HF-ICT

Diplomarbeit

Kassenzettelverwaltung mit iOS-App

Timo Dörflinger

19. Juni 2018

Inhaltsverzeichnis

1 Einleitung

In meiner Diplomarbeit erstelle ich eine App zur Kassenzettel-Verwaltung. Für meinen Auftraggeber und mich persönlich sind die Kassenzettel verlorene Daten, ausser man verwendet Sie zur persönlichen Analyse bzw. zum Führen eines Haushaltsbuchs. Daher erstelle ich in meiner Diplomarbeit eine iOS-App, mit welcher ein Kassenzettel fotografiert werden kann. Aus diesem erstellten Abbild werden dann gewisse Werte bzw. Daten mit Hilfe eines bereits bestehenden OCR-Frameworks ausgelesen und zur weiteren Verarbeitung zwischengespeichert. Hier kann also unter anderem der Gesamtbetrag ausgelesen und mit einem gesetzten monatlichen Budget verrechnet werden. Dies soll dann auch kategorisiert werden können. Daraus ist ersichtlich, was in dem laufenden Monat bereits in den verschiedenen Kategorien ausgegeben wurde. Die Kategorien können jeweils individuell erstellt werden.

2 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 mein 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 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 mich entschieden, die Diplomarbeit mittels Scrum zu führen.

Da ich, wie bereits erwähnt, einen engen Zeitrahmen von 60 Stunden 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` entdeckt. Es sah einfach aus und war für ein Projekt kostenlos. Das Management findet in der Cloud des Anbieter statt. Bei dem ersten Test nach der Anmeldung musste ich aber feststellen, das manche Seiten lediglich auf Französisch zur Verfügung stehen. Da ich aber so gut wie keine Französischkenntnisse habe, empfand ich das als nicht sonderlich hilfreich und dachte es ist besser nach einer Alternative zu suchen.

Nach einer weiteren Suche bin ich dann auf OpenProject gestossen. OpenProject ist eine *Open source project management software* mit Scrum integriert. OpenProject bietet eine Community-Version an, die kostenlos als auch lokal verwendet werden kann und 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 die Diplomarbeit.

Ich habe das OpenProject in meiner lokalen Docker-Umgebung auf dem MacBook installiert. Dafür bin ich der Anleitung von OpenProject gefolgt und bin nach diese Anleitung prompt in ein Problem gelaufen, dass mich zeit-

lich etwas aufgehalten hatte. Der Anleitung zu folge, sollte für eine produktive Nutzung eine erweiterte Installation gemacht werden. So kann gesichert werden, dass bei einem Neustart des Container keine Daten verloren gehen und auch die Log-Dateien lokal in einer selbst gesetzten Ordnerstruktur auf dem System zu finden ist.

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

```
1 docker run -d -p 8080:80 --name diplomopenproject -e SECRET_KEY_BASE=secr
2   -v /Users/Timo/diplom-kassenzettelverwaltung/projektmanagement/openproj
3   -v /Users/Timo/diplom-kassenzettelverwaltung/projektmanagement/openproj
4   -v /Users/Timo/diplom-kassenzettelverwaltung/projektmanagement/openproj
5   openproject/community:7
```

