

# **Hochschule Darmstadt**

– Fachbereich Informatik–

## **Synergie von DLT und IOT: Anforderungsanalyse und praktische Verprobung**

Abschlussarbeit zur Erlangung des akademischen Grades

Master of Science (M.Sc.)

vorgelegt von

**Sebastian Kanz**

Matrikelnummer: 735176

Referent : Prof. Dr. Michael Braun

Korreferent : Prof. Dr. Martin Stiernerling



## ERKLÄRUNG

---

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

*Darmstadt, 15. Oktober 2019*

---

Sebastian Kanz

## ABSTRACT

---

The internet of things (IOT) as an emerging technology enables a fully-automated machine-to-machine (M2M) communication to make our everyday lives easier: smart homes, the vision of smart cities or latency sensible connected cars are only some examples of possible research areas in this business. A huge amount of data needs to be processed, transmitted and stored. The presentation of the next generation mobile standard 5G and high-performance devices enable the transmitting and processing. To securely store the data a suitable solution must be found.

Besides that the distributed ledger technology (DLT) earns high attention due to the newly developed use cases which benefit from the trustless and decentralized properties of DLT.

It is obvious that an analyzation of those two relatively new technologies might result in a benefit for IOT and DLT. There could be a synergy so that both technologies could complement each other.

This work evaluates a selection of well-established DLTs for their suitability in the context of a representative IOT use case. Therefore the requirements for the use case are analyzed and applied to the DLTs. In a next step the DLTs are evaluated and ranked by the fulfillment of the requirements. A prototypical implementation of the IOT use case with the best suitable DLT is created. For a real-world adaption the use case is created by different IOT devices and sensors that transmit data over the DLT to other devices. Load-tests are used to check the actual performance.

As a result this work presents a comprehensible list of DLTs with an evaluation and ranking for their IOT-suitability in the M2M-area. On top that the feasibility of IOT use cases with DLT technology is shown.

## ZUSAMMENFASSUNG

---

Das Internet der Dinge (engl. IOT; Internet of Things) erhält immer mehr Einzug in das tägliche Leben. Smart-Home Lösungen, vernetzte und latenzempfindliche Connected-Cars oder die Vision einer Smart-City prägen die Forschungsarbeiten in den jeweiligen Bereichen. Ziel ist eine vollautomatische Machine-to-Machine (M2M) Abwicklung von Prozessen, um unseren Alltag zu automatisieren und zu vereinfachen. Dabei fallen eine Menge Daten an, die verarbeitet, übertragen und gespeichert werden müssen. Mit der Einführung des neuen Mobilfunk-Standards 5G und immer leistungsfähigeren Endgeräten sind Übertragung und Verarbeitung der Daten weitestgehend gesichert; bleibt die Frage offen, wo diese großen Datenmengen gespeichert und weiterprozessiert werden. Daneben erfreut sich das Thema Distributed-Ledger-Technology (DLT) immer größerer Beliebtheit: Es werden täglich neue Anwendungsfälle gefunden, die durch die verteilte Infrastruktur, der Trustless-Eigenschaft und der Dezentralität profitieren. Es bietet sich eine Untersuchung an, um zu überprüfen, inwieweit diese beiden noch recht jungen Technologien IOT und DLT Synergien besitzen und sich gegebenenfalls gegenseitig ergänzen können. Die vorliegende Masterarbeit evaluiert eine Auswahl etablierter DLTs anhand ihrer Tauglichkeit für den Einsatz im IOT-Umfeld mit Fokus auf den M2M-Bereich. Dazu wird zunächst ein IOT-Anwendungsfall erstellt, der stellvertretend für den M2M-Bereich für die weiteren Analysen verwendet wird. Anschließend werden konkrete Anforderungen aus verschiedenen Bereichen Infrastruktur, IT-Security, Performance und weiteren aufgestellt, die eine DLT erfüllen muss, um den Anforderungen des beispielhaften Anwendungsfalls gerecht zu werden. Die erstellten Kriterien werden auf eine Auswahl von DLT-Implementierungen angewandt, evaluiert und bewertet. Mit der am besten geeigneten DLT wird eine prototypische Implementierung des Anwendungsfalls vorgenommen, um die Ergebnisse aus der Anforderungsevaluierung zu überprüfen. Um den Use-Case möglichst realistisch zu simulieren werden Daten aus verschiedenen IOT-Sensoren an die DLT übermittelt und eine M2M-Kommunikation zwischen IOT-Devices via DLT erstellt. Anschließende Load-Tests geben detaillierte Informationen über die Performance. Das Ergebnis ist eine strukturierte und nachvollziehbare Bewertung mehrerer, am Markt etablierter DLTs, inwieweit diese für DLT-sinnvolle IOT-Anwendungsfälle im M2M-Umfeld geeignet sind, sowie ein DLT-basierter Prototyp angelehnt an einen realen Use-Case, der beispielhaft als Nachweis der erarbeiteten Bewertung dient.

# INHALTSVERZEICHNIS

---

## I THESIS

1	EINLEITUNG	2
1.1	Motivation & Problemstellung	2
1.2	Zielsetzung & Zielgruppe	3
1.3	Methodik & Aufbau dieser Arbeit	3
2	THEORETISCHE GRUNDLAGEN	5
2.1	Distributed Ledger Technologies	5
2.1.1	Konsensalgorithmen	7
2.1.2	Smart-Contracts	8
2.1.3	Dezentrale Apps (DApps)	9
2.1.4	Klassifizierung von Blockchains	10
2.1.5	Wallets	11
2.1.6	Dezentrale Identitäten	12
2.1.7	State-Channel	13
2.1.8	Blockchain als Kommunikationsprotokoll	15
2.2	Internet of Things	18
2.2.1	Digitaler Zwilling	20
3	VERWANDTE FORSCHUNGSARBEITEN	21
4	ANWENDUNGSFALL: VERMIETUNG VON HAUSHALTSGERÄTEN NACH DEM PAY-AS-YOU-USE PRINZIP	23
4.1	Beschreibung	23
4.2	Technische Lösungsskizze	25
4.2.1	Endgeräte	25
4.2.2	Verträge	26
4.2.3	Benutzerschnittstelle	27
4.2.4	Backend	28
5	ANFORDERUNGEN	29
5.1	Standards und Normen	29
5.2	Ableitung eines Klassifizierungsmodells	31
5.3	Anforderungsanalyse	33
5.4	Anforderungsevaluierung	36
5.4.1	System Anforderungen	37
5.4.2	Software Anforderungen	38
5.5	Anforderungstransfer auf DLT	45
6	AUSWAHL RELEVANTER DLTS	47
6.1	Vorgehen	47
6.2	Marktübersicht DLTS	48
6.3	Anforderungserfüllung	48
6.4	Bewertung, Ranking & Auswahl	50
7	UMSETZUNG	52
7.1	Auswahl der Anwendungsanforderungen	52

7.1.1	Architektur und Aufbau . . . . .	53
7.1.2	Deployment . . . . .	54
7.2	Testaufbau . . . . .	54
7.3	Datenschutz & Privatsphäre . . . . .	55
8	ERGEBNISSE & FAZIT	56
9	DISKUSSION	57
9.1	Wiederaufnahme These Teil 1: Eignung für IOT? . . . . .	57
9.2	Wiederaufnahme These Teil 2: Technische Anforderungen immer gleich? . . . . .	57
10	AUSBLICK	58
 <b>II APPENDIX</b>		
A	APPENDIX: ANFORDERUNGEN	60
B	APPENDIX: CODE	64
 LITERATUR		
		84

## ABBILDUNGSVERZEICHNIS

---

Abbildung 2.1	Bestandteile von DLTs . . . . .	6
Abbildung 2.2	Aufbau von DApps . . . . .	10
Abbildung 2.3	Taxonomie von Blockchains (vgl. [9]) . . . . .	11
Abbildung 2.4	Die Blockchain-Layer im Konext des OSI-Modells . . .	17
Abbildung 2.5	Übersicht IOT im Kontext von DLTs . . . . .	19
Abbildung 4.1	Grafische Veranschaulichung des Anwendungsfalls . .	24
Abbildung 4.2	Grober Ablauf des Anwendungsfalls . . . . .	26
Abbildung 4.3	Aufbau und Bestandteile eines Endgeräts . . . . .	27
Abbildung 5.1	Einordnung der Begriffe und Zusammenhänge unter- schiedlicher Normen und Standards . . . . .	30
Abbildung 5.2	Anforderungen werden nach zwei verschiedenen An- sätzen gruppiert. . . . .	31
Abbildung 5.3	Anforderungsklassifizierung als kombiniertes Modell aus [1], [16] bzw. [15] und [14] . . . . .	33
Abbildung 7.1	Architektur als UML-Komponentendiagramm . . . . .	54
Abbildung 7.2	Testabdeckung der Solidity Smart-Contracts . . . . .	55



## TABELLENVERZEICHNIS

---

Tabelle 5.1	DLT-relevante Anforderungen . . . . .	44
Tabelle 6.1	Erstauswahl von DLT-Lösungen . . . . .	49
Tabelle 6.2	Erfüllung der DLT-relevanten Anforderungen . . . . .	50
Tabelle 7.1	Adressen der Smart-Contracts auf dem öffentlichen Kovan Testnet . . . . .	54

## LISTINGS

---

Listing 2.1	Beispiel einer DID . . . . .	12
Listing 2.2	Beispiel eines DID-Dokuments . . . . .	12
gfx/code/RentalProvider.sol . . . . .		64
gfx/code/PaymentProvider.sol . . . . .		74
gfx/code/IdentityProvider.sol . . . . .		78

## ABKÜRZUNGSVERZEICHNIS

---

API	Application Programming Interface
ARP	Address Resolution Protocol
BABOK	Business Analysis Body of Knowledge
CIOT	Consumer IOT
DApp	Distributed Application
DID	Decentralized Identifier
DLT	Distributed Ledger Technologies
DoD	Definition of Done
DoR	Definition of Ready
EVM	Ethereum Virtual Machine
IEEE	Institute of Electrical and Electronics Engineers
IIBA	International Institute of Business Analysis
IIOT	Industrial IOT
IOT	Internet of Things
ISO	International Organization for Standardization
IPFS	Inter-Planetary File System
M2M	Machine-to-Machine
OSI	Open Systems Interconnection
OSI	Open Systems Interconnection
P2P	Peer-to-Peer
PMBOK	Project Management Body of Knowledge
PMI	Project Management Institute
PoC	Proof-of-Concept
SEBOK	System Engineering Body of Knowledge
SPoF	Single-Point-of-Failure
SWEBOK	Software Engineering Body of Knowledge

TPS	Transactions per Second
UI	User-Interface
UX	User-Experience
VC	Verifiable Credential
W <sub>3</sub> C	World Wide Web Consortium

Teil I

THESIS

## EINLEITUNG

---

Die Blockchain ist eine Technologie, deren grundlegende Ansätze keineswegs neu sind: Vor über 1500 Jahren suchten die Einwohner der Insel Yap im Westpazifik östlich der Philippinen eine Lösung für ihr Zahlungsmittel-Problem. Ihre Währung - Rai - waren großen runde Steintafeln, die teils mehrere Tonnen wogen und als alltägliches Zahlungsmittel ungeeignet waren. Ihre Lösung: Die Dorfältesten merkten sich, wem welcher Rai-Stein gehörte. Fand eine Transaktion statt, so wurde diese den Dorfältesten mitgeteilt. Das Wissen, wem welcher Stein gehörte und wer von wem etwas gekauft hatte, war über mehrere Köpfe verteilt; die dezentrale Informationsspeicherung war geboren [11].

1991 beschrieben die Forscher Stuart Haber und W. Scott Stornetta die Idee hinter der Blockchain-Technologie [4], 2008 veröffentlichte Satoshi Nakamoto das Whitepaper zu Bitcoin, dicht gefolgt von Ethereum, welches 2013 von Vitalik Buterin erstmals vorgestellt wurde. Und wie sieht es heute aus? Mittlerweile ist ein ganzes Ökosystem an Blockchain- und Distributed Ledger Technologies (DLT)-Lösungen entstanden: Es existieren digitale Vermögenswerte wie Bitcoin, verteilte Anwendungen auf Basis des Blockchain-Protokolls wie Distributed Application (DApp)s auf Ethereum und digitale Zahlungsmittel von fast über 5000 gelisteten Kryptowährungen.

### 1.1 MOTIVATION & PROBLEMSTELLUNG

Das Telekommunikationsunternehmen Cisco prognostiziert, dass bis zum Jahr 2030 mehr als 500 Milliarden mit dem Internet verbundene Internet of Things (IoT)-Geräte in verschiedenen Bereichen unseres alltäglichen Lebens Einzug erhalten haben werden [8]. Vernetzte Dinge unseres Alltags wie Kühlschränke, Kaffeemaschinen, die automatisierte Supply-Chain aus dem Business-Umfeld oder eine Smart-City sind nur einige wenige Beispiele dieses Geschäftsfeldes. Das Konzept von IoT ist nach wie vor sehr theoretisch, obwohl bereits einige Anwendungsfälle erarbeitet wurden. Um das große Potential von IoT vollumfänglich nutzbar zu machen und entsprechende Visionen umzusetzen, muss eine passende IT-Lösung für den entsprechenden Anwendungsfall bereitgestellt werden. Viele verschiedene Hersteller und Service-Provider benötigen eine einheitliche Plattform, auf der sie ihre IoT-Geräte, Services, Geschäftslogiken und Kunden miteinander vernetzen und ein sicheres Bezahlssystem integrieren können. Es stellt sich die Frage, ob und inwiefern die zwei innovativen Technologien DLT und IoT voneinander profitieren können und ob sich DLT als skalierende, performante und sichere Technologie für IoT-Anwendungsfälle eignet.

## 1.2 ZIELSETZUNG & ZIELGRUPPE

Das Ziel dieser Arbeit ist die Untersuchung und prototypische Verprobung der folgenden These: 'DLT eignet sich als Technologie für IOT und die technischen / nicht-funktionalen Anforderungen sind für alle Anwendungsfälle gleich'. Es wird gezeigt, inwieweit sich die Technologie DLT für IOT-Anwendungsfälle eignet, welche Anforderungen dafür erfüllt sein müssen, und welche Implementierung für die Umsetzung in Frage kommt. Dazu wird im Verlauf der Arbeit das zu lösende Problem genauer spezifiziert und herausgearbeitet: Nachdem ein konkreter, stellvertretender IOT-Anwendungsfall vorgestellt wurde und alle DLT-relevanten Anforderungen ermittelt und evaluiert sind, ergibt sich das konkrete Problem als die Umsetzung eines speziellen IOT-Anwendungsfalls mit einer ausgewählten DLT-Lösung. Der explizite Lösungsraum, also welche Anforderungen umgesetzt werden, wird zuvor genau beschrieben. Das Problem gilt als gelöst, sobald die zuvor abgeleiteten Anforderungen mit dem Proof-of-Concept (PoC) erfüllt werden können. Abschließend wird die eingangs formulierte These diskutiert und evaluiert.

Diese Arbeit richtet sich an IT-Spezialisten aus dem Umfeld DLT und IOT, die sich über die Synergie beider Konzepte informieren, sowie Fachleuten aus der Industrie, die entsprechende IOT-Anwendungsfälle ausarbeiten möchten. Ein solides Grundverständnis für die grundlegenden Konzepte und Wordings wird an dieser Stelle vorausgesetzt; auf entsprechende Grundlagenliteratur wird gegebenenfalls verwiesen. In Kapitel 2 wird das benötigte Basiswissen vermittelt, welches für das Verständnis dieser Arbeit und der Zusammenhang von Nöten ist.

## 1.3 METHODIK & AUFBAU DIESER ARBEIT

In dieser Arbeit werden die Themenbereiche 'DLT' und 'IOT' vorgestellt, klassifiziert und 'DLT' in das Open Systems Interconnection (OSI) Referenzmodell eingeordnet. Die Synergie beider Bereiche wird herausgearbeitet und es wird dem Leser vorgestellt, wie diese Technologien voneinander profitieren können (Kapitel 2). Zur entsprechenden Einordnung dieser Arbeit werden verwandte Forschungsarbeiten vorgestellt (Kapitel 3). Ein beispielhafter IOT-Anwendungsfall wird entwickelt (Kapitel 4) und eine detaillierte Auflistung aller Anforderungen erarbeitet (Kapitel 5). Im nächsten Schritt werden die die ermittelten Anforderungen schrittweise auf eine Untermenge von fundamentalen Anforderungen reduziert, die relevant für IOT in Verbindung mit DLT sind. Mehrere am Markt etablierte DLTs werden anschließend vorgestellt und auf Basis dieser Anforderungsuntermenge evaluiert (Kapitel 6). Es wird geprüft, ob und inwieweit sie sich als Lösung für den IOT-Anwendungsfall qualifizieren. Die vielversprechendste Lösung wird in einem PoC prototypisch umgesetzt (Kapitel 7), um die Anforderungsliste zu evaluieren. Es wird gezeigt, dass die gewählte DLT zielbringend eingesetzt werden kann (8). Abschließend wird diskutiert, ob und warum die nicht-funktionalen Anforderungen für DLT-geeignete IOT-Anwendungsfälle - unabhängig vom tat-

sächlichen Anwendungsfall selbst - stets die gleichen sind 9). Es wird ein Ausblick über weitere, mögliche Forschungsgebiete in diesem Umfeld gegeben 10).

