
PRAKTIKUMSBERICHT

Herr
Hannes Steiner

Entwicklung einer Dokumenten-Scanner-App

**Digitalisierung der Verwaltung an der Hochschule
Mittweida**

2020

Fakultät **Angewandte Computer- und
Biowissenschaften**

PRAKTIKUMSBERICHT

Entwicklung einer Dokumenten-Scanner-App

**Digitalisierung der Verwaltung an der Hochschule
Mittweida**

Autor:
Hannes Steiner

Studiengang:
Softwareentwicklung

Seminargruppe:
IF17wS-B

Matrikelnummer:
46540

Erstprüfer:
Prof. Dr. Mark Ritter

Zweitprüfer:
N.N.

Mittweida, März 2020

Bibliografische Angaben

Steiner, Hannes: Entwicklung einer Dokumenten-Scanner-App, Digitalisierung der Verwaltung an der Hochschule Mittweida, 31 Seiten, 7 Abbildungen, Hochschule Mittweida, University of Applied Sciences, Fakultät Angewandte Computer- und Biowissenschaften

Praktikumsbericht, 2020

Dieses Werk ist urheberrechtlich geschützt.

Referat

In dem vorliegendem Praktikumsbericht...

I. Inhaltsverzeichnis

Inhaltsverzeichnis	I
Abbildungsverzeichnis	II
Tabellenverzeichnis	III
Abkürzungsverzeichnis	IV
1 Einleitung	1
2 Hochschule Mittweida	2
3 Problemstellung	3
3.1 Digitalisieren der Klausur-Daten	3
3.2 Klausuren-Vorlage	3
3.3 Vorlage verbessern	4
3.4 Weitere Anmerkungen	4
4 Anforderungen	5
5 Konzept der Dokumenten-Scanner-App	7
6 Entwicklung des ersten Prototyps	10
6.1 Anforderungsplanung	10
6.2 Planung und Vorbereitung bzw. Analyse und Definition	10
6.3 Grundlagen	11
Model-View-ViewModel:	11
Redux.js:	12
6.4 Entwurf und Design	12
6.5 Implementierung	15
6.5.1 Registrieren und Anmelden	16
6.5.2 Scan-Vorlagen erstellen und speichern	17
Scan-Vorlage erstellen	17
Regionen erstellen	17
Links erstellen	18
Vorlage speichern	19
6.5.3 Vorlage verwenden	20

6.6	"Abnahme"	21
7	Weitere Entwicklung und Besonderheiten	23
7.1	API	23
8	Grenzen der App	24
8.1	Probleme beim Erkennen von Dokumenten	24
8.2	Probleme der Klausur-Vorlage beim Scannen	24
8.3	Weitere ...	24
9	Templates	25
10	Ausblick	26
A	Workflow	27
B	Tätigkeitsbericht	28
	Literaturverzeichnis	30

II. Abbildungsverzeichnis

3.1 Umsetzung der Klausuren-Vorlage der Fakultät CB.	4
6.1 Architektur Schemata	14
6.2 Willkommen- Registrier- und Anmelde-View	16
6.3 Die ersten Views zur Erstellung einer Scan-Vorlage	17
6.4 Views zur Erstellung von Regionen	18
6.5 Seiten-Vorschau-View	19
6.6 Views zur Erstellung von Links	20

III. Tabellenverzeichnis

IV. Abkürzungsverzeichnis

- API application programming interface, auf deutsch Programmierschnittstelle, Seite 15
- bzw. beziehungsweise, Seite 3
- CRUD Das Akronym CRUD steht für Create/Erstellen, Read/Lesen, Update/Aktualisieren und Delete/Löschen und umfasst die vier grundlegenden Operationen persistenter Speicher, Seite 6
- Fakultät CB Fakultät für angewandte Computer- und Biowissenschaften, Seite 2
- HSMW Hochschule Mittweida - university of applied science, Seite 2
- IDE integrated development environment, auf deutsch Integrierte Entwicklungs-umgebung, Seite 15
- MVVM Model View ViewModel, Seite 11
- OCR optical character recognition, deutsch: optische Zeichen Erkennung und im deutschen Synonym für Texterkennung, Seite 5

1 Einleitung

Digitalisierung wird gängig als Integration von digitaler Technologie in den Alltag verstanden, und soll helfen Zeit einzusparen. Mit diesem Gedanken initiierten die Mitarbeiter Holger Langner und Falk Schmidsberger der Hochschule Mittweida, das Projekt *Memo Space*. Im Zuge dessen sollen kleinere Forschungsergebnisse entstehen, die richtungsweisend für die Digitalisierung der Verwaltung von Lehr- und Forschungseinrichtung sind.

Im Rahmen eines zwölfwöchigem Forschungspraktikums an der Hochschule Mittweida arbeiteten der Student Tobias Kallauke und der Verfasser des Berichts, gemeinsam an einem Forschungsprojekt von Memo Space. Dabei entwickelten sie ein Software-System, mit dem die Arbeit von vielen Hochschulmitarbeitern erleichtert und auch Zeit eingespart werden soll.

2 Hochschule Mittweida

Die Hochschule Mittweida - university of applied science (HSMW) wurde vor über 150 Jahren gegründet. Heute lehrt und forscht sie mit ca. 6000 Studenten in fünf Fakultäten und vier Forschungsschwerpunkten [2] . Eines der Schwerpunkte ist die angewandte Informatik, in dem Memo Space angesiedelt ist.

Nach eigener Einschätzung, schreibt jeder Student ca. 5 Prüfungen pro Semester, was bedeutet, dass im Jahr um die 60.000 Klausuren kontrolliert werden. Dazu kommt, dass nachdem die Klausuren kontrolliert wurden, die Zensur, sowie die Eckdaten der Studenten, die an der Prüfung teilgenommen haben, in ein digitales Format gebracht werden muss. Grund dafür ist, dass die Übertragung der Noten in das Notensystem der Hochschule. Da die Mitarbeiter der Fakultät *Angewandte Computer- und Biowissenschaften* (Fakultät CB) Holger Langner und Falk Schmidsberger selbst Klausuren kontrollieren und die Problematik genau kennen, entstand hier eines der ersten Ideen für Memo Space.

3 Problemstellung

An der Kontrolle von Klausuren sitzen zum Ende eines Semesters Hochschulmitarbeiter über Tage dran. Diese Aufgabe muss stets mit hoher Konzentration erledigt werden, und lässt sich aber in den meisten Fällen nur schwer durch Maschinen ersetzen. Unter keinen Umständen dürfen bei der Bewertung Fehler vorkommen, was jedoch bei der kognitiven Last der Prüfer immer wieder passiert. Auch nach der Durchsicht der Prüfungsaufgaben ist eine hohe Achtsamkeit wichtig. Denn anschließend wird die Benotung in eine digitale Tabelle überführt. In diese muss die Matrikelnummer, der Vor- und Nachname, sowie die Note des Studenten eingetragen werden. Hier kommt es vor allem bei der Matrikelnummer und der Zensur auf die Richtigkeit jedes Zeichens drauf an.

3.1 Digitalisieren der Klausur-Daten

Für genau diesen Vorgang des Digitalisierens wird eine Lösung gesucht. Die Prüfer sollen so bequem und möglichst zeitsparend diese Aufgabe verrichten, ohne dabei ihre Aufmerksamkeitsspanne zu überlasten. Des Weiteren müssen die Ergebnisse der Prüfungen, sowie die Eckdaten der Studenten in ein geeignetes digitales Format gebracht werden, um es der Notenfreigabe weiterzuleiten. Darüber hinaus empfiehlt es sich, digitale Kopien der Klausuren abzuspeichern, um sie nicht nur analog zu archivieren.

3.2 Klausuren-Vorlage

Eine Klausuren-Vorlage bzw. ein Gestaltungsleitfaden für Klausuren der Fakultät *Angewandte Computer- und Biowissenschaften* bietet außerdem die Möglichkeit der Kontrolle des Prüfers an. Genauer ist es durch das vorgegebene Layout der Klausur möglich, einen Teil der Arbeit des Prüfenden auf Fehler zu untersuchen. Auf dem Deckblatt der Klausur ist eine Tabelle, mit drei Zeilen und für jede Aufgabe eine Spalte. In der ersten Zeile befinden sich die Nummern der Aufgaben. In der zweiten, die zu erreichenden Punkte der Aufgabe. Und in der dritten Zeile trägt der Prüfer die erbrachten Punkte des Studenten ein. Unter der Tabelle befindet sich ein Feld für die erreichte Gesamtpunktzahl, sowie ein Feld für die aus den Punkten resultierende Note. Die Punkte pro Aufgabe, die Gesamtpunktzahl und die Zensur stehen in Relation zu einander, so dass aus den Punkten der Aufgaben die beiden anderen Felder errechnet werden und mit den Ergebnissen des Prüfers abgeglichen werden könnten. Eine weiteres Merkmal der Klausuren-Vorlage ist ein Feld, für die vom Studenten erreichten Punkte über jeder Aufgabenstellung. Die dortige Angabe sollte mit der, in der Tabelle auf dem Deckblatt übereinstimmen und bietet somit noch eine weitere Möglichkeit der Kontrolle an.

