



Institut für Informatik  
Lehrstuhl für Organic Computing  
Prof. Dr. rer. nat. Jörg Hähner

**Masterarbeit**

**Learning drinking patterns with  
Q-Learning and Feature Selection on an  
Ethereum Blockchain**

Fabian Pieringer

Erstprüfer: Prof. Dr. rer. nat. Jörg Hähner

Zweitprüfer: Prof. Dr. Albert Einstein

Betreuer: Betreuer am Lehrstuhl, M.Sc.

Matrikelnummer: 123456

Studiengang: Informatik (Master)

Eingereicht am: 9. Mai 2019

# **Abstract**

In dieser Masterarbeit ...



# Inhaltsverzeichnis

<b>Abstract</b>	<b>i</b>
<b>1 Einleitung</b>	<b>1</b>
1.1 Aufbau dieser Arbeit . . . . .	1
1.2 Problemstellung . . . . .	1
<b>2 Stand der Technik</b>	<b>3</b>
2.1 Blockchain . . . . .	3
2.1.1 Funktionsweise & Konzepte . . . . .	3
2.1.2 Ethereum . . . . .	3
2.2 Machine Learning . . . . .	3
2.2.1 Überblick . . . . .	3
2.2.2 Q-Learning . . . . .	3
<b>3 System Architektur</b>	<b>5</b>
3.1 Überblick . . . . .	5
3.1.1 Architektur . . . . .	5
3.1.2 Workflow . . . . .	11
3.1.3 Entwicklungsprozess . . . . .	17
3.2 Blockchain . . . . .	21
3.2.1 Genesis Block . . . . .	21
3.2.2 CoffeeCoin . . . . .	24
3.2.3 Beverage-List . . . . .	26
3.3 Learner . . . . .	27
3.3.1 Modellierung . . . . .	27
3.3.2 Q-Learning . . . . .	27

## *Inhaltsverzeichnis*

3.3.3	Lernprozess & Ablauf . . . . .	27
3.4	Tablet-App . . . . .	27
3.4.1	Interface . . . . .	28
3.4.2	Internal Workflow . . . . .	32
<b>4</b>	<b>Studie</b>	<b>35</b>
4.1	Testphase . . . . .	35
4.2	Evaluation . . . . .	35
<b>5</b>	<b>Zusammenfassung</b>	<b>37</b>
5.1	Weiterführende Forschungsfragen . . . . .	37
5.2	Ausblick . . . . .	37
	<b>Literaturverzeichnis</b>	<b>I</b>
	<b>Abbildungsverzeichnis</b>	<b>III</b>
	<b>Tabellenverzeichnis</b>	<b>V</b>
<b>A</b>	<b>Anhang A</b>	<b>VII</b>
<b>B</b>	<b>Anhang B</b>	<b>IX</b>

# **1 Einleitung**

## **1.1 Aufbau dieser Arbeit**

## **1.2 Problemstellung**





## **2 Stand der Technik**

### **2.1 Blockchain**

#### **2.1.1 Funktionsweise & Konzepte**

#### **2.1.2 Ethereum**

### **2.2 Machine Learning**

#### **2.2.1 Überblick**

#### **2.2.2 Q-Learning**



## **3 System Architektur**

### **3.1 Überblick**

Im folgenden wird nun die System Architektur und der Workflow erläutert, welche als Grundlage für die Studie dienen, um die in 1.2 geschilderte Problemstellung abzubilden und letztendlich zu lösen.

#### **3.1.1 Architektur**

Um ein besseres Bild davon zu bekommen, wie die einzelnen Komponenten zusammenhängen bzw. welche Aufgaben diese in Wechselwirkung zu anderen Instanzen übernehmen und ausführen, wird zunächst die Architektur des Systems erläutert. Als Basis soll dabei die Abbildung 3.3 dienen, anhand jener vorrangig die jeweiligen Komponenten bezüglich ihrer Funktionsweise beschrieben werden. Das Zusammenspiel der Anwendungen wird im Anschluss unter 3.1.2 detailliert beleuchtet.

### 3 System Architektur

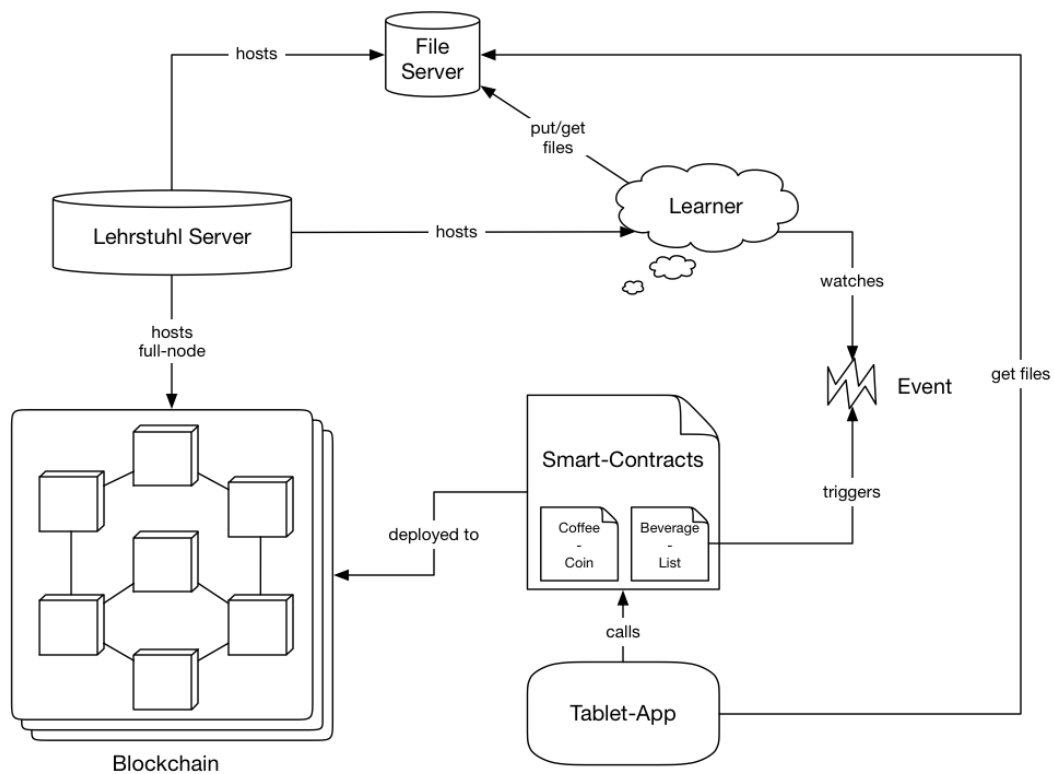


ABBILDUNG 3.1: Systemarchitektur

#### Lehrstuhl Server

Der Lehrstuhlserver ist dafür zuständig einen Großteil der Anwendungen zu hosten bzw. zu starten. Dies gilt sowohl für den HTTP-Fileserver als auch für die Learner-Anwendung, welche dessen Betriebssystem als Plattform nutzen. Auch die Blockchain wird auf dem Server gestartet und verwendet diesen zudem als full-node, um Transaktionen zu minen. Der Server ist im Grunde der Anwendungen, dessen primäre Funktionsweise darin besteht jenen eine Plattform zu bieten und mit Rechenleistung zu versorgen.

#### Learner

Der sogenannte “Learner” ist eine in Golang implementierte Softwareanwendung, dessen Hauptaufgabe darin besteht, das Kaffeetrinkverhalten der Nutzer zu erlernen - wovon auch die Namensgebung der Anwendung stammt. Um dies

zu erreichen, wurde die Anwendung in Submodule unterteilt, welche einen dedizierten Aufgabenbereich abdecken und diesen eigenständig bearbeiten. Auch wenn jene für sich autark agieren können, kann das Trinkverhalten letztendlich erst in der gegenseitigen Wechselwirkung jener erlernt werden.

Die Submodule lauten wie folgt:

- Q-Learning
- Worker
- Watcher
- (Smart Contract Deployment Skript)

Das **Q-Learning** ist, wie der Name bereits impliziert, für das eigentliche Erlernen des Trinkverhaltens zuständig. Es ist im Grunde die Implementierung des Q-Learning Algorithmus, sowie die damit einhergehende Zustandsraummodiellerung, welches aber unter `refsubsec:ql` genauer erläutert wird.

Der **Worker** ist einerseits für die Userverwaltung und andererseits für die, in einem festgelegten Intervall, Ausführung des Q-Learning Algorithmus, zuständig. (vgl. Kap. 3.3.3)

Der **Watcher** beobachtet Events, die vom Smart-Contract “Beveragelist” ausgelöst wurden. Die Daten, welches das Event beinhaltet, werden daraufhin verwendet um den Q-Learning Algorithmus zu befüllen und aufgrund diesen das Trinkverhalten zu erlernen.