Diese Arbeit zeigt die Eignung von verschiedenen DLTs für IOT-Anwendungsfälle anhand eines beispielhaften PoCs. Es werden nur solche Bereiche von IOT betrachtet, die auch grundsätzlich für die Implementierung auf DLTs geeignet sind. Es gibt darüber hinaus weitere Bereiche, die sich nicht eignen, um auf DLTs umgesetzt zu werden und müssen auf einer anderen technologischen Basis implementiert werden (vgl. Kapitel 2.2). Des weiteren wird die in dieser Arbeit durchgeführte Analyse anhand eines PoC belegt. Aufgrund von Restriktionen wie der Zeitlimitierung und der praktischen Umsetzbarkeit könnten unter Umständen nicht alle fundamentalen Anforderungen gezeigt werden, die für einen IOT-DLT-Anwendungsfall erfüllt sein müssen (vgl. Kapitel 7).



## THEORETISCHE GRUNDLAGEN

---

Dieses Kapitel stellt dem Leser benötigtes Basiswissen zur Verfügung. Es wird ein grundlegendes Wissen zu den Themenbereichen [DLT](#) und [IOT](#) vermittelt und auf weitere Informationsquellen verwiesen, welche tiefergehende Informationen bereitstellen.

### 2.1 DISTRIBUTED LEDGER TECHNOLOGIES

Die Begriffe Blockchain und Distributed Ledger Technology (kurz: [DLT](#)) werden häufig synonym verwendet<sup>1</sup>, wobei es sich bei Blockchain um eine Unterklasse von [DLT](#) handelt. Grundsätzlich werden Daten in Blockchains zu Blöcken zusammengefasst und mittels Hashketten in eine feste Reihenfolge gebracht. [DLTs](#) hingegen nutzen oftmals andere Strukturen als Blöcke, indem sie Daten beispielsweise durch Bäume oder Graphen persistieren. Abbildung [2.1](#) zeigt die Bestandteile einer [DLT](#) auf. Diese werden im Folgenden detailliert beschrieben und erklärt.

Es handelt sich bei [DLTs](#) um dezentrale Peer-to-Peer ([P2P](#))-Netzwerke, in denen Knoten (engl.: Nodes) Daten dezentral speichern, einen Konsens-Mechanismus zur Synchronisierung einsetzen und asymmetrische Verschlüsselungsverfahren zur Integrität und Absicherung des Systems nutzen. Hauptbestandteil eines Distributed Ledgers ist der Ledger selbst (deutsch: Kassenbuch), der eine Historie aller erfolgten Transaktionen speichert. Eine Transaktion stellt eine Wert- oder Informationsübertragung zwischen Entitäten dar. Sie beinhaltet einen Sender, einen Empfänger und eine Anzahl Einheiten, die versendet werden sollen. Je nach Implementierung können weitere Inhalte wie zum Beispiel Nutzdaten oder Software-Code dazukommen. Dieser Code, auch Smart-Contract genannt, wird in einer separierten Laufzeitumgebung (Virtual Machine) ausgeführt. Durch das Zusammenspiel mehrerer Smart-Contracts ist man in der Lage, ganze Anwendungen onchain auszuführen; hierbei spricht man von sogenannten [DApps](#). Smart-Contracts sind nur in der Lage, auf den internen (onchain) State des Ledgers zuzugreifen. Werden darüber hinaus weitere (offchain) Informationen benötigt<sup>2</sup>, so können diese durch sogenannte Oracles an den Smart-Contract übermittelt werden. Diese handeln als "Trusted Data Provider", da die bereitgestellten Informationen nicht validiert werden können. Transaktionen, die durch Smart-Contracts, [DApps](#) oder Benutzer ausgeführt werden, müssen von Nodes im Netzwerk validiert werden, bevor sie dem Ledger hinzugefügt werden können. Da-

---

<sup>1</sup> In dieser Arbeit werden die Begriffe ebenfalls synonym verwendet, andernfalls wird explizit darauf hingewiesen.

<sup>2</sup> Beispiele hierfür können Wetterdaten, Aktien- und Wechselkurse oder Lottozahlen sein

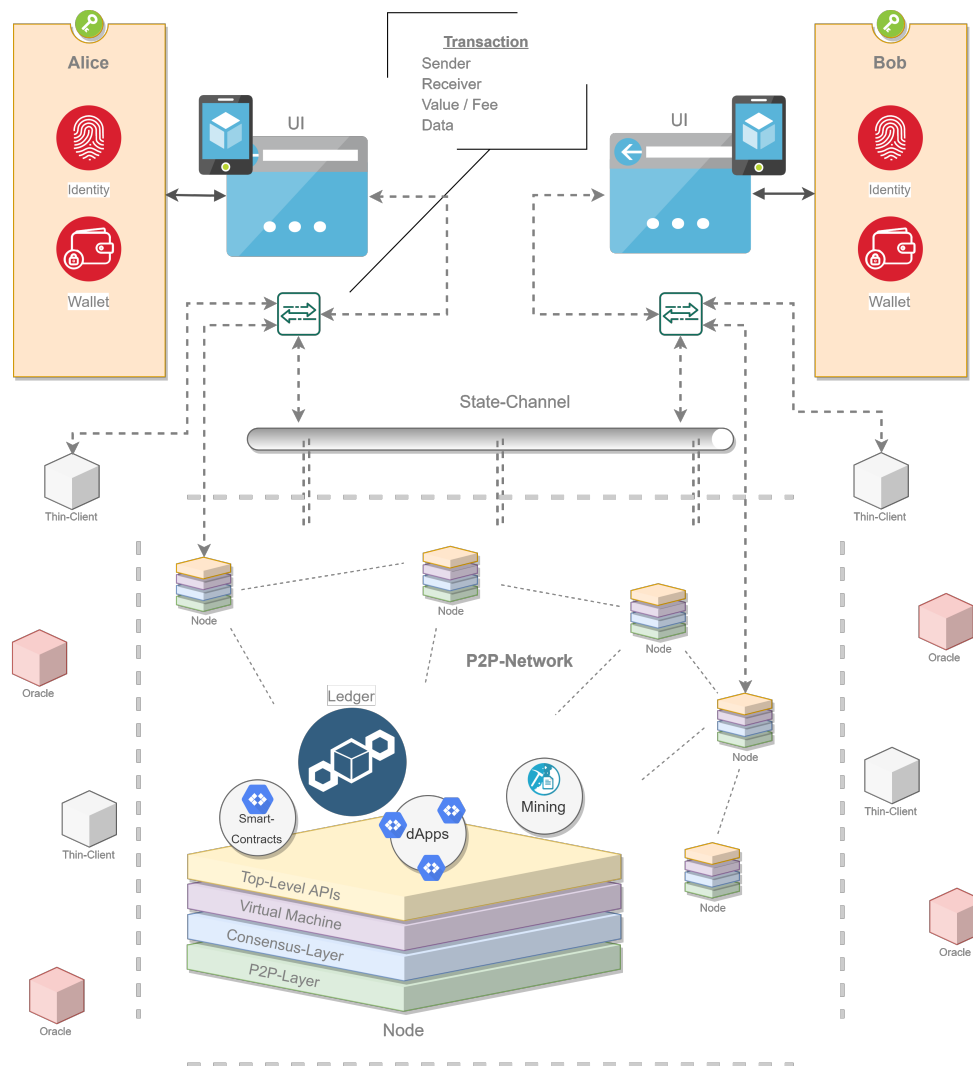


Abbildung 2.1: Bestandteile von DLTs

zu werden diese durch komplexe Berechnungen zusammengefasst (Mining), dem lokalen Ledger hinzugefügt und anschließend an andere Knoten versendet. Das Konsensprotokoll definiert die Regeln, wann ein Block hinzugefügt werden darf, welche Eigenschaften dieser erfüllen muss und ob das Mining korrekt durchgeführt wurde. Dürfen beliebige Nodes am Konsens teilnehmen, so spricht man von einer permissionless Blockchain; sind nur privilegierte Nodes befähigt, am Konsens teilzunehmen, so spricht man von einer permissioned Blockchain. Darüber hinaus lassen sich Blockchain-Netzwerke klassifizieren, ob Transaktionen und Blöcke öffentlich erreichbar und lesbar sind (Public Blockchain) oder ob auch hier Restriktionen vorliegen (Private Blockchain). Besondere Anforderungen wie möglichst niedrige Transaktionskosten, eine hohe Transaktionsgeschwindigkeit, ein eingeschränkter Wirkungskreis der Anwendung oder Beschränkungen durch das Geschäftsmodell sind einige Beispiele für die Zulassungsbeschränkung und beschränkte Teilnahme am Konsensverfahren; je nach den gegebenen Anforderungen

müssen Blockchain-Protokolle für den jeweiligen Anwendungsfall gewählt oder angepasst werden. Alle Knoten, die dem Konsens-Protokoll folgen und eingehende Blöcke nach diesem Protokoll validieren heißen Full-Nodes. Daneben existieren sogenannte Light-Clients, die nur Teile des Ledgers vorhalten und meist nur die Block-Header auf Validität überprüfen. Thin-Clients sind genauso wie Oracles nicht Teil des P2P-Netzwerkes sondern greifen mittels Application Programming Interface (API) auf die Nodes zu. Möchte ein Benutzer (in der Abbildung durch Alice und Bob dargestellt) mit der Blockchain kommunizieren oder diese als Kommunikationsmedium nutzen, so geschieht dies ebenfalls über entsprechende APIs. Über eine grafische Benutzeroberfläche können Benutzer meist per Browser oder Smartphone Transaktionen auslösen und diese über das Blockchain-Netzwerk verschicken. Jeder Benutzer besitzt eine Wallet, die aus einem öffentlichen und einem privaten Schlüssel besteht (Public / Private Key). Der Public-Key gleicht der Kontoadresse eines Bankkontos, der dazugehörige Private-Key der PIN-Nummer. Damit besitzt jeder Nutzer eine eindeutige Identität. Möchte Alice eine Transaktion im Wert von 10€ an Bob senden, so kann sie dies auf unterschiedlichen Wegen tun. Zunächst einmal kann sie die gewünschten Informationen auf ihrem Smartphone mittels einer Wallet-App eingeben und per Knopfdruck an das Blockchain-Netzwerk übermitteln. Diese wird dort über die entsprechende API empfangen, verarbeitet und im Ledger persistiert. Bob kann ebenfalls mittels seiner Smartphone-App verfolgen, wann sich sein Kontostand ändert und wer eine Transaktion an seine Adresse gesendet hat. Eine alternative Lösung hierzu wäre der Einsatz von sogenannten State-Channels. Diese können Alice und Bob dazu nutzen, beliebig viele Transaktionen untereinander offchain auszutauschen und lediglich die endgültige Bilanz aller untereinander ausgetauschten Transaktionen onchain in den Ledger zu übertragen. Die Vor- und Nachteile beider Varianten sowie die in diesem Abschnitt aufgezeigten Fachbegriffe werden im weiteren Verlauf genauer erläutert.

### 2.1.1 Konsensalgorithmen

Transaktionen und Daten müssen über das Gesamtsystem hinweg konsistent sein. Um dies und die Integrität der Daten zu gewährleisten, bedarf es eines Mechanismus, welcher sicherstellt, dass alle Knoten die gleichen Daten halten. Solche Mechanismen werden durch Konsensalgorithmen beschrieben: Es handelt sich dabei um ein Protokoll, welches Regeln definiert, wie und welche Daten gespeichert und welche verworfen werden. Im Folgenden werden einige bekannte Konsensverfahren genannt und kurz vorgestellt.

**PROOF-OF-WORK (POW)** Transaktionen werden zu Blöcken zusammengefasst. Diese Blöcke werden durch das Lösen eines kryptografischen Puzzles (Mining) wie zum Beispiel die Ermittlung eines Hashwertes abgesichert. Ein Knoten, der einen Block dem Ledger hinzufügen möchte, muss zuerst eine zufallsbasierte Berechnung durchführen, um dem Netzwerk die Validität zu beweisen. Andere Knoten können mit sehr

wenig Aufwand überprüfen, ob die Berechnung korrekt war und der Knoten ehrlich gehandelt hat. Dieses Konsensverfahren ist meist mit einem hohen Stromverbrauch verbunden, da die zeitaufwändige Berechnung viele Ressourcen benötigt.

**PROOF-OF-STAKE (POS)** Unter der Annahme, dass Knoten mit hoher finanzieller Beteiligung (Stake) ehrlich agieren, um der Plattform nicht zu schaden und ihre finanziellen Mittel nicht zu gefährden, werden diese bei der Auswahl bevorzugt, eine Transaktion dem Ledger hinzuzufügen. Dieses Verfahren ist deutlich stromsparender als PoW, allerdings durch die Miteinbeziehung des Vermögens auch ungerechter im Vergleich zu PoW.

**DELEGATED-PROOF-OF-STAKE (DPOS)** Dieses Verfahren ergänzt PoS um ein Delegierten-System, was die Anzahl an potentiellen Konsensknoten stark einschränkt und damit die Performance erhöht. Knoten, die bössartig handeln, können aus dem Konsens herausgewählt werden.

**PRACTICAL BYZANTINE FAULT TOLERANCE (PBFT)** Um in der Lage zu sein, bis zu einem Drittel an bössartigen Knoten handhaben zu können (Byzantinischer Fehler, siehe [XYZ]), müssen alle Knoten des Gesamtsystems bekannt sein. Dieses Verfahren löst das Konsensproblem, indem der Konsens in Runden eingeteilt wird und jede Runde Transaktionen in den Ledger übernommen werden. Indem die Knoten untereinander ihre Validierungsergebnisse austauschen kann ein Mehrheitsentscheid durchgeführt werden und damit die korrekten Transaktionen identifiziert werden.

**PROOF-OF-AUTHORITY (POA)** Dieses Verfahren wird meist in privaten und zugangsberechtigten Blockchains eingesetzt. Die Konsensknoten sind fest definiert und fungieren als Autorität des Gesamtsystems. Da kein Mining existiert, können Transaktionskosten minimal gehalten werden und das System kann aufgrund des einfachen Aufbaus des Konsens sehr gut skalieren.

Detailliertere Informationen sowie weitere Konsensverfahren werden von [19], [23] und [25] vorgestellt und analysiert.

### 2.1.2 *Smart-Contracts*

Bei Smart-Contracts handelt es sich um Software-Code, der auf der Blockchain (öchain") auf allen Knoten ausgeführt wird und eine Zustandsänderung meist in Form von ausgehenden Transaktionen zur Folge hat. Smart-Contracts werden in einer abgeschirmten Laufzeit-Umgebung (engl.: Runtime-Environment) ausgeführt; im Falle von Ethereum spricht man von der Ethereum Virtual Machine (EVM). Um auf Daten und Informationen außerhalb des Blockchain-Netzwerkes zugreifen zu können, können sich Smart-Contracts

sogenannter Oracle-Services bedienen. Dabei handelt es sich um Informationsanbieter (Trusted Data Provider), die zum Beispiel Wetterdaten, Lottozahlen oder Nahverkehrsinformationen an einen Smart-Contract senden, damit dieser, basierend auf den empfangenen Daten, seine Logik ausführen und die Daten verarbeiten kann. Der grundlegende Nachteil von Oracle-Services liegt in der Vertrauensfrage: Während eine Blockchain grundsätzlich mittels Konsensverfahrens die Vertrauensfrage beantwortet, obliegt dem Besitzer bzw. Betreiber eines Oracles die Macht über die Richtigkeit der Daten. Dieser Problematik kann durch den Einsatz mehrerer, mittels eines separat implementierten Konsens abgestimmter Oracle-Services entgegengewirkt werden. Mit dem Software-Code von Smart-Contracts können komplexe Logiken, so auch Kauf- oder Mietverträge, abgebildet und automatisiert abgearbeitet werden. Bei der Umsetzung und Automatisierung von juristischen Verträgen ist das jeweils geltende Recht eines Landes zu beachten: Die Form und der Aufbau sowie der abzubildende Inhalt eines Vertrages muss gewissen Normen entsprechen und alle benötigten Informationen enthalten, damit ein Vertrag rechtskräftig ist.

### 2.1.3 Dezentrale Apps (DApps)

Eine dezentrale Applikation (DApp) ist eine Anwendung, die auf dem Blockchain-Netzwerk dezentral ausgeführt wird. Die Applikationslogik wird dabei durch einen Smart-Contract oder durch den Zusammenschluss mehrerer Smart-Contracts abgebildet. Eine DApp kann demnach - genauso wie ein einzelner Smart-Contract - nicht auf Informationen außerhalb der Blockchain zugreifen; es sei denn, Informationen werden durch Oracles bereitgestellt. Da DApps! (DApps!) als solche keine dedizierte Benutzerschnittstelle besitzt, sondern lediglich über Blockchain-typische APIs erreichbar ist, trifft man DApps! meist in Verbindung mit einer Web-Applikation an, die dem Benutzer die Interaktion ermöglicht. Da eine Datenhaltung onchain sehr kostenintensiv ist, bietet es sich an, diese offchain zu speichern. Im Sinne einer dezentralen Anwendung bieten sich hierbei der Einsatz von Technologien wie InterPlanetary File System (IPFS)<sup>3</sup> an. Daten können dezentral und kostengünstig außerhalb des Blockchain-Netzwerkes aufbewahrt, allerdings onchain eindeutig referenziert werden. In Abbildung 2.2 wird ein beispielhafter Aufbau einer DApp aufgezeigt.

