

# Kassenzettelverwaltung mit iOS-App

*Verfasser: Timo Dörflinger*

HF-ICT

HE15

*Examinator: Ronald Tanner*

*Eaminator: Beat Holm*

*Auftraggeber: Marc Hoffmann*

PRATTELN

28. Juni 2018

# I Abstract

Der Auftraggeber Marc Hoffmann führt für einen guten Überblick über seine Finanzen ein Haushaltsbuch, welches er anhand der Kassenzettel der Einkäufe führt. Hier möchte Herr Hoffmann gerne eine Vereinfachung mittels einer App. Der bisherige Aufwand verlangt viel Zeit. Die Kassenzettel müssen mit nach Hause genommen und dort manuell in das Haushaltsbuch übertragen werden.

Der Wunsch des Auftraggebers ist es, dass die Kassenzettel mit einer App fotografiert werden können. Diese Kassenzettel-Abbilder sollen dann von der App ausgelesen und die Daten weiterverarbeitet werden. Es sollen die ausgelesenen Endbeträge mit einem gesetzten monatlichen Budget verrechnet werden und somit ersichtlich sein, wieviel Budget für den laufenden Monat noch zur Verfügung steht.

Für diesen Auftrag habe ich mich für eine iOS-App entschieden, da der Auftraggeber ein iPhone hat. Der Auftraggeber kann mit der iOS-App ein Kassenzettel-Abbild erstellen oder ein bereits erstelltes Abbild aus der Bildergalerie öffnen und zur Verarbeitung in der App bereitstellen. Hat der Auftraggeber also ein Abbild erstellt oder eines ausgewählt, wird der komplette Text ausgelesen und als neuer Eintrag in eine lokale Datenbank geschrieben. Zusätzlich zu diesem Datenbank-Eintrag wird der gezahlte Endbetrag aus dem ausgelesenen Text extrahiert und ebenfalls dem Datenbank-Eintrag hinzugefügt. Ebenfalls wird noch die Verbindung zum Kassenzettel-Abbild sowie das Erstellungsdatum hinzugefügt.

Zusätzlich kann eine Upload-Funktion aktiviert werden, mit der das Kassenzettel-Abbild in die iCloud hochgeladen werden kann. Das ist für eine verlustlose Sicherung bei speziellen Kassenzetteln mit Garantieansprüchen wichtig. So können diese Kassenzettel selbst beim Verlust der Daten oder des iPhones später wiederhergestellt werden.

Auf der Benutzeroberfläche der App ist dann das angegebene monatliche Budget angegeben und darunter das noch übrige Budget, welches mit den ausgelesenen Endbeträgen verrechnet wird.

# Inhaltsverzeichnis

<b>I</b>	<b>Abstract</b>	<b>1</b>
<b>II</b>	<b>Ehrenwörtliche Erklärung</b>	<b>3</b>
<b>III</b>	<b>Vorwort</b>	<b>4</b>
<b>IV</b>	<b>Glossar</b>	<b>5</b>
<b>1</b>	<b>Einleitung</b>	<b>6</b>
<b>2</b>	<b>Theoretischer Teil</b>	<b>7</b>
<b>3</b>	<b>Methodische Vorgehensweise</b>	<b>8</b>
3.1	GitHub-Repository . . . . .	8
3.2	Projektmanagement - Scrum . . . . .	8
<b>4</b>	<b>Praktischer Teil</b>	<b>10</b>
4.1	Basis-App . . . . .	10
4.2	Evaluierung des Datenbank-Frameworks . . . . .	11
4.2.1	SQLite . . . . .	11
4.2.2	Apple Core Data . . . . .	11
4.2.3	Realm . . . . .	11
4.3	Datenbank-Schema . . . . .	12
4.4	Datenbank Realm . . . . .	13
4.4.1	Realm Installation . . . . .	13
4.4.2	Realm Implementierung . . . . .	14
4.5	Benutzeroberfläche . . . . .	17
4.6	iCloud Upload . . . . .	19
4.6.1	Installation . . . . .	19
4.6.2	Programmierung . . . . .	21
<b>5</b>	<b>Schlussfolgerung</b>	<b>25</b>
<b>6</b>	<b>Anhang</b>	<b>26</b>
6.1	Quellenverzeichnis . . . . .	26
6.2	Bilderverzeichnis . . . . .	26

## **II Ehrenwörtliche Erklärung**

Ich versichere, dass ich die vorliegende Diplomarbeit selbstständig und ohne Benutzung anderer als der im Literaturverzeichnis angegebenen Quellen und Hilfsmittel angefertigt habe.

Die wörtlich oder inhaltlich den im Literaturverzeichnis aufgeführten Quellen und Hilfsmitteln entnommenen Stellen sind in der Arbeit als Zitat bzw. Paraphrase kenntlich gemacht.

Diese Diplomarbeit ist noch nicht veröffentlicht worden. Sie ist somit weder anderen Interessenten zugänglich gemacht noch einer anderen Prüfungsbehörde vorgelegt worden.  
Pratteln, 28.06.2018 Unterschrift:

### **III Vorwort**

Diese Diplomarbeit ist meine Abschlussarbeit des Studiums der hf-ict, welches ich vom Sommer 2015 bis zum Sommer 2018 besuchte. Während diesem Studium habe ich unter anderem auch das Programmieren nach der Ausbildung noch weiter vertiefen dürfen. Dabei habe ich ein Interesse für die App-Entwicklung, besonders auch für iOS-Apps, entwickelt.

Ich möchte mich bei meinem Auftraggeber, Marc Hoffmann, für sein Vertrauen und seine Unterstützung bedanken. Die Zusammenarbeit war stets positiv. Diese Arbeit hat mich in meinen Fähigkeiten gefördert.

Auch bedanken möchte ich mich bei Elke Dörflinger, die viel Zeit in das Korrekturlesen dieser Arbeit investiert hat. Auch hier hat mir die Zusammenarbeit stets Freude bereitet.

Des Weiteren möchte ich mich bei allen Dozenten bedanken, die mich bei Fragen oder Herausforderungen stets unterstützten.

## **IV Glossar**

OCR - Die "Texterkennung oder auch Optische Zeichenerkennung (englische Abkürzung OCR von englisch optical character recognition) ist ein Begriff aus der Informationstechnik und bezeichnet die automatisierte Texterkennung innerhalb von Bildern." (<https://de.wikipedia.org/w> 23. Februar 2018, o.S.)

Xcode - Xcode ist die Entwicklungsumgebung, in der iOS-App erstellt und programmiert werden.

Cocoapods - Cocoapods ist ein Dependency-Manager. Mit einem Dependency-Manager können Frameworks über einfache Terminal-Befehle einem Xcode-Projekt hinzugefügt werden.

Tesseract - Tesseract ist ein Open Source-Framework, welches den Text aus einem Bild auslesen kann.

Realm - Realm ist ein Datenbank-Framework, welches Datenbank-Records als Objekte behandelt und lokal in einer iOS-App installiert werden kann.

Swift - Swift ist die Entwicklungssprache, mit der iOS-Apps erstellt und programmiert werden.

Scrum - Scrum ist eine Projektmanagement-Methode, die sich besonders für agile Software-Entwicklungs-Projekte eignet.

# 1 Einleitung

Der Auftraggeber führt sein Haushaltsbuch manuell, mit Hilfe der Kassenzettel. Dies ist jedoch mit einem enorm hohen zeitlichen und administrativen Aufwand verbunden. Dieser hohe Aufwand soll durch die Automatisierung mittels einer iOS-App vereinfacht werden.

Die App soll ein Kassenzettel-Abbild erstellen oder aus der Bildergalerie öffnen können. Von diesem Abbild soll dann der Text ausgelesen werden. Aus diesem Text wiederum soll der Endbetrag herausgenommen werden. Diese Ausarbeitung ist bereits in einem Projekt vor der Diplomarbeit und während dem letzten Studien-Semester erarbeitet worden.

In der Diplomarbeit soll nun der herausgenommene Endbetrag aus dem ausgelesenen Text weiterverarbeitet und eine Datenbank lokal in der App installiert werden. Eine externe Datenbank, wie zum Beispiel eine in der Cloud, ist vorerst nicht zu beachten. Es sollen neben dem ausgelesenen Endbetrag noch weitere Daten in die Datenbank eingetragen werden.

