. Das Backend würde dies, und andere Erweiterungen der Aufzeichnung, bereits ohne Änderungen dank dem flexiblen Datenformat unterstützen.

Das Konzept der Interaktionsdatenaufzeichung kann auf beliebige weitere Prozesse angewandt werden. Ein Beispiel wäre die bereits im Praxismodul genannte Stapelverarbeitung von Klausuren. Statt die Klausuren per Hand einzeln zu scannen, könnte der gesamte Stapel über einen automatischen Einzug eingelesen werden. Damit würde der physische Aufwand für den Prüfer entfallen. Die Arbeitslast sollte nach dem Scannen der Klausuren ausschließlich aus dem Prüfen und Korrigieren der Texterkennung bestehen. Nach dem Einlesen der Klausuren ähnelt dieser Prozess dem Kontrollieren der Texterkennungsergebnisse der iOS App und könnte genau wie in der App auch aufgezeichnet werden.

Eine weitere bereits im Praxismodul beschriebene Idee ist die Verbesserung der Extraktion von Angaben durch Anpassung der Klausurdeckblätter. Vor dem Einscannen einer Klausur muss zuerst eine Vorlage erstellt werden, mit der alle Bereiche markiert werden, in denen Angaben extrahiert werden sollen. Dies könnte durch die Markierung der Felder auf dem Deckblatt selbst vermieden werden. Durch den Druck von bestimmten Mustern, wie z. B. QR-Codes, könnte die Texterkennung ohne weitere Angaben des Benutzers die Lage der Felder finden. Dies hätte außerdem den Vorteil, dass die Texterkennung unabhängiger von Bildqualität und Genauigkeit der markierten Bereiche ist. Wenn ein Bild falsch eingescannt wurde, wird das Anwenden der Vorlage wahrscheinlich keine guten Ergebnisse liefern.

Anhang A: Testdaten

Matrikelnummer	Vorname	Nachname	Seminargruppe	Note
1305	Bob	Müller	IF17wS-B	2,0
3506	Lina	Schröder	IF17wS-B	3,3
3577	Elias	Klein	IF17wS-B	1,0
3648	Finn	Richter	IF17wS-B	1,3
3790	Marie	Lehmann	IF17wS-B	1,7
3861	Johanna	Peters	IF17wS-B	3,3
1589	Martina	Rabe	MI18w1-B	1,3
1944	Paula	Hoffmann	MI18w1-B	2,0
2299	Anna	Koch	MI18w1-B	2,3
2654	Lukas	Vogt	MI18w1-B	2,3
3009	Markus	Keller	MI18w1-B	2,0
3364	Elisa	Hartmann	MI18w1-B	5,0
1234	Max	Mustermann	FO17w2-B	3,3
1731	Tom	Zimmermann	FO17w2-B	3,3
2228	Jana	Wolf	FO17w2-B	1,7
2725	Jonas	Winter	FO17w2-B	Krank
3435	Julia	Braun	IF17wS-B	5,0
3293	Lena	Fuchs	IF17wS-B	5,0
3151	Sabine	Weiß	IF17wS-B	1,3
3080	Oliver	Jung	IF17wS-B	2,3
1376	John	Fischer	IF17wS-B	2,3
1447	Erika	Jansan	IF17wS-B	3,0
1518	Manfred	Kanz	IF17wS-B	1,0
1660	Jutta	Schmidt	IF17wS-B	1,7
1802	Susanne	Fischer	IF17wS-B	3,7
1873	Nidine	Weber	IF17wS-B	2,3
2015	Lukas	Wagner	IF17wS-B	4,0
3222	Lisa	König	FO17w2-B	2,3
2086	Alexander	Schulz	IF17wS-B	4,0
2370	Peter	Graf	IF17wS-B	2,7
2441	Ben	Sauer	IF17wS-B	1,3
2512	Emma	Seidel	IF17wS-B	1,7
2583	Sofia	Stein	IF17wS-B	5,0
2796	Klaus	Schuster	IF17wS-B	4,0
2867	Christian	Sommer	IF17wS-B	4,0
2938	Paul	Beck	IF17wS-B	2,7
2157	Florian	Becker	IF17wS-B	3,3
3719	Emil	Schäfer	FO17w2-B	3,0

Abbildung A.1: Für der Demonstration benutze Testdaten

Literaturverzeichnis

[1] Vue Property Decorator, Software. Verfügbar unter: https://github.com/kaorun343/vue-property-decorator, Abgerufen am 21.11.2020.

[2] Berners-Lee, T., Fielding, R. T., und Masinter, L. Uniform Resource Identifier (URI): Generic Syntax. STD 66, RFC Editor (2005). Verfügbar unter: http://www.rfc-editor.org/rfc/rfc3986.txt, Abgerufen am 13.11.2020.