DApps haben den Vorteil, dass sie alle Stärken der Blockchain besitzen: Sie sind manipulationssicher, dezentral organisiert und ausfallsicher. Durch Kryptowährungen wie Ethereum oder Token wie können Zahlungsabwicklung nativ integriert werden. Auf der anderen Seite sind zum Beispiel Performanz und Ausführungskosten an die Kapazitäten und Eigenschaften des Blockchain-Netzwerkes gebunden. Zeitkritische oder Performanz-lastige Anwendungen sind nur in Ausnahmefällen für eine Implementierung auf einer Blockchain geeignet.

<sup>3</sup> Dezentrales, verteiltes Dateisystem, basierend auf einem P2P-Netz. Dateien werden versioniert und sind mittels Fingerprint(Hashing) eindeutig identifizierbar.

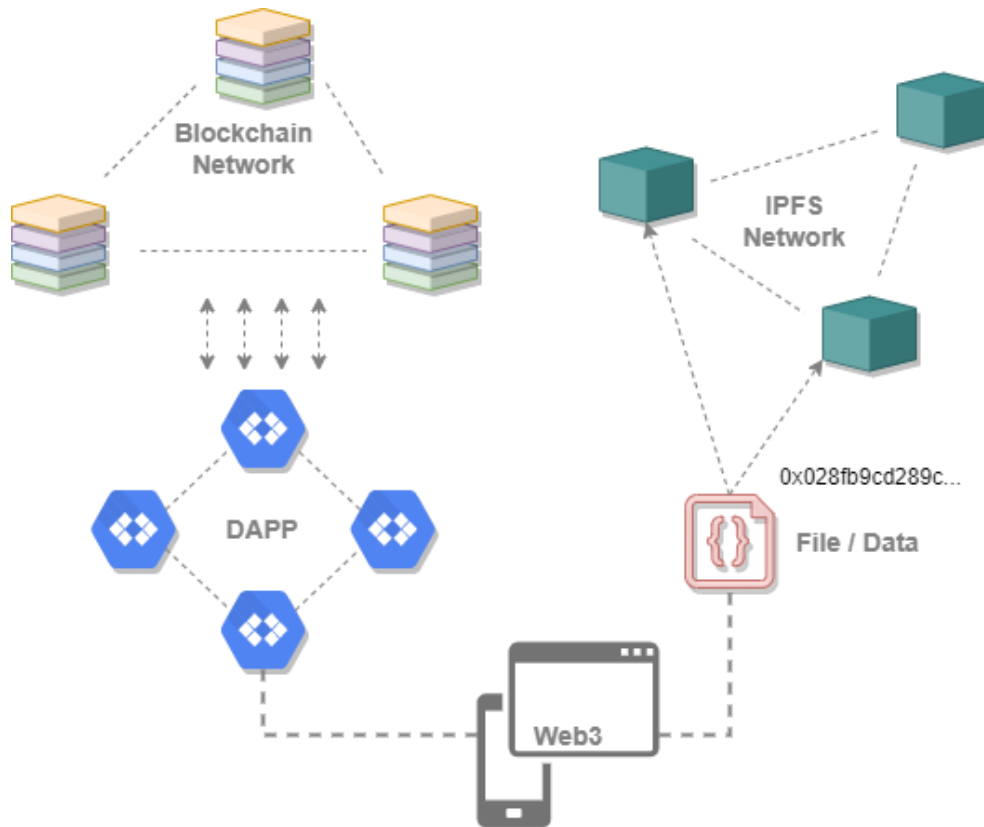


Abbildung 2.2: Aufbau von DApps

#### 2.1.4 Klassifizierung von Blockchains

Blockchains können nach [23] klassifiziert werden, indem man die Sichtbarkeit von Informationen sowie die aktive Teilnahme von Netzwerkknoten am Konsensmechanismus betrachtet. Ist die Blockchain öffentlich erreichbar und können Transaktionen und Blöcke von beliebigen Knoten eingesehen werden, so handelt es sich um eine öffentliche (Public) Blockchain. Ist der Zugang zum Blockchain-Netzwerk beschränkt und können nur berechtigte Knoten Transaktionen und Blöcke einsehen, so spricht man von einer privaten (Private) Blockchain. Eine besondere Form der Private-Blockchain ist die Konsortial-Blockchain; hierbei handelt es sich meist um einen Zusammenschluss mehrerer Entitäten, meist Unternehmen, welche die Zugangsbeschränkung zur Blockchain verwalten. Zur besseren Verständlichkeit können Private- und Public-Blockchains analog zu Intranet (firmenintern) und Internet (weltweit für jeden zugänglich) gesehen werden.

Darüber hinaus lassen sich Blockchains über die Teilnahme am Konsensalgorithmus, also der Entscheidung, welche Transaktionen in die Blockchain übernommen werden, klassifizieren: Ist jeder Netzwerkknoten, der die Blockchain erreichen und damit Informationen einsehen kann (unabhängig von public oder private), berechtigt am Konsensverfahren teilzunehmen, so spricht man von einer beschränkungslosen (Permissionless) Blockchain. Ist das Konsensverfahren auf privilegierte Knoten beschränkt, so handelt es

sich um eine zugangsbeschränkte (Permissioned) Blockchain. Die vorgestellte Klassifizierung in Public, Private, Permissioned und Permissionless lassen sich darüber hinaus kombinieren, wodurch es vier mögliche Arten von Blockchains gibt. Abbildung 2.3 stellt diese übersichtlich dar und listet dazu einige bekannte Umsetzungen auf.

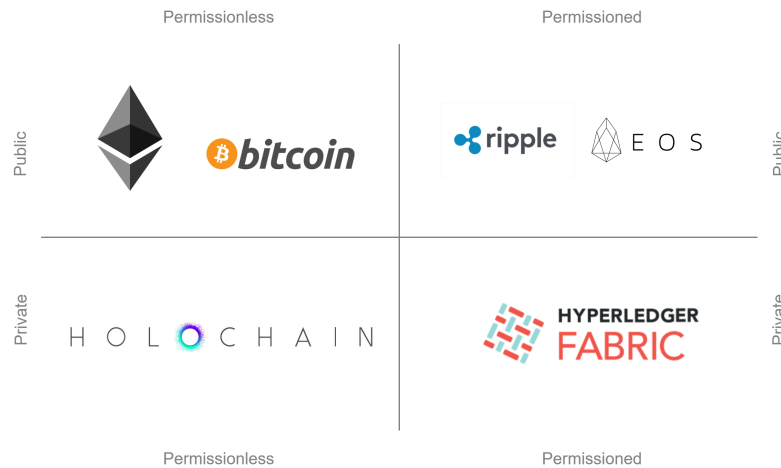


Abbildung 2.3: Taxonomie von Blockchains (vgl. [9])

Die bekanntesten Vertreter Bitcoin und Ethereum sind öffentliche und beschränkungslose Blockchains, an denen jeder teilnehmen kann und alle Daten öffentlich einsehbar sind. Die Klassifizierung ergibt sich als logische Konsequenz aus besonderen Anforderungen an die Blockchain: So können Transaktionskosten, Transaktionsgeschwindigkeiten, der Wirkungskreis der Anwendung oder das Geschäftsmodell Argumente für die Zulassungsbeschränkung und beschränkte Teilnahme am Konsensverfahren sein. Daraus resultiert, dass Blockchain-Protokolle je nach gegebenen Anforderungen des Anwendungsfalls ausgewählt und abgewägt werden müssen.

#### 2.1.5 Wallets

Einem Wallet ist ein Kontostand zugeordnet, besteht aus einem privaten Schlüssel (Private Key) und dem dazugehörigen öffentlichen Schlüssel (Public Key) und hat einen eindeutigen Eigentümer - den Besitzer des Private Keys. Der Public Key ist (in den meisten Fällen) die öffentliche Adresse, die für Transaktionen genutzt wird und entspricht in etwa einer Kontonummer einer Bank. Der Private Key fungiert als PIN-Nummer und ist nur dem Eigentümer der Wallet bekannt.

Multisignatur-Wallets sind Wallets, auf dessen Inhalt nur durch den Einsatz mehrerer Private Keys zugegriffen werden kann. Dabei wird ein Multisignatur-Wallet von  $n$  Schlüsseln erzeugt. Um auf den Inhalt zugreifen zu können, bedarf es der Signaturen von  $t$  von  $n$  Schlüsseln. Dadurch können verschiedene Anwendungsfälle wie zum Beispiel Treuhandkonten (2 von 3), Zwei-Wege-Authentifizierungen (2 von 2) und viele weitere abgedeckt werden. Multisignatur-Wallets können entweder Teil des Blockchain-Protokolls sein



und damit nativ unterstützt oder beispielsweise durch die Umsetzung mittels Smart-Contract realisiert werden. Letzteres ist dabei denkbar einfach: Es handelt sich um Code, der erst dann eine Transaktion ausführt, wenn  $t$  von  $n$  der hinterlegten Schlüssel unterschrieben haben. Die Höhe von  $t$  und  $n$  ist im Code festgelegt.

#### 2.1.6 Dezentrale Identitäten

Eine Identität zeichnet sich durch eine Menge von Informationen, Daten und Eigenschaften aus, die diese eindeutig identifizieren. Nur der Inhaber einer Identität kann mit dieser auch agieren, da er der einzige ist, der Zugriff auf die geschützten Informationen hat, die das Innehaben einer Identität bestätigen. Diese können zum Beispiel ein Passwort, eine Geburtsurkunde, ein Fingerabdruck oder Ähnliches sein.

Eine dezentrale Identität ist eine neuartige Technologie, die es dem Inhaber einer solchen erlaubt, seine Identität digital, dezentral und sicher durch den Einsatz asymmetrischer Verschlüsselung selbst zu verwalten. Sie ist kryptografisch verifizierbar und der Inhaber entscheidet selbst, welche Informationen er teilen möchte und welche nicht. Das World Wide Web Consortium (W3C) entwickelt aktuell (Stand: Dezember 2019) einen Industrie-Standard, der zur Verifizierung und Authentifizierung persönlicher Informationen vor Dritten im Web 3.0 eingesetzt werden soll und sich derzeit in der Version 1.0 befindet [22]. Eine dezentrale Identität besteht aus einer Decentralized Identifier (DID), welche weltweit einzigartig ist, und einem dazugehörigen DID-Dokument, welches Informationen über den beschriebenen Gegenstand enthält. Die folgenden Beispiele zeigen eine DID und ein dazugehöriges DID-Dokument (entnommen aus [22]).

Listing 2.1: Beispiel einer DID

```
did:example:123456789abcdefghi
```

Listing 2.2: Beispiel eines DID-Dokuments

```
{
  "@context": "https://www.w3.org/ns/did/v1",
  "id": "did:example:123456789abcdefghi",
  "authentication": [{
    "id": "did:example:123456789abcdefghi#keys-1",
    "type": "RsaVerificationKey2018",
    "controller": "did:example:123456789abcdefghi",
    "publicKeyPem": "-----BEGIN PUBLIC KEY...END PUBLIC KEY-----\r\n"
  }],
  "service": [{
    "id": "did:example:123456789abcdefghi#vcs",
    "type": "VerifiableCredentialService",
```



```

    "serviceEndpoint": "https://example.com/vc/"
  }
}

```

Durch den Einsatz von Blockchain-Technologie können **DIDs** manipulationssicher, hochverfügbar und für jeden zugänglich gespeichert werden. Die **DID** besteht aus drei Teilen: Zunächst dem Schlüsselwort "did", welches beschreibt, dass es sich um eine **DID** handelt. Anschließend folgt die DID-Methode (im Beispiel: example"), die definiert, wie die **DID** aufzulösen und weitere Informationen zu dieser Identität zu finden sind. Der letzte Teil ist eine ID ("123456789abcdefghi"), die für jede Methode einzigartig ist und somit eindeutig ermittelt werden kann.

Darüber hinaus existieren sogenannte Verifiable Credential (**VC**), die einer **DID** von vertrauenswürdigen Instanzen ausgestellt werden können. Dabei handelt es sich um verifizierbare Berechtigungsnachweise. Der Aussteller bescheinigt dem Empfänger eine bestimmte Eigenschaft und stellt einen Service-Endpoint zur Verfügung, an dem ein Dritter diesen **VC** verifizieren kann. So kann zum Beispiel eine Universität mit ihrer eigenen **DID** "did:hda:12345" einem Student "did:test:sebastiankanzlein" ein **VC** ausstellen, welches dem Student bescheinigt, aktuell an der Universität eingeschrieben zu sein. Möchte sich der Student nun an der Universitätsbibliothek authentifizieren, so kann er dort das **VC** der Universität vorzeigen und bekommt Zugriff auf die Bibliotheksausleihe. Die Bibliothek kann das **VC** verifizieren, indem es unter der Methode "hda" die Universität identifiziert und anschließend die Signaturen überprüft.

### 2.1.7 State-Channel

Blockchains gehen aufgrund ihrer Beschaffenheit einen Kompromiss ein. Durch den Konsensalgorithmus ist unabhängig von Blockgröße und Netzwerkkapazität ein natürliches Limit gegeben: Das Bitcoin-Netzwerk ist durch die komplexe Berechnung von **PoW!** (**POW!**) auf eine Blockzeit von 10 Minuten beschränkt. Ist die Blockgröße sehr klein, können Blöcke sehr schnell über das Netzwerk propagiert werden, allerdings nur wenige Transaktionen auf einmal übertragen werden. Ist die Blockgröße sehr groß, ist es sehr schwer für Knoten sich zu synchronisieren, dafür können mehr Transaktionen auf einmal übertragen werden. Durch die Limitierung von Blockgröße und Blockdauer ist dem gesamten Netzwerk eine natürliche Grenze gesetzt; im Fall von Bitcoin sind derzeit (Stand 01/2020) durchschnittlich etwa 7 Transaktionen pro Sekunde möglich. Andere Implementierungen nutzen gegebenenfalls andere Konsensmechanismen und andere Parameter, allerdings liegen auch dort natürliche Schranken vor. Es wird deutlich, dass mit steigenden Anforderungen vor allem an die Transaktionsverarbeitung pro Zeitintervall [17] (meistens Transactions per Second (**TPS**)) eine verbesserte Performanz und neue Lösungsansätze erforderlich werden. Es wird eine Lösung für die schlechte Skalierung von Blockchains gesucht.

Als eine mögliche Antwort auf das Skalierungsproblem von Blockchains

werden sogenannte State-Channels entwickelt. Ziel ist es dabei, jegliche Arten von Status-ändernden Operationen off-chain zu prozessieren, die typischerweise auf der Blockchain ausgeführt und on-chain gespeichert werden. Dadurch können die Anzahl an Zugriffen auf die Blockchain sowie die Anzahl an Transaktionen verringert und gleichzeitig die Interaktionszeit zwischen einzelnen Parteien verbessert werden. Im Kontext von Zahlungen (Payments) werden dadurch sogenannte Micro-Payments ermöglicht; bei State-Channels, die sich auf die Zahlungsabwicklung beschränken, spricht man von Payment-Channels. Diese können schneller und günstiger als normale Transaktionen erfolgen. Die Kosten solcher Micro-Payments können sehr gering gehalten werden, da nicht alle Transaktionen auf der Blockchain gespeichert werden. Die Grundidee ist dabei folgende: Alice und Bob reservieren einen Teil ihres Vermögens auf der Blockchain, sodass sie vorerst nicht darüber verfügen können und eröffnen einen Payment-Channel zueinander. Diese Transaktion, also das Eröffnen des Payment-Channels, wird auf der Blockchain gespeichert. Das Volumen des Channels, also das Vermögen, das Alice und Bob nun zwischen einander austauschen können, entspricht der Summe der reservierten Vermögen. Ihr gewünschtes Vermögen, welches für den Payment-Channel reserviert werden soll, kann entweder mittels Multisignatur-Wallet (vgl. Kap. 2.1.5) oder Smart-Contract (vgl. Kap. 2.1.2) reserviert werden. Alice und Bob haben nun einen State von jeweils 50 Euro. Anschließend können sich beide signierte off-chain Transaktionen schicken (also nicht über das Blockchain-Netzwerk). Diese Transaktionen enthalten den neuen State: Überweist Bob 10 Euro an Alice, so ändert sich der State von Alice von 50 Euro auf 60 Euro. Schickt Bob erneut eine Transaktion von 10 Euro, so ändert sich der State von Alice auf 70 Euro. Diesen Vorgang können beide nun solange wiederholen, wie sie möchten, solange sie sich in dem Volumen von 100 Euro bewegen. Zum Schließen eines Channels sendet Alice oder Bob eine Transaktion an das Blockchain-Netzwerk, welche den finalen State enthält (im Beispiel hat Alice 70 Euro und Bob 30 Euro). Für theoretisch unendlich viele Transaktionen zwischen Alice und Bob müssen lediglich die eröffnende und die schließende Transaktion des Payment-Channels on-chain gespeichert werden.