Auch die Benutzeroberfläche soll an ein bereits erstelltes Mockup angepasst werden. Dieses Mockup ist auch bereits in dem Projekt zuvor mit meinem Auftraggeber zusammen erstellt worden. Gewisse Daten sollen dann aus der Datenbank auf dieser Benutzeroberfläche angezeigt werden. Darüber hinaus sollen über die Benutzeroberfläche die Kassenzettel-Abbilder erstellt und dann mit weiteren Optionen abgespeichert werden können. Die weiteren Optionen sind zum einen eine Auswahl an Kategorien, die die App-Anwenderin, der App-Anwender passend zu dem Kassenzettel anwählen muss. Zum anderen kann die App-Anwenderin, der App-Anwender auswählen, ob der Kassenzettel in der iCloud hochgeladen und damit zusätzlich gesichert werden soll.

Auf einer weiteren Seite der Benutzeroberfläche soll dann noch das Budget gesetzt werden können, welches dann mit den Daten der Kassenzettel aus der Datenbank verrechnet werden soll.

## 2 Theoretischer Teil

Kassenzettel sind schon länger in der Kritik. Immer wieder gibt es Meldungen in den Nachrichten, dass Einkaufsläden auf die Herausgabe der Kassenzettel verzichten wollen oder diese lediglich auf Verlangen der Kunden herausgeben. So soll das Papier und damit vor allen Dingen der Abfall verringert werden. So heisst es in einem Bericht "«D Quittig?», fragen Kassiererinnen von Coop, Migros und anderen Läden täglich hundertfach. Oft drücken sie einem den Kaufbeleg zusammen mit dem Rückgeld in die Hand. Und nicht selten lässt der Kunde die Quittung dann an der Kasse liegen, verliert sie auf dem Weg nach draussen oder wirft sie in den nächsten Papierkorb. Coop und Lidl wollen deshalb weniger Belege drucken. Der Discounter Lidl stellt ab Ende Januar nur noch dann eine Quittung aus, wenn der Kunde dies wünscht." (<https://www.derbund.ch/wirtschaft/standard/Das-Kassezeddeli-wird-zum-Auslaufmodell/story/17537630>, geschrieben von Yvonne Debrunner, 20.12.2016, o.S.).

Und wer kennt es nicht, ist der Kassenzettel für eine gewisse Zeit im Geldbeutel, verliert dieser je nach Dauer die Tinte und ist damit nicht mehr lesbar. Oder wie oft war man bereits auf der Suche nach dem einen Kassenzettel, den man extra in einer besonderen Schublade aufbewahrt hat, weil dieser einen wichtigen Garantieanspruch sichert. Und dann ist auch bei diesem Kassenzettel bereits die Tinte nicht mehr lesbar oder der Kassenzettel selbst ist nicht mehr auffindbar. In beiden Fällen verliert man den Garantieanspruch.

Ich persönlich denke, Kassenzettel werden früher oder später komplett durch digitale Alternativen ersetzt. Jedoch sind die Kassenzettel bis zu diesem Zeitpunkt ein gutes Mittel zur Führung eines Haushaltsbuchs, egal ob analog oder digital.



## 3 Methodische Vorgehensweise

### 3.1 GitHub-Repository

Für diese Diplomarbeit verwende ich Git und GitHub. Auf GitHub habe ich ein Repository mit dem Namen *diplom-kassenzettelverwaltung* erstellt. Auf diesem werden alle erstellten Dateien gesichert. Dafür habe ich das Repository auf meinem MacBook über den Terminal geklont. In diesem lokalen Klon werden alle Dateien wie die Dokumentation, die erstellte iOS-App und weitere Dateien erstellt und regelmässig mit den passenden *Commits* auf das Repository auf GitHub *gepusht*.

### 3.2 Projektmanagement - Scrum

Die Diplomarbeit ist ein wichtiges Projekt mit einem geringen zeitlichen Rahmen. Um in diesem eng gesetzten Rahmen die Aufgaben und deren Verteilung im Überblick zu haben, habe ich mich entschieden die Diplomarbeit mit einer Projektmanagement-Methode zu führen. Da ich bereits in einem zuvor geführten Projekt mit HERMES5 schlechte Erfahrungen in einem Software-Engineering-Projekt gemacht hatte, habe ich Scrum für die Diplomarbeit ausgewählt.

Da ich, wie bereits erwähnt, einen engen Zeitrahmen für die Diplomarbeit habe, wird ein Scrum-Sprint das komplette Projekt abdecken. Um die Diplomarbeit bestmöglich nach Scrum führen zu können, habe ich diverse unterstützende Programme herausgesucht. Bei der Evaluierung bin ich auf zwei interessante Programme gestossen.

Nach der ersten Suche hatte ich iceScrum (<https://www.icescrum.com>, 2018, o.S.) entdeckt. Es sah einfach aus und war für ein Projekt kostenlos. Das Management findet in der Cloud des Anbieters statt. Bei dem ersten Test nach der Anmeldung musste ich aber feststellen, dass manche Seiten lediglich auf Französisch zur Verfügung stehen. Da ich aber so gut wie keine Französischkenntnisse habe, empfand ich das als nicht hilfreich und habe daher nach einer Alternative gesucht.

Ich bin dann auf OpenProject (<https://www.openproject.org>, 2018, o.S.) gestossen. OpenProject ist eine *Open source project management software* mit Scrum integriert. OpenProject bietet eine Community-Version an, die kostenlos und lokal anstatt in einer Cloud verwendet werden kann. Auch die Darstellung des Programs auf ihrer Homepage hat mich angesprochen. Daher habe ich es getestet und empfinde es als die richtige Lösung für mich und meine Diplomarbeit.

Ich habe das OpenProject in meiner lokalen Docker-Umgebung auf dem MacBook installiert. Dafür bin ich der Anleitung (<https://www.openproject.org/docker/>, 2018) von OpenProject gefolgt und nach dieser Anleitung prompt auf ein Problem gestossen, dass mich zeitlich etwas aufgehalten hat. Der Anleitung zu folge, sollte für eine produktive Nutzung eine erweiterte Installation ausgeführt werden. So kann gesichert werden, dass bei einem Neustart des Containers keine Daten verloren gehen und auch die Log-Dateien

lokal in einer selbst gesetzten Ordnerstruktur auf dem System zu finden sind.

Dafür habe ich einen Ordner in dem GitHub-Repository erstellt, in dem dann diese Daten und Logs gesichert werden. Dafür habe ich den angegebenen Konsolen-Befehl für Docker, mit der angepassten Ordnerstruktur, ausgeführt.

```
1 docker run -d -p 8080:80 --name diplomopenproject
2   -e SECRET_KEY_BASE=secret \
3   -v /Users/Timo/diplom-kassenzettelverwaltung/projektmanagement/
4     openproject/pgdata:/var/lib/postgresql/9.6/main \
5   -v /Users/Timo/diplom-kassenzettelverwaltung/projektmanagement/
6     openproject/logs:/var/log/supervisor \
7   -v /Users/Timo/diplom-kassenzettelverwaltung/projektmanagement/
8     openproject/static:/var/db/openproject \
9     openproject/community:7
```

Dieser ist auch erfolgreich durchgelaufen und hat den Container wie erwartet erstellt. Die Webseite konnte ich aber dann nicht im Localhost <http://localhost:8080> aufrufen. Nach mehreren Versuchen habe ich dann den zuvor aufgeführten Befehl wie folgt angepasst:

```
1 docker run -it -p 8080:80 --name diplomopenproject
2   -e SECRET_KEY_BASE=secret \
3   -v /Users/Timo/diplom-kassenzettelverwaltung/projektmanagement/
4     openproject/pgdata:/var/lib/postgresql/9.6/main \
5   -v /Users/Timo/diplom-kassenzettelverwaltung/projektmanagement/
6     openproject/logs:/var/log/supervisor \
7   -v /Users/Timo/diplom-kassenzettelverwaltung/projektmanagement/
8     openproject/static:/var/db/openproject \
9     openproject/community:7
```

Mit diesem Befehl konnte ich den Container erfolgreich erstellen und dann auch im Browser darstellen. Danach konnte ich den Container mit dem Befehl *docker stop diplomopenproject* stoppen und diesen mit *docker start diplomopenproject* wieder starten.

Nach der Installation ist bereits ein Administrator-Account erstellt, der für den ersten Login verwendet werden kann. Die Zugangsdaten stehen in der Installations-Anleitung. Nach dem Login mit dem Admin muss zuerst einmal ein neues Projekt erstellt werden. In diesem Projekt können nun *Work packages*, also Arbeitspakete definiert werden, welchen nur einzelne Aufgaben oder ganze Meilensteine zugewiesen werden. Da diese Einstellungen bzw. Erstellungen der Arbeitspakete als Projekt-Administrator mit dem bereits installierten Administrator definiert werden, habe ich noch einen weiteren Benutzer mit meinem Namen hinzugefügt. Dieser Nutzer dient als ausführende Person der Aufgaben und bekommt daher die Arbeitspakete und deren enthaltene Aufgaben und Meilensteine zugewiesen. In der Scrum-Übersicht sind nun alle Aufgaben übersichtlich zu sehen.