Das **Smart Contract Deployment Skript**, ist in der Form zwar nicht in der Systemarchitektur vorhanden, da es aber auch ein Submodul des Learners und für das gesamte Konstrukt dahingehend essentiell ist, da es die Smart Contracts auf der Blockchain installiert und im Zuge dessen erst die Verbindung zwischen Blockchain und Learner ermöglicht, wird es in dieser Auflistung trotzdem aufgeführt.

#### **Fileserver**

Der Fileserver ist eine Go-Anwendung, welche eine rudimentäre REST [Wikd] Api [Wika] zur Verfügung stellt. Von den sogenannten CRUD [Wikc] Operationen, welche als grundlegend für alle persistenten Datenspeicher angesehen werden können, implementiert dieser nur das “GET” und das “PUT”. Sowohl die “PATCH” als auch die “DELETE” Operation bieten keinen Mehrwert für die Gesamtarchitektur bzw. den Workflow und sind in Anbetracht dessen nicht implementiert.

Das bedeutet die Hauptaufgabe des Fileservers besteht darin, Dateien zu empfangen und zu speichern (PUT) und diese auf Anfrage (GET) an einen Antragsteller wieder zu versenden.

Außerdem bietet die Anwendung zusätzlich zur Api einen UDP-Broadcast, welcher v.a. beim Testing und beim Setup eine große Erleichterung darstellt. Dieser Broadcast versendet in seinen Nachrichten lediglich die IP-Adresse des Lehrstuhl-Servers und somit auch seine eigene und die der Blockchain. Da die IP-Adresse und der Port des Broadcasts stets gleich bleiben, sich aber die Host-IP der Blockchain und des Fileservers je nach Deployment theoretisch ändern können - was in der Entwicklungsphase sehr oft der Fall war. Müssen sich sowohl der Learner als auch die Tablet-App lediglich auf den Broadcast “subscriben” und können dadurch die IP der Blockchain und des Fileservers erfahren. Durch diese dynamische Zuweisung der IP-Adresse, müssen keine Updates beim Learner und der App durchgeführt werden, sollte die Blockchain und der Fileserver auf einem anderen Host deployed werden.

Aufgrund der Tatsache, dass sich die IP-Adresse des Lehrstuhl-Servers während der Studie nicht ändert, ist der UDP-Broadcast auch nicht in der Abbildung 3.1 der Systemarchitektur berücksichtigt worden. Der Anwendungsbereich ist trotz alledem im Bereich der Testphase und auch für die künftige Projekte, bei denen das System Verwendung findet, definitiv vorhanden.

#### **Blockchain**

Die Blockchain ist eine private, eigens für die Studie erstellte Ethereum-Blockchain, dessen “Genesis-Block” aus dem JSON-File (vgl. Abbildung ??) generiert wird. Die Erläuterungen zu den jeweiligen Key-Value-Pairs sind unter Kap. 3.2.1 zu finden. Das Generieren und das Starten der Blockchain erfolgt auf dem Lehrstuhlserver. Dabei hostet der Server zudem eine sogenannte “full-node” (auch “miner” genannt) der Blockchain, welche dafür zuständig ist Transaktionen zu berechnen und zu bestätigen. Aus Ressourcengründen ist dieser “miner” der einzige im Gesamtsystem, was aus theoretischer Sicht einen “Single Point of Failure” [Wike] als Nachteil mit sich zieht. Das bedeutet sollte diese “full-node” ausfallen, würden keine Transaktionen mehr bestätigt werden. Da es weder während der Entwicklungsphase noch während der Studie zu einem einzigen Ausfall kam, ist dieser Nachteil als sehr klein einzuschätzen, weswegen auch keine weitere “full-node” zum System hinzugefügt wurde. Der große Vorteil besteht allerdings darin, dass Transaktionen sehr schnell bestätigt werden, da es keine weiteren “node’s” gibt, die um die Berechnung eines Block’s konkurrieren. Was vor allem aus Sicht der User-Experience [Wikf] einen großen Mehrwert darstellt, da dieser in wenigen Sekunden erfährt, ob seine Transaktion erfolgreich durchgeführt wurde. Dies kann bei anderen Blockchains wie z.B. Bitcoin bis zu 10 Minuten dauern [Rap09], was im Kontext der Systemarchitektur nicht tragbar wäre.

Um letztendlich mit der Blockchain kommunizieren und dessen Potential in voller Gänze ausschöpfen zu können, werden auf diese sogenannte Smart-Contracts [Dav15] deployed. Im Rahmen der Systemarchitektur sind es zwei dedizierte Smart-Contracts (*Coffe-Coin*, *Beverage-List*), welche komplett unabhängig voneinander agieren.

#### Smart Contracts

Die beiden Smart Contracts welche auf die Blockchain deployed werden, werden mit *Coffe-Coin* und *Beverage-List* betitelt. Diese decken zwei völlig unterschiedliche Aufgabenbereiche ab, weswegen sie keinen Einfluss aufeinander haben und deswegen unabhängig voneinander operieren. So löst nur der *Beverage-List Contract* ein Event aus, sobald eine bestimmte Funktion dessen aufgerufen wird.

Die Ausführung (*call*) beider erfolgt jedoch stets von Seiten der *Tablet-App*. Diese ist auch die einzige Instanz, welche in Form von Transaktionen mit der Blockchain interagiert.

#### Tablet-App

Die *Tablet-App* ist eine mit React-Native [RN:] erstellte Crossplattform App [Wikb], welche auf einem Android Tablet installiert ist. Die Hauptaufgabe der App ist es Funktionen der beiden Smart Contracts aufzurufen, in dem es die benötigten Daten an den Smart Contract übergibt, um schlussendlich Transaktionen auszulösen.

Damit eine Kommunikation mit einem Smart Contract überhaupt zustande kommt, schickt die App einen Request an den Fileserver, welcher mit den angefragten Smart Contract Daten in Form einer Datei antwortet.

#### Event

Das Event beschreibt im Grunde die indirekte Kommunikation zwischen dem *Learner* und der *Tablet-App* mit dem Smart Contract *Beverage-List* als Mittelsmann. So wird jenes im Zuge eines Funktionsaufrufs des Smart Contracts von Seiten der App ausgelöst und vom *Learner* detektiert und der Inhalt zum Erlernen des Kaffeetrinkverhaltens verwendet.



### 3.1.2 Workflow

Die unter Kap. 3.1.1 beschriebene Architektur wird im folgenden unter dem Gesichtspunkt des Workflows, also dem Zusammenspiel der einzelnen Komponenten und dem Gesamtablauf, näher betrachtet. Dabei beschreibt der Gesamtablauf die einzelnen Schritte startend beim Setup der Komponenten hin zum eigentlichen Durchlauf der einzelnen Softwareanwendungen, was letztlich im Erlernen des Kaffeetrinkverhaltens resultiert. Im Zuge dessen werden auch einzelne Algorithmen der Instanzen und Kommandos kurz erläutert, um ein besseres Verständnis für die Funktionsweise der Anwendungen zu bekommen.

Der Workflow lässt sich in zwei Phasen unterteilen. In der ersten werden die einzelnen Komponenten konfiguriert und gestartet und die zweite beschreibt den eigentlichen Ablauf und das Zusammenwirken der Instanzen.

#### Setup

1. Blockchain
  - 1.1. erstellen & konfigurieren
  - 1.2. starten
2. Fileserver
  - 2.1. REST Api starten
  - 2.2. UDP Broadcast starten
3. Smart Contracts
  - 3.1. deploy Beveragelist Smart Contract und sende JSON-File mit ABI und Adresse an Fileserver
  - 3.2. deploy CoffeeCoin Smart Contract und sende JSON-File mit ABI und Adresse an Fileserver

### 3 System Architektur

#### 4. Learner

##### 4.1. Worker starten

##### 4.2. Watcher starten

#### 5. Tablet App

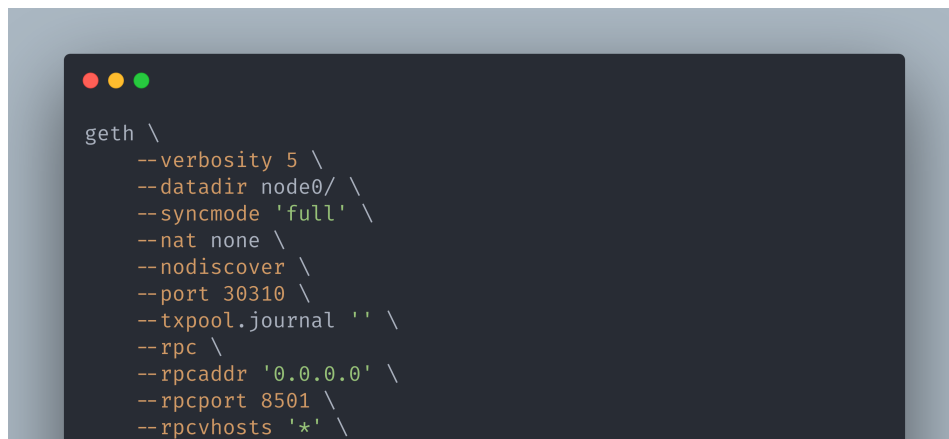
##### 5.1. installieren

##### 5.2. starten

Die Punkte 1. und 2. sowie 4. und 5. können auch parallel ausgeführt bzw. deren Reihenfolge vertauscht werden.