Einen weiteren, großen Vorteil liefert die durch den State-Channel ermöglichte Asynchronität von Transaktionen. Befinden sich Teilnehmer nicht im Blockchain-Netzwerk (zum Beispiel durch Konnektivitätsprobleme), so können keine Transaktionen durchgeführt werden. Hängen beispielsweise reale Aktionen wie zum Beispiel das Öffnen einer Schranke oder das Prozessieren einer Aktion mit einer Blockchain-Statusaktualisierung zusammen, so würde bei Konnektivitätsverlust der reale Prozess zum Stillstand kommen, bis die Verbindung wiederhergestellt wird. State-Channel könnten hierbei eingesetzt werden, um eine redundante Verbindung zu schaffen, die auch bei Nicht-Erreichbarkeit der Blockchain genutzt werden könnte. Einige Beispiel-Implementierungen von State-Channels sind Bitcoins Lightning Network, Ethereums Raiden Network oder die Implementierung von Neo namens Trinity.

Eine weitere Möglichkeit, Blockchain-Anwendungen zu skalieren, kann durch den Einsatz von Side-Chains geschaffen werden. Hierbei handelt es sich um separate Blockchains, die parallel zur Haupt-Blockchain (auch oft Eltern-Blockchain oder Main-Chain genannt) laufen. Um eine Side-Chain zu eröffnen, muss zunächst der Beweis erbracht werden, dass alle Assets, deren Status in der Side-Chain verändert werden sollen, auf der Main-Chain gesperrt oder reserviert sind, sodass der Eigentümer temporär nicht über diese verfügen kann (beispielsweise durch Zero-Knowledge Proofs, siehe XYZ). Diese gesperrten Assets können anschließend in die Side-Chain transferiert werden. Dort kann der Status verändert werden; beispielsweise kann eine Geld-Transaktion durchgeführt werden. Soll ein Asset zurück auf die Main-Chain transferiert werden, so muss der Beweis erbracht werden, dass dieses Asset auf der Side-Chain gesperrt wurde. Dadurch werden Effekte wie das Double-Spending verhindert. Ähnlich wie bei State-Channels handelt es sich auch bei Side-Chains um einen Skalierungsansatz auf Layer-2 der Blockchain-Architektur. Das bedeutet, dass diese Ansätze nicht direkt auf der Blockchain selbst (Layer-1) ausgeführt werden, sondern auf einem separaten System.

#### 2.1.8 Blockchain als Kommunikationsprotokoll

Das OSI Modell gilt seit Mitte der 80er-Jahre als Standard zur Einordnung von Netzwerkprotokollen. Es wurde von der International Organization for Standardization (ISO) entwickelt und besteht aus sieben Schichten ([24], [2]):

**PHYSICAL LAYER** Die Übertragung des Bit-Datenstroms über ein physikalisches Medium (Hardware) findet auf dieser Ebene statt.

**DATA LINK LAYER** Diese Schicht kapselt Daten in Datenframes; es werden grundlegende Funktionalitäten zur Fehlererkennung und Fehlerbehebung bereitgestellt. Bekannte Protokolle dieser Ebene sind zum Beispiel Ethernet und Address Resolution Protocol (ARP).

**NETWORK LAYER** Diese Schicht kapselt Daten in Datenpaketen und implementiert Sequenznummern, Flusskontrolle und Funktionalitäten fürs Routing.

**TRANSPORT LAYER** Diese Schicht ist zuständig für die Ende-zu-Ende Übermittlung von Daten, die sie vom Session Layer empfängt. Die Daten werden in sogenannte Segmente unterteilt und über das Netzwerk versendet und auf der Empfängerseite ebenfalls von der Transport-Schicht wieder zusammengesetzt.

**SESSION LAYER** Kommunikationskanäle werden auf dieser Schicht geöffnet, geschlossen und verwaltet und heißen Sessions.

**PRESENTATION LAYER** Diese Schicht ist zuständig für Datenkonvertierungen, -kompressionen und stellt Funktionalitäten wie Ver- und Entschlüsselung bereit.

**APPLICATION LAYER** Diese Schicht stellt die Schnittstelle zum Benutzer dar.

Bei einem dezentralen Netzwerk wie **DLT**, liegt es nahe, eine Einordnung in das **OSI** Modell durchzuführen. Dazu werden die verschiedenen Bestandteile eines **DLTs** vorgestellt und den Schichten des **OSI** Modells zugeordnet. Die Schichten eins bis drei (Physical, Data Link und Network) werden hierbei nicht betrachtet, da die Kommunikation über das Internet erfolgt und demnach auf bekannten Protokollen aufbaut (Ethernet, IP).

Eine Blockchain basiert auf einem **P2P**-Netz, in dem alle Knoten miteinander verbunden sind und miteinander kommunizieren können. Diese Kommunikation geschieht nach dem Ende-zu-Ende Prinzip und nicht nach dem Punkt-zu-Punkt Prinzip. Die **P2P**-Kommunikation erfolgt meist über die entsprechenden Transportprotokolle TCP oder UDP, welche Daten als Segmente zwischen Sender und Empfänger austauschen. Demnach findet die **P2P**-Kommunikation auf dem Transport-Layer (Schicht 4) statt.

Der Konsensalgorithmus legt zum einen fest, welche Knoten am Konsensverfahren partizipieren dürfen und welche nicht. Dazu werden Verbindungen zu anderen Knoten aufgebaut und ggfs. wieder geschlossen. Zum anderen wird definiert, welchem Schema die Kommunikation folgt. Eingehende Datenssegmente der **P2P**-Schicht werden entgegengenommen und in Form von Transaktionen gemäß der Konsensregeln verarbeitet. Die Ergebnisse werden wiederum an die **P2P**-Schicht zurückgespiegelt und über das Netzwerk an die anderen Knoten publiziert. Diese Vorgänge sind in den Session-Layer des **OSI**-Modells einzuordnen.

Die virtuelle Maschine einer Blockchain sorgt dafür, dass die Verarbeitungsschritte auf allen Knoten zum selben Ergebnis führen. Dazu werden Smart-Contracts aus Transaktionen extrahiert, in Byte-Code übersetzt und in separierten Laufzeitumgebungen ausgeführt. Verschlüsselte Nutzdaten von Transaktionen werden ebenfalls in dieser Laufzeitumgebung entschlüsselt. Die Funktionen der Smart-Contracts werden der nächst-höheren Schicht zur Verfügung gestellt. Darüber hinaus werden Daten, die in Byte-Form aus Smart-Contracts entstehen, konvertiert und in Form von Transaktionen an die untere Ebene weitergereicht. Diese Funktionalitäten können in den Presentation-Layer (Schicht 6) eingeordnet werden.

Die Schnittstellen zum Benutzer stellen Top-Level-APIs dar, über die der Benutzer mit der Blockchain kommunizieren kann. Sendet ein Benutzer eine API-Anfrage an einen Node, kann dieser gemäß des Protokoll-Stacks Aktionen in die unteren Schichten weiterleiten und in das Netzwerk propagieren. API-Calls werden in diesem Kontext meist von einer Web-Anwendung abgesendet und die Ergebnisse mittels User-Interface (**UI**) dem Benutzer präsentiert. Damit stellen die genannten API-Calls eine Schnittstelle zum Benutzer bzw. zur Benutzeranwendung dar und können dem Application-Layer (Schicht 7) zugeordnet werden.

In Abbildung 2.4 werden die zugeordneten Schichten der Blockchain noch einmal aufgezeigt und mit den sieben Schichten des **OSI**-Modells dargestellt.

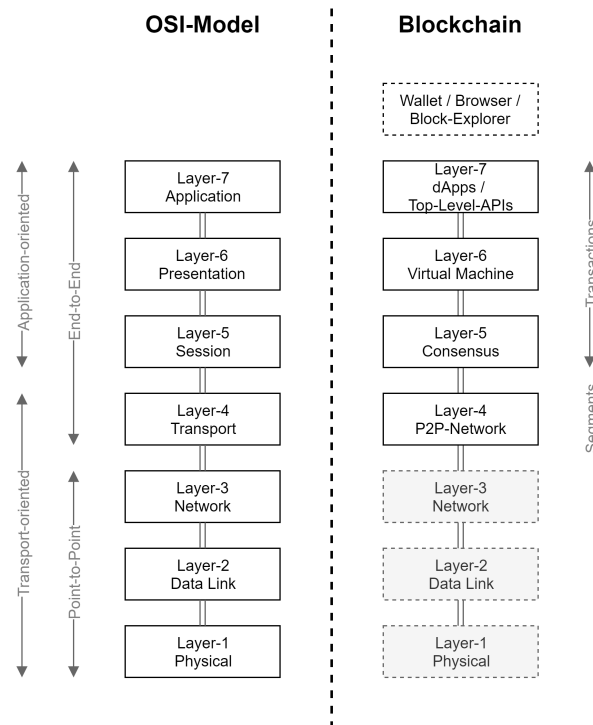


Abbildung 2.4: Die Blockchain-Layer im Konext des OSI-Modells

Der Duden definiert den Begriff Protokoll als „Festlegung von Standards und Konventionen für eine reibungslose Datenübertragung zwischen Computern“. Ein Kommunikationsprotokoll ist gemäß des Gabler Wirtschaftslexikons „eine Übermittlungsvorschrift bei der Datenübertragung, die die gesamten Festlegungen für Steuerung und Betrieb der Datenübermittlung in einem Übermittlungsabschnitt [...] umfasst“. Diese Definitionen zusammen mit der Einordnung der Blockchain-Technologie in das OSI-Modell machen deutlich, dass es sich bei dabei um ein Kommunikationsprotokoll handelt. Zusammen mit der Zahlungsabwicklung und der Abbildung von Eigentumsverhältnissen beziehungsweise von Eigentumsübergängen spricht man auch von einem sogenannten „Value Exchange Protocol“[3].

Durch die Einordnung in das OSI-Modell kann auch die Sinnfrage beantwortet werden, weshalb die Blockchain-Technologie eine vielversprechende Alternative zu klassischen IT-Systemen darstellt: Betrachtet man die Technologie objektiv, abseits des Hypes und des Anwendungsfalls der Kryptowährungen, so erkennt man eine junge Technologie mit hohem Potenzial. Die Blockchain als Kommunikationsprotokoll gibt dem modernen Softwareentwickler umfangreiche Werkzeuge und Out-of-the-Box Funktionalität über vier Schichten des OSI-Modells hinweg an die Hand. Eine dezentrale Plattform mit integrierter Datenbank, Security, Ver- und Entschlüsselung, P2P-Netzwerkkommunikation, Entwicklungs- und Laufzeitumgebung für dezentrale Anwendungen, sicherer Zahlungsabwicklung und hoher Verfügbarkeit ist oftmals per Knopfdruck binnen Sekunden erstellt. Alle diese Bestandteile

und Funktionalitäten sind in der technischen Spezifikation einer jeden Blockchain in Form eines Kommunikationsprotokolls enthalten und klar definiert.

## 2.2 INTERNET OF THINGS

Der Begriff Internet der Dinge - kurz **IOT** - ist ein Sammelbegriff und bezeichnet die Vernetzung von Gegenständen untereinander (meist über das Internet). Es wird eine autonome Machine-to-Machine (**M2M**)-Kommunikation ermöglicht, die wiederum den Automatisierungsgrad in dem jeweiligen Einsatzgebiet erhöht. Das bedeutet, dass die Kommunikation zwischen den **IOT**-Geräten selbstständig erfolgt, also ohne das Eingreifen eines Menschen. Nach [20] lässt sich das Themenfeld **IOT** in zwei Bereiche untergliedern: Consumer IOT (**CIOT**) und Industrial IOT (**IIOT**). **CIOT** findet Anwendungen im privaten Umfeld, vor allem geht es hier um Smart-Home und den damit verbundenen Applikationen: Smart-Gardening, Smart-Lights, intelligente Türschließsysteme und Heizungssteuerungen. **IIOT** fokussiert sich auf den kommerziellen Bereich und versucht Anwendungen im deutlich größeren Stil zu entwickeln: Die Bereiche Automotive, Energie und Supply-Chain sind hierbei einige wichtige Vertreter und treten als Smart-Factory, Smart-City, Connected-Cars und Weitere in Erscheinung. Die Abbildung 2.5 gibt eine Übersicht über die grobe **IOT**-Architektur im Kontext von **DLT**.

Um physische Geräte miteinander zu vernetzen werden sogenannte digitale Zwillinge (engl.: Digital Twins) erzeugt. Dabei handelt es sich um virtuelle Abbildungen der Geräte, die mittels Sensoren und Aktoren miteinander verbunden sind. Somit können Zustandsänderungen vom physischen Teil zum virtuellen Teil übertragen werden und umgekehrt. Jedes Gerät besitzt eine eindeutige Identität, die im Blockchain-Umfeld mittels dezentraler Identität abgebildet wird (vgl. 2.1.6). Darüber hinaus und abhängig vom Anwendungsfall besitzen Geräte eine Wallet, um automatisierte Bezahlvorgänge **M2M** durchzuführen. Jedes Gerät hat eine bestimmte Funktionalität und trägt einen Teil zum IOT-System bei, wobei die Geräte über ein lokales Gateway mit dem Internet verbunden sind. Dabei kann es sich um einen einfachen WLAN-Router, einen Mobilfunk-Chip oder eine andere Art von Gateway handeln. Die Kommunikation erfolgt über gängige Kommunikationsprotokolle wie MQTT oder HTTP oder - wie im Falle von Blockchain - über das Blockchain-Protokoll. **IOT**-Anfragen werden über das Blockchain-Netzwerk prozessiert; aus Sicht eines **IOT** Geräte-Herstellers und Konsumenten formulieren Christidis et al. ([7]) die Vorteile davon sehr treffend:

„From the manufacturer’s side, the current centralized model has a high maintenance cost consider the distribution of software updates to millions of devices for years after they have been long discontinued. From the consumer’s side, there is a justified lack of a trust in devices that phone home in the background and a need for a security through transparency approach.“ Christidis et al. nennen zwei zentrale Kerneigenschaften eines Distributed Ledgers: Zum einen die Verteilung und die einfache Anbindung und Erreichbarkeit von **IOT**-Geräten, die die Wartung seitens des Herstellers erleichtern



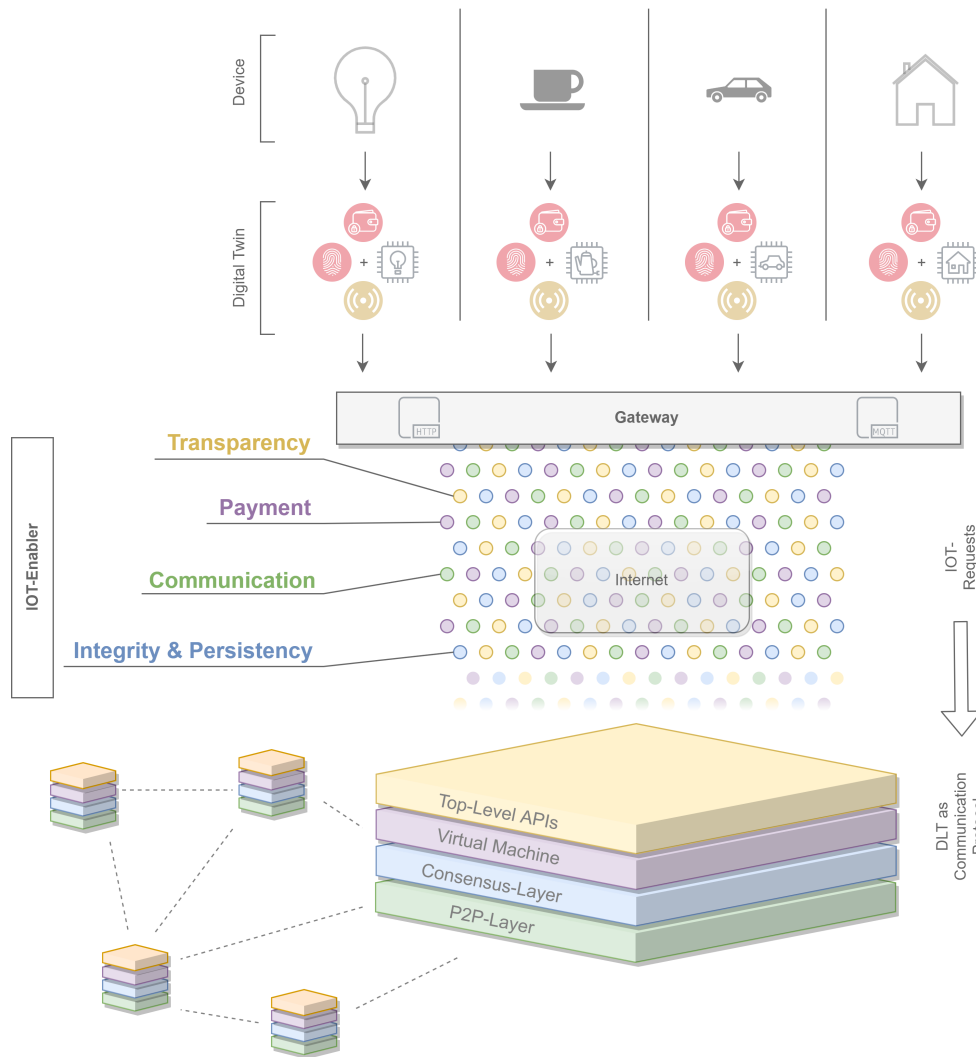


Abbildung 2.5: Übersicht IOT im Kontext von DLTs

können. Zum anderen sprechen die Autoren die Vertrauensfrage seitens der Kunden in Bezug auf Datensicherheit und Privatsphäre an. Hier kann die Blockchain-Technologie „Sicherheit durch Transparenz“ erzielen und für eine größere Akzeptanz sorgen.