Dort können nun die Aufgaben je nach aktuellem Stand, in die dazu entsprechenden Zustand-Spalten verschoben werden. Dies übermittelt einen einfach Überblick über die noch bestehenden Aufgaben und welche Aufgaben aktuell welchen Stand haben.

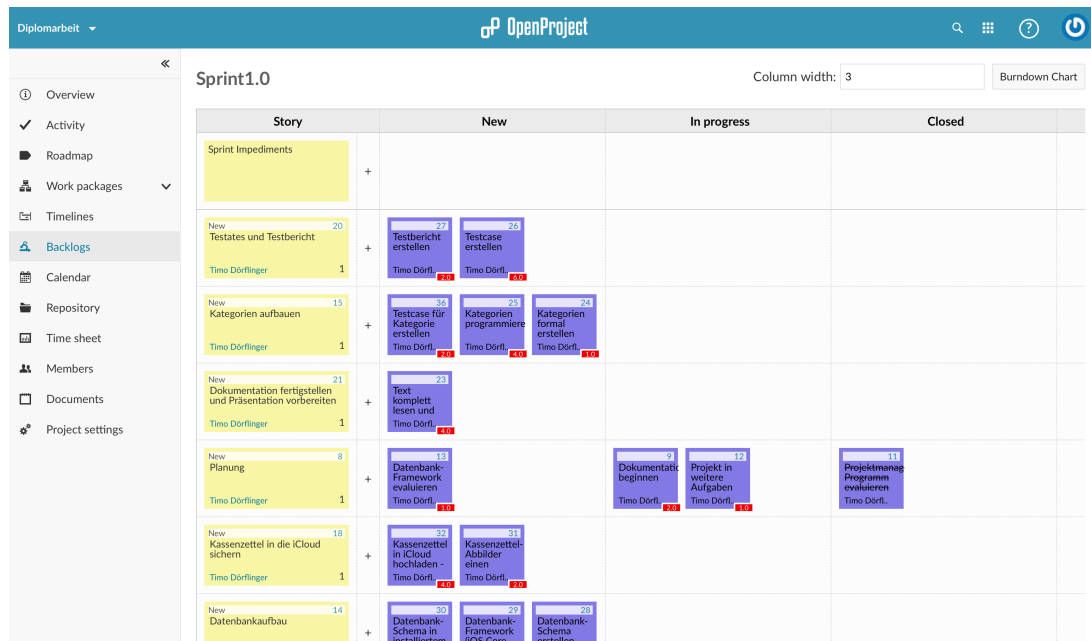


Abbildung 1: Datenbank-Schema

Scrum war mir eine grössere Hilfe als es Hermes5 in Projekten zuvor war, da mit Scrum eine agilere Planung der Aufgaben möglich ist. So konnte ich während der Erarbeitung einzelne Aufgaben in der Reihenfolge der Abarbeitung ändern.

## 4 Praktischer Teil

### 4.1 Basis-App

In einem Projekt im letzten Semester habe ich bereits eine App erstellt, die als Basis für diese Diplomarbeit dienen soll. Diese App hat Tesseract implementiert. Tesseract ist ein OCR-Framework, welches den Text von einem Bild auslesen kann. Die App ist so aufgebaut, dass das Abbild eines Kassenzettels mit OpenCV durch diverse Bildaufbereitungs-Funktionen soweit aufbereitet wird, dass die Ausleserate von Tesseract verbessert wird. So kann die App nach dem Aufbereiten des Kassenzettel-Abbildes mit Tesseract den Text auslesen. Dieser ausgelesene Text wird dann so weiterverarbeitet, dass der gezahlte Endbetrag aus dem ausgelesenen Text gefiltert wird.

Der ausgelesene und gefilterte Endbetrag wird dann für die weitere Verarbeitung bereitgestellt. Hier setzt die Diplomarbeit nun an und verarbeitet den ausgelesenen Endbetrag weiter.

## 4.2 Evaluierung des Datenbank-Frameworks

Im zuvor durchgeführten Projekt hatte ich bereits die Vorstellung, zwischen dem ausgelesenen Text aus den Kassenzettel-Abbildern und der Benutzeroberfläche, eine Datenbank zu verwenden. In diese Datenbank können dann die ausgelesenen Endbeträge der Kassenzettel mit dem jeweiligen Erfass-Datum und noch weiteren Daten zwischengespeichert werden. Ich hatte mich bereits während dem ersten Projekt bezüglich einer lokalen Datenbank für eine iOS-App informiert. Hier sind mir nach diversen Berichten *SQLite* und *Apple Core Data* im Gedächtnis geblieben. Bei der aktuellen Evaluierung bin ich dann noch auf eine weitere Möglichkeit gestossen, die Daten lokal in der App abspeichern zu können. Diese Option heisst *Realm* und ist ein externes Framework für einen Datenbank-Aufbau in einer App.

Da leider nicht genug Zeit in der Diplomarbeit übrig war, konnte ich diese drei Optionen nicht selbst testen und vergleichen. Die nachfolgende Aufstellung bezieht sich daher auf den Bericht von OPHIR (<https://rollout.io/blog/ios-databases-sqlite-core-data-realm/>, 15.02.2016, o.S.) , der diese drei Optionen verglichen hat.

### 4.2.1 SQLite

So beschreibt OPHIR das SQLite als "SQLite is the most used database engine in the world and its open source as well. It implements a transactional SQL database engine with no configuration and no server required. SQLite is accessible on Mac OS-X, iOS, Android, Linux, and Windows. It delivers a simple and user-friendly programming interface as it is written in ANSI-C. SQLite is also very small and light and the complete database can be stored in one cross-platform disk file." (<https://rollout.io/blog/ios-databases-sqlite-core-data-realm/>, OPHIR, 15.02.2016, o.S.).

### 4.2.2 Apple Core Data

Apples Core Data wird von OPHIR als "Core Data is the second main iOS storage technology available to app developers. Depending on the type of data and the amount of data you need to manage and store, both SQLite and Core Data have their pros and cons. Core Data focuses more on objects than the traditional table database methods. With Core Data, you are actually storing contents of an object which is represented by a class in Objective-C." (<https://rollout.io/blog/ios-databases-sqlite-core-data-realm/>, OPHIR, 15.02.2016, o.S.) beschrieben.

### 4.2.3 Realm

Und Realm wird von OPHIR wie folgt beschrieben: "There's a new(ish) player in town called Realm. Realm was designed to be faster and more efficient than the previous database solutions. This new solution is a cross-platform mobile database called Realm. It is availa-

ble in Objective-C and Swift, and it's designed for iOS and Android." (<https://rollout.io/blog/ios-databases-sqlite-core-data-realm/>, OPHIR, 15.02.2016, o.S.).

Ich habe mich daher für die Nutzung von Realm entschieden, da es auf mich einen einfach zu implementierenden Eindruck hinterlässt. Da ich nicht viel Zeit zur Verfügung habe, mich in Core Data einzuarbeiten, werde ich Realm für die App verwenden.

### 4.3 Datenbank-Schema

Für eine saubere Darstellung des Datenbank-Aufbaus habe ich ein Schema erstellt, welches ich in MySQL-Workbench (<https://www.mysql.com/de/products/workbench/>, Version 6.3) erstellt habe. Mit MySQL-Workbench können Datenbank-Schemas mit einer guten Darstellung aufgebaut und veranschaulicht werden.