(a) Deckblatt der Klausur

(b) 1. Seite der Klausur

Abbildung 3.1: Umsetzung der Klausuren-Vorlage der Fakultät CB.

Die Bilder sind beim Erstellen einer Scan-Vorlage mit der App entstanden.

3.3 Vorlage verbessern

Nachdem ein erster Prototyp zur Digitalisieren der Klausur-Daten entstanden ist, soll außerdem Prüfungsvorlagen entstehen, die für die Digitalisierung optimiert ist. Dabei sollen Probleme die beim Einstellen der aktuellen Vorlage erkannt wurden, behoben werden.

3.4 Weitere Anmerkungen

Ferner soll bei der Lösung von der Anschaffung neuer Technologie und Geräte abgesehen werden. Grund dafür sind neben den Anschaffungskosten, die Idee, dass das Ergebnis des Forschungsprojekts in weiteren Lehr- und Forschungseinrichtung Anwendung finden sollte. Außerdem hat so gut wie jeder Mitarbeiter an einer Lehr- und Forschungseinrichtung ein eigenes oder Zugang zu einem Smartphone oder Tablet, welche durch die eingebaute Technik in der Lage sind, diese Aufgabe zu übernehmen.

4 Anforderungen

Es soll eine iOS-App entstehen mit der Klausuren digitalisiert werden können. Genauer müssen die, für die Notenfreigabe relevanten Daten der Klausur, in ein tabellarisches Format gebracht werden. Um unterschiedliche Klausuren oder auch andere Dokumente zu unterstützen, soll das Anlegen von Scan-Vorlagen in der App möglich sein. Hierfür (aber auch zum Einscannen) benötigt die App die Berechtigung für das Kamera-System. Diese muss, bei der ersten Benutzung einmalig erteilt werden.

- Bild Scan-Vorlage: Aus Foto wird Dokument extrahiert und Auf der Seite werden Regionen markiert. -> Sammlung an Seiten mit Regionen

Eine Scan-Vorlagen bestehen aus zwei Komponenten. Die eine sind zugeschnittene Fotos der Seiten eines Dokuments (3.1). Und die zweite sind die Regionen auf den Bildern, wo sich die zu digitalisierenden Informationen befinden. Dem Benutzer muss es möglich sein, zuerst die Fotos zu machen und anschließend Regionen auf den Seiten zu markieren. Weiter benötigen die Regionen Namen bzw. Typen, die der Benutzer angeben muss. Die Angabe des Typs ist wichtig, da dadurch eindeutig wird, ob es sich um die Note oder die Matrikelnummer des Studenten handelt. Diese Eindeutigkeit wird nicht nur für die automatische Erstellung der Tabelle benötigt, sondern auch für die Texterkennung.

Optical character recognition (OCR), auf deutsch optische Zeichen Erkennung, soll dazu benutzt werden, die Informationen der Regionen zu digitalisieren. Um die Texterkennung zu verbessern, soll der Typ der Regionen verwendet werden. Damit könnte dem OCR mögliche Ergebnisse mitgeteilt oder die Ergebnisse angepasst werden. Beispielsweise bestehen Zensuren immer aus Dezimalzahlen von Eins bis Sechs und mit nur einer Nachkommastelle. Wird der Buchstabe O anstelle der Ziffer Null erkannt, kann so der Fehler korrigiert werden.

Zudem muss die Texterkennung auf dem Gerät selbst statt finden. Allerdings darf die App auch Texterkennung auf externen Servern unterstützen. Hierfür müssen dann die entsprechenden API-Schnittstellen implementiert oder für einen eigenen Server entwickelt werden.

Die digitalisierten Daten, die für die Texterkennung entstandenen Dokumenten-Bilder und die erstellten Scan-Vorlagen, soll außerhalb der App auf einem Server gespeichert werden. Somit ist die Möglichkeit gegeben, die Vorlagen wieder zu verwenden und anderen Benutzern der App zur Verfügung zu stellen. Des Weiteren hat diese zentralen Stelle den Vorteil alle anfallenden Daten zu verwalten, was die Benutzung der App trotz vieler Benutzer vereinfacht.

Damit Synchronität der Daten auf den Geräten und dem Server gewährleistet werden kann, benötigt die App hierfür ebenfalls Schnittstellen. Diese sollten den standardmäßigen *CRUD*-Operationen entsprechen.

Da die gesamte Kommunikation über das Internet geschieht, muss das Softwaresystem die üblichen Datenschutz- und Datensicherheit-Richtlinien entsprechen, bzw. implementieren.

5 Konzept der Dokumenten-Scanner-App

Wie in der Problemstellung und Anforderung beschreiben, soll die Dokumenten-Scanner App wichtigen Daten auf dem Deckblatt einer Klausur, wie Vor- und Nachname des Studenten, seine Matrikelnummer sowie die Note erkennen, digitalisieren und in ein für die Notenfreigabe geeignetes Format bringen. Die digitalisierten Daten sollen anschließend an einen Server gesendet werden, wo sie und die beim Einscannen entstandenen Bilder gespeichert werden. Verwendet eine einzuscannende Klausur die Klausuren-Vorlage der Fakultät CB oder ähnliche, die eine Punkteübersicht haben, dann soll es außerdem möglich sein, die Punkte sowie die Note auf Richtigkeit zu überprüfen.

Zur Digitalisierung und Kontrolle der Daten werden Scan-Vorlagen verwendet. Diese muss vor dem Einscannen der ausgefüllten Klausuren erstellt werden, um zu gewährleisten, dass die richtigen Daten digitalisiert werden. Beim Erstellen einer Scan-Vorlage geht man wie folgt vor:

- Zuerst wird jede Seite der Klausur fotografiert (6.3b). Dabei wird in jedem Bild das Dokument erkannt, vom Hintergrund getrennt oder genauer gesagt ausgeschnitten und perfekt ausgerichtet (3.1). Zudem wird der Kontrast erhöht, so dass die Schrift leichter lesbarer wird.
- Anschließend markiert man diejenigen Regionen auf jeder Seite, die zu digitalisieren und/oder kontrollieren sind (6.5b). Zusätzlich muss jeder Region ein Name und einer der folgenden Typen zugeordnet werden: Unbekannt, Name, Matrikelnummer, Seminargruppe, Punkte und Note. Mithilfe des Typs wird die Texterkennung verbessert und falsche Ergebnisse automatisch korrigiert. Dazu später mehr.
- Des Weiteren kann man aus allen markierten Regionen diese auswählen, die in Relation stehen. Diesen Relationen werden danach noch weitere Eigenschaften zugeteilt, um sie später bei der Kontrolle richtig zu analysieren. So eine Eigenschaft könnte sein, dass es sich um einen Vergleich zwischen zwei in Relation stehenden Regionen handelt. Z. B. ist die eine Region die Zelle in der Punkteübersicht-Tabelle, in der die erreichten Punkte zu Aufgabe 1 rein geschrieben werden sollen. Und die zweite Region die Stelle über der Aufgabenstellung von Aufgabe 1, in der ebenfalls die erreichten Punkte eingetragen werden sollen. Weitere Beispiele für solche Relationen und Eigenschaften sind die Summanden für die erreichte Gesamtpunktzahl und die Summe selbst, oder auch noch die daraus resultierende Note.
- Als letztes speichert man die Vorlage ab, welche dann automatisch an einen Server gesendet wird, so dass andere diese Vorlage ebenfalls nutzen können.

Nach der Erstellung einer Vorlage kann das Dokument digitalisiert werden. Dazu scannt

man die Seiten in derselben Reihenfolge, wie in der Scan-Vorlage ein. Die App digitalisiert mithilfe von OCR den Inhalt, in den vordefinierten Regionen. Der Typ der Region nimmt nun Einfluss auf das Ergebnis. Durch ihn wird während der Worterkennungsphase eine Liste an vordefinierten möglichen Ergebnissen eingespeist. Diese Liste hat Vorrang vor dem Standard-Lexikon und verbessert dadurch die Texterkennung. Ein Beispiel für solch eine Liste könnten, alle möglichen Noten sein, wie schon in dem Kapitel 4 Anforderungen beschrieben. Auch vorstellbar ist, dass alle Seminargruppen oder Namen von Personen, die an der Klausur teilgenommen haben, dort verwendet werden.

Weiter wäre eine zusätzliche Möglichkeit der Korrektur erdenklich. Angenommen, jemand verschreibt sich bei seiner Seminargruppe und vergisst ein Zeichen. Die Texterkennung erkennt zwar jedes Zeichen richtig, jedoch ist das Ergebnis keine korrekte Seminargruppe. Deshalb könnte man auch im Nachhinein das Ergebnis mit einer Seminargruppe ersetzt, die die größte Übereinstimmung mit dem erkannten Wort hat. Bei diesem Vorgehen sollten natürlich auch nur die Seminargruppen verwendet werden, die tatsächlich die Klausur geschrieben haben. Ähnliches gilt wieder für die Namen oder Matrikelnummern der Studenten.

Zudem kommt noch, dass die eingetragenen Punkte kontrolliert werden. Möglich wäre auch, die Note zu kontrollieren, jedoch müsste dafür der jeweilige Notenspiegel integriert werden. Für die Umsetzung dieser Kontrollinstanz werden die angelegten Relationen zwischen den Regionen verwendet. Genauer soll es zwei Typen von Links bzw. Relationen geben.