Der Einsatz von Blockchain-Technologie ist allerdings nicht immer ratsam oder möglich. So kann beispielsweise aufgrund des Anwendungsfalls oder aufgrund von technischen Beschränkungen und Gegebenheiten der Einsatz eines DLTs im IOT-Umfeld unvorteilhaft sein. Ersteres könnte zum Beispiel eine Smart-Home Anwendung sein, die eine Temperaturregelung und Überwachung der eigenen vier Wände vorsieht. Ein dezentraler Ledger wäre hier überdimensioniert und für diesen Zweck überqualifiziert. Der Aufwand stünde in keinem Verhältnis zum Nutzen. Darüber hinaus zeigt dieser IOT-Anwendungsfall auch keine Charakteristika einer DLT-Anwendung auf: Es handelt sich um eine einzige beteiligte Partei in einem vertrauten Umfeld mit nur wenig Endgeräten. Auf der anderen Seite können technische Hür-

den den Einsatz von **DLT** verhindern. Gerade im Automotive-Bereich ist die Verarbeitung von Sensor- und Aktordaten in Echtzeit ein kritischer Punkt. Dezentrale Ledger eignen sich hierzu nicht, da sie nicht in der Lage sind, zeitkritische Anwendungen zu betreiben.

Es wird deutlich, dass die Beschaffenheit des **IOT**-Anwendungsfalls sehr entscheidend dazu beiträgt, ob der Einsatz einer Blockchain-Lösung sinnvoll sein kann oder nicht. Die Blockchain bietet aufgrund ihrer Kern-Eigenschaften einige IOT-Enabler nativ an: Transparente und überprüfbare Prozesse aller Geräte, integrierte und automatisierte **M2M**-Zahlungsabwicklung, **P2P**-Kommunikation sowie Datenintegrität und -persistenz.

### 2.2.1 Digitaler Zwilling

Ein digitaler Zwilling (engl. Digital Twin) ist nach [20] eine virtuelle Kopie physikalischer Objekte. Digitale Zwillinge werden in automatisierten IT-Prozessen benötigt, die sie als Schnittstelle zwischen physischer Welt und dessen digitalen Pendant fungieren. Der Zustand eines physikalischen Objekts wird in den digitalen Zwilling gespiegelt, welcher wiederum eine digitale Zustandsüberwachung und die Manipulation seines physischen Gegenstücks ermöglicht [20]. Der Ansatz von digitalen Zwillingen und die Thematik **IOT** haben gegenseitig enormen Einfluss aufeinander und befähigen einander zu neuen Anwendungsfällen. Diese können unter Anderem die Abbildung von Fabriken und Maschinen in digitale Automatisierungsprozesse sein, indem die Geräte und Teile mit Sensoren, Konnektivität und einer Steuerungslogik ausgestattet werden. Dadurch können Predictive Maintenance, Bedarfsplanungen und Prozessoptimierungen durchgeführt werden.

Im Umfeld Blockchain und **IOT** werden die Begriffe Digital Twin und Dezentrale Identität oft synonym verwendet, da beide ein physikalisches Objekt virtuell abbilden. Dezentralen Identitäten beschreiben als **W3C**-Standard die konkrete Umsetzung, wie virtuelle Identitäten nach der Idee eines Digital Twins standardisiert erzeugt und verwaltet werden können.



VERWANDTE FORSCHUNGSARBEITEN

---

Die Themenbereiche **DLT** und **IOT** sind bereits an vielen Stellen untersucht und beschrieben worden. Darüber hinaus existieren auch einige Werke, die sich mit den Synergien zwischen beiden Themenbereichen befassen.

In [10] werden **IOT**-Szenarien vorgestellt, die mit Blockchain umsetzbar sind. Es werden Anwendungsfälle aus den Bereichen Gesundheit, Logistik und Smart-City aufgezeigt. Darüber hinaus gehen die Autoren auf die praktischen Limitierungen ein, die durch Blockchain-Lösungen entstehen und beschreiben, an welchen Stellen weitere Forschung betrieben werden muss. Das Resultat ist, dass die Autoren keine allgemeingültige Blockchain-Lösung für **IOT**-Anwendungsfälle identifizieren konnten, die Technologie aber ihrer Meinung nach großes Potential mit sich führt.

Eine ausführliche Untersuchung vieler Konsensalgorithmen hinsichtlich Tauglichkeit für **IOT** wurde von den Autoren von [19] durchgeführt. Darüber hinaus werden einige der bekanntesten DLT-Implementierungen wie Hyperledger Fabric, Corda, Ethereum, Bitcoin und weitere gegenübergestellt und hinsichtlich Skalierbarkeit, Durchsatz, Latenz und weiteren untersucht. Es werden gewünschte Eigenschaften identifiziert, die eine **DLT** mitbringen sollte, um für Anwendungsfälle im **IOT**-Umfeld geeignet zu sein. Die Autoren unterscheiden nicht nach unterschiedlichen Anforderungen verschiedener Anwendungsfälle sondern betrachten **IOT** als Gesamtes. Das Fazit lautet, dass es derzeit keine konkrete **DLT**-Implementierung gebe, die alle Anforderungen von **IOT** vollumfänglich erfüllen. Es müsse ein bestehender Konsensalgorithmus erweitert werden oder die Vorteile verschiedener Implementierungen kombiniert werden.

In [21] wird untersucht, wo Blockchain-Knoten in einem **IOT**-Umfeld gehostet werden können. Die Autoren beschreiben, dass die Umsetzung eines Blockchain-Knoten auf einem **IOT**-Device aufgrund von fehlender Rechenleistung, hohem Stromverbrauch und geringer Bandbreite keine ratsame Lösung sei. Es wird die Umsetzung mittels Cloud- oder Fog-Computing vorgeschlagen. Als Ergebnis von Performance-Messungen kommen die Autoren zu dem Schluss, dass das Fog-Computing eine bessere Lösung darstelle als die Cloud-Variante.

Die Autoren von [12] sind der Überzeugung, dass Konsensalgorithmen, die auf dem Byzantinischen Fehler basieren, Potential bieten, um hoch-skalierende Anwendungen zu betreiben und geeignet für **IOT** zu sein. Dazu untersuchen sie verschiedene **BFT!** (**BFT!**) Konsensalgorithmen und führen Performance-Tests durch. Die abschließende Erkenntnis der Autoren ist, dass weitere Forschung zur Lösung des Konsensproblems im **IOT**-Umfeld nötig sei.

Nach den Autoren von [18] können **IOT** und **DLT** voneinander profitieren, man müsse jedoch zunächst Lösungen für Probleme wie den hohen Res-

ressourcenverbrauch, zeitliche Verzögerungen, Bandbreite und weitere lösen. Dazu beschreiben sie die einzelnen Komponenten sowie gängige Konsensverfahren einer Blockchain. Es werden Konsensverfahren und bestehende DLTs vorgestellt und nach Aspekten wie Durchsatz, Latenz, Sicherheit und Ressourcenverbrauch untersucht. Die Autoren unterscheiden nicht in unterschiedliche IOT-Anwendungsfälle und kommen zu dem Ergebnis, dass einheitliche Protokolle für IOT-Anwendungen auf DLTs definiert werden müssen und weitere Forschung besonders im Umfeld Konsensmechanismen notwendig sei.

## ANWENDUNGSFALL: VERMIETUNG VON HAUSHALTSGERÄTEN NACH DEM PAY-AS-YOU-USE PRINZIP

---

Qualitativ sehr hochwertige Haushaltsgeräte und Geräte für den professionellen Einsatz im Gastronomie-Umfeld haben hohe Anschaffungskosten, die der Privatanwender oder der Inhaber eines kleinen Gewerbes oftmals nicht leisten kann. Ein professioneller Kaffeevollautomat, eine leistungsfähige Spülmaschine oder eine Waschmaschine, die für hohe Kapazitäten ausgelegt ist, können Anschaffungskosten im vier bis fünfstelligen Euro-Bereich haben<sup>1</sup>. Eine naheliegende Möglichkeit besteht hier bei der Nutzung von Anbietern, die Haushaltsgeräte für eine monatliche oder jährliche Gebühr vermieten. So gibt es beispielsweise Anbieter für Kaffeemaschinen wie Tchibo<sup>2</sup> oder Nespresso<sup>3</sup>, die ihre Produkte direkt vermieten, oder Anbieter, die als Zwischenhändler fungieren und sich auf die Vermietung von Haushaltsgeräten verschiedener Hersteller spezialisiert haben. Dabei kommen klassische Miet- und Bezahlmodelle zum Einsatz, wobei es sich meistens um monatliche oder jährliche Mietgebühren handelt. Einen neuartigen Ansatz verfolgt das Unternehmen Winterhalter mit ihrem Pay-per-Wash<sup>4</sup> Ansatz. Hier bezahlt der Kunde keine monatliche Mietgebühr, sondern pro Waschgang; die Berechnung erfolgt also auf dem tatsächlichen Verbrauch des Kunden und nicht auf einer kalkulierten Pauschale.

Dieses Kapitel beschreibt einen IOT-Anwendungsfall, der die oben beschriebene Problematik aufgreift und das von der Firma Winterhalter eingeführte Pay-per-Wash Bezahlmodell einen Schritt weiterführt. Dabei interagieren verschiedene Stakeholder miteinander nach einem Pay-as-You-Use Prinzip auf einer einheitlichen Plattform.

### 4.1 BESCHREIBUNG

Kunden mieten Haushaltsgeräte (im Consumer-Bereich oder für den professionellen Einsatz) wie Kaffeemaschinen oder Waschmaschinen je nach Anbieter zum Nulltarif von verschiedenen Herstellern, die ihre Geräte auf einer Plattform zur Miete anbieten. Der genaue Verbrauch (Anzahl Kaffees, Menge an gewaschener Wäsche, Wasserverbrauch, etc.) wird mittels integrierter Sensoren an den Geräten erfasst und auf der Plattform persistiert. Damit wird ein genaues, vom tatsächlichen Verbrauch abhängiges Abrechnungsmodelle umgesetzt: Kunden zahlen nur das, was sie auch wirklich verbrauchen. Regelmäßige Reinigungen seitens der Kunden werden erfasst

---

<sup>1</sup> Quelle!

<sup>2</sup> <https://www.tchibo-coffeeservice.de/shop/kaffeevollautomaten/>

<sup>3</sup> <https://www.nespresso.com/pro/de/de/kaffeemaschinen-buero>

<sup>4</sup> [https://www.pay-per-wash.biz/ch\\_de/](https://www.pay-per-wash.biz/ch_de/)

und durch ein entsprechendes Rabattmodell verrechnet. Serviceleistungen wie Wartung und Reparatur durch entsprechende Dienstleister können über die zugrundeliegende Plattform geplant, gesteuert und abgerechnet werden. Die Lieferung von Geräten, Ersatzteilen und Konsumgütern wie Kaffee oder Waschmittel erfolgt durch Lieferanten. Die Bestellung und Abrechnung wird über die zugrundeliegende Plattform koordiniert.

Die folgende Abbildung 4.1 veranschaulicht den erläuterten Anwendungsfall.

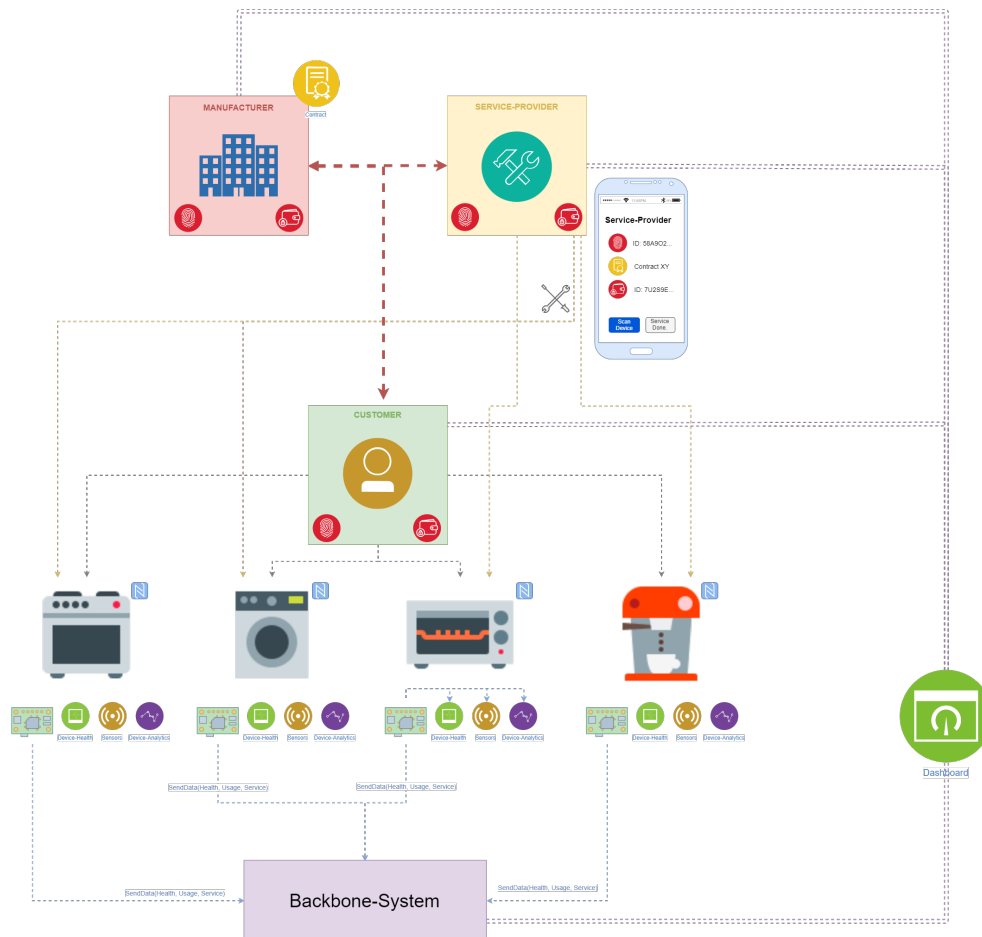


Abbildung 4.1: Grafische Veranschaulichung des Anwendungsfalles

Der vorgestellte Anwendungsfall lässt sich dem Teilgebiet **IIOT** (vgl. Kapitel 2.2) zuordnen und beinhaltet das Zusammenspiel mehrerer Stakeholder:

**HERSTELLER** Der Hersteller der Geräte entwickelt und produziert die zu vermietenden Haushaltsgeräte und bietet diese zur Vermietung an Kunden auf der Plattform an. Er vertreibt Ersatzteile sowie Pflege- und Zusatzprodukte zu seinen Geräten, die Kunden und Service-Dienstleister erwerben können. Die vermieteten Geräte des Herstellers können Service-Dienstleister selbstständig und automatisiert für eine Reparatur oder eine Wartung beauftragen. Die Beauftragung und Abrechnung erfolgt über die Plattform. Für vermietete Geräte erhält der Hersteller nach ei-

nem Pay-as-You-Use Prinzip eine Bezahlung der Kunden entsprechend ihres Verbrauches.

**LIEFERANT** Der Lieferant ist für die Lieferung der Geräte und Zusatzprodukte zu den Kunden und Service-Dienstleistern zuständig. Er holt die Ware beim Hersteller ab und liefert diese aus; die benötigten Adressinformationen sind auf der Plattform hinterlegt. Die Bezahlung für die Auslieferung erfolgt über die Plattform und berechnet sich automatisch über die Distanz der Lieferstrecke und der Abmessung der Ware.

**KUNDE** Der Kunde mietet Geräte vom Hersteller. Die Bestellung und Abrechnung erfolgt über die Plattform nach einem Pay-as-You-Use Prinzip. Die regelmäßige Reinigung der Geräte wird auf der Plattform protokolliert. Bei Einhaltung der vorgeschriebenen Reinigungsintervalle erhält der Kunde eine vertraglich festgelegte Gutschrift, bei Nicht-Beachten eine entsprechende Gebühr. Darüber hinaus kann der Kunde auf Wunsch Konsumgüter wie Kaffee und Reinigungsmittel über die Plattform bestellen; dies geschieht vollautomatisch über das Gerät: Sobald die Menge des Produktes ein gewisses Limit unterschreitet, beauftragt das Gerät selbstständig den Kauf und die Anlieferung der Produkte über die Plattform.

**SERVICE-DIENSTLEISTER** Der Service-Dienstleister ist zuständig für die Wartung und Reparatur der Geräte und wird von den Geräten über die Plattform beauftragt.

Die Abbildung 4.2 zeigt den groben Ablauf beginnend mit der Mietanfrage eines Kunden bis zur monatlichen Bezahlung der Teilnehmer für ihre Leistungen.

## 4.2 TECHNISCHE LÖSUNGSSKIZZE

In dieser Arbeit wird der oben beschriebenen Anwendungsfall basierend auf einer DLT-Lösung prototypisch umgesetzt. Neben den beteiligten Stakeholdern besteht das Gesamtsystem aus einem Frontend, dass jedem Stakeholder die für ihn relevanten Funktionen zur Verfügung stellt und Informationen anzeigt. Das Backend des Systems besteht aus einer DLT-Lösung<sup>5</sup>.