Im ersten Schritt muss die private Blockchain erstellt werden. Dabei müssen zuerst die benötigten Accounts generiert und daraufhin die Blockchain erzeugt werden. Sollte 1a) zu einem früheren Zeitpunkt bereits durchgeführt worden sein, kann dieser Punkt übersprungen und gleich mit 1b) begonnen werden. Sobald 1a) einmal durchgeführt wurde, kann die Blockchain gestartet werden.

Dies geschieht mit folgendem Befehl:

A screenshot of a terminal window with a dark background and light-colored text. The terminal shows the command to start a private Ethereum node using the 'geth' command. The command is split across multiple lines, with each line starting with a backslash to indicate it's a single command. The options include verbosity, data directory, sync mode, NAT, node discovery, port, transaction pool journal, RPC, RPC address, RPC port, and RPC hosts.

```
geth \  
  --verbosity 5 \  
  --datadir node0/ \  
  --syncmode 'full' \  
  --nat none \  
  --nodiscover \  
  --port 30310 \  
  --txpool.journal '' \  
  --rpc \  
  --rpcaddr '0.0.0.0' \  
  --rpcport 8501 \  
  --rpcvhosts '*' \  

```

```
--rpcapi 'personal,db,eth,net,web3,txpool,miner,debug' \
--ws \
--wsaddr '0.0.0.0' \
--wsport 8546 \
--wsorigins '*' \
--wsapi 'personal,db,eth,net,web3,txpool,miner,debug' \
--networkid 50 \
--gasprice '2000000000' \
--targetgaslimit '0x4c4b400000' \
--mine \
--etherbase '0xe8816898d851d5b61b7f950627d04d794c07ca37' \
--unlock '0x02e9f84165314bb8c255d8d3303b563b7375eb61, ...' \
--password=node0/password.txt
```

ABBILDUNG 3.2: geth Befehl zum starten der Blockchain

Dieser Befehl setzt zum einen weitere Konfigurationsparameter der Blockchain. So wird z.B. festgelegt unter welcher IP-Adresse und Port (*--rpcaddr*, *--rpcport*, *--wsaddr*, *--wsport*) die Blockchain erreichbar ist und welche Api-Befehle unter dieser Schnittstelle ausgeführt werden dürfen (*--rpcapi*, *--wsapi*). Zum anderen wird aber zugleich auch eine sogenannte *full node* gestartet (*--syncmode*), die durch das “Flag” *--mine* sofort zum “minen” beginnt. Desweiteren werden alle Accounts entsperrt die unter (*--unlock*) gelistet sind und deren Passwörter in der angegebenen Textdatei bei (*--password*) hinterlegt sind.

Damit eine *Node* Daten speichern kann, muss ein Verzeichnis angegeben werden (*--datadir*), in dem Dateien abgelegt werden können. Hier werden z.B. die Daten der Accounts oder auch die Textdatei mit den Passwörtern (*--password*) gespeichert.

Wurde die Blockchain in Betrieb genommen, wird im nächsten Schritt die REST Api und der UDP Broadcast des Fileservers gestartet. Daraufhin ist es möglich die beiden Smart Contracts zu deployen. Dabei wird jeweils für 3a) und 3b) das identische Bash-Skript mit unterschiedlichen Eingabeparametern ausgeführt. Dieses Skript liest zuerst die Datei des angegebenen Smart Contracts ein und erzeugt daraufhin die Binaries und die ABI, welche schlussendlich dazu verwendet werden ein Go Bindingsfile zu generieren.

### 3 System Architektur

Im Anschluss wird dann ein Go-Skript ausgeführt, welches auf Grundlage des Bindingsfiles den Smart Contract auf der Blockchain installiert und die zurückgelieferte Adresse und die bereits bekannte ABI ein JSON-File packt und an den Fileserver schickt.

Wurden die Smart-Contracts erfolgreich deployed, kann der Learner gestartet werden. Dieser “subscribed” sich im ersten Schritt auf den UDP Broadcast und extrahiert aus den Nachrichten die IP-Adresse der Blockchain und des Fileservers. Daraufhin werden sowohl der Worker als auch der Watcher in Form von “Goroutines” aktiviert. Der Worker iteriert über die Liste aller User und kontaktiert jeweils den Fileserver ob bereits gelernte Daten für diesen Usern vorhanden sind. Sollte das Fall sein, initialisiert er damit die Parameter des Q-Learning Algorithmus des Users.

Der Watcher schickt ebenfalls eine Anfrage an den Fileserver und bekommt als Antwort die Daten der Smart Contracts. Daraufhin kann er sich mit der Blockchain verbinden und sich auf die Events des Beveragelist Smart Contracts “subscriben”.

Abschließend wird die App auf dem Tablet installiert, sollte sich diese noch nicht auf dem Tablet befinden und daraufhin gestartet. Die Initialisierung erfolgt hierbei nach dem selben Prinzip wie beim Learner. Zuerst wird der UDP Broadcast nach der Server Adresse abgefragt, mit welcher anschließend der Request an den Fileserver geschickt wird, um die benötigten Smart Contract Daten zu bekommen. Welche im Anschluss dazu verwendet werden eine Verbindung zur Blockchain bzw. zu den Smart Contracts herzustellen.

Erfolgte eine fehlerlose Abarbeitung dieser Schritte, kann zum eigentlichen Workflow übergegangen werden.

### Workflow

1. App
  - 1.1. User wählt Getränk aus
  - 1.2. *call* CoffeeCoin
  - 1.3. *call* Beveragelist
2. Smart Contracts
  - 2.1. Beveragelist *triggers* Event
3. Learner
  - 3.1. Watcher:
    - 3.1.1. detektiert Event
    - 3.1.2. extrahiert Daten aus Event
    - 3.1.3. befüllt Q-Learning Algorithmus mit den Event-Daten (evaluate & predict)
  - 3.2. Worker (periodisch alle 3h)
    - 3.2.1. “triggers” Q-Learning Algorithmus (evaluate & predict)
    - 3.2.2. sendet gelernte Daten (vgl. Abbildung 3.3) an Fileserver

Diese Auflistung beschreibt einen synchronen, erfolgreichen Durchlauf der Systemarchitektur - die Asynchronität des Workers 3b) außer Acht gelassen. Alternative Abläufe sowie Zustände die aus Fehlern resultieren, werden bei den einzelnen Komponenten nochmals genauer betrachtet.

Der Workflow wird durch den User gestartet indem dieser auf Tablet ein Getränk auswählt und eine Transaktion auslöst. Zuerst wird dabei der CoffeeCoin Contract aufgerufen und das ausgewählte Getränk bezahlt. Nachdem

### 3 System Architektur

diese Transaktion erfolgreich bestätigt wurde, wird als nächstes der Beverage-list Contract ausgeführt. Die dabei aufgerufene Funktion des Smart Contracts verwendet die übergebenen Daten (Zeit, Getränk, Wochentag, Eth-Adresse) und löst damit ein Event aus.

Dieses Event wird vom Watcher detektiert und die Daten (Zeit, Getränk, Wochentag, Eth-Adresse) daraus extrahiert. Daraufhin wird die *Learn-Methode* des Q-Learning Algorithmus aufgerufen, bei der zuerst die vorherige “Prediction” evaluiert und basierend auf dem aktuellen Zustand eine neue “Prediction” gemacht wird.

Zu diesem synchronen Durchlauf führt der Worker am Ende jedes Timeslots (alle 3h) die *Learn-Methode* für jeden bekannten User aus. Das heißt es werden wie auch beim Watcher die “Predictions” des vorangegangenen Timeslots evaluiert, neue “Predictions” für den kommenden Timeslot erstellt und die gelernten Daten als Datei an den Fileserver gesendet.

### 3.1.3 Entwicklungsprozess

Abschließend wird der Prozess der Entwicklung geschildert, aus welchem schließlich die finale Version der Systemarchitektur resultierte.