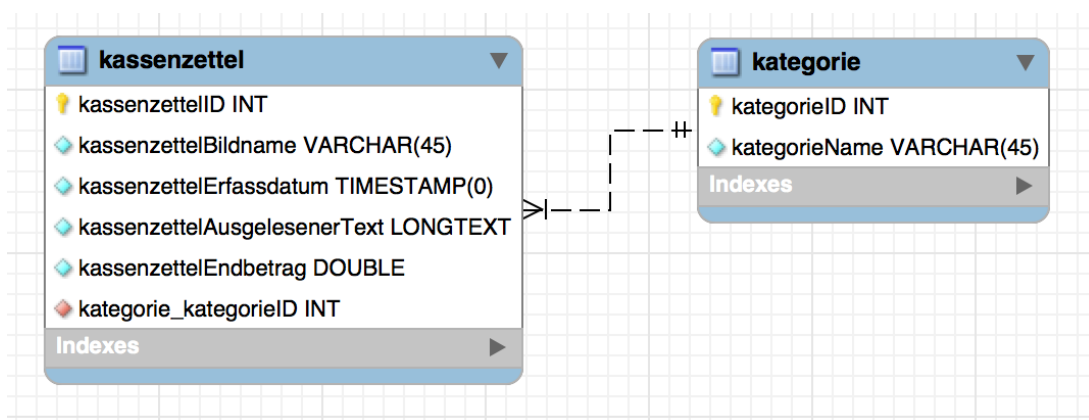


Abbildung 2: Datenbank-Schema

Die Tabelle *kassenzettel* enthält alle für den Moment nötigen Attribute. Durch die begrenzte Projektzeit war es nicht möglich das Ausstellen aller Artikel des Kassenzettels zu erfassen und es wird lediglich der Endbetrag ausgelesen. Somit ist es hier nicht nötig, eine weitere Tabelle oder weitere Attribute für die einzeln gekauften Artikel zu berücksichtigen. Wird ein Kassenzettel fotografiert und in die App eingelesen, bekommt das Kassenzettel-Abbild einen Datenbankeintrag mit einer ID. Darüber hinaus wird der Bildname aus dem System (iPhone) ausgelesen und dem Datenbankattribut *kassenzettelBildname* übergeben. Der von Tesseract ausgelesene Text des Kassenzettel-Abbildes wird dem Attribut *kassenzettelAusgelesenerText* übergeben. Der aus diesem ausgelesenen Text extrahierte Endbetrag wird dem Attribut *kassenzettelEndbetrag* übergeben. Alle bisherigen Attribute sind als *Not-Null* definiert. Das bedeutet, dass diese Attribute zwingend einen Wert für den Datenbankeintrag enthalten müssen.

Die Tabelle *kategorie* enthält die zur Verfügung stehenden Kategorien für die Kassenzettel. Hier sollen in diesem Projekt lediglich die Kategorien *Lebensmittel*, *Tanken* und *Sonstiges* zur manuellen Auswahl für die Anwenderin, den Anwender der App zur Verfügung stehen. In diesem Projekt ist es nur möglich, eine Kategorie pro eingelesenem Kassenzettel zu wählen. Das bringt das Problem mit sich, falls ein Kassenzettel Artikel

beinhaltet, die auf mehrere Varianten zutrifft. Falls also eine App-Anwenderin, ein App-Anwender beim tanken noch Brot und Grillkohle gekauft hat, so würde der Kassenzettel zu allen drei Varianten passen. Das müsste dann für eine produktive Verwendung noch weiter angepasst werden, indem mehrere Kategorien pro Kassenzettel verwendet werden könnten. Auch könnte die manuelle Auswahl einer Kategorie durch eine *KI* bzw. *Machine Learning* ersetzt werden. So müsste die App-Anwenderin, der App-Anwender diese Kategorien-Auswahl nicht mehr selbst erledigen, was die Benutzerfreundlichkeit einfacher machen würde (von einer zuverlässigen Auswahl der Kategorien mittels des Machine Learning vorausgesetzt).

Die Beziehungen zwischen den Tabellen sind 1:n-Beziehungen. So kann ein Kassenzettel immer nur von einer Kategorie zugewiesen werden. Wie bereits erwähnt könnte ein Einkauf mehrere Kategorien entsprechen, in diesem Projekt wird dies aber nicht beachtet. So kann ein Kassenzettel in diesem Projekt lediglich eine Kategorie besitzen.

Dieses Schema dient nur der Veranschaulichung meiner Gedanken hinter dem Aufbau. Leider kann dieser Aufbau nicht direkt in Realm importiert werden, so wie es bei MySQL möglich.

## 4.4 Datenbank Realm

### 4.4.1 Realm Installation

Wie zuvor evaluiert, habe ich mich für Realm entschieden. Um Realm in die iOS-App einbauen zu können, bin ich der Dokumentation (<https://realm.io/docs/swift/latest/>, Version 3.3.1) von Realm gefolgt. So muss der App zuerst das Framework hinzugefügt werden. Dafür verwende ich das Cocoapods. Um damit starten zu können, muss das Programm Cocoapods zuerst installiert werden. Danach wird die Ordnerstruktur des Projekts im Terminal geöffnet und mit dem Befehl *pod init* eine Pod-Datei erstellt. Wie bereits erwähnt, baut die Diplomarbeit auf einer bereits bestehenden App auf, die bereits eine Pod-Datei enthält. Damit ist die Installation und die Erstellung der Pod-Datei nicht mehr nötig.

Um nun fortzufahren, wird über den Terminal die Pod-Datei mit dem vi-Befehl geöffnet. Unter dem aus der Basis-App bereits bestehenden "pod TesseractOCRiOSEintrag muss nun der Eintrag für das Realm-Framework hinzugefügt werden. Dafür wird der Eintrag "pod RealmSwift zwischen "targetünd"end" hinzugefügt. Nach diesem kann noch eine Versions-Nummer angegeben werden, falls eine spezielle ältere Version verwendet werden möchte. Wird keine Versions-Nummer angegeben, so wird die bei Cocoapods aktuellste Version für die Installation verwendet. Im nachfolgenden Bild ist zu erkennen, wie die Pod-Datei dann aussieht.

Dann kann die Pod-Datei gespeichert und geschlossen werden. Danach muss lediglich mit dem Befehl "pod install" der Installations-Modus von Cocoapods gestartet werden. Dies kann eine Weile dauern, da alle dafür nötigen Pakete heruntergeladen werden müssen. Das sieht dann wie folgt aus:

```
# Uncomment the next line to define a global platform for your project
# platform :ios, '9.0'

target 'kassenzettel-management' do
  # Comment the next line if you're not using Swift and don't want to use dynamic frameworks
  use_frameworks!

  # Pods for kassenzettel-management
  pod 'TesseractOCRiOS', '4.0.0'
  pod 'RealmSwift'
end
```

Abbildung 3: Die Pod-Datei vor der Installation des Realm-Frameworks.

```
kassenzettel-management Timo$ vi Podfile
kassenzettel-management Timo$ pod install
Analyzing dependencies
Downloading dependencies
Installing Realm (3.3.1)
Installing RealmSwift (3.3.1)
Using TesseractOCRiOS (4.0.0)
Generating Pods project
Integrating client project
Sending stats
Pod installation complete! There are 2 dependencies from the Podfile and 3 total pods installed.

[!] Automatically assigning platform 'ios' with version '11.2' on target 'kassenzettel-managemen
t' because no platform was specified. Please specify a platform for this target in your Podfile.
See 'https://guides.cocoapods.org/syntax/podfile.html#platform'.
```

Abbildung 4: Der Installations-Vorgang von Cocoapods.

Damit ist das Realm-Framework der App hinzugefügt und kann nun verwendet werden.

#### 4.4.2 Realm Implementierung

Auf der Realm-Homepage ist eine grosse Dokumentation zu finden, wie bereits schon angegeben, die die Verwendung von Realm beschreibt. Dieser Dokumentation zufolge habe ich nun das Datenbank-Schema aufgebaut. Dies sieht wie folgt aus:

```
1 // kassenzettel model
2 class kassenzettel: Object {
3     @objc dynamic var kassenzettelID = ""
4     @objc dynamic var kassenzettelBildname = ""
5     //dem nächsten Attribut würde ich gerne die Function
6     // timestamp() übergeben, funktioniert aber nicht
7     @objc dynamic var kassenzettelErfassdatum = Date()
8     @objc dynamic var kassenzettelAusgelesenerText = ""
9     @objc dynamic var kassenzettelEndbetrag = 0.0
10    //One-to-many Relationship
11    @objc dynamic var kategorie: kategorie?
12    //PrimaryKey überschreiben
13    override static func primaryKey() -> String? {
14        return "kassenzettelID"
15    }
```

```

16     }
17
18     // kategorie model
19     class kategorie: Object {
20         @objc dynamic var kategorieID = 0
21         @objc dynamic var kategorieName = 0
22         //PrimaryKey überschreiben
23         override static func primaryKey() -> String? {
24             return "kategorieID"
25         }
26     }

```

Damit ist das Schema mit PrimaryKeys und den One-to-Many-Beziehungen implementiert. Das einzige was noch fehlt, ist die auto-increment-Funktion für die PrimaryKeys, welche die IDs der Tabellen automatisch pro Eintrag um eins erhöht. Die auto-increment-Funktion ist aber bis zu dieser verwendeten Version von Realm nicht verfügbar. Realm sieht hier keinen grossen Nutzen und beschreibt lediglich, dass stattdessen zufällig generierte Zahlen- oder Buchstabenkonstellationen als PrimaryKeys verwendet werden sollen (<https://academy.realm.io/posts/realm-primary-keys-tutorial/>, 07.03.2017, o.S.). Als Alternative gibt Realm an, den PrimaryKey des letzten Datenbank-Eintrags manuell auslesen und um eins zu erhöhen. Diese Funktion habe ich selbst erstellt, da ich nach den IDs der letzten Einträge filtern und diese somit auf der Hauptseite darstellen kann.