- Der erste Typ ist zum Vergleichen. Dafür benötigt es genau zwei Regionen und wie der Name des Typs ahnen lässt, wird der Inhalt der beiden Regionen auf Gleichheit überprüft. Wie schon vorher erwähnt, sollen so die Punkte auf dem Deckblatt mit den Punkten über den Aufgaben verglichen werden.
- Der zweite Typ ist zum Summieren. Genauer ist dieser Typ nur für die Punkte auf dem Deckblatt der Klausur gedacht. Dabei wählt man alle Regionen aus, deren Inhalt summiert werden sollen und die Region, in die der Prüfer die berechnete Gesamtpunktzahl rein schreibt. Somit kann die Berechnung des Prüfers überprüft werden.

Mithilfe der beiden Typen, wird dem Benutzer nach der Texterkennung mitgeteilt, wo Fehler vorliegen.

Allerdings heißt das nicht, dass das OCR selbst keine Fehler macht. Aus diesem Grund ist die sogenannte confidence des OCR bei jeder Region sichtbar. Diese sagt aus, wie sicher sich die Texterkennung ist, das das Ergebnis stimmt. Jedoch kann es auch hier falsche Positiv-Ergebnisse geben, weshalb auch das kein perfekter Indikator für die Richtigkeit ist. Deshalb wird im Anschluss an die Texterkennung die Möglichkeit geboten, die Resultate noch einmal zu überprüfen und zu korrigieren, bevor sie an den

Server zum Abspeichern gesendet werden. Genauer können die Ergebnisse der Texterkennung zu jeder Region einzeln bearbeitet werden, wodurch sich die Fehlermeldung durch einen Link ändern kann.

6 Entwicklung des ersten Prototyps

Dieses Kapitel beschreibt die Entwicklung der iOS-App in einer ähnlichen Reihenfolge, wie das bekannte Wasserfall-Model über die Verwaltung der Entwicklung großer Softwaresysteme nach Winston W. Royce. - Quelle:...

6.1 Anforderungsplanung

Vor Beginn des Praktikums wurde diskutiert und kalkuliert, welches Thema aus dem Projekt Memo Space für ein zwölfwöchiges Praktikum geeignet ist. Dabei entstand eine Art Durchführbarkeits- / Machbarkeitsstudie, welche zur Entscheidung führte, ein Dokumenten-Scanner-Softwaresystem umzusetzen. Aufgrund der Kenntnisse und Erfahrung, soll ein Backend mit entsprechenden Schnittstellen und einer Android-App von Tobias Kallauke umgesetzt werden, während vom Verfasser eine iOS-App gefordert ist. Die Anwendung soll den Anforderungen, die im gleichnamigen Kapitel 4 zu finden sind, erfüllen. Für weitere Details über den Server und die Android-App siehe im Praktikumsbericht von Tobias Kallauke.

6.2 Planung und Vorbereitung bzw. Analyse und Definition

Die Aufgaben bzw. Ziele dieser Phase sind:

1. die Auseinandersetzung mit der Problemstellung und den Anforderungen,
2. das Betreiben von einer Problemanalyse,
3. die Entwicklung von Ideen und eines genauen Konzepts der iOS-App sowie
4. das Entwickeln eines ersten Prototyps, als Machbarkeitsnachweis.

Bei der Entstehung des Konzepts, welches im Kapitel 5 zu finden ist, spielen die Dokumentationen der Frameworks *Vision*¹, *VisionKit*² und *PhotoKit*³ von *Apple* eine entscheidende Rolle. Aus ihnen wird klar, welche Funktionen der App noch zu entwickeln und welche in Frameworks schon vorhanden sind. Z. B. ist das Erkennen und das gerade Ziehen von Dokumenten in Echtzeit, sowie die Texterkennung auf Bildern in *Vision* und *VisionKit* enthalten. Bei der Entwicklung des ersten Prototyps half eine Beispiel-App von *Apple* [1], die das Erkennen von Objekten in Standbildern mithilfe der genannten Frameworks umsetzt.

¹ Dokumentation von Vision - <https://developer.apple.com/documentation/vision>

² Dokumentation von VisionKit - <https://developer.apple.com/documentation/visionkit>

³ Dokumentation von PhotoKit - <https://developer.apple.com/documentation/photokit>

Durch die Auseinandersetzung mit den Bibliotheken konnte festgestellt werden, dass die zu entwickelnde App nur Geräte mit iOS 13.0 oder neuer unterstützen werden kann. Grund dafür sind die *Apple Frameworks SwiftUI und VisionKit*. *SwiftUI* bietet die Möglichkeit, Benutzeroberflächen für alle *Apple*-Plattformen in *Swift* zu erstellen. Die deklarative *Swift*-Syntax ist einfach zu lesen und schnell zu schreiben, so dass es möglich ist, die App in wenigen Wochen für iPhone und iPad zu schreiben. Als Alternative gibt es noch *UIKit* oder auch *AppKit*, die unter alle iOS Versionen funktionieren. Diese Frameworks sind allerdings nicht deklarativ sodass, Views sowohl im Code als auch in Interface-Dateien getrennt voneinander erstellt und konfiguriert werden müssen [5]. Dadurch dauert die Entwicklung einer App, im Gegensatz zu *SwiftUI* deutlich länger, wie man auch hier in dem Video ⁴ von Paul Hudson einem in der Swift-Community sehr bekannten Programmierer und Autor sieht. Sehr ähnliche Erfahrung hat der Verfasser vor dem Praktikum mit den beiden Frameworks gemacht. *VisionKit* dagegen, ist das Framework zum Scannen der Dokumente. Auch hierfür gibt es eine Alternative. Das Framework ist von den Entwicklern von *WeTransfer* und funktioniert auch auf älteren iOS Versionen. Allerdings unterstützt *WeScan*⁵ noch kein stapelweises Scannen. Das bedeutet, man kann immer nur ein Foto machen, welches erst abgespeichert werden muss, bevor man das nächste machen kann. Das ist für den Benutzer nicht bequem und spart wahrscheinlich auch keine Zeit. Zu Informationen zu anderen verwendeten Frameworks siehe im Kapitel 6 Abschnitt 6.5.

Abschließend zu dieser Phase ist zu erwähnen, dass das Scrum-Konzept, welches sich für agile Softwareentwicklung anbietet, zur Projektplanung und zum Projektmanagement verwendet wurde. Als Versionsverwaltung der Software wurde Git in Kombination mit GitHub Issues und GitHub Projects als Planungstools benutzt. So konnte der Verfasser selbstständig jedem Sprint Aufgaben zuordnen und den Fortschritt nachvollziehen.

6.3 Grundlagen

In den folgenden zwei Absätzen werden wichtige Konzepte und Grundwissen vermittelt, die im folgenden Kapitel Entwurf und Design nochmal aufgegriffen werden.

Model-View-ViewModel: Das Entwurfsmuster Model View ViewModel (MVVM) entstand bei Microsoft im Jahr 2005 mit der *Windows Presentation Foundation* (WPF) und *Silverlight-Technologien*. MVVM verwendet das Konzept eines Schichtmodells und ist eine abstrakte Darstellung einer Benutzeroberfläche, in Form einer Klasse, wie man sie unter objektorientierten Programmiersprachen kennt. Diese Klasse enthält Daten, die auf der Benutzeroberfläche angezeigt werden sollen und Anweisungen, die auf der

⁴ SwiftUI vs UIKit – Comparison of building the same app in each framework - <https://www.youtube.com/watch?v=qk2y-TiLDZo>

⁵ WeScan GitHub Repository - <https://github.com/WeTransfer/WeScan>

Benutzeroberfläche aufgerufen werden können. Dieses sogenannte ViewModel, weiß nichts von Views, wie es sonst bei anderen Entwurfsmustern üblich ist, um Daten auf der Benutzeroberfläche anzuzeigen. Stattdessen verwendet eine MVVM-View eine Bindungsfunktion (data binding) zur bidirektionalen Zuordnung von Daten aus dem View-Model zu den jeweiligen Eigenschaften auf der View. Z. B. Einträge in einer Dropdown-Menü. Aber auch das binden von Daten aus dem Model zu Benutzereingaben durch Maus, Tastatur oder Touch-Screens ist möglich. Beispielsweise kann ein Mausklick eine Anweisung in dem ViewModel auslösen. Diese verändert einen Wert im Model, wodurch die View durch data binding aktualisiert wird. [4] [3]

Redux.js: Redux.js ist eine Bibliothek⁶ für die Programmiersprache JavaScript. Diese stellt einen sogenannten vorhersehbaren Zustandscontainer⁷ für JavaScript Anwendungen bereit. In diesem Container wird der Zustand der gesamten Anwendung in einem Objektbaum innerhalb eines einzigen Speichers gespeichert. Diesen Baum kann man sich vorstellen, wie eine Matroschka die weitere Puppen in sich hat. Der wichtige Unterschied zwischen einer Matroschka und einem Baum ist jedoch, dass in eine Puppe genau nur eine andere gesteckt werden kann. Beim einem Objektbaum hingegen können mehrere Objekte in ein Objekt "gesteckt" werden und so weiter. Im Bezug auf einen Zustandscontainer enthält dieser dann States (Objekte) auf deutsch Zustände und Daten, die unter anderem auf der Benutzeroberfläche angezeigt werden sollen. Diese Aufteilung in die States ist dazu gedacht, einer View oder mehrere zusammenhängende Views nur die wirklich benötigten Daten bereit zustellen. Diese Struktur erleichtert das Testen oder Untersuchen der Anwendung und ermöglicht es, durch hinzufügen eines neuen States den aktuellen Entwicklungsstand der Anwendung beizubehalten und dadurch den Entwicklungsprozess zu beschleunigen.