Der Entwicklungsprozess beinhaltete mehrere Iterationen der einzelnen Komponenten bis hin zum derzeitigen Stand. Das Konzept sah primär die Entwicklung von drei dedizierten Software Anwendungen vor, welche aber im Zuge der Iterationen nochmal in kleinere Module aufgeteilt und ausgelagert wurden. Zudem wurden, um den Workflow und das Testen während der Entwicklungsphase zu erleichtern, Anwendungen entwickelt, welche während der Konzeption in der Art nicht vorgesehen waren, aber partiell Bestandteil der Systemarchitektur wurden.

So wurde mit zwei separaten Repos gestartet, einerseits für den Learning-Part, welcher anfänglich auch die Smart Contracts umfasste, und andererseits eines für die Tablet-App, welches bereits vor der eigentlichen Konzeption erstellt wurde, um in erster Linie bestehende Crossplattform Frameworks, auf Basis der Kompatibilität und Funktionstüchtigkeit mit Libraries, welche die Kommunikation mit der Blockchain ermöglichen, zu evaluieren.

Die Wahl fiel letztendlich auf React-Native, welches zwar nur bis zu einer bestimmten Versionsnummer der Web3.js Library von Ethereum vollends kompatibel ist und nur mit einem kleinen Workaround zum Laufen gebracht werden konnte. Jedoch im Vergleich zu anderen Frameworks (z.B. Nativescript) die beste Development-Experience (geringe Lernkurve, gute Dokumentation, CLI) bot und v.a. hinsichtlich der Requirements alle Aufgaben komplett erfüllen konnte, welche die anderen Frameworks in dieser Gänze nicht replizieren konnten.

Nachdem die erste rudimentäre Version der Tablet-App, welche lediglich eine funktionierende Kommunikation (read/write) mit einem bereits bestehenden Smart-Contract auf einer lokal gehosteten Blockchain bestätigte, erstellt wurde, kam im nächsten Schritt der Learning-Part zum Zuge.

In Anbetracht der kompletten Implementierung des Ethereum Protokolls in Golang und der Schwierigkeiten mit der Javascript Library Web3.js, v.a. im

### 3 System Architektur

Bezug auf das deployen der Smart-Contracts, aus einem vorangegangenen Projekt, fiel die Wahl für diese Instanz auf Golang.

Zuerst wurde der Q-Learning Algorithmus, welcher für das Erlernen des Kaffee Trinkverhalten zuständig ist, implementiert. Die Problematik bestand zum einen darin mit einer neuen Programmiersprache vertraut zu werden und zum anderen den Workflow hinsichtlich der Problemstellung und des daraus resultierenden Zustandsraums vollends abzubilden. Die Umsetzung des Algorithmus in der Programmiersprache ging relativ einfach von der Hand, was jedoch Probleme bereitete war die Simulation des Workflows, um die Algorithmus Parameter zu justieren und dessen Tauglichkeit bezüglich das Erlernen des Nutzerverhaltens zu testen.

Im Anschluss wurde ein erster Smart-Contract erstellt und via dem “go-ethereum” package deployed, woraus das erste Smart-Contract Bindingsfile resultierte, welches für die Kommunikation mit dem Smart Contract vonnöten ist. Da mit jedem neuem Deployment eines Smart Contracts eine neue Smart Contract Adresse und eine neue ABI hervorgeht, welche wiederum beide im Source Code für die Kommunikation mit dem Smart Contract, über alle Instanzen hinweg, die mit einem Smart Contract interagieren wollen, hinterlegt sein müssen, wurde ein kleiner HTTP-Fileserver entwickelt, auf dem diese Informationen gespeichert und gelesen werden können.

Bei jedem neuen Deployment werden daraufhin die Smart-Contract Adresse und die generierte ABI in ein JSON-File gepackt und an den Server geschickt. So konnte während der Entwicklung enorm an Zeit gespart werden, da sich sowohl die Learning-Instanz als auch die App, die benötigten Daten vom Server holen und somit ein ständiges “Hardcodieren” dieser Daten vermieden werden konnte.

Aus diesem Grund findet der Fileserver auch Einzug in die finale Systemarchitektur, da er als persistente Datenquelle eine enorme Erleichterung nicht nur im Entwicklungsprozess, sondern auch im “Live-System” darstellt.

Zudem wird der Fileserver auch für die Verwaltung der Algorithmus-Daten verwendet. Dabei wird bei jedem Worker-Durchlauf (vgl. Kap. 3.3.3) für jeden



Nutzer ein JSON-File erzeugt, welches folgende Key-Value-Pairs beinhaltet (vgl. Kap. Abbildung 3.3):

- qt: die aktuelle Q-Tabelle des Users
- ep: der aktuelle Epsilon-Wert
- negs: Anzahl der falschen Predictions in der aktuellen Woche
- Wk\_negs: Array von negs über alle Wochen hinweg

```

{
  "qt": [
    {
      "weekday": 1, "timeslot": 0, "drinkcount": 0,
      "coffeeCount": 0, "waterCount": 0, "mateCount": 0
    },
    {
      "weekday": 1, "timeslot": 1, "drinkcount": 0,
      "coffeeCount": 0, "waterCount": 0, "mateCount": 0
    },
    {
      "weekday": 1, "timeslot": 2, "drinkcount": 0,
      "coffeeCount": 0, "waterCount": 0, "mateCount": 0
    },
    {
      "weekday": 3, "timeslot": 0, "drinkcount": 0,
      "coffeeCount": 0, "waterCount": 0, "mateCount": 0
    },
    {
      "weekday": 3, "timeslot": 0, "drinkcount": 0,
      "coffeeCount": 0, "waterCount": 0, "mateCount": 0
    },
    {
      "weekday": 3, "timeslot": 0, "drinkcount": 0,
      "coffeeCount": 0, "waterCount": 0, "mateCount": 0
    },
    {
      "weekday": 3, "timeslot": 0, "drinkcount": 0,
      "coffeeCount": 0, "waterCount": 0, "mateCount": 0
    },
    {
      "weekday": 3, "timeslot": 0, "drinkcount": 0,
      "coffeeCount": 0, "waterCount": 0, "mateCount": 0
    },
    {
      "weekday": 3, "timeslot": 0, "drinkcount": 0,
      "coffeeCount": 0, "waterCount": 0, "mateCount": 0
    },
    {
      "weekday": 3, "timeslot": 0, "drinkcount": 0,
      "coffeeCount": 0, "waterCount": 0, "mateCount": 0
    }
  ],
  "ep": "0.988516",
  "negs": 8,
  "Wk_negs": [18, 17, 13, 15, 12, 9]
}

```

ABBILDUNG 3.3: 0x6ecbe1db9ef729cbe972c83fb886247691fb6beb-ql.json  
 Der Name des JSON-Files setzt sich aus der Ethereum-Adresse des Users und der Abkürzung "ql", welches für Q-Learning steht, zusammenhängen.

Der Vorteil liegt in der Möglichkeit den Learner jederzeit upzudaten ohne die gelernten Daten zu verlieren. Denn wird der Learner gestartet, holt sich dieser zuerst die Files vom Server, liest die Daten aus und initialisiert schon im Vorab die Q-Tabelle und das Epsilon eines jeden Users.

Sollte der Learner aus unbestimmten Gründen abstürzen, ist durch den eben beschriebenen Algorithmus die Erhaltung des Lernfortschrittes trotzdem sichergestellt.

Als letztes Modul wurde ein kleiner Node-Server entwickelt, dessen Aufgaben darin bestand die Smart-Contracts zu testen und als primitiver Ersatz

### 3 System Architektur

für die App zu fungieren. Hierbei erzeugte er in einem festgelegten Intervall (7sek) Events mit einem zufällig generierten Daten (User & Getränk) auf der Blockchain, um letztendlich die Event-Erkennung (“Watcher”) des “Learners” zu testen. Dabei wurde sowohl für den “Beverage-List Contract” als auch für den “CoffeeCoin-Contract” eine entsprechende Implementierung angefertigt. In diesen Fällen wurde das Q-Learning-Submodul des Learners gar nicht erst gestartet, da lediglich die Funktionsweise des Watchers getestet werden sollte. Besonders hier zeigte sich die Nützlichkeit des Fileservers, da gerade in der Entwicklungsphase die Smart-Contracts noch häufigen Änderungen unterlagen und dahingehend sehr oft neu deployed werden mussten, was ohne den Fileserver dazu geführt hätte die Smart Contract Daten bei jeder Iteration neu im Sourcecode zu hinterlegen.

Nach einer längeren Testphase, in der eine einwandfreie Kommunikation mit den beiden Smart Contracts attestiert werden konnte, wurde mit der eigentlichen Entwicklung der App begonnen, für jene auch Teile der Node-Server Implementierung übernommen werden konnten.