Nun kann das Framework in der Swift-Datei importiert werden, in der diese Verbindung benötigt wird. Dann kann eine Verbindung mit der Realm-Datenbank erstellt werden. Danach kann ein neues Objekt der Tabelle erstellt und mit Daten in die Datenbank geschrieben werden. Das Erstellen eines solchen Objekts ist im nächsten Code-Ausschnitt zu sehen. Diese Implementierungen beziehen sich alle auf die Dokumentation von Realm.

```

1 import RealmSwift
2 ...
3 var realm: Realm!
4 ...
5 //Diese Funktion erstellt ein neues Objekt als DB-Eintrag
6 // und pusht es in die DB
7 func DatabaseVerarbeitung(ID: Int, Bildname: String,
8     AusgelesenerText: String, Betrag: Double){
9
10     //erstellt ein neues Object als Realm-Eintrag mit den
11     // Attributen aus den Parametern
12     let newBon = kassenzettel()
13     newBon.kassenzettelID = ID

```

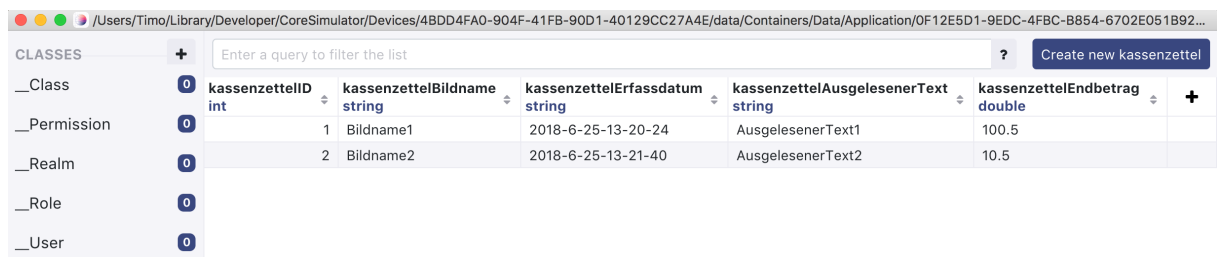


```

14 newBon.kassenzettelBildname = Bildname
15 newBon.kassenzettelErfassdatum = timestamp()
16 newBon.kassenzettelAusgelesenerText = AusgelesenerText
17 newBon.kassenzettelEndbetrag = Betrag
18
19 //versucht das Object als neuen Eintrag in die Datenbank
20 // zu schreiben, oder gibt sonst eine Fehlermeldung aus
21 do {
22     let realm = try Realm()
23     try! realm.write {
24         realm.add(newBon)
25     }
26 } catch let error as NSError {
27     print("Realm-Error:", error)
28 }
29 }

```

Realm stellt hier ein Programm zur Verfügung, mit dem die erstellte Datenbank-Datei geöffnet und die Einträge überprüft oder sogar editiert werden können. Dieses Programm heisst *Realm Studio* (<https://realm.io/products/realm-studio/>, 2018, .o.S.). Über einen bestimmten Befehl kann der lokale Pfad zu der Datenbank-Datei im simulierten iPhone angegeben werden. Wird die dort befindliche Datenbank-Datei *default.realm* im Realm Studio geöffnet, sind dort die Tabellen und deren Einträge ersichtlich. Diese Einträge können hier nun überprüft, editiert und auch neue manuelle erstellt werden.



The screenshot shows the Realm Studio interface. On the left, a sidebar lists classes: \_\_Class, \_\_Permission, \_\_Realm, \_\_Role, and \_\_User, each with a count of 0. The main area displays a table with the following columns: kassenzettelID (int), kassenzettelBildname (string), kassenzettelErfassdatum (string), kassenzettelAusgelesenerText (string), and kassenzettelEndbetrag (double). There is a search bar at the top with the placeholder 'Enter a query to filter the list' and a 'Create new kassenzettel' button. The table contains two rows of data:

kassenzettelID	kassenzettelBildname	kassenzettelErfassdatum	kassenzettelAusgelesenerText	kassenzettelEndbetrag
1	Bildname1	2018-6-25-13-20-24	AusgelesenerText1	100.5
2	Bildname2	2018-6-25-13-21-40	AusgelesenerText2	10.5

Abbildung 5: Realm-Studio mit zwei Testeinträgen

Sind nun Daten in die Datenbank eingetragen, können diese natürlich auch aufgerufen werden. Realm zu folge können alle Einträge einer Tabelle in eine Liste geladen werden. Aus dieser Liste können dann bestimmte Objekte und Daten heraus gefiltert oder gesucht werden.

```

1 //Ruft alle Daten-Objekte aus der Tabelle in eine Liste
2 var kassenzettelList: Results<kassenzettel>{
3     get {
4         return realm.objects(kassenzettel.self)

```

```
5 | }
6 | }
```

Leider funktionierte dieser Aufruf nicht. Es erschien die Fehlermeldung "Thread 1: Fatal error: Unexpectedly found nil while unwrapping an Optional value". Ich konnte nicht herausfinden, ob es ein Fehler von meiner Seite war, oder ein Problem mit der installierten Realm-Version.

```
35 var realm: Realm!
36
37 var kassenzettelList: Results<kassenzettel> {
38     get {
39         return realm.objects(kassenzettel.self)
40     }
41 }
```

Thread 1: Fatal error: Unexpectedly found nil while unwrapping an Optional value

Abbildung 6: Fehlermeldung bei dem abrufen aller gespeicherten Daten-Objekte einer Tabelle in eine Liste

Alternativ konnte ich aber auf die Objekte direkt in der Datenbank zugreifen, diese sortieren und damit die grösste und damit aktuellste ID heraussuchen. Von dieser ausgehend konnte ich dann manuell die drei letzten Einkäufe in der Tabelle auf der Hauptseite darstellen. Mit ist bewusst, dass dies absolut nicht professionell gelöst wurde. Immerhin konnte ich so aber die erfolgreiche Implementierung darstellen.

Resumee:

Realm hat mich sehr überzeugt. Die Implementierung ist recht einfach und auch die Handhabung ist äusserst simpel.

## 4.5 Benutzeroberfläche

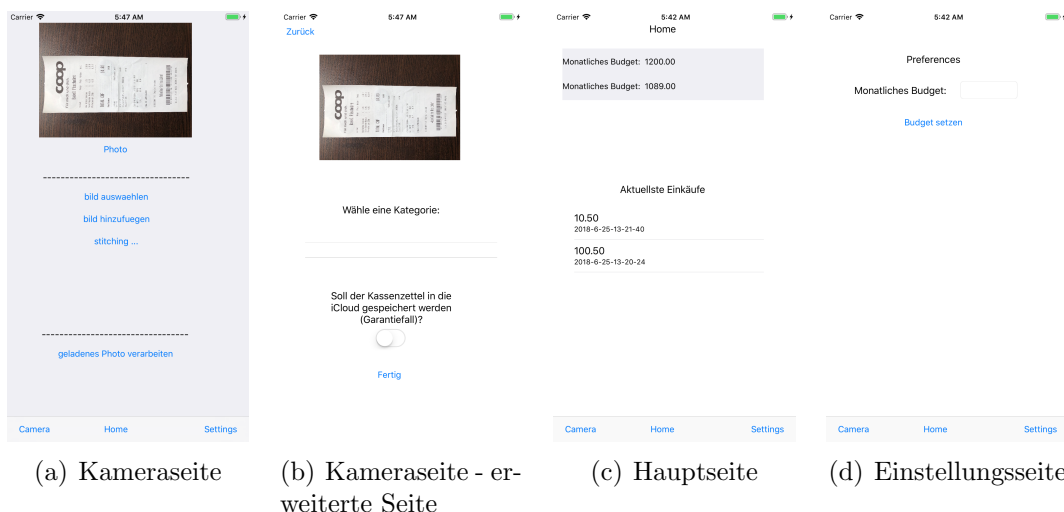


Abbildung 7: Benutzeroberfläche

Die Benutzeroberfläche habe ich annähernd so angepasst, wie es im Projekt zuvor mit dem Auftraggeber in einem Mockup zusammen erarbeitet wurde. Das bedeutet, die App

baut sich aus drei anzuwählenden Seiten zusammen, die über eine Menüleiste zu unterst angewählt werden können. Auf der Hauptseite ist das gesetzte Budget und darunter das noch verfügbare Budget des Monats ersichtlich. Unter diesen Angaben sind in einer Tabelle die letzten drei Einkäufe mit Datum und Endbetrag ersichtlich.