Ein weiteres wichtiges Prinzip⁸ von Redux ist, dass die Daten in den States Schreibgeschützt sind. Die einzige Möglichkeit den Zustand zu ändern, besteht darin, eine Aktion auszulösen, die beschreibt, was passiert. Dadurch wird sichergestellt, dass weder die Views noch Netzwerk-Rückrufe jemals direkt an den Zustand schreiben werden. Stattdessen bringen sie nur die Absicht zum Ausdruck, den Zustand zu verändern und lösen eine Aktion aus. Da alle Änderungen zentralisiert sind und eine nach der anderen in einer strikten Reihenfolge erfolgen, gibt es weniger Programmfehler-Quellen.

6.4 Entwurf und Design

- Quelle Schubert Folien

Eine typische Aufgabe dieser Entwurfsphase ist die Entscheidung über die Datenhal-

⁶ Redux.js Github Repository - <https://github.com/reduxjs/redux>

⁷ Redux Core Concepts - <https://redux.js.org/introduction/core-concepts>

⁸ Redux Three Principles - <https://redux.js.org/introduction/three-principles>

tung, die Verteilung im Netz und die Benutzeroberfläche des Software-Systems. Jedoch standen diese Grundsatzentscheidungen schon zu Beginn des Praktika, durch die Kenntnisse von Tobias Kallauke und dem Verfasser fest. Der Server verwendet zur Datenspeicherung Postgres, eine relationale Datenbank, sowie das Dateiverzeichnis für Bilder, während die App keine Daten persistent speichert. Durch die Aufteilung von App und Server handelt es sich um eine sogenannte Client/Server-Architektur und für die Benutzeroberfläche wird das deklarative SwiftUI verwendet. Bei Fragen zum Backend und der Android App siehe im Praktikumsbericht von Tobias Kallauke.

Jedoch mussten noch Arbeit in den Workflow und in die Architektur der App gesteckt werden. Zur Definition einzelner Teil-Workflows, die zusammenhängende Aufgaben umfassen, wurden eine Art Zustandsautomaten bzw. Flussdiagramm beschrieben. Ein Beispiel, was noch einmal geändert in der App verwendet wurde befindet sich im Anhang A. Die Modelle halfen dabei Views zu entwickeln und deren Design festzulegen. Das Aussehen der App sollten sich jedoch im Laufe der Zeit noch ändern, denn erst einmal stand die Umsetzung des Konzepts im Vordergrund.

Aus den Designs und den Teil-Workflows heraus entstand ein grober Plan, wie die Daten in der App gehandhabt werden sollten. Da *SwiftUI* das Entwurfsmuster MVVM umsetzt, benötigt es eine Struktur für das ViewModel und für den Datenfluss der asynchronen Server-Rückrufe. Jedoch stellte sich die Entwicklung dieser Strukturen, auch schon während des allerersten Prototyps, als problematisch heraus. Denn auch bei steigender Komplexität soll das ViewModel noch übersichtlich bleiben, um eine schnelle Weiterentwicklung zu gewährleisten. Aus diesen Gründen begann die Suche nach einer besseren Lösung, bei der die JavaScript-Bibliothek *Redux.js* zur Verwaltung von Zustandsinformationen in Webanwendung entdeckt wurde.

Redux hilft durch dessen Konzept, komplexe Views mit vielen Daten schnell und einfach zu implementieren. Auch werden so Serverantworten und zwischengespeicherte Daten sowie lokal erstellte Daten, die noch nicht auf dem Server gespeichert wurden, strukturiert und zentral abgelegt. Das erleichtert nicht nur das Wiederverwenden von Daten, sondern spart auch wiederholte Server-Aufrufe ein. Es erschien nun möglich, trotz der begrenzten Zeit im Praktikum, möglichst viel mit wenig Fehlern umzusetzen. Denn *Redux* ermöglicht durch die modulare Aufteilung des State Containers eine schnelle Weiterentwicklung der App, auch wenn die Komplexität der Anwendung steigt. Und die Vorteile von *Redux* hinsichtlich des Testens und Untersuchens der App sollten helfen, Fehler möglichst schnell ausfindig zu machen, trotz asynchroner Programmabschnitte.

Daher lag es nah die wesentlichen Konzepte von *Redux* umzusetzen und einen *Redux* ähnlichen State Container, als ViewModel zu implementieren. Aus der Definition von Teil-Workflows sollte der Container oder auch Store genannt mindestens 5 States haben:

- für Authentifizierung sowie Registrierung,
- für das Anlegen von Vorlagen, um Zwischenergebnisse zu speichern,
- für das Ausführen von Server-Aufrufen zum Speichern und Abrufen von Vorlagen,
- für die Steuerung des Workflows bzw. den stellvertretenden Views sowie
- für sonstige Daten, die sehr häufig verwendet werden.

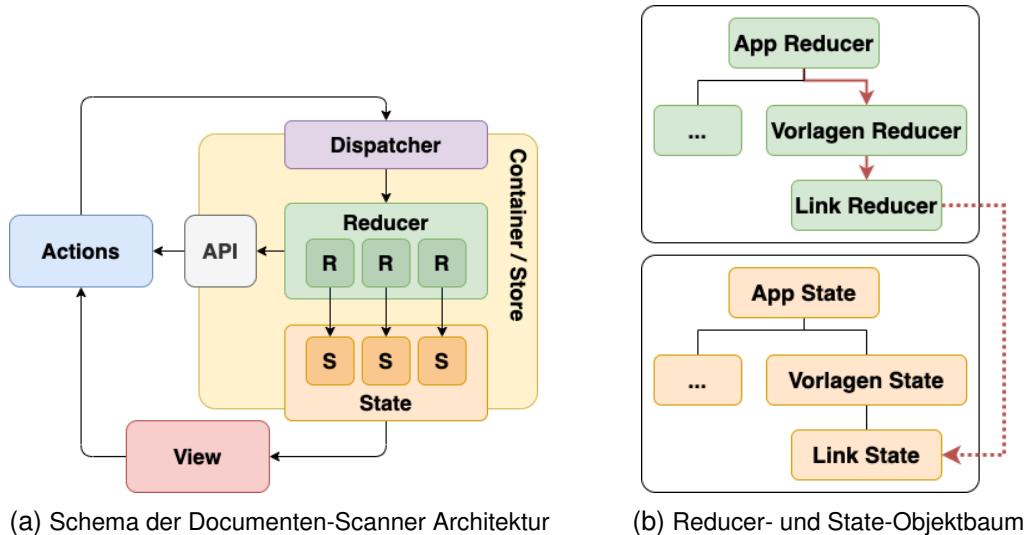


Abbildung 6.1: Architektur Schemata

Das Schema 6.1a zeigt, die in der App umgesetzte Redux ähnliche Struktur. Über eine View (rot) können Aktionen (blau) aufgerufen werden. Diese gelangt zuerst in einen sogenannten Dispatcher (lila), welcher als Verteiler dient. Anhand der Art einer Aktion, wird der entsprechende Reducer (grün) vom Dispatcher beauftragt, die Aktion auszuführen. Ein Reducer ist für genau eine Art von Aktionen zuständig. Allerdings ist, wie in der Abbildung 6.1b zu sehen ist, eine Schachtelung von Reducern (grün) möglich. Dies bedeutet auch, dass Aktionen-Arten ineinander geschachtelt werden können. Das hat den Hintergrund, dass die States wie ein Objektbaum aufgebaut sind. So hat jeder State seinen eigenen Reducer, was für die oben erwähnte Modularität sorgt. Möchte man beispielsweise eine Änderung im Link State (siehe 6.1b) vornehmen, muss die eigentliche Aktion in einer Link Reducer Aktion gekapselt und diese wiederum in einer Vorlagen Reducer Aktion. Zum ausführen der Aktion, werden dann die entsprechenden Reducer die Kapselung von außen nach innen auflösen. Nachdem die Aktion ausgeführt und eine State-Änderung herbeigeführt wurde, aktualisiert sich durch das data binding von MVVM die View. Genauer werden alle Views, die eine Bindung zu dem geänderten Datum haben, über die Änderung benachrichtigt und daraufhin aktualisieren diese sich.

Ein weiterer wichtiger Punkt sind die Server-Aufrufe. Für diese gibt es einen eigenen Reducer und eine besondere Schnittstelle, welche in der Abbildung 6.1a grau markiert und mit API beschriftet ist. Diese hat die Besonderheit selbst Aktionen an den Container zu senden. Beispielsweise löst ein Knopfdruck in einer View die Aktion aus, alle Vorla-

gen vom Server zu laden. Diese Aktion gelangt über den vorgesehenen Reducer zu der Schnittstelle. Dort wird ein entsprechender Server-Aufruf gestartet und ohne andere Prozesse anzuhalten auf den Server-Rückruf gewartet. Sobald die Antwort des Servers angekommen ist, wird diese in einer Aktion wieder zum Container gesendet. Dort wird sich wieder ein entsprechender Reducer um eventuelle Fehlerbehandlung oder das Ablegen der Vorlagen kümmern.