[3] Bogard, J. AutoMapper, Software. Verfügbar unter: https://github.com/AutoMapper/AutoMapper, Abgerufen am 11.11.2020.

[4] Bray, T. The JavaScript Object Notation (JSON) Data Interchange Format. STD 90, RFC Editor (2017). Verfügbar unter: https://tools.ietf.org/html/rfc8259, Abgerufen am 21.11.2020.

[5] Chopin, A., Chopin, S., Parsa, P., et al. Nuxt.js, Software. Verfügbar unter: https://nuxtjs.org, Abgerufen am 22.11.2020.

[6] Fielding, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. Dissertation, University of California, Irvine (2000).

[7] Fielding, R. T., Gettys, J., Mogul, J. C., Nielsen, H. F., Masinter, L., Leach, P. J., und Berners-Lee, T. Hypertext transfer protocol – http/1.1. RFC 2616, RFC Editor (1999). Verfügbar unter: https://tools.ietf.org/html/rfc2616, Abgerufen am 07.11.2020.

[8] Fowler, M. und Rice, D. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, Buch (2003). ISBN 978-0-321-12742-6.

[9] Jones, M., Bradley, J., und Sakimura, N. JSON Web Token (JWT). RFC 7519, RFC Editor (2015). Verfügbar unter: http://www.rfc-editor.org/rfc/rfc7519.txt, Abgerufen am 21.11.2020.

[10] Kallauke, T. Entwicklung eines Dokumenten-Scan-Systems. Praktikumsbericht, Hochschule Mittweida (2020).

[11] Microsoft. ASP.NET Core, Software. Verfügbar unter: https://github.com/dotnet/aspnetcore, Abgerufen am 11.11.2020.

[12] Microsoft. EF Core, Software. Verfügbar unter: https://github.com/dotnet/efcore, Abgerufen am 11.11.2020.

[13] MICROSOFT. Typescript, Software. Verfügbar unter: https://www.typescriptlang.org.

[14] OLOFINJANA, M. Vuex Class Component, Software. Verfügbar unter: https://github.com/michaelolof/vuex-class-component, Abgerufen am 21.11.2020.

[15] OPENAPI INITIATIVE. *OpenAPI Specification - Version 2.0*. Verfügbar unter: https://swagger.io/specification/v2/, Abgerufen am 21.11.2020.

[16] POSTGRESQL GLOBAL DEVELOPMENT GROUP. PostgreSQL, Software. Verfügbar unter: https://www.postgresql.org/, Abgerufen am 29.11.2020.

[17] REDOCLY. Redoc, Software. Verfügbar unter: https://github.com/Redocly/redoc, Abgerufen am 21.11.2020.

[18] STEINER, H. Entwicklung einer Dokumenten-Scanner-App. Praktikumsbericht, Hochschule Mittweida (2020).

[19] SUTER, R. NSwag, Software. Verfügbar unter: https://github.com/RicoSuter/NSwag, Abgerufen am 11.11.2020.

[20] THOMAS, J. Bulma, Software. Verfügbar unter: https://bulma.io, Abgerufen am 21.11.2020.

[21] TOMMASI, W. Buefy, Software. Verfügbar unter: https://buefy.org, Abgerufen am 21.11.2020.

[22] VAHRENKAMP, R. UND SIEPERMANN, C. Enterprise-Resource-Planning-System. In *Gabler Wirtschaftslexikon*. Springer Gabler. Verfügbar unter: https://wirtschaftslexikon.gabler.de/definition/enterprise-resource-planning-system-51587/version-274748, Abgerufen am 23.11.2020.

[23] WEB HYPERTEXT APPLICATION TECHNOLOGY WORKING GROUP. HTML Living Standard. Spezifikation. Verfügbar unter: https://html.spec.whatwg.org, Abgerufen am 23.11.2020.

[24] WORLD WIDE WEB CONSORTIUM. Extensible Markup Language (XML) 1.1 (Second Edition). Spezifikation. Verfügbar unter: https://www.w3.org/TR/xml11, Abgerufen am 23.11.2020.

[25] YOU, E. Vue Class Component, Software. Verfügbar unter: https://class-component.vuejs.org, Abgerufen am 21.11.2020.

[26] YOU, E. Vue.js, Software. Verfügbar unter: https://vuejs.org, Abgerufen am 21.11.2020.

[27] YOU, E. Vuex, Software. Verfügbar unter: https://vuex.vuejs.org, Abgerufen am 21.11.2020.

[28] ZABRISKIE, M. Axios, Software. Verfügbar unter: https://github.com/axios/axios, Abgerufen am 21.11.2020.

Erklärung

Hiermit erkläre ich, dass ich meine Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die Arbeit noch nicht anderweitig für Prüfungszwecke vorgelegt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Mittweida, 04.12.2020