Auf die Kameraseite gelangt man über den linken Menü-Button. Hier kann über den Photo-Button ein Bild eines Kassenzettels erstellt oder eines aus der Bildergalerie geöffnet werden. Darunter befindet sich die Stitching-Funktion, welche aus mehreren Bildern ein Panoramabild erstellt. Damit kann ein Abbild eines besonders langen Kassenzettels erstellt werden. Diese Funktion wurde in dem Projekt vor der Diplomarbeit während dem Semester erstellt und ist nicht Teil dieser Diplomarbeit. Danach kann über den *geladenes Foto verarbeiten*-Button fortgefahren werden. Es erscheint eine weitere Seite, auf der die Kategorien ausgewählt werden können. Hier kann auch angewählt werden, ob der Kassenzettel in die iCloud hochgeladen werden soll, oder nicht. Mit dem Fertig-Button wird die Verarbeitung abgeschlossen, der Kassenzettel wird dann im Hintergrund ausgelesen und alle nötigen Funktionen werden aufgerufen. Danach zeigt die App wieder die Startseite an, auf der dann der eben verarbeitete Kassenzettel als letzter aktiver Einkauf angezeigt wird. Das noch verfügbare Budget wurde auch mit dem ausgelesenen Endbetrag des Kassenzettels verrechnet und damit verringert.

Über den rechten Menü-Button wechselt die App auf die Einstellungs-Seite. Dort kann das monatliche Budget angepasst werden. Hier steht noch genug Platz für eventuelle Erweiterungen bereit, zum Beispiel eine Tabelle, in der alle in die iCloud hochgeladenen Kassenzettel-Abbilder durchsucht werden können.

Neben der Benutzeroberfläche, habe ich die bestehenden Swift-Dateien aus dem Projekt vor der Diplomarbeit, umbenannt. Dies schafft eine gewisse Logik bei dem Aufbau dieser Dateien in Verbindung mit den drei bzw. vier Benutzeroberflächen.

Da zu meiner App für mich auch ein tolles App-Logo gehört, habe ich von einem Bekannten, der als Grafikdesigner arbeitet, noch ein App-Logo kreieren lassen. Dieses habe ich der App noch hinzugefügt.



Abbildung 8: Das App-Logo

## 4.6 iCloud Upload

### 4.6.1 Installation

Ein Teil dieser Diplomarbeit ist, Kassenzettel-Abbilder in die iCloud hochladen zu können. Dies ist für Kassenzettel gedacht, die spezielle Garantiesprüche haben und damit besonders gesichert werden sollen. Durch den Upload in die iCloud kann bei einem Verlust der lokalen Daten, lokalen Kassenzettel-Abbilder oder sogar des gesamten Handys, diese Kassenzettel später aus der iCloud wieder hergestellt werden.

Diese Upload-Funktion ist für die App-Anwenderin, den App-Anwender nicht obligatorisch. Die Anwenderin, der Anwender kann nach dem Erstellen des Kassenzettel-Abbildes über einen Button die iCloud-Upload-Funktion aktivieren oder deaktiviert lassen.

Die Integration dieser iCloud-Verbindung begann bereits mit einer Herausforderung. So ist bei Apple nirgends beschrieben, dass es für die Entwicklung mit iCloud den Zutritt zum *Apple Developer Program* benötigt. Zu Beginn muss im Xcode die iCloud-Entwicklung zur bisherigen Entwicklung zusätzlich aktiviert werden. Diese Aktivierung stand aber im Xcode nicht zur Verfügung. Nach einer langen Suche im Internet bin ich in einem Forum (<https://stackoverflow.com/questions/46896595/xcode-9-push-notification-capability-missing?rq=1>, 26.01.2018, o.S.) fündig geworden, in dem der User Oluwatobi Omotayo beschreibt, dass es für die Entwicklung einiger Erweiterungen die Teilnahme bei dem *Apple Developer Program* benötigt.

Ich habe mich dann bei diesem Programm eingeschrieben, was mich 100.00 SFr. kostete und mir einen Zugang für ein Jahr gewährt. Zum Glück hatte ich das an einem Freitag bemerkt, denn die Aktivierung zu diesem Programm benötigte bei Apple 48 Stunden Bearbeitungszeit. Danach konnte ich mit der Entwicklung dieser Upload-Funktion beginnen.

Die folgenden Schritte liegen der Dokumentation von Apple zugrunde ([https://developer.apple.com/library/archive/documentation/DataManagement/Conceptual/CloudKitQuickStart/Introduction/Introduction.html#//apple\\_ref/doc/uid/TP40014987-CH1-SW1](https://developer.apple.com/library/archive/documentation/DataManagement/Conceptual/CloudKitQuickStart/Introduction/Introduction.html#//apple_ref/doc/uid/TP40014987-CH1-SW1), 19.09.2017, o.S.). Am Anfang muss, wie bereits erwähnt, die iCloud-Funktion in Xcode aktiviert werden. Dafür gibt es in den Einstellungen zu der entwickelten App ein Menü-Punkt "Capabilities". Ist man nun Teil des *Apple Developer Program*, ist dort die Option iCloud zu sehen, welche aktiviert werden kann. Ist diese aktiviert, wird das iCloud-Framework der App-ID hinzugefügt. Dann stehen noch weitere Unterfunktionen der iCloud zur Auswahl bereit. Eine der Unterfunktionen, die für den Upload von Kassenzettel-Abbildern nötig ist, ist das CloudKit. Dieses muss als nächstes in diesem iCloud-Menü zusätzlich aktiviert werden.

Daraufhin wird ein Container erstellt, der automatisch eine Bezeichnung nach der App-ID erhält. Diese Container-Bezeichnung ist völlig einzigartig und steht nun lediglich dieser App und mir als Entwickler zur Verfügung.

In diesem Bild ist auch die ID für diesen Container zu erkennen, die wie folgt lau-