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

```
1 docker run -it -p 8080:80 --name diplomopenproject -e SECRET_KEY_BASE=sec
2   -v /Users/Timo/diplom-kassenzettelverwaltung/projektmanagement/openproj
3   -v /Users/Timo/diplom-kassenzettelverwaltung/projektmanagement/openproj
4   -v /Users/Timo/diplom-kassenzettelverwaltung/projektmanagement/openproj
5   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* den Container stoppen und mit *docker start diplomopenproject* den Container wieder starten und es wieder im Browser darstellen.

Nach der Installation ist bereits ein Administrator-Account erstellt, der für den ersten Login verwendet werden kann. Die Zugangsdaten stehen in der Installations-Anleitung. Nachdem 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 können. 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. Diesem Benutzer werden dann die Arbeitspakete und deren enthaltener Aufgaben und Meilensteine zugewiesen.

4 Basis-App

In einem Projekt während dem Semester vor der Diplomarbeit habe ich bereits eine App erstellt, die als Basis für diese Diplomarbeit dienen soll. Diese App hat Tesseract implementiert. Tesseract ist ein OCR-Framework[1], welches den Text von einem Bild auslesen kann. Die App ist soweit aufgebaut, dass das Abbild eines Kassenzettels mit OpenCV durch diverse Bildaufbereitungs-Funktionen soweit aufgebereitet 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.

Dieser ausgelesene und gefilterte Endbetrag wird dann für die weitere Verarbeitung bereitgestellt. An dieser Stelle setzt die Diplomarbeit nun an verarbeitet den ausglesenen Endbetrag nun weiter.

5 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 bereitzustellen. In diese Datenbank können dann die ausgelesenen Endbeträge der Kassenzettel, mit dem jeweiligen Erfass-Datum und noch weiteren Daten zwischengespeichert werden. So hatte ich mich bereits während dem ersten Projekts bezüglich einer lokalen Datenbank für eine iOS-App schlau gemacht. Hier sind mir nach diversen Berichten *SQ-Lite* und *Apple Core Date* 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, mit einer steigenden Begeisterung bei Programmierern.

Da leider nicht genug Zeit in der Diplomarbeit übrig war, konnte ich diese drei Optionen selbst nicht mit einem Testaufbau testen. Die nachfolgenden Vergleiche beziehen sich daher auf verschiedene Berichte, die diese drei Optionen verglichen haben.

[1] OCR: (englisch: *optical character recognition*) steht für eine automatisierte Erkennung eines Texts in einem Bild

5.1 SQLite

SQLite ist die schlankere Form von MySQL und damit ein vollumfängliches Datenbank-Framework für iOS das auf Tabellen und Beziehungen (Relations) aufbaut, wie es wir es im Datenbank-Unterricht gelernt haben. SQLite kann mit einer grossen Menge Daten umgehen und die Daten können mit einfachen SQL-Querys (Abfragen) eingefügt oder aufgerufen werden. Jedoch sind laut den verschiedenen Internet-Beiträgen haben diese Querys als iOS-Framework keine hohe Performance und ist wohl recht aufwändig in der Implementierung in die iOS-App.

5.2 Apple Core Data

Apple Core Data ist der von Apple entwickelte Datenbank-Aufbau. Wobei Core Data anders als die üblichen Datenbank-Aufbauten gehandhabt wird. Core Data selbst ist keine Datenbank. Es baut auf SQLite auf und verwaltet die Daten als Objekte. Das macht die Verarbeitung von Daten recht schnell. Allerdings ist der Aufbau von Core Data recht kompliziert und verlangt eine grosse zeitliche Einarbeitung.

5.3 Realm

Realm ist ein noch relativ junges Framework und verarbeitet die Daten, ähnlich wie Core Data, in Objekten. Es ist aber ein relativ schlankes Framework, welches eine sehr hohe Performance bietet. Auch die Verwendung von Realm kann mit weniger Code-Zeilen verwendet werden, als mit Core Data. Was ich als einen weiteren Vorteil sehe, ist dass Realm auch Plattformübergreifend verwendet werden kann. Soll die App also später in einer produktiven Umgebung auf das Android-System migriert werden, können aus dem Realm-Aufbau der iOS-App einfach die Datenbank-Dateien in einen Android-Realm-Aufbau übernommen werden.

Aus den oben beschriebenen Vergleichen werde ich die Datenbank für dieses Projekt in Realm umsetzen.

6 Datenbank-Schema