6.5 Implementierung

Zur Realisierung der entworfenen Systemkomponenten wurde ausschließlich, die von Apple entwickelte IDE Xcode, verwendet. Diese stellt Geräte-Simulatoren zur Verfügung, auf denen die App getestet werden kann. Die Simulatoren bieten unter anderem auch die Möglichkeit an, die Anwendung schnell auf verschiedenen iOS bzw. iPadOS Versionen zu testen. Auch ist das Testen der App auf echten Geräten durch Xcode möglich, was bei der Entwicklung unumgänglich war. Grund dafür ist die Benutzung der Kamera, die bei den Simulatoren zu gewollten Abstürzen führt, da diese keinen Zugriff auf eine Kamerasystem besitzen.

Die App wurde ausschließlich in der Programmiersprache Swift geschrieben. Zusätzlich wurden die Frameworks SwiftUI, Vision und VisionKit von Apple, sowie das Framework Kingfisher⁹, zum Downloaden und Cachen von Bildern verwendet. Das Vision-Framework führt Erkennung von Gesichts- und Gesichtsmarkierungen, Texterkennung, Barcode-Erkennung, Bildregistrierung und allgemeine Merkmalsverfolgung durch. Jedoch wurde in der App nur die integrierte Texterkennung benutzt. Es ist aber anzunehmen, dass VisionKit Algorithmen von Vision zur Dokumenten-Erkennung verwendet.

Für mehr Informationen über die Frameworks SwiftUI und VisionKit siehe im Kapitel 4 Abschnitt 6.1.

Zum Ändern und Testen des Backends wurde außerdem Visual Studio Community bzw. Visual Studio Code¹⁰ mit dem REST Client Plugin¹¹, zum Testen der Server-Schnittstellen verwendet. Das Backend konnte dann als lokaler Server auf dem Entwicklungscomputer gestartet und über die IP-Adresse 0.0.0.0 sogar im lokalen Netzwerk aufgerufen werden. Für die Verwaltung der PostgreSQL-Datenbank genügte das Tool PgAdmin4¹². Damit ist es möglich einzelne Einträge oder auch ganze Tabellen zu bearbeiten.

⁹ Kingfisher GitHub Repository - <https://github.com/onevcat/Kingfisher>

¹⁰ Visual Studio Internetseite - <https://visualstudio.microsoft.com/de/>

¹¹ REST Client, ein Visual Studio Code Plugin - <https://marketplace.visualstudio.com/items?itemName=humao.rest-client>

¹² PgAdmin Internetseite - <https://www.pgadmin.org/>

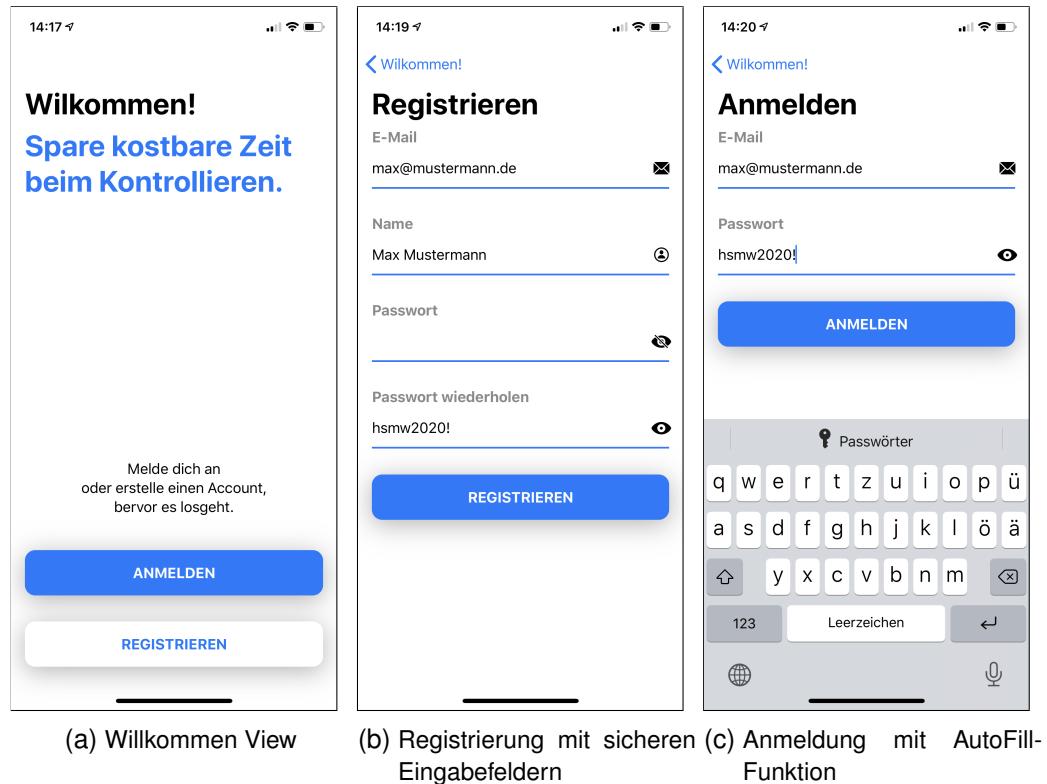


Abbildung 6.2: Willkommen-, Registrier- und Anmelde-View

6.5.1 Registrieren und Anmelden

Die technische Implementierung der Registrierung und Anmeldung beinhaltet typische Charakteristiken von Anmelde- und Registrier-Formularen. Die E-Mail-Textfelder besitzen, wie auch das Passwort-Feld der Anmeldung eine AutoFill-Funktion. Dadurch kann das iOS-Gerät Registrier- und Anmelde-Daten vorschlagen und automatisch in die entsprechenden Felder einsetzen. Dies sieht man im dritten Bild 6.2c anhand des "Passwörter"-Knopfes über der Tastatur. Weiter sind alle Passwort-Felder gesichert. Das bedeutet, dass der Inhalt zum Schutz der Privatsphäre standardmäßig ausgeblendet und mit Punkten ersetzt wird. Durch das drücken des Auges rechts des Textfeldes kann der Inhalt jedoch eingesehen werden. Im mittlerem Bild 6.2b sind wegen der Sicherheitsstandards von iOS, bei einer Aufnahme des Bildschirms nicht einmal die Punkte des ersten Passwort-Textfelds zu sehen. Zusätzlich validieren alle Text-Felder ihren Inhalt mithilfe von regulären Ausdrücken. Beispielsweise muss ein Passwort aus 8 Zeichen bestehen und zwei von den drei folgenden Eigenschaften erfüllen:

1. Es ist mindestens ein Sonderzeichen enthalten.
2. Es ist mindestens ein Großbuchstabe enthalten.
3. Es ist mindestens eine Zahl enthalten.

Die Validierung ist jedoch nicht standardmäßig, sondern wurde selbst entwickelt.

6.5.2 Scan-Vorlagen erstellen und speichern

Bei der Implementierung des ersten Arbeitsschritts der Scan-Vorlagen wurde das schriftliche Flussdiagramm, welches im Anhang A zu finden ist, leicht verändert benutzt.