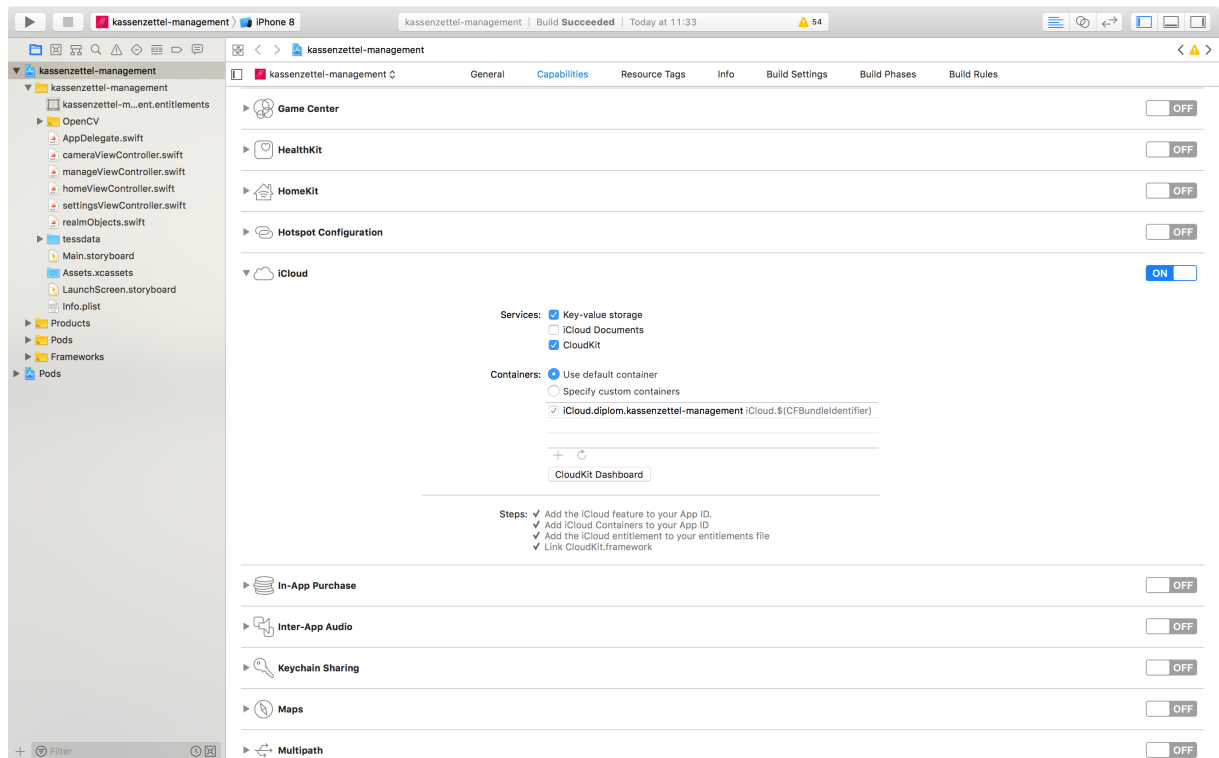


Abbildung 9: Hinzufügen der iCloud und des CloudKit

tet *iCloud.diplom.kassenzettel-management*. Darunter befindet sich ein Button *CloudKit Dashboard*. Wenn dieser angeklickt wird, wird man auf eine Webseite weitergeleitet (<https://icloud.developer.apple.com/dashboard>, 2018, o.S.). Auf dieser Seite muss man sich zuerst mit einem Apple Developer Account einloggen. Danach ist man auf der Management-Seite von CloudKit, welche alle Container anzeigt, die mit dem Apple Developer Account erstellt wurden. Hier können mehrere angezeigt werden, falls parallel weitere App-Entwicklungen mit iCloud betrieben werden.

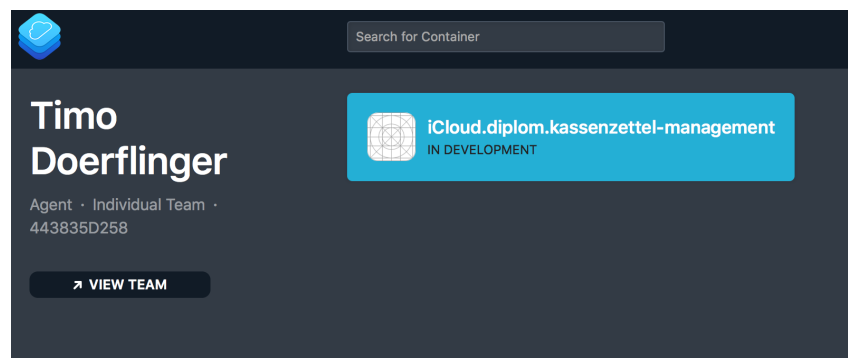


Abbildung 10: Die Oberfläche und die Anzeige der Container im CloudKit Dashboard

Wählt man dann den Container aus, den man aktuell verwenden möchte, gelangt man auf eine weitere Seite. Auf dieser Seite muss man sich entscheiden, ob man sich aktuell noch in der Entwicklung (Development) oder bereits in einer produktiven App (Production) befindet. Hier hat man dann je nach Option verschiedene Möglichkeiten. Unter

anderem können Log-Dateien der App ausgewertet oder Telemetrydaten der laufenden App betrachtet werden. Dies ist aber nicht Teil dieser Diplomarbeit. Hier ist lediglich die Option *Data* interessant.

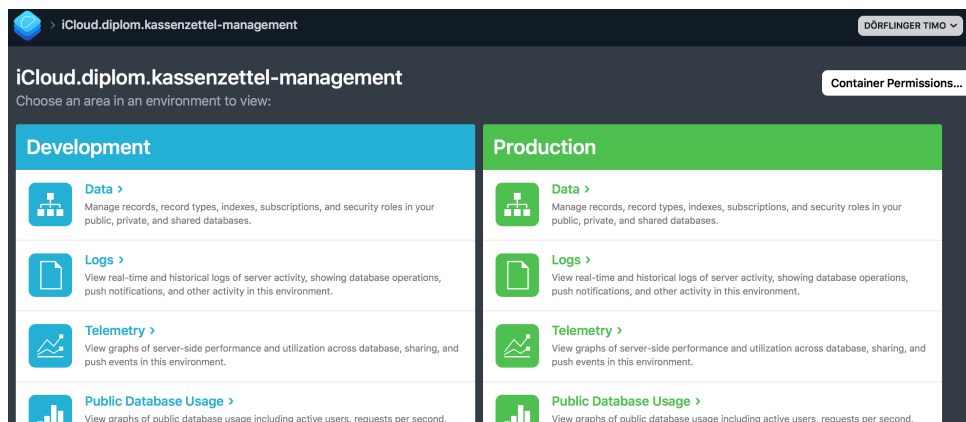


Abbildung 11: Hinzufügen der iCloud und des CloudKit

Wird die Option *Data* ausgewählt, erscheint eine Datenbank des Containers. In diese Datenbank können nun unter anderem String- oder Integer-Werte gespeichert werden. Diese Werte erlauben lediglich eine Speicherung bis zu 1MB Grösse. Sollen grössere Dateien oder Daten gespeichert werden, was für die Kassenzettel-Abbilder zutrifft, dann können die bereitgestellten Asset-Speicher verwendet werden. Diese erlauben eine Speicherung von bis zu 250MB.

Damit ist das CloudKit und die iCloud bereit für den Upload der Kassenzettel-Abbilder.

### 4.6.2 Programmierung

In der Datenbank in diesem Container der App kann nun, wie bei einer MySQL-Datenbank, ein Schema von Tabellen erstellt werden. Dies wird auch gut in der Dokumentation von Apple Development beschrieben. Jedoch ist die Beschreibung des Erstellens von Records und des Speicherns dieser Records in die iCloud nicht sehr ausführlich. Es vermittelt zwar, wie es generell funktioniert. Um es aber selbst anwenden zu können, ist es zu wenig dokumentiert.

Dies fällt besonders bei der Beschreibung der Speicherung von Assets auf. Diese Assets sind die Speichertypen für grössere Dateien und Daten, wie zum Beispiel für Bilder. Um aber ein Asset-Objekt erstellen zu können, verlangt dieses als Attribut einen Pfad zu der zu speichernden Datei. Hier besteht nun ein Problem. Bei der Swift-Programmierung für iOS ist es nicht möglich, den lokalen Pfad des erstellten oder aufgerufenen Bildes aus der Bildergalerie zu ermitteln. Daher ist es der Dokumentation von Apple Development zufolge nicht klar, wie sich dieser Pfad zusammensetzen soll, wenn dieser zur Datei nicht ermittelt werden kann.

Daher habe ich mich im Internet umgesehen und bin auf einen Bericht von Gabriel Theodoropoulos (<https://www.appcoda.com/cloudkit-introduction-tutorial/>, 22.04.2015, o.S.) gestossen, der die Verwendung von CloudKit in einem Tutorial beschreibt. Gabriel Theodoropoulos führt auf, dass das Pfad-Problem gelöst werden kann, indem die zu verwendende Datei, in diesem Fall das Kassenzettel-Abbild, lokal in einem temporären Ordner gespeichert werden soll. Dafür muss der Pfad manuell angegeben werden. Das gespeicherte Bild kann in diesem temporären Ordner mittels diesem Pfad dem Assets-Objekt übergeben werden.