Schwierigkeiten traten dabei erst in der Testphase auf, in der festgestellt wurde, dass zu wenig Events vom Learner detektiert werden. Die Ursache dafür lag an der sehr alten Android Version des Tablets, die nicht ermöglichte eine direkte Verbindung zum Uni-Netzwerk herzustellen. Dies gelang nur mit einem Workaround, bei dem sich das Tablet mit einem öffentlichen Wlan-Netzwerk verband und sich daraufhin über eine VPN-Verbindung in das Uni-Netzwerk einwählen konnte.

Das führte jedoch dazu, dass die Verbindung zum Wlan-Netzwerk in unregelmäßigen Abständen abbrach und dadurch auch zur Blockchain. Da so ein unvorhergesehenes Verhalten wurde in der ersten Implementierung der App nicht vorgesehen war, musste dies in einem Update der App berücksichtigt werden (vgl. Kap. 3.4), sodass keine Daten verloren gingen, sollte die Verbindung abbrechen.

Schlussendlich waren es sechs dedizierte Software Anwendungen, welche jeweils in eigenen Git-Repositories gehostet werden. Dazu zählten:

- Learner

- Tablet-App
- Go Fileserver
- Smart Contracts: Beverage-List, CoffeeCoin
- Web3 Node-Server
- Dockerimage für die Blockchain

Das Dockerimage fand in der Hinsicht keine größere Erwähnung, da es nur zu Test- und Weiterbildungszwecken entwickelt wurde und auch keine Verwendung in der finalen Architektur fand.

## 3.2 Blockchain

Das folgende Kapitel erläutert im Detail den Setup der privaten Blockchain, sowie die beiden Smart Contracts welche ebenso ein Teil der Systemarchitektur darstellen.

### 3.2.1 Genesis Block

Der große Vorteil einer privaten (Ethereum) Blockchain gegenüber einer öffentlichen, ist die Möglichkeit die Blockchain nach den eigenen Vorstellungen und Anwendungszwecken zu konfigurieren. So können, wie auch bei einer öffentlichen Blockchain, Smart Contracts erstellt und Transaktionen durchgeführt werden, allerdings ohne dabei wirkliches Ether zu besitzen. Denn eine Besonderheit eines sogenannten “Testnet’s” ist das Erzeugen von “privatem” Ether, welcher Accounts zugeordnet und somit Transaktionen durchgeführt werden können.

Die Konfiguration dessen erfolgt durch ein sog. “genesis.json file” (3.5). Diese Datei ist die Grundlage für den *Genesis Block* der zu erstellenden Blockchain,

### 3 System Architektur

welcher der erste Block in der Kette ist und somit auch keinen Vorgänger besitzt.

Das in 3.5 abgebildete JSON Objekt zeigt die in der Systemarchitektur verwendete Datei einen solchen *Genesis Block* zu erzeugen. Nicht alle “properties” bedingen einer Erklärung, die essentiellen werden allerdings kurz erläutert:

A screenshot of a code editor with a dark background and light-colored text. The editor displays the content of a JSON file named 'genesis.json'. The JSON structure includes a 'config' object with fields like 'chainId', 'homesteadBlock', 'eip150Block', 'eip150Hash', 'eip155Block', and 'eip158Block'. It also includes fields for 'difficulty', 'nonce', 'timestamp', 'gasLimit', 'mixHash', 'coinbase', and 'alloc' (which contains two account entries with addresses and balances). Finally, there is a 'parentHash' field at the bottom. The code is syntax-highlighted, with strings in quotes and numbers in a different color.

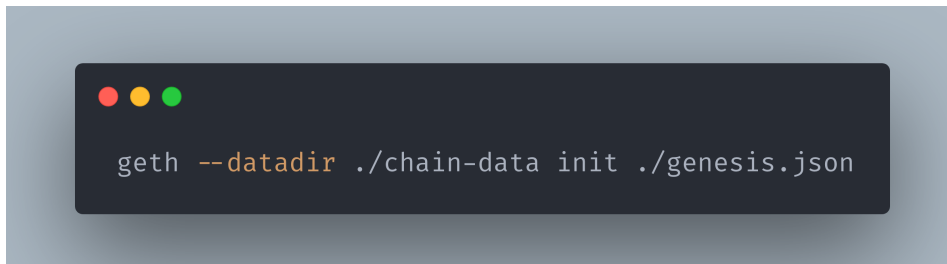
```
{
  "config": {
    "chainId": 50,
    "homesteadBlock": 1,
    "eip150Block": 2,
    "eip150Hash": "0x0000000000000000000000000000000000000000000000000000000000000000",
    "eip155Block": 3,
    "eip158Block": 3,
    ...
  },
  "difficulty": "0x1",
  "nonce": "0x0",
  "timestamp": "0x5af1ffac",
  "gasLimit": "0x4c4b40000",
  "mixHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "coinbase": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "alloc": {
    "0xe8816898d851d5b61b7f950627d04d794c07ca37": {
      "balance": "0x56BC75E2D63100000"
    },
    "0x5409ed021d9299bf6814279a6a1411a7e866a631": {
      "balance": "0x56BC75E2D63100000"
    },
    ...
  },
  "parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000"
}
```

ABBILDUNG 3.4: genesis.json  
JSON-File welches zur Erstellung des Genesis Blocks verwendet wurde.

- config: the config block defines the settings for our custom chain and has certain attributes to create a private blockchain
- chainId: ist eine einzigartige Id für die private Blockchain
- eip150Block/eip155Block/eip158Block: eip steht für “Ethereum Improvement Proposal”. Diese drei Community getriebenen “Proposals” beschreiben sog. “hard forks”, welche dafür sorgen sollten Fehler im Protokoll zu beheben.

- homesteadBlock: Homestead ist die zweite “major version” von Ethereum, welche einige Änderungen an dem Protokoll vornahm
- difficulty: beschreibt die Schwierigkeitsstufe für einen Miner einen validen Block zu finden. Das heißt je höher der Wert desto mehr Berechnungen müssen statisch durchgeführt werden und desto mehr Zeit wird benötigt, um eine Transaktion zu bestätigen. Im Falle eines Testnets ist es deshalb ratsam einen sehr niedrigen Wert zu wählen
- gasLimit: beschreibt das Limit für eine Transaktion wie viel an Gas verbraucht werden darf.
- alloc: hier können Accounts schon im Voraus mit “fake ether” befüllt werden.
- nonce: ist ein Zähler für die Anzahl der durchgeführten Transaktionen einer Adresse

Sobald die genesis.json Datei fertiggestellt ist, kann die Blockchain mit folgendem Befehl erstellt werden:

A screenshot of a terminal window with a dark background and light text. The window has three colored window control buttons (red, yellow, green) in the top-left corner. The command entered in the terminal is `geth --datadir ./chain-data init ./genesis.json`.

```
geth --datadir ./chain-data init ./genesis.json
```

Das Flag `--datadir` ist mit dem aus Abbildung 3.2 identisch. Das bedeutet das Verzeichnis welches hier im Befehl angegeben ist, muss beim Kommandozeilenbefehl in Abbildung 3.2 exakt gleich sein. Andernfalls ist es nicht möglich die Blockchain zu starten.

Nachdem der Setup der privaten Blockchain abgeschlossen ist, kann mit der Entwicklung eines Smart Contracts begonnen werden.

### 3 System Architektur

Die nächsten beiden Unterkapitel befassen sich mit den Smart Contracts, welche im Zuge dieser Arbeit entwickelt wurden.

#### 3.2.2 CoffeeCoin

Der Smart Contract “CoffeeCoin” stellt einen sogenannten ERC-20 Token mit gewissen Abwandlungen dar. Dabei ist ein Token im Grunde eine zusätzliche Währung zur eigentlichen Währung von Ethereum dem Ether. Das bedeutet Smart Contracts können somit eine eigenständige Währung abbilden. Für solche Smart Contracts gibt es mittlerweile einige Standardisierungen, die Vorschreiben welche Methoden und Datenstruktur ein Smart Contract zu implementieren hat. Der am weit verbreitetste Standard ist der ERC-20, bei welchem es folgende Funktionen und Events zu implementieren gilt:



```
contract ERC20Interface {
    function totalSupply() public view returns (uint);
    function balanceOf(address tokenOwner) public view returns (uint balance);
    function allowance(address tokenOwner, address spender) public view returns (uint
remaining);
    function transfer(address to, uint tokens) public returns (bool success);
    function approve(address spender, uint tokens) public returns (bool success);
    function transferFrom(address from, address to, uint tokens) public returns (bool success);

    event Transfer(address indexed from, address indexed to, uint tokens);
    event Approval(address indexed tokenOwner, address indexed spender, uint tokens);
}
```

ABBILDUNG 3.5: ERC-20 Interface

- totalSupply: Gesamtanzahl der existierenden Tokens
- balanceOf: Anzahl der Tokens eines bestimmten Users
- allowance: besagt wie viele Tokens eines bestimmten Users durch einen bestimmten “spender” abgehoben werden dürfen
- transfer: transferiert die Inputanzahl an Tokens des “senders” (Adresse welche diese Funktion aufgerufen hat) an die angegebene Adresse

- `approve`: Erlaubniserteilung an den “spender” die in *allowance* festgelegte Anzahl an Tokens abzubuchen
- `transferFrom`: transferiert die angegebene Anzahl von Tokens von der “from Adresse” zur “to Adresse”
- `event Transfer`: wird von den beiden *transfer* Funktionen ausgelöst
- `event Approval`: wird von der *approve* Funktion ausgelöst

Der eben erläuterte Token Standard wurde im Falle des CoffeeCoin Smart Contracts um zusätzliche Funktionen und Datenstrukturen erweitert. Diese Erweiterungen stellen eine gesonderte Abstraktionsebene nach außen hin da, um die Kommunikation seitens der Tablet-App zu erleichtern und einfacher zu gestalten.