Für eine saubere Darstellung des Datenbank-Aufbaus, habe ich ein Schema erstellt. Das Schema habe ich in MySQL-Workbench erstellt. Mit MySQL-Workbench können Datenbank-Schemas mit einer guten Darstellung aufgebaut und veranschaulicht werden.

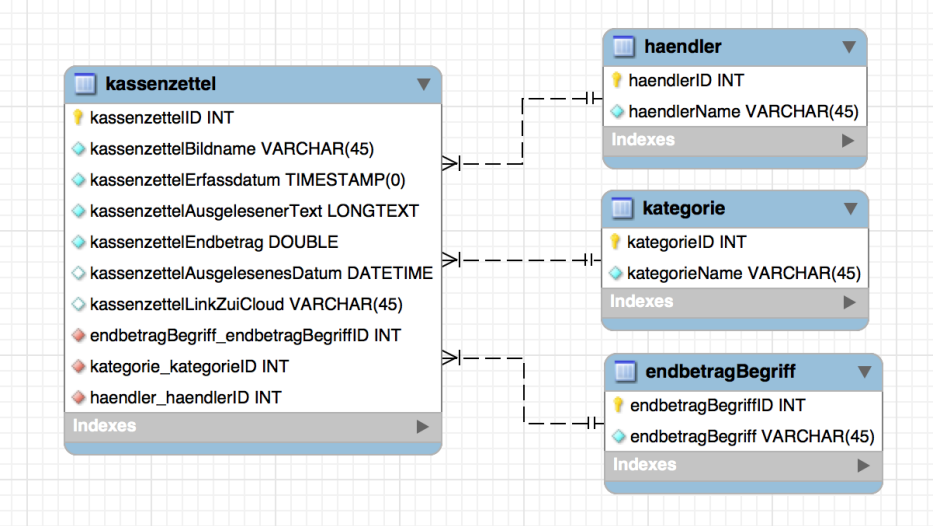

Abbildung 1: Datenbank-Schema

Die Tabelle *kassenzettel* enthält alle für den Moment nötigen Attribute. Da nicht genug Zeit zur Verfügung steht, alle gekauften Artikel von dem Kassenzettel auszulesen, wird lediglich der Endbetrag ausgelesen. Daher ist es hier nicht nötig, eine weitere Tabelle oder weitere Attribute für die einzeln gekauften Artikel zu berücksichtigen. Wir ein Kassenzettel fotografiert und in die App eingelesen, bekommt das Kassenzettel-Abbild einen Datenbankeintrag mit einer einzigartigen (anderen Begriff finden) 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 als dieser in dem Attribut *kassenzettelAusgelesenerText* übergeben. Der aus diesem ausgelesenen Text extrahierte Endbetrag wird dem Attribut *kassenzetteEndbetrag* übergeben. Alle bisherigen Attribute sind als *Not-Null* definiert. Das Bedeutet dass diese Attribute zwingend einen Wert für den Datenbankeintrag erhalten müssen. Das Attribut *kassenzettelAusgelesenesDatum* und *kassenzettelLinkZuiCloud* sind nicht zwingend nötig. Das erstere der beiden Attribute wird nicht immer erfolgreich von Tesseract ausgelesen und soll damit nicht Teil dieses Projekts sein. Das zweite der beiden Attribute ist nicht zwingend, da nicht jeder Kassenzettel in die iCloud gesichert werden möchte. Diese Funktion soll speziell für Kassenzettel zur Verfügung stehen, die wichtige Garantieansprüche beinhalten.

Neben der Tabelle *kassenzettel* gibt es noch die Tabelle *haendler*, welche die einzelnen Supermärkte, Tankstellen oder Läden beinhaltet.

Die Tabelle *kategorie* enthält die zur Verfügung stehenden Kategorien für die Kassenzettel. Hier sollen in diesem Projekt lediglich die Kategorien Le-

bensmittel, Tanken und XXX zur manuellen Anwahl für den Anwender der App zur Verfügung stehen. In diesem Projekt ist es möglich, lediglich eine Kategorie pro eingelesenem Kassenzettel zu wählen. Das bringt natürlich das Problem mit sich, falls ein Kassenzettel Artikel beinhaltet, die auf mehrere Varianten zutrifft. Falls also 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 Anwahl einer Kategorie durch eine *KI* bzw. *Machine Learning* ersetzt werden. So müsste der App-Anwender diese Kategorien-Anwahl nicht mehr selbst erledigen, was die Benutzerfreundlichkeit einfacher machen würde (von einer zuverlässigen Auswahl der Kategorien mittels des Machine Learning vorausgesetzt).

Die Tabelle *endbetrag* enthält die auf den Kassenzettel befindlichen Endbetrags-Begriffe. So werden die gezahlten Endbeträge auf den Kassenzetteln mit den verschiedensten Begriffen angeführt. So zum Beispiel mit Summe, Bar, Total und vielen weiteren Begriffen mehr. Diesbezüglich gibt es keine Landesweite oder eine andere Beschränkung oder eine gewisse Norm des Kassenzettel-Aufbaus. Hier könnte für eine produktive App die Erweiterung dieser Begriffe von dem App-Anwender über die Einstellungen getätigt werden, falls der Begriff vor dem Endbetrag auf dem Kassenzettel der App nicht bekannt ist und dieser somit in der Suchliste erweitert werden müsste. Auch diese Funktion könnte mittels *KI* oder *Machine Learning* eventuell gelöst und damit vereinfacht werden.

Die Beziehungen zwischen den Tabellen sind alle 1:n-Beziehungen. So kann ein Kassenzettel immer nur von einem Händler ausgestellt worden sein und lediglich einen Endbetrag enthalten. Wie bereits erwähnt könnte ein Einkauf mehrere Kategorien entsprechen, in diese 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 ist.

7 Datenbank Realm

7.1 Realm Installation

Wie zuvor evaluiert, habe ich mich für Realm entschieden. Um Realm in die iOS-App einbauen zu können, muss der App zuerst das Framework hin-

zugefügt werden. Dafür verwende ich das Cocoapods. Cocoapods ist ein Dependency-Manager[2] für Swift und Objective-C. 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. Die Diplomarbeit baut auf einer bereits bestehenden App auf, die bereits eine Pods 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 RealmSwiftßwischen "targetünd ënd"hinzugefügt. Nach diesem Eintrag 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.