Ich bin dem Tutorial von Gabriel Theodoropoulos weiter gefolgt und konnte mit seiner Hilfe und der Beschreibung des Code-Aufbaus erfolgreich ein Kassenzettel-Abbild in die iCloud hochladen. Ihm zufolge kann der Code wie im nachfolgenden Code-Abschnitt aufgebaut werden und damit erfolgreich verwendet werden.

```
1 func iCloudUpload(bild: UIImage, ID: Int){
2
3     //https://www.appcoda.com/cloudkit-introduction-tutorial/
4     var imageURL: NSURL!
5     let documentsDirectoryPath = NSSearchPathForDirectoriesIn
6         Domains(.documentDirectory, .userDomainMask, true)
7         [0] as NSString
8
9     //set a name for the image, that will save temporarily in the
10    // directory
11    let tempImageName = "temp_image.jpg"
12
13    //migrate the image from the parameter to NSData
14    let imageData: NSData = UIImagePNGRepresentation(bild)!
15        as NSData
16    //set the directory with the name for the image
17    let path = documentsDirectoryPath.appendingPath
18        Component(tempImageName)
19    //set the directory as URL path
20    imageURL = NSURL(fileURLWithPath: path)
21    imageData.write(to: imageURL as URL, atomically: true)
22
23    //set an ID for the record
24    let kassenzettelRecordID = CKRecordID(recordName: ID)
25
26    //set the type of the record with the recordID
27    let kassenzettelRecord = CKRecord(recordType: "kassenzettel",
28        recordID: kassenzettelRecordID)
```

```

29 //give the record the attribute "kassenzettelDatum" with
30 // the timestamp()-output
31 kassenzettelRecord.setObject(timestamp() as NSString,
32     forKey: "kassenzettelDatum")
33
34 //set a CKAsset with the local fileURL from above
35 let imageAsset = CKAsset(fileURL: imageURL as URL)
36 //give the record the attribute "kassenzettelBild" with the
37 // image as CKAsset
38 kassenzettelRecord.setObject(imageAsset, forKey:
39     "kassenzettelBild")
40
41 //set the connection to the container with the private
42 // database from the App
43 let privateDatabase = CKContainer.default()
44     .privateCloudDatabase
45
46 //write the record to the database in the container
47 privateDatabase.save(kassenzettelRecord,
48     completionHandler: { (record, error) -> Void in
49     if (error != nil) {
50         //print(error!)
51     }
52 })
53 }

```

Der erfolgreiche Upload kann nun über das CloudKit Dashboard überprüft werden. Dafür öffnet man wieder das CloudKit Dashboard mit dem Container, dem Container der App und wählt wieder die Option "Dataäus. Dann öffnet sich das Dashboard in dem Menü-Punkt *Records* und zeigt alle hochgeladenen Kassenzettel-Abbilder als Assets an. Dort sind auch die mitgegebenen Zeitstempel zu sehen.

Leider hat an dieser Stelle die Zeit nicht mehr gereicht, noch eine weitere Funktion zu implementieren, die die hochgeladenen Kassenzettel-Abbilder auch wieder herunterlädt und in der App darstellt. Hier könnte ich mir vorstellen, dass die Kassenzettel in eine eigene Galerie in der App geladen und ein bestimmter Kassenzettel herausgenommen werden kann. Es könnte aber auch der Record des Kassenzettel-Abbildes in der Datenbank des Containers mit gewissen Suchbegriffen erweitert werden. So könnte der App-Anwender beim Hochladen des Kassenzettels gewisse Suchbegriffe mitgeben, die mit dem Kassenzettel-Abbild in die iCloud gespeichert werden. Es könnte dann später durch diese Suchbegriffe ein bestimmter Kassenzettel aus der iCloud wiederhergestellt werden.

Resumee: Ich empfinde dieses Thema als äusserst spannend. Besonders für diese App



ZONES	RECORDS	RECORD TYPES	INDEXES	SUBSCRIPTIONS	SUBSCRIPTION TYPES	SECURITY ROLES	
LOAD RECORDS FROM:		Record Name	Record Type	Fields	Ch. Tag	Created	Modified
Private Database timo_doerflinger92@web.de		1	kassenzettel	kassenzettelDatum 2018-6-25-15-26-46 kassenzettelBild 292.8kB Asset	jiuaqb9b	Mon Jun 25 2018 15:26:49 GMT+0200 (CEST)	Mon Jun 25 2018 15:26:49 GMT+0200 (CEST)
_defaultZone		2	kassenzettel	kassenzettelDatum 2018-6-25-16-21-58 kassenzettelBild 44.9kB Asset	jiucp9hm	Mon Jun 25 2018 16:21:59 G MT+0200 (CEST)	Mon Jun 25 2018 16:21:59 G MT+0200 (CEST) ✖

USING: Query Fetch Changes

QUERY FOR RECORDS OF TYPE: kassenzettel

Filter by: Add filters...

Sort by: Add sorts...

Query Records

Query for records of the type "kassenzettel" that are in the "\_defaultZone" zone of the "private" database.

Abbildung 12: Die hochgeladenen Kassenzettel-Abbilder als Assets im Dashboard des CloudKit-Containers.

ist eine Möglichkeit des Uploads in die iCloud ein sehr spannender Zusatz. Die Entwicklung ist jedoch sehr umfangreich und eine schnelle Einarbeitung für eine grössere Implementierung ist nicht möglich.

## 5 Schlussfolgerung

Das Thema dieser Diplomarbeit konnte leider wegen dem zeitlichen Rahmen nicht komplett abgeschlossen werden. Es sind noch Probleme vorhanden, die noch gelöst werden müssen. Dennoch hat das Thema der Kassenzettel-Verwaltung mit einem Handy im allgemeinen viele Vorteile. Bei einem weiteren Ausbau dieser App besteht hier ein sehr grosses Potenzial für eine reelle Markteinführung.

Ich persönlich bin sehr stolz auf das, was ich in dieser Diplomarbeit und mit dieser App aufgebaut habe. Anfänglich habe ich den Umfang und das ganze Ausmass dieser App unterschätzt. Ich hatte zu Beginn einen einfachen Grundrahmen für diese App im Kopf, den ich mit der Zeit der Erarbeitung kontinuierlich erweitern musste. Dennoch konnte ich in dem doch engen, zeitlichen Rahmen eine funktionierende App aufbauen, die die wichtigsten Funktionen abdeckt.

Der Lerngewinn durch die Erstellung der App ist für mich sehr positiv. Nicht nur bezüglich des Programmierens mit Swift, sondern auch durch die allgemeine Durchführung dieses Projekts. Ich konnte mit der Verwendung von Scrum als Projektmanagement eine weitere Methode verwenden. In den Semestern zuvor hatten wir bereits Hermes5 als Projektmanagement-Methode kennen und nutzen gelernt. Doch Hermes5 eignet sich nicht besonders für solch ein agiles Software-Entwicklungs-Projekt. Hier ist Scrum mit seinem agilen Aufbau deutlich nützlicher.

Des Weiteren konnte ich mit Realm ein sehr interessantes Datenbank-Framework kennenlernen. Da Realm auch für andere Plattformen genutzt werden kann, behalte ich das sicherlich als nützliches Datenbank-Framework im Hinterkopf.

Auch die Arbeit mit der iCloud und dem CloudKit war mehr als interessant. Nachdem die Einarbeitung nicht einfach war, ist das Erfolgserlebnis, wenn nach der investierten Zeit das Ergebnis funktioniert und verwendet werden kann, umso grösser.

Für eine weitere Entwicklung können sicherlich viele erarbeitete Teile wiederverwendet werden. Jedoch müsste in die OCR-Funktion, also in das Auslesen des Textes der Kassenzettel-Abbilder, noch mehr Zeit investiert werden. Dies ist sozusagen das Herz der Applikation, ohne das die restlichen Komponenten nicht funktionieren. Vielleicht müsste hier auch ein anderes OCR-Framework verwendet werden.

Für mich hat die Kassenzettel-Verwaltung als App sehr grosses Potential für eine weitere Erarbeitung und Verwendung.

## 6 Anhang

### 6.1 Quellenverzeichnis

Glossar - OCR: <https://de.wikipedia.org/wiki/Texterkennung>

Theoretischer Teil - Internetbericht: <https://www.derbund.ch/wirtschaft/standard/Das-Kassezeddeli-wird-zum-Auslaufmodell/story/17537630>

Methodische Vorgehensweise - IceScrum: <https://www.icescrum.com>

Methodische Vorgehensweise - OpenProject: <https://www.openproject.org>

Methodische Vorgehensweise - OpenProject - Anleitung: <https://www.openproject.org/docker/>

Datenbank-Schema: <https://www.mysql.com/de/products/workbench/>

Evaluierung des Datenbank-Frameworks: <https://rollout.io/blog/ios-databases-sqlite-core-data-realm/>

Realm Installation: <https://realm.io/docs/swift/latest/>

iCloud Installation: <https://stackoverflow.com/questions/46896595/xcode-9-push-notification-capability-missing?rq=1> <https://icloud.developer.apple.com/dashboard>

iCloud Programmierung: <https://www.appcoda.com/cloudkit-introduction-tutorial/>

### 6.2 Bilderverzeichnis

Abbildung 1: Selbst erstellt

Abbildung 2: Selbst erstellt

Abbildung 3: Selbst erstellt

Abbildung 4: Selbst erstellt

Abbildung 5: Selbst erstellt

Abbildung 6: Selbst erstellt

Abbildung 7: Alle vier Screenshots selbst erstellt

Abbildung 8: Selbst erstellt

Abbildung 9: Selbst erstellt

Abbildung 10: Selbst erstellt

Abbildung 11: Selbst erstellt

Abbildung 12: Selbst erstellt