### 4.2.1 Endgeräte

Jedes Gerät besitzt eine eindeutige Identifikationsnummer und eine eindeutige Referenz zu dessen Eigentümer (Hersteller). Befindet sich ein Gerät in Vermietung, so existiert zwischen dem Mieter (Kunde) und dem Vermieter (Hersteller) ein Vertrag, der auf der Plattform persistiert wird. Das Gerät hat über das Internet Zugriff auf diese Plattform und damit auf den Vertrag,

<sup>5</sup> Die konkrete Implementierung, die als Backend-System eingesetzt wird, wird im Laufe dieser Arbeit ermittelt.

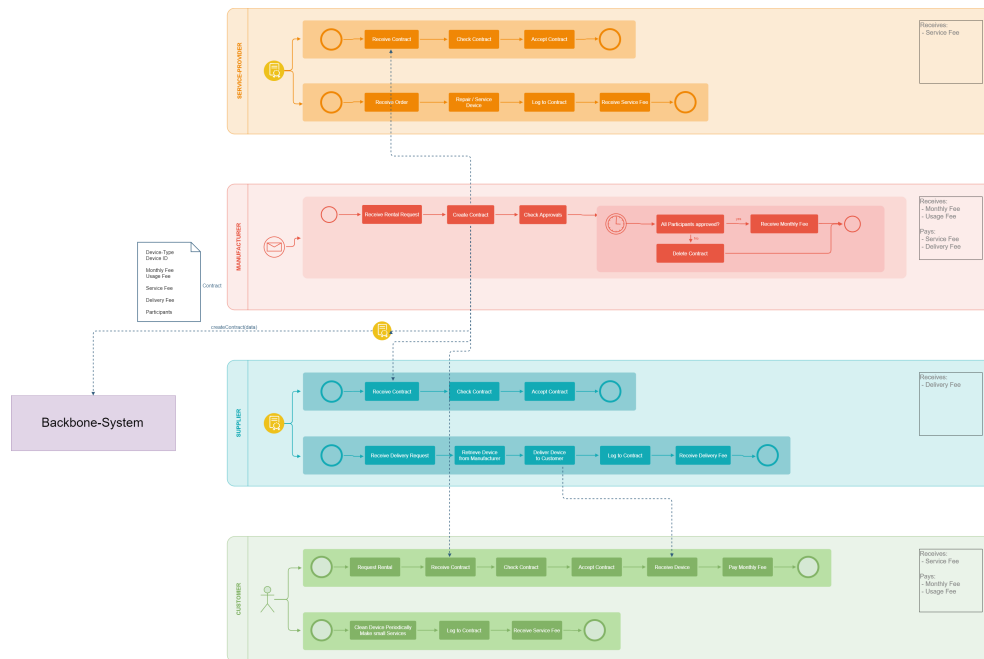


Abbildung 4.2: Grober Ablauf des Anwendungsfalls

in dem wichtige Informationen zu den Rahmenbedingungen wie der Dauer des Vertragsverhältnisses oder die Kosten einer verbrauchten Einheit. Die Sensoren zum Registrieren des Verbrauchs und des Gerätestatus befinden sich auf dem Gerät selbst. Diese melden alle gesammelten Daten an eine Sammelstelle am Gerät. Dort werden die Daten aufbereitet, nach der Gerätelegitimität verarbeitet und gesammelt. In einem regelmäßigen Intervall meldet das Gerät alle relevanten Daten wie Nutzung, Reinigungs- und Wartungsarbeiten und sonstige Informationen an das Backend und den verknüpften Vertrag, der wiederum die entsprechenden Geld-Transfers in die Wege leitet (siehe unten). In Abbildung 4.3 wird ein Endgerät schematisch dargestellt.

#### 4.2.2 Verträge

Verträge (Mietverträge, Service-Verträge, Lieferverträge) werden von allen beteiligten Parteien digital unterschrieben und im Backend gespeichert. Sie halten verschiedene Informationen, unter anderem über die Vertragslaufzeit, die Kosten sowie die zu erbringenden Leistungen der Parteien. Der Vertrag beinhaltet Mechanismen zur Begutschriftung und zur Belastung der Konten aller Beteiligten; die folgende Auflistung nennt alle wichtigen Geld-Transfers:

- Nutzung durch den Kunden (Sender ist der Kunde, Empfänger ist der Hersteller)
- Reinigung durch den Kunden (Sender ist der Hersteller, Empfänger ist der Kunde)

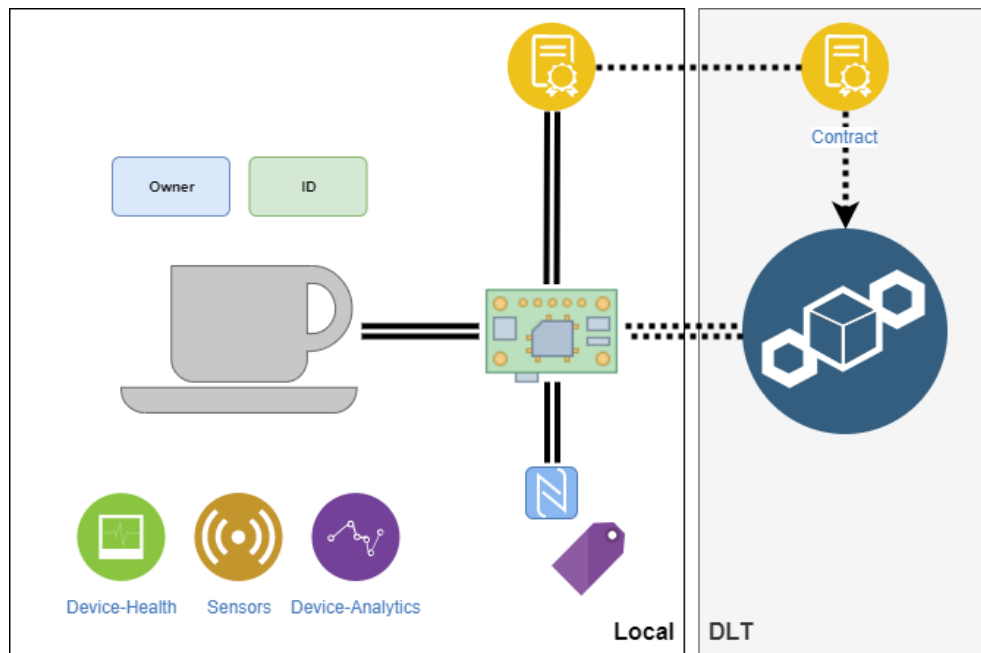


Abbildung 4.3: Aufbau und Bestandteile eines Endgeräts

- Wartung durch den Service-Dienstleister (Sender ist der Hersteller, Empfänger ist der Service-Dienstleister)
- Lieferung durch den Lieferanten (Sender ist der Absender, Empfänger ist die Lieferant)

#### 4.2.3 Benutzerschnittstelle

Das Frontend stellt eine grafische Benutzerschnittstelle bereit, die je nach Stakeholder die relevanten Informationen und Funktionalitäten bereitstellt. Um an der Plattform teilnehmen zu können, muss ein Registrierungsprozess durchlaufen werden und die Rolle bestimmt werden (Hersteller, Kunde, ...). Jeder Rolle ist der Zugriff auf eine Ansicht gestattet:

**HERSTELLER-ANSICHT** Eine Übersicht über alle Geräte sowie deren Status, ob sie sich derzeit in Vermietung befinden, gibt dem Hersteller Aufschluss über die momentane Gesamtlage. Laufende Verträge können eingesehen und aktuelle Mietanfragen bearbeitet werden.

**KUNDEN-ANSICHT** Die gemieteten Geräte sowie die damit verbundenen, laufenden Verträge werden angezeigt. Es besteht Transparenz über Verbrauchs- und Statusinformationen, die die Geräte an die Plattform übermitteln. Laufende Kosten und aktueller Verbrauch werden übersichtlich dargestellt. Es können Verträge gekündigt und neue Geräte gemietet werden.

**SERVICE-DIENSTLEISTER-ANSICHT** Eine Übersicht über alle aktuell laufenden Service-Verträge wird angezeigt. Alle Meldungen über Service-

Anfragen und Aufträge werden aufgelistet. Die Einnahmen durch Reparaturen und Services sowie den Kontostand werden detailliert dargestellt.

**LIEFERANTEN-ANSICHT** Eine Übersicht über alle aktuell laufenden Lieferverträge wird angezeigt. Die Einnahmen durch Auslieferungen sowie den Kontostand werden detailliert dargestellt.

#### 4.2.4 Backend

Als dezentrale Plattform verwaltet und speichert das Backend alle Verträge sowie die Identitäten, Konten (Wallets) und Interaktionen der oben aufgelisteten Stakeholder. Die Implementierung auf einer DLT kann in zwei verschiedenen Ausprägungen erfolgen, welche im Folgenden kurz dargelegt werden:

Die einfachste Lösung eine DLT zur Versionierung und Speicherung von Daten einzusetzen, entspricht einer simplen, dezentralen Hash-Datenbank. Hierbei wird der Status bestehend aus Mietverträgen, Kontoständen, Benutzerinteraktionen und Weiteren als Hashwert abgebildet und in der DLT persistiert. Somit wird ein einfaches Logging ermöglicht; die Stärken einer DLT werden allerdings nicht eingesetzt. An dieser Stelle ergibt die Frage, welchen Vorteil die umfangreiche Konfiguration und Einrichtung einer DLT im Vergleich zu einer klassischen verteilten Datenbank mit sich bringt.

Die zweite Variante setzt die Stärken einer DLT geschickt ein und geht weit über das einfache Dokumentieren von Revisionsständen hinaus: Informationen wie Benutzerinteraktionen, Miet- und Service-Verträge werden in die Blockchain geschrieben. Darüber hinaus geschieht die Vertragsabwicklung wie Zahlungstransaktionen, Vertragslogiken, etc. direkt auf der Blockchain. Es werden umfangreiche Geschäftslogiken abgebildet und manipulationssicher verarbeitet. Zahlungen werden instantan ohne Integration eines Drittanbieters wie zum Beispiel PayPal oder Ähnliche abgewickelt und können nachvollziehbar und transparent persistiert werden. Zusammengefasst bedeutet das, dass die optimale Lösung dieses Anwendungsfalls auf Basis einer DLT die Fähigkeiten dieser voll ausnutzt, indem Logging, Vertragsabwicklung, Payment und Manipulationssicherheit onchain ausgeführt werden.

Das Ziel der Umsetzung dieses Anwendungsfalls ist die Implementierung der zweiten Variante, um einen realistischen Anwendungsfall zielbringend zu implementieren und um einen entsprechenden Mehrwert durch den Einsatz einer DLT zu generieren.



## ANFORDERUNGEN

---

In diesem Kapitel werden die Anforderungen für den im vorherigen Kapitel aufgezeigten Anwendungsfall aufgestellt. Ziel ist es, die **DLT**-relevanten Anforderungen zu identifizieren, um dadurch im nächsten Kapitel eine geeignete **DLT**-Lösung für das Backend-System zu finden. Dazu werden zunächst einige Standards vorgestellt, wie Anforderungen klassifiziert und eingeordnet werden können. Diese Standards werden anschließend in einem optimierten Modell zusammengeführt und auf die konkreten Anforderungen angewandt. Die Klassifizierung dient als Werkzeug zur Einteilung der unübersichtlichen Gesamtmenge der Anforderungen und soll die weitere Identifikation **DLT**-relevanter Anforderungen vereinfachen. Als Zwischenergebnis existieren verschiedene Anforderungsklassen, die auf **DLT**-Relevanz geprüft werden. Durch schrittweises Ausschließen und Reduzieren der Anforderungsklassen werden jene Klassen identifiziert, die für die Umsetzung auf einer **DLT**-basierten Lösung entscheidend sind.

### 5.1 STANDARDS UND NORMEN

In diesem Abschnitt werden die verwendeten Standards und Normen aufgelistet und kurz beschrieben:

**BABOK** Das Business Analysis Body of Knowledge (**BABOK**) wird vom International Institute of Business Analysis (**IIBA**) herausgegeben und stellt einen Leitfaden für die Business-Analyse dar, genauere Informationen können [15] entnommen werden. Es unterteilt in sogenannte Knowledge-Areas und vermittelt Techniken und Kompetenzen im Umfeld der Business-Analyse. Anforderungen werden im **BABOK** nach Abstraktionsebene gruppiert: Die Business-Ebene, die Anforderungen abstrakt aus Sicht der gesamten Organisation betrachtet, die Stakeholder-Ebene, die die Anforderungen aus Sicht der verschiedenen Stakeholder beschreibt und die Solution-Ebene, die in funktionale und nicht-funktionale Anforderungen unterscheidet. Darüber hinaus gibt es eine Transition-Ebene, die temporäre Übergangsanforderungen zwischen dem Ausgangs- und dem Zielzustand des Gesamtsystems beschreibt.

**PMBOK** Das Project Management Body of Knowledge (**PMBOK**) wird vom Project Management Institute (**PMI**) herausgegeben und ist der State-of-the-Art Standard im Bereich Projektmanagement, siehe [16]. Das Werk teilt seine Sektionen ebenfalls wie das **BABOK** in sogenannte Knowledge-Areas ein und kennt die gleiche Gruppierung im Bereich Anforderungsmanagement. Neben der Einteilung in Business, Stakeholder, So-

lution und Transition Anforderungen kennt das **PMBOK** noch Quality und Project Anforderungen.

**SWEBOK** Das Software Engineering Body of Knowledge (**SWEBOK**) wurde von der Institute of Electrical and Electronics Engineers (**IEEE**) erstellt und stellt ein Standardwerk aus dem Bereich Software-Engineering dar, genauere Informationen können [1] entnommen werden. Anforderungen werden in System- und Software-Anforderungen unterteilt. Letztere werden untergliedert in Funktionale, Nicht-Funktionale, Produkt und Prozess Anforderungen.

**SEBOK** Das System Engineering Body of Knowledge (**SEBOK**) wurde unter Anderem von der IEEE erstellt und stellt ein Standardwerk aus dem Bereich System-Engineering dar, genauere Informationen können [6] entnommen werden. Anforderungen werden sehr detailliert unterteilt, unter Anderem in die Klassen Functional, Usability, Interface, Performance, Policies and Regulations, etc.

**ISO29148** Die Norm 29148 der **ISO** beschreibt das Anforderungs-Engineering **ISO** als Teilbereich des Software-Engineering, genauere Informationen können [13] entnommen werden. Anforderungen werden ähnlich wie beim **SEBOK** untergliedert in Functional, Usability, Interface. Darüber hinaus kennt die Norm Human Factors und Process Anforderungen.

**ISO25010** Die Norm 25010 der **ISO** beschreibt Qualitätskriterien eines Software-Produktes, siehe [14]. Die Kriterien Performance-Efficiency, Compatibility, Usability, Reliability, Security, Maintainability, Portability sowie deren Unterkriterien werden zur detaillierten Beschreibung von Nicht-Funktionalen Anforderungen eingesetzt.

Das Schaubild 5.1 zeigt die unterschiedlichen Normen und Standards grafisch auf und stellt die verschiedenen Anforderungsgruppierungen zueinander in Beziehung.

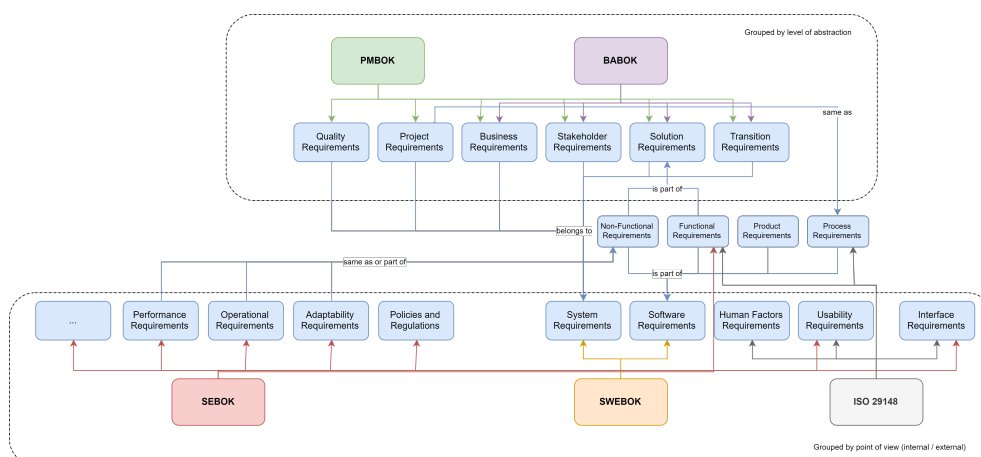


Abbildung 5.1: Einordnung der Begriffe und Zusammenhänge unterschiedlicher Normen und Standards

Grundsätzlich unterscheiden die vorgestellten Ansätze zur Anforderungsklassifizierung in zwei Sichtweisen: **PMBOK** und **BABOK** unterscheiden Anforderungen nach Abstraktionslevel während **SWEBOK**, **SEBOK** und **ISO29148** primär nach der Perspektive der Stakeholder gruppieren. Das Schaubild 5.2 stellt diesen Zusammenhang grafisch dar.