```
# 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 2: Die Pod-Datei vor der Installation des Realm-Frameworks.

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

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

7.2 Realm Implementierung

Auf der Realm-Homepage ist eine grosse Dokumentation zu finden, die die Verwendung und Anwendung von Realm beschreibt. Nach dieser Dokumentation habe ich nun das Datenbank-Schema aufgebaut. Dies sieht wie folgt aus:

[2] Mit einem Depencency-Manager (Dependency englisch: Abhängigkeit) können Frameworks über einfache Terminal-Befehle einem Xcode-Projekt hinzugefügt werden.

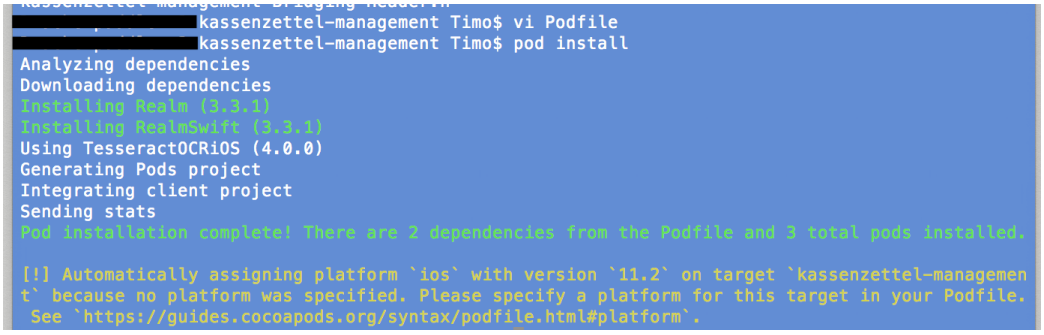

Abbildung 3: Der Installations-Vorgang von Cocoapods.

```
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   //folgende zwei Attribute sind optional
11   @objc dynamic var kassenzettelAusgelesenesDatum
12       : Data? = nil
13   @objc dynamic var kassenzettelLinkZuiCloud
14     : String? = nil
15   //One-to-many Relationship
16   @objc dynamic var endbetragBegriff
17     : endbetragBegriff?
18   @objc dynamic var kategorie: kategorie?
19   @objc dynamic var haendler: haendler?
20   //PrimaryKey überschreiben
21   override static func primaryKey() -> String? {
22     return "kassenzettelID"
23   }
24 }
25
26 // haendler model
27 class haendler: Object {
28   @objc dynamic var haendlerID = 0
```

```
29      @objc dynamic var haendlerName = ""
30      //PrimaryKey überschreiben
31      override static func primaryKey() -> String? {
32        return "haendlerID"
33      }
34    }
35
36    // kategorie model
37    class kategorie: Object {
38      @objc dynamic var kategorieID = 0
39      @objc dynamic var kategorieName = 0
40      //PrimaryKey überschreiben
41      override static func primaryKey() -> String? {
42        return "kategorieID"
43      }
44    }
45
46    // endbetragBegriff model
47    class endbetragBegriff: Object {
48      @objc dynamic var endbetragBegriffID = 0
49      @objc dynamic var endbetragBegriff = ""
50      //PrimaryKey überschreiben
51      override static func primaryKey() -> String? {
52        return "endbetragBegriffID"
53      }
54    }
```

8 Produktiver Einsatz

Hier Beschreiben, was für einen produktiven Einsatz nötig wäre: Zuverlässigere OCR-Software, hier vielleicht auf keine Open Source-Lösung setzen (auf die Lizenzierung achten)

9 Verwendete Software

Hier die Beschreibung der verwendeten Software, Programme und Frameworks.

Ganz wichtig hier ist auch die Beschreibung der Versionen der Programme!!!!!!!!!!!!!!!!!!!!!!!!

9.1 MySQL-Workbench

Diese können dann auch direkt in MySQL für die Datenbank importiert werden, womit dann die entsprechenden Tabellen und deren Beziehungen aufgebaut werden.

10 Quellenverzeichnis

Anleitung für OpenProject in Docker `https://www.openproject.org/docker/`
Video-Anleitung für Scrum Nutzung in `https://www.openproject.org/collaboration-software-features/scrum-agile-project-management/`
OCR: `https://de.wikipedia.org/wiki/Texterkennung` Cocoapods `https://cocoapods.org`

11 Bilderverzeichnis