Scan-Vorlage erstellen

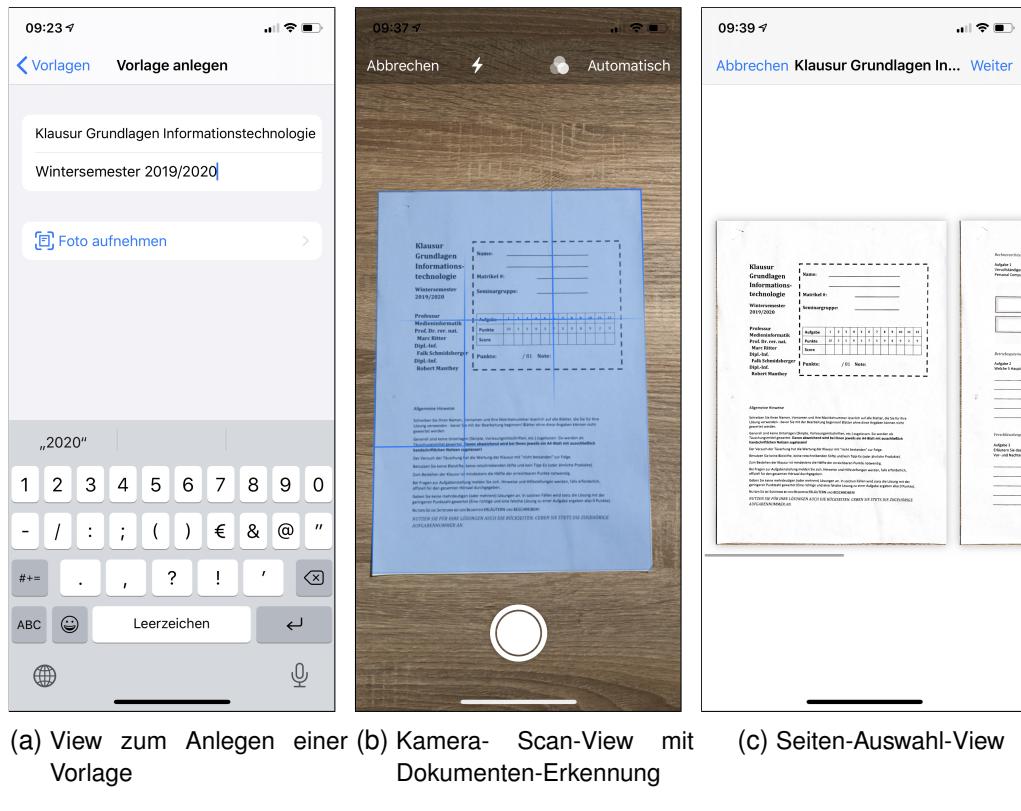


Abbildung 6.3: Die ersten Views zur Erstellung einer Scan-Vorlage

Im ersten Schritt sind ein Name und weitere Informationen, zu der Vorlage an zugegeben (6.3a), um im Anschluss die Fotos aufzunehmen. Bei der Kamera-View (6.3b) handelt es sich um die Scan-View des Frameworks VisionKit. Mithilfe von Kantenerkennung und anderen Algorithmen, die Tobias Kallaue in seinem Bericht beschreibt, kann dass Dokument sobald es erkannt ist, automatisch fotografiert werden. In Echtzeit wird das erkannte Dokument aus dem Bild ausgeschnitten und gerade gezogen. Ein manuelles Auslösen des Fotos und Anpassen der Dokumenten-Kanten im Bild, ist ebenfalls möglich. Des Weiterem werden alle erstellten Bilder zu einer Gruppe gesammelt. Diese können vor dem Abspeichern angeschaut und nochmal bearbeitet werden.

Regionen erstellen

Im nächsten Schritt sind die Regionen auf den Dokumenten-Seiten zu markieren, deren Inhalt beim Einscannen digitalisiert werden soll. Zu erst wird die gewünschte Seite aus

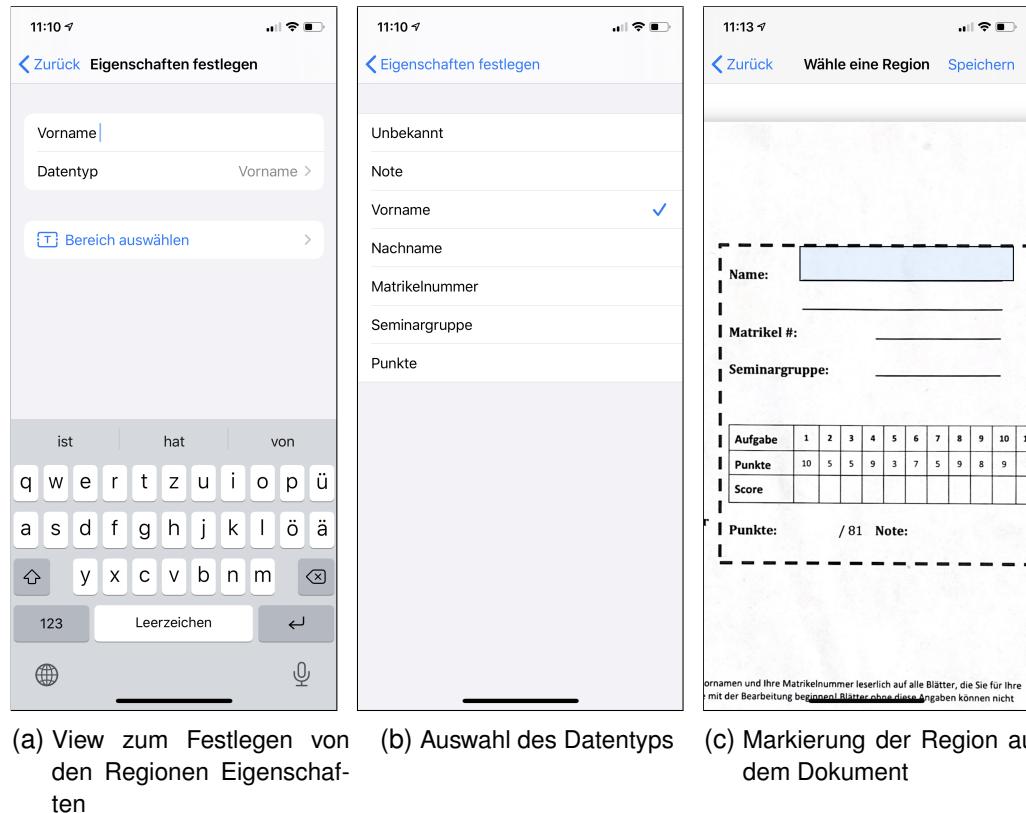


Abbildung 6.4: Views zur Erstellung von Regionen

einer Übersicht (6.3c) ausgewählt. Anschließend ist eine Vorschau der Seite mit alle eingetragenen Regionen zu sehen (6.5a bzw. 6.5b). Über einen Button können weitere Regionen hinzugefügt werden. Dazu legt man einen Namen (6.4a) und einen Datentyp (6.4b) fest. Die Datentypen sind für die Texterkennung und für die Erstellung der Tabelle für die Notenfreigabe wichtig. Dazu später mehr. Wenn die Eigenschaften der Region festgelegt sind, muss diese noch auf dem Bild markiert werden. Dazu zieht man mit einem Finger ein Rechteck in einer beliebigen Größe ein (6.4c). Die markierte Region kann anschließend noch bewegt oder neu gemacht werden. Um kleinere Regionen präzise zu markieren kann mit einer Zwei-Finger-Geste auch an die Seite heran bzw. auch heraus gezoomt werden. Dieses Vorgehen muss dann für alle nötigen Regionen auf den jeweiligen Seiten wiederholt werden.

Links erstellen

Zum Schluss können noch die sogenannte Links erstellt werden. Diese Funktion ist allerdings noch nicht sehr weit fortgeschritten und beinhaltet aktuell nur den Link-Typ zum Vergleichen von Regionen, wie im Kapitel 5 Konzept beschrieben ist. Beim Erstellen eines Vergleich-Links wählt man zwei Regionen aus, deren Inhalt nach der Texterkennung auf Gleichheit überprüft wird. Genauer werden nur die IDs der Regionen in einem Link gespeichert. Diese werden dann, beim Einscannen benötigt, um den Inhalt der Regionen zu den IDs zu Analysieren.

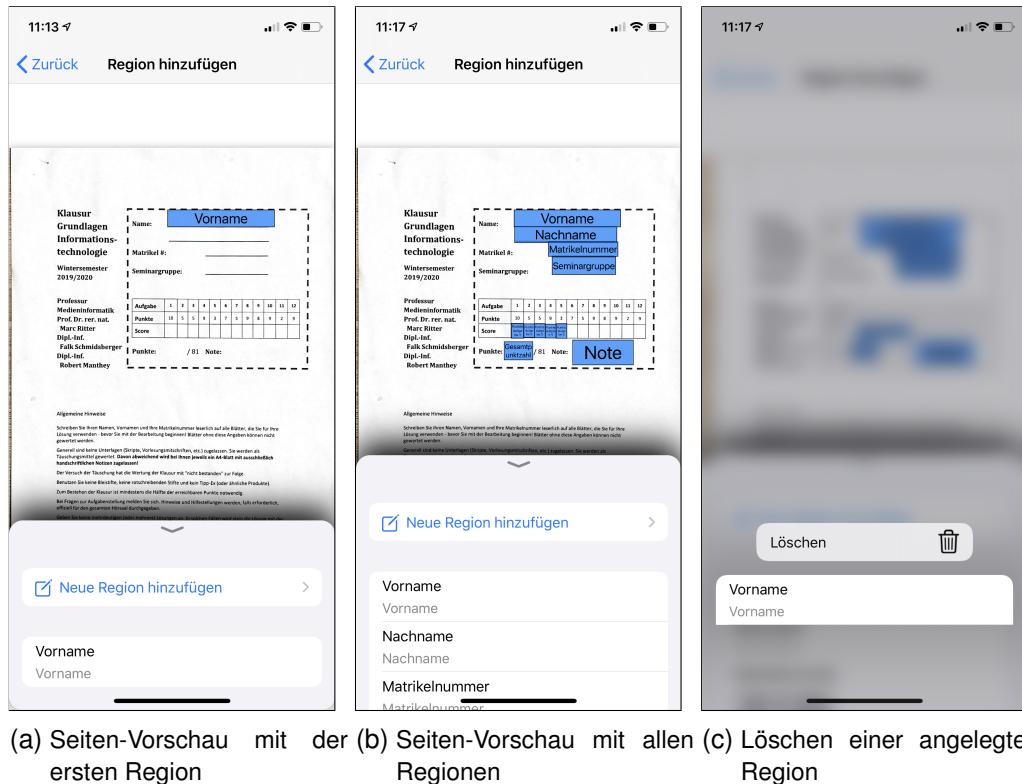


Abbildung 6.5: Seiten-Vorschau-View

Vorlage speichern

Wenn eine Vorlage gespeichert werden soll, passiert das in mehreren Schritten. Dazu werden die entsprechenden Server-Schnittstellen aufgerufen und auf die Server-Rückrufe gewartet. Für genauere Details zur API des Servers siehe im Praktikumsbericht von Tobias Kallauke.