So werden schon beim Deployment die einzelnen Preise der Getränke und die Adresse, zu jener die Tokens beim Bezahlen eines Getränks überwiesen werden, gesetzt.

Dies ermöglicht die Implementierung folgender Funktionen:



```

contract CoffeeCoinInterface {
    function payCoffee() public returns (bool success);
    function payWater() public returns (bool success);
    function payMate() public returns (bool success);
    ...
}

```

ABBILDUNG 3.6: CoffeeCoin Interface Auszug

Diese Funktionen bieten eine Abstraktionsebene zur ERC-20 Funktion *transferFrom*. Da bereits beim Deployment die Parameter für die Getränke und die Zieladresse gesetzt werden, benötigen diese Funktionen keine weiteren Daten von der Seiten der User (Tablet-App). Somit kann nach Auswahl des Getränks die entsprechende Funktion aufgerufen werden, ohne sich dabei mit den Details

### 3 System Architektur

der Transaktion beschäftigen zu müssen.

Desweiteren wird einem User, beim ersten Aufruf einer jener Funktionen (erste ausgelöste Transaktion des Users), eine festgelegte Anzahl an Tokens als “Startguthaben” überwiesen, sodass dieser stets über genügend Token verfügt und jederzeit Transaktionen durchführen kann.

Im Kontext der Problemstellung liegt die Zweckmäßigkeit des Smart Contracts grundsätzlich in der Bezahlung der Getränke in Form des Tokens. Dabei soll vor allem die Möglichkeit einer solchen Bezahlungsmethode aufgezeigt werden, weswegen die Transaktionen lediglich auf exemplarischer Ebene durchgeführt werden. Das bedeutet, den Usern wird, wie gerade beschrieben, eine nahezu unendliche Menge an Tokens zugewiesen ohne eine Gegenleistung zu fordern.

Der Sourcecode ist im Anhang unter REF SO UND SO zu finden.

#### 3.2.3 Beverage-List

Im Gegensatz zur CoffeeCoin basiert der Beveragelist Contract auf keinem festgelegten Standard, sondern ist in voller Gänze an die Problemstellung angepasst.

Dessen Zweck besteht im Grunde darin eine Getränkeliste abzubilden, in welcher jede Getränktransaktion eines Users vorzufinden ist. Dabei wird pro Transaktion nicht nur das Getränk, sondern auch das aktuelle Datum inklusive Uhrzeit und der aktuelle Wochentag, gespeichert. Diese Daten sollen es dem Learner schlussendlich ermöglichen das Kaffeetrinkverhalten des Users zu erlernen. Damit der Learner ohne großen Aufwand auf diese Informationen zugreifen kann, löst der Smart Contract, mit den eben genannten Daten beinhaltend, ein Event aus, sobald seine Methode “setDrinkData” aufgerufen wird. Diese Funktion hinterlegt die übergebenen Daten (Eth-Adress, Zeit, Wochentag, Getränk) in festgelegten Datenstruktur und löst zugleich das Event für den Learner aus.



Dieser Smart Contract umfasst noch weitere Funktionen, welche aber vor allem zu Testzwecken implementiert wurden und in der finalen Systemarchitektur keine Verwendung finden.

## **3.3 Learner**

### **3.3.1 Modellierung**

### **3.3.2 Q-Learning**

### **3.3.3 Lernprozess & Ablauf**

## **3.4 Tablet-App**

Das folgende Unterkapitel befasst sich mit dem Aufbau und der Funktionsweise der App und schildert zudem den verwendeten Algorithmus, welcher die Getränk- und Userdaten auf die Blockchain schreibt.

Die entwickelte App basiert auf dem Crossplattform Framework “React Native” [RN:], dieses erlaubt es mit einer einzigen Codebasis Apps für unterschiedliche Plattformen (z.B. iOS, Android) zu entwickeln. Der wesentliche Vorteil allerdings liegt in der Verfügbarkeit einer offiziellen Library, mit der es erst möglich ist eine Verbindung zur Blockchain bzw. den Smart Contracts herzustellen. Diese Library (Web3.js) wurde von Ethereum dafür entwickelt, um sog. DApp’s (“decentralized apps”) [DAp] auf Basis von Javascript erstellen zu können.

### 3 System Architektur

So ist die Entwicklung einer DApp in einer nativen Programmiersprache (Java/Kotlin/Swift) bisher nur mit “third-party libraries” möglich, weswegen die Umsetzung letztlich mit ReactNative erfolgte.

#### 3.4.1 Interface

Eine essentielle Eigenschaft von ReactNative ist der komponentenbasierte Ansatz. Dabei setzt sich eine App aus vielen einzelnen Komponenten zusammen, welche jeweils einen dedizierten Aufgabenbereich abdecken.

Im Falle der entwickelten App existieren jeweils zwei “page components”, die wiederum mehrere kleine Komponenten in sich vereinen. Da es aber den Rahmen dieser Arbeit sprengen würde auf jede einzelne Komponente und deren Funktionsweise einzugehen, werden nur die Hauptkomponenten anhand ihrer Funktion und Bedienung geschildert.

Wird die App gestartet und es besteht eine Verbindung zum Internet bzw. zur Blockchain, so findet der User folgende Startseite vor:

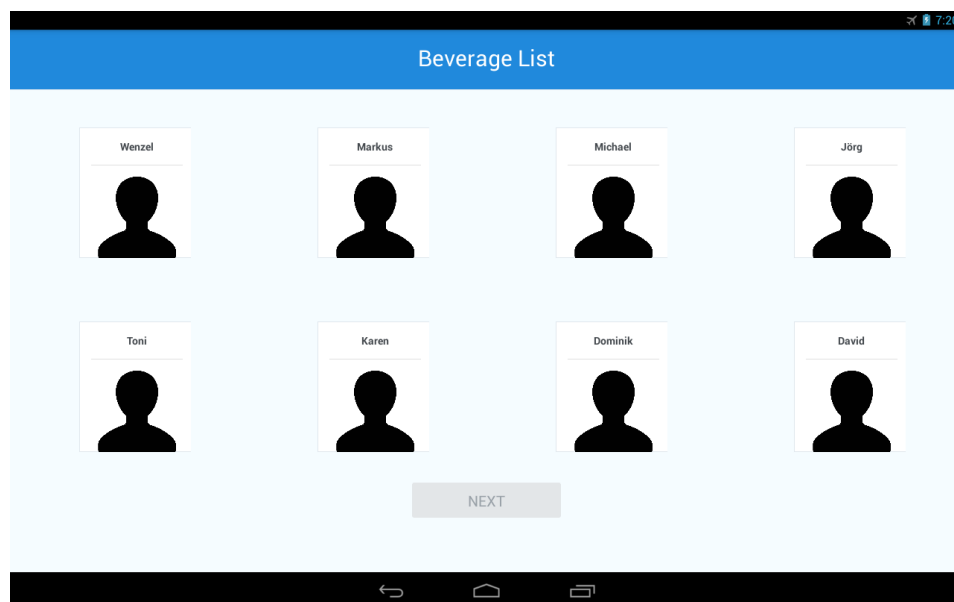


ABBILDUNG 3.7: Mitarbeiter Page: kein Mitarbeiter ausgewählt

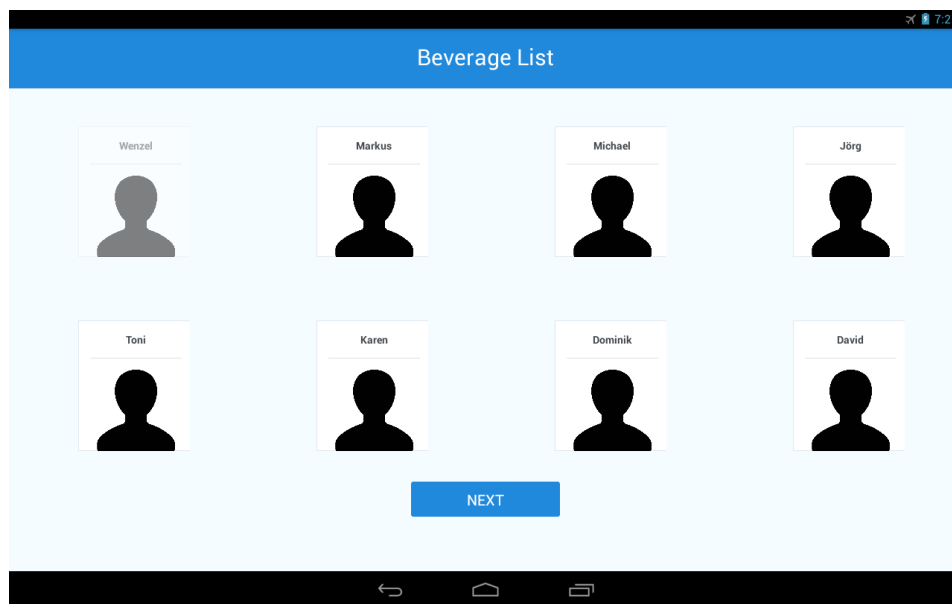


ABBILDUNG 3.8: Mitarbeiter Page: Mitarbeiter ausgewählt

Hier kann der User seinen Avater selektieren und deselektieren. Ist ein Avater ausgewählt wird der “NEXT” Button aktiviert (vgl. Abbildung 3.8) und durch dessen Betätigung gelangt der User zur nächsten Seite:

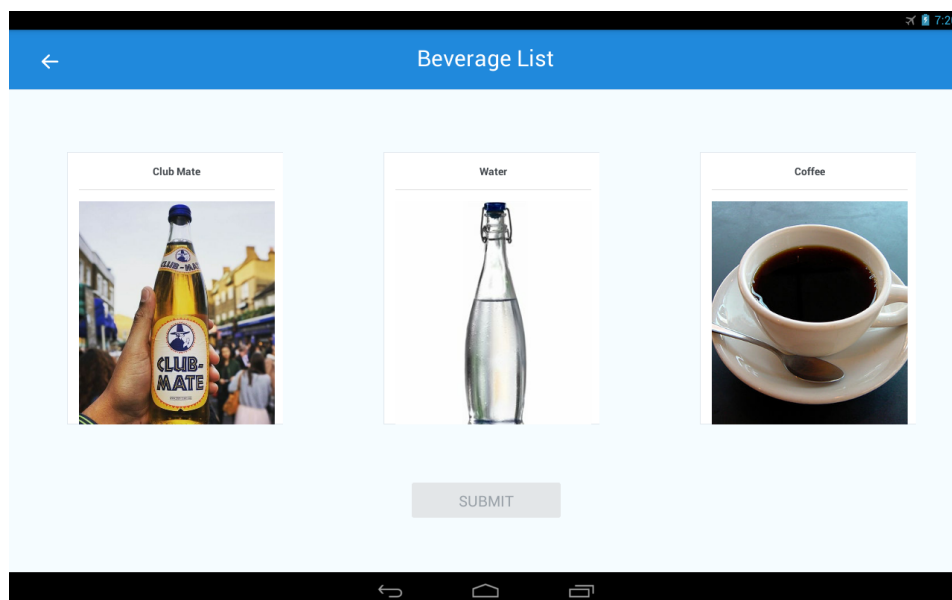


ABBILDUNG 3.9: Drinks Page: kein Getränk ausgewählt

### 3 System Architektur

Hier besteht eine Auswahl aus folgenden Getränken: Club Mate, Wasser und Kaffee. Ausgewählt kann jedoch immer nur eines werden (vgl. Abbildung 3.12). Das Prinzip der Selektion und Deselektion ist identisch mit dem der vorherigen Seite. So wird der “SUBMIT” Button aktiv, sobald ein Getränk ausgewählt ist und inaktiv wenn das Getränk wieder deselektiert wird (vgl. Abbildung 3.9).

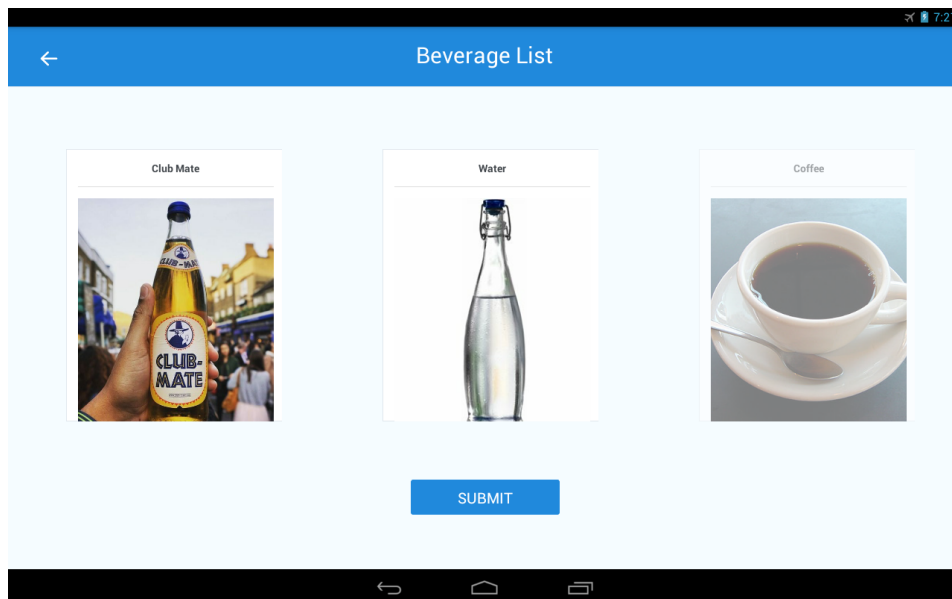


ABBILDUNG 3.10: Drinks Page: Getränk ausgewählt

Wird schließlich der “SUBMIT” Button gedrückt und die Transaktion als erfolgreich bestätigt erscheint folgendes Overlay:

Nach 4 Sekunden wird dieses Overlay wieder ausgeblendet und automatisch zur Startseite (vgl. Abbildung 3.7) navigiert, wodurch der Workflow von neuem startet.

Aufgrund der Tatsache, dass die Verbindung zum Uni-Netzwerk (eduroam) nur über einen Workaround hergestellt werden konnte, bei dem die Verbindung zum Netzwerk trotzdem nach einer unbestimmten Zeit immer wieder abgebrochen ist, wurde eine weitere Komponente entwickelt.