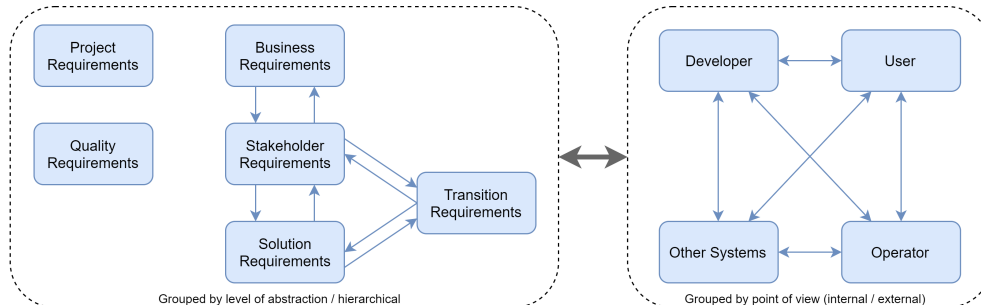


Abbildung 5.2: Anforderungen werden nach zwei verschiedenen Ansätzen gruppiert.

## 5.2 ABLEITUNG EINES KLASSIFIZIERUNGSMODELLS

Die Vorteile mehrerer der vorgestellten Ansätze zur Anforderungsklassifizierung können kombiniert werden, indem Anforderungen stufenweise in Unterklassen unterteilt werden. Dabei wird sich der Klassifizierung des [1], [16] bzw. [15] und [14] bedient und die folgende Einordnung erstellt:

**SYSTEM ANFORDERUNGEN** Anforderungen dieser Klasse beziehen sich auf das Gesamtsystem als solches.

**SOFTWARE ANFORDERUNGEN** Anforderungen dieser Klasse beziehen sich auf die Softwarekomponente des Gesamtsystems.

**BUSINESS ANFORDERUNGEN** Diese Anforderungen gehören zu der Klasse der System-Anforderungen und beschreiben Anforderungen, die sich an das Geschäftsmodell hinter dem Gesamtsystem richten. Der Scherpunkt liegt auf dem Mehrwert für die Organisation und dem damit verbundenen Nutzen des Gesamtsystems.

Leitfrage: „Welche Geschäftsfälle gibt es und wie werden diese abgedeckt? Welche Richtlinien und Vorgaben müssen beachtet werden?“

**STAKEHOLDER ANFORDERUNGEN** Diese Anforderungen gehören zu der Klasse der System-Anforderungen und beschreiben Anforderungen, die die Interessen der beteiligten Stakeholder widerspiegeln und sich keiner anderen Klasse zuordnen lassen.

Leitfrage: „Was muss das Gesamtsystem aus Sicht von [Stakeholder] können?“

**TRANSITIONSANFORDERUNGEN** Diese Anforderungen gehören zu der Klasse der System-Anforderungen und beschreiben den Übergang vom IST-Zustand des Systems in den SOLL-Zustand. Beispiele hierfür sind benötigte Anwenderschulungen oder Datenkonvertierungen.

Leitfrage: „Was muss gegeben sein, damit sich das Gesamtsystem von Zustand A in den Zustand B überführen lässt? “

**PROJEKT ANFORDERUNGEN** Diese Anforderungen gehören zu der Klasse der System-Anforderungen und beschreiben die Rahmenbedingungen an das Entwicklungsprojekt. Beispiele hierfür können die Projektsprache und Dokumentationsrichtlinien sein.

Leitfrage: „Welche Rahmenbedingungen sind dem Entwicklungsprojekt gegeben? “

**QUALITÄTSANFORDERUNGEN** Als Unterklasse der System-Anforderungen beschreiben die Qualitätsanforderungen die Qualitätsansprüche an das System und die Entwicklung und definieren Akzeptanzkriterien ähnlich zu Definition of Ready (DoR) bzw. Definition of Done (DoD).

Leitfrage: „Welche Qualitätsansprüche werden an das Gesamtsystem gestellt? “

**NICHT-FUNKTIONALE ANFORDERUNGEN** Diese Anforderungen werden gemäß ISO-Norm 25010 zur Software-Qualität definiert und sind Teil der Software-Anforderungen. Dazu zählen zum Beispiel die Performanz, die Kompatibilität und die Benutzbarkeit. Eine ausführliche Auflistung aller Klassen unter dem Sammelbegriff der nicht-funktionalen Anforderungen sowie die genauen Definitionen der Begriffe kann unter [14] eingesehen werden.

Leitfrage: „Wie gut muss die Software etwas können? “

**FUNKTIONALE ANFORDERUNGEN** Diese Unterklasse der Software-Anforderungen beschreibt, was das Software-System leisten muss und welche Aufgaben es erfüllen muss.

Leitfrage: „Was muss die Software können? “

**PROZESS ANFORDERUNGEN** Als Untergruppe der Software-Anforderungen bündelt diese Klasse alle Anforderungen, die den Prozess beschreiben, damit die Software so wird wie gefordert. Typischerweise sind Anforderungen an den Softwareentwicklungsprozess enthalten.

Leitfrage: „Was ist beim Entwickeln der Software zu beachten? “

Die Abbildung 5.3 fasst die Anforderungstypen zusammen und stellt sie hierarchisch strukturiert dar. Es wird deutlich, dass das entwickelte Modell beide Sichtweisen (vgl. Abbildung 5.2) aufgreift. Auf Seite der System-Anforderungen werden verschiedene Abstraktionslevel wie zum Beispiel die Business-Anforderungen und die Stakeholder-Anforderungen unterschieden. Stakeholder-Anforderungen wiederum spiegeln die Interessen der Beteiligten wider und betrachten die Anforderungen zusammen mit den Software-Anforderungen aus unterschiedlichen Perspektiven.

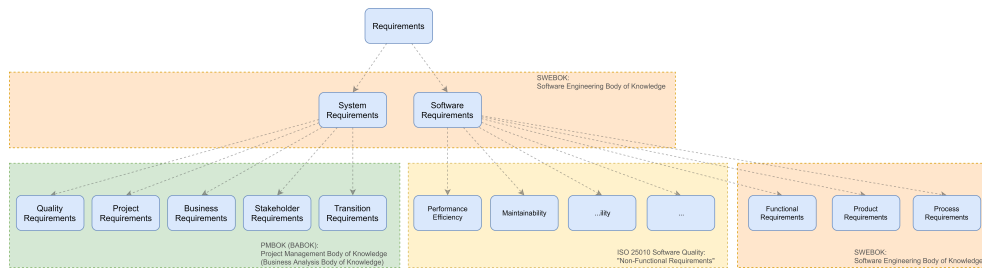


Abbildung 5.3: Anforderungsklassifizierung als kombiniertes Modell aus [1], [16] bzw. [15] und [14]

### 5.3 ANFORDERUNGSANALYSE

In diesem Abschnitt werden die Anforderungen zu dem in Kapitel 4 vorgestellten Anwendungsfall ermittelt. Dazu werden zunächst alle beteiligten Stakeholder identifiziert und kurz beschrieben. Anschließend werden aus der jeweiligen Perspektive heraus User-Stories gebildet und die daraus abgeleiteten Tasks aufgelistet. Eine detaillierte Beschreibung aller daraus resultierenden Anforderungen sowie die genaue Zuordnung zu den User-Stories sind im Anhang A zu finden.

Die Rollen Hersteller, Kunde, Service-Dienstleister und Lieferant wurden bereits unter 4.1 beschrieben. Die folgende Auflistung zeigt weitere Anforderungsrollen, die über die bereits genannten hinausgehen:

**PLATFORM-BETREIBER** Der Plattform-Betreiber ist verantwortlich für den Betrieb der Plattform und ist hauptsächlich an einem stabilen System und einer einfachen Wartung der Software interessiert.

**IT-SECURITY-BEAUFTRAGTER** Für den IT-Security-Beauftragten stehen alle sicherheitsrelevanten Themen im Fokus. Dazu zählen insbesondere Verschlüsselung, Datensicherheit und Datenschutz sowie die Authentizität von Daten.

**BUSINESS-DEVELOPER** Der Business-Developer beschäftigt sich mit der Unternehmensentwicklung und hat Anforderungen an das Geschäftsmodell, die Wirtschaftlichkeit und die Zielerreichung des Gesamtsystems.

**SYSTEM-ARCHITEKT** Für den Software-Architekten stehen alle Fragen rund um die IT-Architektur der Plattform im Vordergrund. Dazu zählen APIs, Modularisierung und der generelle Aufbau der Software.

Um konkrete Anforderungen zu erstellen, werden Userstories aus der Perspektive jedes Stakeholders erarbeitet. Userstories beschreiben eine Funktionalität des Systems, welche aus Sicht der jeweiligen Rolle benötigt wird, um ein bestimmtes Ziel zu erreichen oder einen bestimmten Zweck zu erfüllen. Somit haben alle Userstories die einheitliche Grundstruktur: „Als [Rolle] möchte ich [Funktion], um [Ziel / Zweck]“. Zu jeder User-Story werden Tasks beschrieben, die die User-Stories in logische Teile untergliedern. Im

letzten Schritt werden diese Tasks feingranular in Anforderungen unterteilt. Die Klassifizierung wird anschließend nach dem erarbeiteten Modell aus 5.2 durchgeführt.

Die erste Userstory (A1) fasst grundlegende Tätigkeiten über das Agieren auf der Plattform als Akteur (Hersteller, Kunde, Service-Dienstleister, Lieferant, Endgerät) zusammen. Dies beinhaltet sämtliche Interaktionen und Bedingungen, die für alle Akteure gleich sind. Die Userstory A1 beinhaltet vier Tasks:

- Die Zugangsberechtigung zur Plattform ist in fünf Anforderungen untergliedert, vier davon sind als *Funktionale Anforderungen* klassifiziert und eine als *Security Anforderung*.
- Die Kommunikation zwischen Rollen ist durch fünf Anforderungen definiert und beinhaltet eine *Funktionale Anforderung*, drei *Security Anforderungen* und eine *Performance-Efficiency Anforderung*.
- Die Vertragsgestaltung ist in elf Tasks untergeleitet, zwei davon sind *Security Anforderungen* und neun *Funktionale Anforderungen*.
- Grafische Oberflächen, die alle Akteure zur Interaktion mit der Plattform benötigen, sind in zwei *Funktionale Anforderungen* unterteilt.

Die Userstories M1 bis M3 sind aus Sicht eines Manufacturers beschrieben:

- Userstory M1 beschreibt die Vermietung von Endgeräten und beinhaltet zwei Tasks mit insgesamt sechs Anforderungen. Fünf der sechs Anforderungen sind *Funktionale Anforderungen*, eine wurde als *Portability Anforderung* klassifiziert.
- Userstory M2 beschreibt das Erzeugen von Verträgen und beinhaltet einen Task mit drei Anforderungen, welche alle der Klasse der *Funktionalen Anforderungen* zugeordnet wurden.
- Userstory M3 beschreibt die Abrechnung von Verträgen und beinhaltet zwei Tasks mit insgesamt drei Anforderungen, wobei eine Anforderung als *Funktionale Anforderung*, eine als *Performance-Efficiency Anforderung* und eine als *Security Anforderung* bestimmt wurden.

Die Userstories C1 bis C3 sind aus Sicht des Customers beschrieben:

- Userstory C1 beschreibt die Ansicht verfügbarer Geräte, also eine Oberfläche, die dem Customer bereitgestellt werden muss, auf der er alle zur Miete verfügbaren Geräte gelistet bekommt. Diese Userstory beinhaltet zwei Tasks mit insgesamt zwei *Funktionalen Anforderungen*.
- Userstory C2 beschreibt die Wartung und Reinigung der Geräte durch den Customer und beinhaltet vier Tasks mit insgesamt sechs Anforderungen. Dabei handelt es sich um vier *Funktionale Anforderungen*, eine *Security Anforderung* und eine *Performance-Efficiency Anforderung*.

- Userstory C3 beschreibt die Bedienbarkeit aus Nutzersicht und beinhaltet zwei Tasks mit insgesamt drei Anforderungen. Dabei handelt es sich um eine *Transition Anforderung* und zwei *Usability Anforderungen*.

Die Userstories SP1 und SP2 sind aus Sicht des Service-Providers beschrieben:

- Userstory SP1 beschreibt das Anbieten eigener Service-Dienstleistungen auf der Plattform mit einem Task und zwei *Funktionalen Anforderungen*.
- Userstory SP2 beschreibt das Abschließen von Service-Aufträgen nach getätigtem Service an den Geräten vor Ort und ist unterteilt in zwei Tasks mit je zwei *Funktionalen Anforderungen*.

Die Userstories SEC1 bis SEC3 sind aus Sicht des Security-Beauftragten beschrieben:

- Userstory SEC1 beschreibt die sichere Zahlungsabwicklung und ist in zwei Tasks mit insgesamt fünf Anforderungen unterteilt. Dabei handelt es sich um zwei *Funktionale Anforderungen* und drei *Security Anforderungen*.
- Userstory SEC2 beschreibt die sichere Kommunikation mittels signierter Nachrichten und untergliedert sich in zwei Tasks mit insgesamt zwei *Quality Anforderungen*.
- Userstory SEC3 beschreibt die Manipulationssicherheit und ist in zwei Tasks mit insgesamt drei *Security Anforderungen* unterteilt.

Die Userstories BD1 bis BD4 sind aus Sicht des Business-Developers beschrieben:

- Userstory BD1 beschreibt das Geschäftsmodell und beinhaltet zwei Tasks mit insgesamt sechs Anforderungen, wobei fünf davon als *Funktionale Anforderungen* und eine als *Business Anforderung* klassifiziert wurden.
- Userstory BD2 beschreibt den Zugang von Geschäftspartnern zu der Plattform und ist in zwei Tasks mit insgesamt drei Anforderungen untergliedert. Diese wurden als *Business Anforderung*, *Transition Anforderung* und *Stakeholder Anforderung*.
- Userstory BD3 beschreibt die Plattform als Hersteller-übergreifend und ist in drei Tasks mit insgesamt drei Anforderungen unterteilt. Dabei wurde eine Anforderung als *Business Anforderung* und zwei als *Compatibility Anforderungen* klassifiziert.
- Userstory BD4 beschreibt die Abrechnungsmodelle der Plattform und ist in einen Task mit zwei Anforderungen unterteilt. Dabei handelt es sich um eine *Business Anforderung* und einer *Maintainability Anforderung*.



Die Userstories SA1 und SA2 sind aus Sicht des System-Architekten beschrieben:

- Userstory SA1 beschreibt die einfache Einbindung der Plattform in bestehende Infrastruktur und ist in einen Task mit einer *Process Anforderung* unterteilt.
- Userstory SA2 beschreibt die Robustheit des Gesamtsystems bei Ausfällen und ist in zwei Tasks mit je zwei Anforderungen unterteilt. Dabei handelt es sich um zwei *Security Anforderungen* und zwei *Reliability Anforderungen*.

Die Userstory P1 ist aus Sicht des Plattform-Betreibers beschrieben:

- Userstory P1 beschreibt die automatisierte Bereitstellung der Software und ist in zwei Tasks mit je einer Anforderung untergliedert. Diese sind als *Maintainability Anforderung* und als *Portability Anforderung* klassifiziert.

Insgesamt wurden 19 Userstories beschrieben, die in 39 Tasks unterteilt wurden. Die 83 daraus resultierenden Anforderungen wurden klassifiziert, sodass sich 9 Level-1 *System Anforderungen* und 74 Level-1 *Software Anforderungen* ergaben. Für jede Level-2 Klasse als Unterklasse der *System Anforderungen* wurden Anforderungen ermittelt bis auf die Klasse der Projekt-Anforderungen. Im Rahmen dieser Arbeit wird ein PoC entwickelt, womit diese Arbeit aus Sicht der Anforderungserstellung als Entwicklungsprojekt angesehen werden kann. Rahmenbedingungen wie die zeitliche Begrenzung des Projektes oder den Reifegrad eines PoCs könnten als Projekt-Anforderungen definiert werden. Da dies aber keine konkrete Auswirkung auf den Anwendungsfall als solchen hat, wird an dieser Stelle auf diese Klasse verzichtet. Bei der Entwicklung eines marktreifen Produktes in einem realen Entwicklungsprojekt ist diese Klasse allerdings zu beachten.

Auf Seiten der Software-Anforderungen wurden alle Level-2 Klassen bis auf die *Functionality-Suitability Anforderungen* abgedeckt. Bei dieser Klasse handelt es sich um Funktionale Anforderungen, welche bereits durch eine eigene Klasse abgedeckt werden.

## 5.4 ANFORDERUNGSEVALUIERUNG

Die Anforderungsevaluierung hat zum Ziel, die im vorherigen Abschnitt beschriebenen und klassifizierten Anforderungen schrittweise zu reduzieren, um die DLT-relevanten Anforderungen zu identifizieren. Dabei handelt es sich um Anforderungen, die relevant für eine technische Umsetzung auf Basis einer DLT-Lösung sind.

Um die Relevanz für den Kontext DLT festzustellen, wurde die Anforderungsliste mit den Fachexperten der Abteilung für Distributed Ledger Technologies der Firma MaibornWolff GmbH in mehreren Diskussionsrunden durchgearbeitet und nach Einschätzung der Experten entsprechend gewertet.