Im folgenden Abschnitt, ist beschrieben, in welcher Reihenfolge das Hochladen einer Vorlage zum Server geschieht. Dabei werden mögliche Fehler von Seiten des Server, der Internetverbindung und des Clients ignoriert. Jedoch ist im aktuellen Stand der App das Abfangen des Fehler schon integriert, die Fehlerbehandlung aber noch nicht.

1. Der Name und die Beschreibung der Vorlage sowie die Liste an Links wird gesendet. In der Antwort des Servers befindet sich dann die Vorlagen-ID, die in den nächsten Schritten benötigt wird.
2. Die Bilder der Seiten werden gesendet. Als Antwort zu jedem Bild wird der Pfad geschickt, wo das Bild gespeichert wurde. Dieser Pfad wird im nächsten Schritt benötigt. Grund dafür ist, dass
3. Die Seiten mit einer Seiten Nummer und dem Pfad zu dem Bild, wird mit der Vorlagen-ID gesendet. Als Antwort wird eine ID zurückgegeben, die im nächsten Schritt verwendet wird.
4. Die Regionen der Seiten werden gesendet. Eine Region, auch Attribut genannt

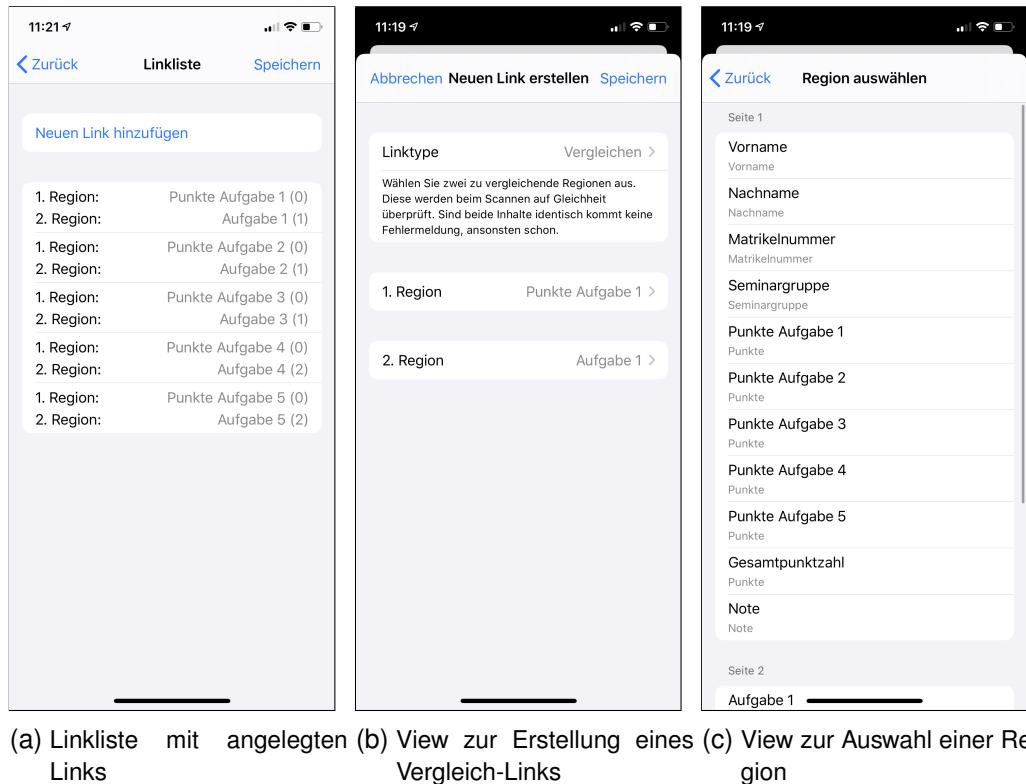


Abbildung 6.6: Views zur Erstellung von Links

hat X- und Y-Koordinaten. Diese repräsentieren den Abstand vom Bildursprung, der sich in jedem Bild oben links befindet. Außerdem besitzt eine Attribut eine Höhe und Breite, einen Namen und einen Datentyp. Die Koordinaten, sowie die Höhe und Breite sind alle in Pixel angegeben. Zusammen mit der Seiten-ID wird das Attribute an den Server gesendet.

6.5.3 Vorlage verwenden

Um eine Klausur zu digitalisieren, muss die vorher erstellte Vorlage ausgewählt werden. Danach bietet die Benutzeroberfläche über einen Button die Möglichkeit an, eine Klausur einzuscannen. Wichtig dabei ist, dass die Seiten beim Fotografiert die selbe Reihenfolge, wie in der Vorlage haben. Auch muss die Anzahl der Eingescannten Seiten mit denen in der Vorlage übereinstimmen. Bei der Implementierung der Scan-View wurde auf die Selbe zurückgegriffen, wie sie auch beim Vorlagen Erstellen zu finden ist. Allgemein konnten viele Komponenten der gesamten Benutzeroberfläche dank SwiftUI immer wieder verwendet werden. Durch die Scan-View werden die Seiten wieder aus dem Bild rausgeschnitten, geglättet und der Kontrast des Bildes wird ebenfalls leicht angepasst, damit Buchstaben leichter zu erkennen sind und Schatten bzw. Falten verschwinden.

Nach dem Einstellen der Klausur-Seiten beginnt die Texterkennung. Durch Aktivitäts-

anzeigen wird dem Benutzer mitgeteilt, dass die Anwendung gerade beschäftigt ist. Zuerst werden die Positionen der Regionen im neuen Bild berechnet. Das hat den Hintergrund, da die Regionen in der Vorlage absolut zum Bildursprung liegen. Jedoch hat die neue eingescannte Seite mit hoher Wahrscheinlichkeit, nicht die selben Pixel-Maße, wie das Bild der Seite aus der Vorlage. Das liegt dann daran, dass sich der Winkel und der Abstand der Kamera zum Dokument geändert hat. Diese Besonderheit lässt sich auch in den Abbildungen 3.1a und 3.1b beobachten. Beide Bilder sind beim Einstellen einer Scan-Vorlage entstanden. Jedoch haben sie unterschiedliche Maße, was an dem Größenunterschied zu erkennen ist. Die Abbildung 3.1a ist leicht größer, als die Abbildung 3.1b, da dieses Bild von näher dran und in einem geeigneterem Winkel aufgenommen wurde. Würde man immer den selben Abstand und Winkel garantieren, wie das beispielsweise in einem richtigen Kopiergerät oder Scanner der Fall ist, könnte man die Neuberechnung der Positionen der Regionen weglassen.

Mit der Berechnung der relativen Position der Region, an Hand der Maße der Vorlagen-Seite und der neuen Seite, wird außerdem noch die relative Höhe und Breite der Region berechnet. Durch das Umrechnen von absoluten in relative Positionen ist auch möglich, verschiedenen Auflösungen von Kameras zu kompensieren. Nachdem die neue Position und Maße aller Regionen einer Seite bestimmt sind, entstehen daraus Bildausschnitte für die Texterkennung. Die Regionen dienen als eine Art Schablone, so dass alles, was sich außerhalb der Regionen befindet, nicht verwendet wird. Das bedeutet auch, dass es vorkommen kann, dass zu digitalisierende Teile weggeschnitten werden können. Bei richtiger Erstellung der Vorlagen und bei richtigem Einstellen der Klausur passiert das jedoch nicht.

größte beachten siehe 3.1

- keine Unit Tests etc. keine Zeit

6.6 "Abnahme"

Abnahme ...

- was wurde umgesetzt was nicht - Accessibility von Haus aus etwas... dank swiftui - wie benutze ich die App wie installiere ich die -

Weitere Punkte die in Kapitel 6 rein könnten/sollten:

- beschreiben wie die einzelnen Views/Seiten nun aussehen und funktionieren? oder eher den Prozess der Entwicklung?

- Bild mit Wireframe aller Views und deren Workflow?
- erklären deskew / Dokument ausrichten?
- Umrechnung der Bilder vom Template aus und zuschneiden der neuen Regionen erklären
- Attribute hinzufügen (den Vorgang) -> Rechteck einzeichnen...
- Verwendete Programme, Sprache, etc.
- ein paar Worte, wie gut die Entwicklung lief (Simulator vs. echtes Gerät), wo sind die Grenzen des Simulators (Keine Kamera), wo waren Probleme mit dem echten Gerät...

7 Weitere Entwicklung und Besonderheiten

7.1 API

- auf die API eingehen? oder nur Tobias?

7.2

- Keyboards - Accsessebility

8 Grenzen der App

8.1 Probleme beim Erkennen von Dokumenten

- Gleich-farbiger Hintergrund
- Wenig Licht
- Hintergrund mit starken Kanten
- Runde Ecken, keine Ecken
- Starke Kanten im Bild (schwarzer Kreditkartenstreifen)
- zu jedem möglichem vlt. dann ein Beispiel Bild

8.2 Probleme der Klausur-Vorlage beim Scannen

- Schrift zu klein
- zu wenig Platz
- zu viel Platz
- Überschneidungen
- ...