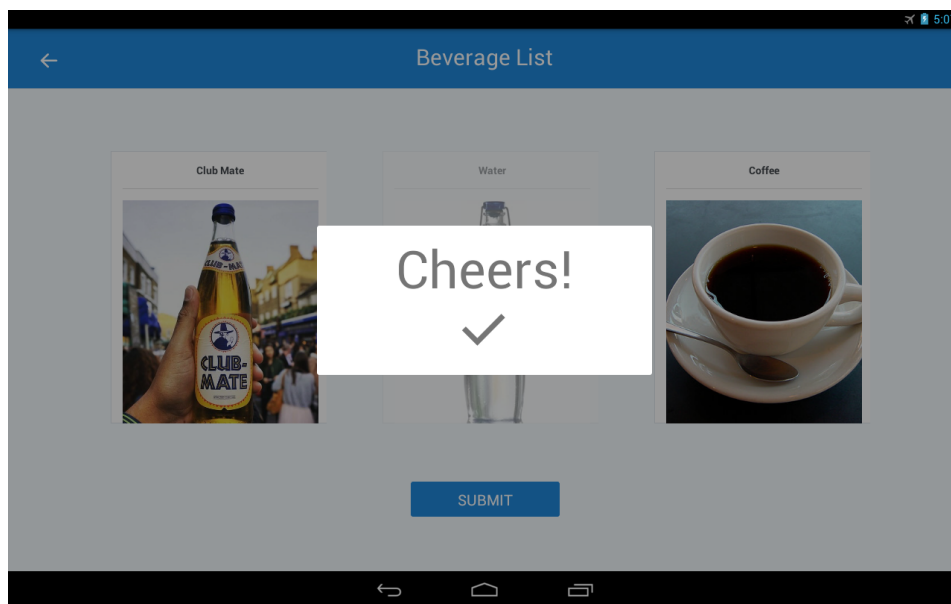


ABBILDUNG 3.11

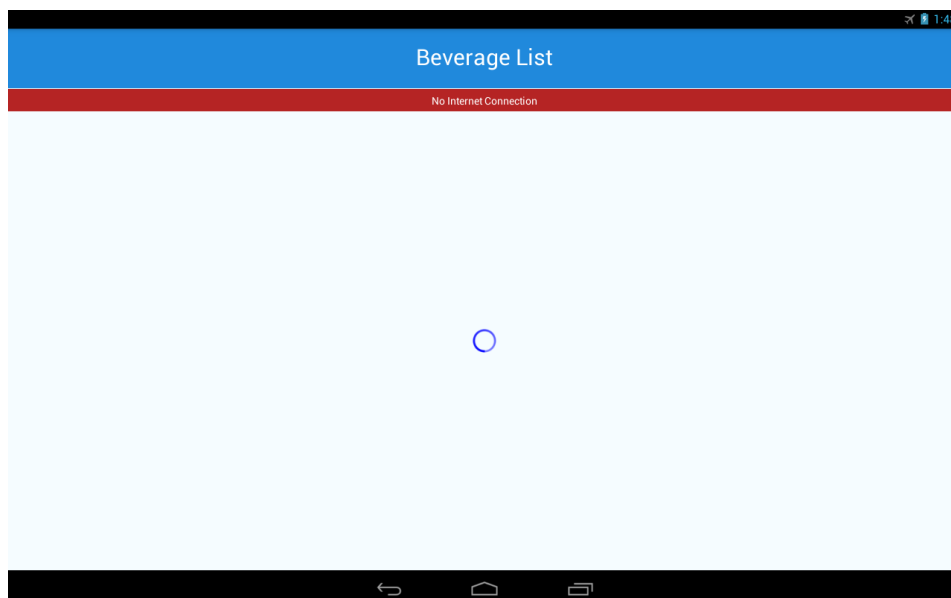


ABBILDUNG 3.12

Diese wird sofort eingeblendet sobald die Verbindung zum Internet unterbrochen ist, allerdings nur wenn sich der User auf der Startseite befindet. Wird die Seite mit den Getränken angezeigt, so wird dem User die Möglichkeit gegeben seine Transaktion abzuschließen. Da in diesem Moment jedoch keine Verbin-

dung zur Blockchain hergestellt werden kann, werden die Transaktionsdaten zwischengespeichert und diese durchgeführt sobald wieder eine Verbindung zum Netzwerk besteht (vgl. Kap. 3.4.2). Die Loading Animation sowie das “No Internet Connection” Label werden wieder ausgeblendet sobald die App eine erneute Verbindung zum Uni-Netzwerk detektiert.

#### 3.4.2 Internal Workflow

Basierend auf der Problematik eines unerwarteten Verbindungsabbruch und dem einhergehenden Verlust von essentiellen Lerndaten, wird im folgenden der implementierte Algorithmus geschildert, der dieser Komplikation entgegenwirkt.

1. Initialisierungsphase:
  - 1.1. “GET” Smart Contract Daten von Fileserver
  - 1.2. Initialisierung Web3.js
  - 1.3. Wiederhole für jedes Transaktionsfile:
    - 1.3.1. Transaktion durchführen
    - 1.3.2. bei “Success” Transaktionsdatei löschen
2. Warten auf Usereingabe:
  - 2.1. User ausgewählt → Ethereum-Adresse
  - 2.2. Getränk ausgewählt → Getränk
  - 2.3. Submit → (Getränk & Ethereum-Adresse)
3. Wiederhole für jedes User-Transaktionsfile:
  - 3.1. Transaktion durchführen
  - 3.2. bei “Success” Transaktionsdatei löschen

#### 4. Generierung Transaktionsdaten:

- Gas Estimate
- Datum (inkl. Zeit)
- Wochentag
- Ethereum-Adresse & Getränk aus 2.3

#### 5. Transaktion starten:

##### 5.1. Erstellung der Transaktionsdatei

##### 5.2. Transaktion durchführen

##### 5.3. bei "Success" Transaktionsdatei löschen

##### 5.4. gehe zu 1.

Der interne Workflow beginnt mit der Initialisierungsphase, dabei werden zuerst die Smart Contract Files vom Fileserver runtergeladen und mit den beinhaltenden Daten das Web3.js Modul initialisiert. Damit steht Verbindung und die Kommunikation mit den Smart Contracts und es wird im Anschluss über alle vorhandenen Transaktion Files iteriert. Dabei wird zuerst der CoffeeCoin Contract und dann der Beveragelist Contract aufgerufen. Wenn beide ihre Transaktionen bestätigt haben, wird die Datei gelöscht, andernfalls bleibt diese bestehen. Dies geschieht noch bevor der User das Interface zu Gesicht bekommt.

Nachdem 1. abgeschlossen ist wird auf die Eingabe des Users gewartet. Wählt dieser einen Avater aus klickt "NEXT" wird seine hinterlegte Ethereum-Adresse temporär gespeichert. Wird dann im Anschluss ein Getränk selektiert und "SUBMIT" gedrückt, werden die Ethereum-Adresse und das Getränk an das interne Blockchain-Modul weitergereicht und die Transaktion gestartet.

Dabei wird erneut über die Transaktionsdaten iteriert, allerdings nur über die des Users. Damit soll stets die richtige Reihenfolge der getrunkenen Getränke

### 3 System Architektur

sichergestellt werden, da dies ansonsten auf Seiten des Q-Learning Algorithmus zu falschen Lerneffekten führen würde.

Daraufhin kann mit der Generierung der fehlenden Transaktionsdaten angefangen werden. Es wird zuerst ein “Gas Estimate” für beide Smart Contract Transaktionen durchgeführt. Ein “Gas Estimate” ist, wie der Name bereits impliziert, eine grobe Schätzung wie viel Gas beim Funktionsaufruf eines Smart Contracts benötigt wird. Abschließend werden das Datum (inkl. Zeit in Sekunden) und der Wochentag ermittelt und alle benötigten Daten an die jeweiligen Smart Contract Funktionsaufrufe übergeben. Im Falle des Smart Contracts Beveragelist: *Gas-Estimate*, *Datum*, *Wochentag*, *Ethereum-Adresse*, *Getränk*. Bei CoffeeCoin wird anhand des Getränks die entsprechende Smart Contract Funktion ausgewählt und lediglich das *Gas-Estimate* und die *Ethereum-Adresse* des Users übergeben.

Bevor jedoch beide Transaktionen durchgeführt werden, wird ein Transaktionsfile mit dem Namen “«ethereum-adresse»-«datum».json” und den Daten *Datum*, *Wochentag*, *Ethereum-Adresse*, *Getränk* gespeichert. Sogleich werden beide Funktionsaufrufe getätigt und bei erfolgreicher Bestätigung beider Transaktionen wird das gerade erstellte File wieder gelöscht. Somit soll sichergestellt werden, dass keine Transaktionen aufgrund von Verbindungsabbrüchen verloren gehen.

Anschließend erfolgt eine “Pseudo-Reload” der App und der Ablauf beginnt wieder bei 1.



## **4 Studie**

### **4.1 Testphase**

### **4.2 Evaluation**



# **5 Zusammenfassung**

## **5.1 Weiterführende Forschungsfragen**

## **5.2 Ausblick**

# Literaturverzeichnis

[DAp] Dapp.

[Dav15] David Tuesta. Smart contracts: the ultimate automation of trust?, 2015-10-15.

[Rap09] Raphael Honig. So lange dauert mining bei bitcoins | kryptopedia, 2018-04-09.

[RN:] React native.

[Wika] Wikipedia. Application programming interface - wikipedia.

[Wikb] Wikipedia. Cross-platform software - wikipedia.

[Wikc] Wikipedia. Crud - wikipedia.

[Wikd] Wikipedia. Representational state transfer - wikipedia.

[Wike] Wikipedia. Single point of failure - wikipedia.

[Wikf] Wikipedia. User experience - wikipedia.



# Abbildungsverzeichnis

3.1	Systemarchitektur . . . . .	6
3.2	geth Befehl zum starten der Blockchain . . . . .	13
3.3	0x6ecbe1db9ef729cbe972c83fb886247691fb6beb-ql.json Der Name des JSON-Files setzt sich aus der Ethereum-Adresse des Users und der Abkürzung "ql", welches für Q-Learning steht, zusammenhängen. . . . .	19
3.4	genesis.json JSON-File welches zur Erstellung des Genesis Blocks verwendet wurde. . . . .	22
3.5	ERC-20 Interface . . . . .	24
3.6	CoffeeCoin Interface Auszug . . . . .	25
3.7	Mitarbeiter Page: kein Mitarbeiter ausgewählt . . . . .	28
3.8	Mitarbeiter Page: Mitarbeiter ausgewählt . . . . .	29
3.9	Drinks Page: kein Getränk ausgewählt . . . . .	29
3.10	Drinks Page: Getränk ausgewählt . . . . .	30
3.11	. . . . .	31
3.12	. . . . .	31



# **Tabellenverzeichnis**





# **A Anhang A**



## **B Anhang B**