8.3 Weitere ...

- Abstürze, Memory Leaks,

9 **Templates**

- Template und den Entwicklungsprozess vorstellen,
- wieso weshalb warum muss das nun so aussehen?
- Was kann an der neuen Vorlage immer noch optimiert werden?
- Welche Probleme konnten behoben werden?

10 Ausblick

- Es müssen nicht nur Klausuren sein, sondern alles mögliche (Krankenscheine, Urlaubsscheine, was auch immer, nur DB muss angepasst werden. (oder automatisches generieren von DB-Tabellen an Hand der Template Attribute))
- Server mit Bildern als Klausuren-Einsicht nutzen -> allerdings viele Probleme (Klausuren würden kopiert werden -> Profs mehr Arbeit, keinen direkten Kontakt zum Prof wegen Fragen, Verbesserungen oder Anmerkungen, ...) (warum genau, sollen die gespeichert werden? meine/unsere Idee Online Klausureneinsicht -> ins Fazit/Ausblick (hat viele "Probleme"))
- QR-Code-Idee um Vorlagen weg zu lassen -> Programm oder Plugin (Word/LaTeX) was die QR-Codes dann automatisch erstellt und richtig einfügt. (Wichtige Daten sind da hinterlegt, Vor- und Nachteile von QR-Codes, ...)
- Ist die App nun schneller als der normale Umgang bleibt offen -> Bachelorarbeit knüpft da an...
- ImageCaptureCore für macOS
- neuer Workflow. Klausur hat immer Name, Matrikel Nummer, etc. (Nicht extra Regionen-Namen vergeben)

Anhang A: Workflow

Vorlage erstellen

1. "Neue Vorlage erstellen"
2. Foto machen
3. Frage: Ist Foto gut?
 - a) Ja: gehe zu 4.
 - b) Nein: gehe zu 2.
4. Neues Attribut hinzufügen
5. Bereich auf Bild auswählen
6. Frage: Ist Bereich gut?
 - a) Ja: gehe zu 7.
 - b) Nein: gehe zu 5.
7. Name für Attribut festlegen
8. Datentyp für Attribut festlegen (Name, Matrikelnummer, Note, ...)
9. Frage: Sind alle Attribute vorhanden?
 - a) Ja: gehe zu 10.
 - b) Nein: gehe zu 4.
10. Fertig
11. Vorlage an Server senden

Anhang B: Tätigkeitsbericht

24.02. - 01.03. Ich habe mich mit der Problemstellung auseinander gesetzt, Ideen gesammelt, Problemanalyse betrieben und einen kleinen Prototypen entwickelt. Dazu erstellte ich einen minimalen Projektplanung, arbeitete mich in die Frameworks *Vision* und *VisionKit* ein und setzte eine Versionsverwaltung auf. Zusätzlich suchte ich nach einer passenden App-Architektur, die geeignet für das deklarative GUI-Framework SwiftUI, sowie für asynchrone Aufgaben, wie z. B. API-Aufrufe ist. Dabei stieß ich auf *Cleancode Architecture* und *Redux*.

02.03. - 08.03. In dieser Woche habe ich die Texterkennung auf den berechneten Regionen eines neuen Fotos implementiert, den Workflow sowie viele andere Kleinigkeiten in der App verbessert und alle Fehler der letzten Woche behoben, sodass ich neue Dinge implementieren konnte. Zudem probierte ich CI sowie Lint für das Projekt aus. Da CI für eine iOS-App mit *Github Actions* schwer aufzusetzen war und ab April etwas kosten würde, lies ich es sein. Des Weiteren pflegte ich das Projekt Management durch *Issues* und *Project-Boards* in GitHub. Anschließend programmierte ich den App-Workflow so um, dass nun mehr als eine Seite aufgenommen und analysiert werden konnte. Abgesehen von neuem Quellcode fing ich an den Praktikumsbericht zu schreiben und arbeitet mich dafür in \LaTeX und die Bachelorarbeit-Vorlage für \LaTeX der Hochschule Mittweida ein.

09.03. - 15.03. Zu Beginn der dritten Woche schaute ich mir Möglichkeiten für serverseitiges OCR an. Genauer sammelte ich Informationen zu dem Framework Vapor und Swift unter Linux. Jedoch funktionieren die Frameworks *Vision* und *CoreML* von Apple unter Linux nicht, weshalb sich IronOCR als beste Option herausstellte. Ich entwickelte ein Datenbankmodell, mithilfe der in der App verwendeten Datentypen und erstellte dazu noch eine JSON-Struktur die später für die APIs verwendet werden könnte. Außerdem gab es ein Meeting, in dem wir unseren aktuellen Stand präsentieren sollten, um weitere Schritte und Aufgaben zu planen. Bis zum Ende der Woche arbeitete ich weiter an meinem Beleg und schrieb den Datenfluss in der App um. Nun ähnelt er sehr dem Redux-Model.

16.03. - 22.03. Anfangs habe ich weiter an meinem Praktikumsbericht geschrieben, neue Issues hinzugefügt und bearbeitet. Außerdem gepflegte ich die Dokumentation und betrieben Projekt Management, um nun Links zwischen Regionen hinzuzufügen. Dabei entstanden neue Views und der Redux-Store musste dadurch angepasst werden. Es kam eine Erweiterung für die Texterkennung hinzu, so dass man durch die Auswahl eines Datentyps, das Resultat der Erkennung verbessern konnte. Des Weiteren habe ich bis zum Ende der Woche die Vergleich-Links vollständig implementiert und die App auf Fehler und Abstürze kontrolliert, sowie den Beleg um einige Kapitel erweitert.

23.03. - 29.03. Ich begann den Workflow und die Navigation in der App, zu verbessern und vereinfachen. Dabei beseitigte ich Quellcode des Prototyps, erweiterte die Dokumentation und behob einige Fehler. Anschließend überarbeitet ich einige Views, so dass sie übersichtlicher und einfacher zu benutzen sind. Nach dem iOS 13.4 Update in der Mitte der Woche, funktionierte ein Teil der App nicht, da sich das Verhalten von Views geändert hat. Ich behob die Fehler, testete ausgiebig die App und fügte iPad Unterstützung hinzu. Des Weiteren entstand ein neues verbessertes Template und ich schrieb einen großen Teil am Bericht.

30.03. - 05.04. Das Backend für die App war soweit, dass ich es aufsetzen und die API-Schnittstellen implementieren konnte. Dazu erstellte ich Views für Registrieren und Anmelden, die mithilfe von regulären Ausdrücken, die Eingaben überprüfen. Außerdem entwickelte ich einen Schicht im App-Store, für die anfallenden asynchronen Aufgaben. Dabei laß ich mich in das Framework Combine ein und überlegte mir einen geeigneten Aufbau. Da der Ansatz von Combine sehr neu für mich war, dauerte es zwei Tage, bis ein erster API-Service mit Fehler-Handling funktionierte. Zum Ende der Woche waren alle der Create-Schnittstellen implementiert, getestet und dokumentiert. Nebenbei erstellte ein paar Issues für das Backend und sprach mich mit Tobias über OCR auf dem Server ab.

06.04. - 12.04. Diese Woche startete mit dem Umschreiben der Regionen-Links und deren Analyse. Anschließend integrierte ich die Neuerungen vom Backend und erstellte die API für den Upload von Bildern. Nachdem das fertig war fügte ich die APIs zusammen, um Vorlagen vollständig auf dem Server zu speichern und abzurufen. Dazu schrieb ich einen eigenen JSON Decoder-Funktion, um die App internen Datentypen zu unterstützen. Zusätzlich wurde die App etwas benutzerfreundlicher und ein Problem mit dem Start der iPad Version wurde behoben. Nach einem Meeting folgten noch weitere Absprachen mit Tobias und ich arbeitete weiter an dem Bericht.

14.04. - 19.04. Ein kritisches Problem mit den Sessions aus der vorherigen Woche konnte in dieser endlich gelöst werden. Dazu konnte ich die Anwendung etwas optimieren und einen sehr großen Teil des Beleges fertig stellen. Dabei half auch die Beantwortung vieler Fragen während eines Meetings mit dem Betreuer. Allerdings entstanden Probleme mit der Datenbank, die die Funktionalität der App einschränkt.

20.04. - 26.04.

27.04. - ??05.

Literaturverzeichnis

- [1] Detecting Objects in Still Images | Apple Developer Documentation. Verfügbar unter: https://developer.apple.com/documentation/vision/detecting_objects_in_still_images.
- [2] Hochschule Mittweida: Portrait. Verfügbar unter: <https://www.hs-mittweida.de/hochschule/portrait.html>.
- [3] BRAGGE, M. *Model-View-Controller architectural pattern and its evolution in graphical user interface frameworks*. Bachelorarbeit (2013). Verfügbar unter: <https://lutpub.lut.fi/handle/10024/92156>.
- [4] FREEMAN, A. *Pro ASP.NET Core MVC 2*. Expert's Voice in .NET. Apress, 2 (30. Juni 2010) Edition (2017). ISBN 978-1-4842-3150-0. OCLC: 1007700442. Verfügbar unter: <http://www.books24x7.com/marc.asp?bookid=135443>.
- [5] SILLMANN, T. Einstieg in SwiftUI. Library Catalog: www.heise.de. Verfügbar unter: <https://www.heise.de/developer/artikel/Einstieg-in-SwiftUI-4594018.html>.

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, 13.03.2020