#### 5.4.1 System Anforderungen

Die erste Level-2 Subklasse der *System Anforderungen* ist die Klasse der *Quality Anforderungen*. Diese Klasse definiert die Qualitätskriterien, die die Plattform erfüllen muss. Anforderungen dieser Klasse fungieren oft als Enabler für weitere Anforderungen anderer Klassen. Für den beschriebenen IOT-Anwendungsfall wurden zwei Anforderungen identifiziert, die dieser Klasse zuzuordnen (Anforderungen SEC2.1.1, SEC2.2.1) und Teil der Userstory SEC2 sind. Sie beschreiben, dass die Verwendung von HTTPS bzw. SSL/TLS ein vorgeschriebenes Qualitätskriterium ist. Darüber hinaus müssen Passwortregeln hinterlegbar sein, um die Sicherheit der auf der Plattform verwendeten Passwörter zu gewährleisten. Es wird deutlich, dass die Anforderungen dieser Klasse primär Rahmenbedingungen darstellen und keine direkten Auswirkungen auf die technische Basis haben. Diese Qualitätsansprüche, die mit den genannten Anforderungen einhergehen, haben keine Auswirkung auf eine mögliche Umsetzung der Plattform durch eine DLT-Lösung. Damit werden die Anforderungen im weiteren Verlauf nicht tiefergehend betrachtet.

Die zweite Level-2 Subklasse der *System Anforderungen* ist die Klasse der *Project Anforderungen*, die Rahmenbedingungen an das Projekt beschreiben. Die Entwicklung des PoC im Rahmen dieser Arbeit stellt ein solches Entwicklungsprojekt dar, ist allerdings für die Betrachtung in diesem Kontext hinsichtlich DLT-Relevanz nicht weiter zu berücksichtigen. Somit werden die *Project Anforderungen* nicht in die weitere Analyse miteinbezogen.

Die dritte Level-2 Subklasse der *System Anforderungen* ist die Klasse der *Business Anforderungen*, die die Geschäftsfälle und -anforderungen von einer abstrakteren Perspektive betrachten. Im Fokus stehen die Bedürfnisse und Rahmenbedingungen des Unternehmens, welches die Plattform beauftragt hat. Im Rahmen des IOT-Anwendungsfalls wurden vier Anforderungen dieser Klasse identifiziert (Anforderungen BD1.1.1, BD2.1.1, BD3.1.1 und BD4.2.1); betroffen sind die Userstories BD1 bis BD4. Die Anforderungen decken das Abrechnungsmodell Pay-As-You-Use ab und beschäftigen sich mit der aktuellen und zukünftigen geschäftlichen Ausrichtung der Plattform. Dies hat keine Auswirkungen auf die technische Basis, die die zugrundeliegende Plattform verwendet: Es handelt sich um keine Technologieentscheidende Anforderung. Anforderung BD4.2.1 beschreibt, dass ein Abrechnungsmodell (konkret: Pay-As-You-Use) in einem Vertrag abgebildet wird. Diese Anforderung ist DLT-relevant: Zum einen impliziert diese Anforderung, dass eine zugrundeliegende DLT-Plattform in der Lage ist, Verträge abzubilden. Im Umfeld von DLTs redet man Smart-Contracts: Code, der auf onchain ausgeführt wird und eine Vertragslogik widerspiegeln kann. Zum anderen muss gewährleistet sein, dass die Smart-Contract Implementierung mächtig genug ist, damit das Abrechnungsmodell Pay-As-You-Use dort codiert werden kann. Diese Anforderung muss in die weitere Analyse mitein-

bezogen werden.

Die vierte Level-2 Subklasse der *System Anforderungen* ist die Klasse der *Stakeholder Anforderungen*, welche Anforderungen speziell aus der Sicht einzelner Stakeholder beschreiben, die mit anderen Klassen noch nicht abgedeckt werden konnten. Im Kontext des Anwendungsfalls wurde eine Anforderung identifiziert (BD2.2.1), die dieser Klasse zuzuordnen ist und zur Userstory BD2 gehört. Es wird der Onboarding-Prozess eines Geschäftspartners beschrieben, indem dieser als Partner identifiziert werden muss. Dieser Prozess muss entsprechend der Anforderungen gestaltet werden, womit es sich um eine abstrakte Beschreibung dessen, wie ein Prozess auszusehen hat, handelt. Es werden keine technischen Details gefordert, wodurch keine Abhängigkeiten zu der technischen Umsetzung der Plattform entstehen. Damit ist diese Klasse für die weitere Analyse nicht relevant und kann vernachlässigt werden.

Die fünfte Level-2 Subklasse der *System Anforderungen* ist die Klasse der *Transition Anforderungen*, die sämtliche Anforderungen von einem IST-Zustand (zeitlich vor der Einführung der Plattform) in einen SOLL-Zustand (zeitlich nach Einführung der Plattform) kapseln. Zwei Anforderungen (Anforderungen C3.1.1 und BD2.1.2) wurden identifiziert, die dieser Klasse zuzuordnen und Teil der Userstories C3 und BD2 sind. Es handelt sich in dem vorliegenden Kontext um Anforderungen, die die schnelle Erlernbarkeit durch den Benutzer sowie die Schulung von Mitarbeitern zur Nutzung der Plattform beschreiben und somit die User-Experience (UX) in den Vordergrund stellen. Da es sich in diesem Fall um Aufbau, Verständlichkeit und Benutzbarkeit einer grafischen Oberfläche (Schnittstelle zum Benutzer) handelt, kann diese Klasse im weiteren Verlauf der Analyse vernachlässigt werden.

#### 5.4.2 Software Anforderungen

Anforderungen dieser Level-1 Klasse stellen den Großteil aller Anforderungen dar. In einem realistischen Entwicklungsprojekt sind die individuellen Rahmenbedingungen, Qualitätskriterien, Business-Richtlinien und Integrationsrichtlinien des jeweiligen Unternehmens zu beachten. Die in dieser Arbeit aufgestellten System-Anforderungen decken nur die grundlegendsten Anforderungen dieser Kategorie ab. Die *Software Anforderungen*, die in diesem Abschnitt evaluiert werden, sind unabhängig von den *System Anforderungen* stets dieselben.

Die erste Level-2 Subklasse der *Software Anforderungen* sind die *Process Anforderungen*, die Anforderungen an den Entwicklungsprozess der Software stellen. Für den vorliegenden IOT-Anwendungsfall wurde eine Anforderung (SA1.1.1) ermittelt, die zu der Userstory SA1 gehört. Während der Entwicklung ist darauf zu achten, dass die Modularität der Softwarekomponenten

gewahrt bleibt, damit diese später unabhängig voneinander bereitgestellt und gewartet werden können. Dies hat keine Auswirkung auf die Umsetzung auf Basis einer DLT-Lösung. Im Allgemeinen haben Anforderungen dieser Klasse einerseits eine technische Relevanz, da sie zum Beispiel einzusetzende Technologien für Schnittstellen beschränken oder die Art und Weise definieren, wie Software entwickelt werden soll. Andererseits kann jede Software durch den Einsatz einer geeigneten Middleware miteinander verbunden werden, sollten vorhandene Schnittstellen und Softwarekomponenten nicht Standard-konform oder kompatibel sein. Somit können die Anforderungen dieser Klasse ebenfalls vernachlässigt werden.

Die zweite Level-2 Subklasse der *Software Anforderungen* sind die *Funktionalen Anforderungen*, die beschreiben, welche Funktionalität die Plattform anbieten muss. Für den vorliegenden IOT-Anwendungsfall wurden 46 solcher Anforderungen ermittelt, die insgesamt zehn Userstories betreffen. Um die Übersichtlichkeit zu bewahren, wurde diese Klasse in verschiedene, logische Abschnitte gegliedert. Diese wurden nach der inhaltlichen Thematik der Anforderungen definiert und folgen keiner speziellen Klassifizierung.

GUI 12 der 46 Anforderungen dieser Klasse betreffen direkt oder indirekt die grafische Darstellung für den Benutzer. Es handelt sich hierbei lediglich um das Frontend, welches Daten aufbereitet darstellt und Schaltflächen zur Interaktion mit dem Backend bereitstellt. Somit werden die Anforderungen dieser Klasse für weiterführende Analysen nicht beachtet. Konkret handelt es sich um die Anforderungen A1.1.4, A1.3.4, A1.4.1, A1.4.2, M2.1.1, M2.1.2, M2.1.3, C1.1.1, C1.2.1, SP1.1.1, SP1.1.2 und SP2.2.1.

ENDGERÄT 15 der 46 Anforderungen beziehen sich auf Funktionalitäten, die das Endgerät bereitstellen muss, um auf der Plattform vermietet werden zu können. Dabei geht es primär um die Konnektivität zu der Plattform, die Funktionsweise der verbauten Sensoren sowie der Kommunikation mit dem Customer und dem Service-Provider. Die Endgeräte fungieren in ihrer Rolle als Peripherie; es kann sich um Haushaltsgeräte aller Art handeln - von der Kaffeemaschine bis zur Waschmaschine. Die Schnittstellen hin zur Plattform sind klar definiert. Auf der einen Seite wird deutlich, dass die meisten Anforderungen an die Geräte unabhängig von der technischen Umsetzung der Plattform sind und keinen Einfluss auf deren Umsetzung haben. Bei Inkompatibilität eines Endgeräts zur Plattform wäre der Einsatz einer Middleware sinnvoll, die die Integration zur Plattform sicherstellen könnte. Eine Möglichkeit wäre die Verwendung eines lokalen Gateways, das die Daten der Geräte konvertiert und an die Plattform übermittelt. Standardmäßig sollten die Geräte die Kommunikation mit der Plattform bereits Hersteller-seitig gewährleisten. Auf der anderen Seite spielt gerade die Konnektivität eine ganz entscheidende Rolle: Die Tatsache, dass Endgeräte nicht jederzeit mit der Plattform verbunden sind - bedingt

durch Verbindungsprobleme oder Ähnliches - stellt eine zentrale Anforderung an die zugrundeliegende Plattform dar. Um auch im Offline-Modus mit seinem Umfeld interagieren zu können, zum Beispiel um einen Kaffee zu servieren, muss das Gerät zeitnah auf Benutzereingaben reagieren, da ansonsten ein schlechtes Benutzererlebnis entstünde. Würde das Gerät erst dann reagieren, wenn es alle benötigten Informationen von der Plattform erhalten hat, also nachdem es wieder eine Verbindung aufbauen konnte, wäre der Anwendungsfall inpraktikabel und hätte keine wirtschaftliche Daseinsberechtigung. Deshalb muss es möglich sein, dass das Gerät voll funktionsfähig arbeitet, auch wenn es keine Konnektivität zur Plattform hat. Anlaufende Kosten und Informationsengpässe müssen handhabbar sein und zu einem späteren Zeitpunkt synchronisiert werden können. Diese Anforderung hat eine sehr hohe **DLT**-Relevanz und muss für die folgende Analyse genauestens berücksichtigt werden. (Anforderungen M1.1.1, M1.1.2, M1.1.3, M1.2.1, M1.2.2, M1.2.3, M1.2.4, M3.1.1, C2.1.1, C2.1.2, C2.3.1, C2.4.1, SP2.1.1, SP2.1.2 und SP2.2.2)

**KOMMUNIKATION** Drei der 46 Anforderungen beziehen sich auf die Kommunikation zwischen Akteuren auf der Plattform (Anforderungen A1.2.1, A1.3.1 und A1.3.3). Anforderung A1.3.1 beschreibt die Kommunikation über Verträge, die die Akteure der Plattform miteinander abschließen können. Im Kontext einer **DLT**-Lösung bedeutet das, dass die Plattform in der Lage sein muss, einen Vertrag abzubilden (siehe Business Anforderungen oben). Diese Anforderung hat eine **DLT**-Relevanz und muss in der weiteren Analyse betrachtet werden. Die übrig gebliebenen zwei Anforderungen beschreiben die allgemeinere Aspekte der Kommunikation auf der Plattform und können vernachlässigt werden.

**FINANZEN** Sieben der 46 Anforderungen (A1.3.11, SEC1.1.1, SEC1.1.3, BD1.2.1, BD1.2.2, BD1.2.3 und BD1.2.5) beschreiben den Geldfluss zwischen Akteuren aufgrund ihrer vertraglichen Vereinbarungen. Geldtransfers werden geloggt und vor Ausführung überprüft. Diese Anforderungen sind für eine Umsetzung auf einer **DLT**-Lösung nicht relevant, da sie lediglich die Richtung und Menge des Geldflusses sowie Rahmenbedingungen an Geldtransfers festlegen. Damit allerdings erbrachte Leistungen kostenpflichtig gemäß des Abrechnungsmodells abgerechnet werden können, muss die Plattform zum einen das Abrechnungsmodell implementieren und zum anderen ein digitales Zahlungsmittel bereitstellen. Ersteres kann auf einer **DLT**-Lösung mittels Smart-Contracts umgesetzt werden. Zweiteres bedarf einer internen Währung, um Leistungen in Echtzeit nach Verbrauch abzurechnen. Damit sind diese zwei Anforderungen relevant und müssen in der weiteren Analyse beachtet werden.

**ROLLENMANAGEMENT** Drei der 49 Anforderungen (A1.1.2, A1.1.3 und A1.1.5) definieren das Rollenmanagement und den Zugang zu der Plattform mittels Registrierung und Anmeldung. Letzteres stellt keine Relevanz

dar, da es sich um eine standardmäßige Zugangsbeschränkung handelt und nicht abhängig von der Backend-Lösung ist. Im Gegensatz dazu werden Rollen in den Verträgen genutzt, um Berechtigungen der Akteure zu prüfen. Es handelt sich um Informationen, die von Verträgen auf der Plattform einsehbar sein müssen. Content- oder allgemeiner Informationsprovider, die Informationen auf einer [DLT](#)-Umgebung bereitstellen, nennt man Oracles (siehe Kapitel XYZ). Damit liegt hier eine Relevanz in Bezug auf die Umsetzung auf Basis einer [DLT](#)-Lösung vor und muss im weiteren Verlauf beachtet werden.

**VERTRAGSKONSTRUKT** Sechs der 49 Anforderungen (A1.3.2, A1.3.6, A1.3.7, A1.3.8, A1.3.10 und BD1.2.4) beschreiben das Vertragskonstrukt: Eigenschaften wie Individualität und Zugriffsregelung haben keinen Einfluss auf die technische Lösung, wohingegen die Komplexität und Editierbarkeit (drei Anforderungen) große Relevanz haben. Zum einen muss die Implementierung eines Vertrages mittels Smart-Contracts auch komplexe Konstrukte abbilden können. Zum anderen sind Smart-Contracts - sind sie einmal in der [DLT](#) gespeichert - unveränderbar und können als solches nicht überarbeitet werden. Die zugrundeliegende [DLT](#)-Technologie muss also einen entsprechenden Mechanismus anbieten, um Smart-Contracts zu warten und zu aktualisieren. Darüber hinaus handelt es sich bei den Verträgen, die mittels Smart-Contracts abgebildet werden sollen, um rechtskräftige Miet- und Serviceverträge. Demnach müssen die Implementierungen rechtssicher und rechtskonform abgebildet werden können. Diese Punkte sind bei der Wahl der [DLT](#)-Lösung zu beachten.

Die dritte Level-2 Subklasse der *Software Anforderungen* unter dem Sammelbegriff der *Nicht-Funktionalen Anforderungen* heißt *Kompatibilität* (Compatibility). Für den vorliegenden IOT-Anwendungsfall wurden zwei solcher Anforderungen ermittelt (BD3.2.1, BD3.3.1), welche die Userstory BD3 betreffen. Die Kompatibilität stellt die Fähigkeit des Gesamtsystems dar, Informationen mit anderen System auszutauschen und sich eine Umgebung mit anderen Systemen zu teilen. Es handelt sich hierbei um Anforderungen, die im Kontext des [IOT](#)-Anwendungsfalls keine Relevanz für die Wahl der technischen Basis haben. Sollten Systeme nicht kompatibel sein, so könnte im Zweifelsfall eine Middleware für die Übersetzung bzw. die Kompatibilität sorgen. Somit werden die Anforderungen dieser Klasse für die weitere Analyse nicht weiter beachtet.

Die vierte Level-2 Subklasse der *Software Anforderungen* unter dem Sammelbegriff der *Nicht-Funktionalen Anforderungen* heißt *Wartbarkeit* (Maintainability). Anforderungen dieser Klasse stellen die Fähigkeit des Gesamtsystems dar, effizient wartbar zu sein um z.B. die Funktionalität zu erweitern und zu verbessern. Konkret im Kontext des [IOT](#)-Anwendungsfalls beziehen sich die Anforderungen BD4.2.2 und P1.1.1 der zwei Userstories BD4 und P1 auf den Aufbau der Software: Einzelne Module können separat gewartet werden und die Testabdeckung ist hoch genug, damit die Wartung einzel-

ner Module keine unerwünschten Nebeneffekt mit sich führt. Es handelt sich also um generische Anforderungen, die keinen Bezug auf die technische Umsetzung haben und daher keine DLT-Relevanz besitzen. Somit wird diese Anforderungsklasse im weiteren Verlauf nicht weiter berücksichtigt.

Die fünfte Level-2 Subklasse der *Software Anforderungen* unter dem Sammelbegriff der *Nicht-Funktionalen Anforderungen* heißt *Performance-Effizienz* (Performance-Efficiency) und beschreibt die Leistung des Gesamtsystems in Bezug auf die zur Verfügung stehenden Ressourcen. Die Anforderungen dieser Klasse (A1.2.2, M3.2.1 und C2.4.2) im konkreten Anwendungsfall betreffen drei Userstories (A1, M3 und C2) und beschreiben alle das zeitliche Verhalten von übermittelten Daten: Die Kommunikation zwischen Akteuren, Geräten und Verträgen auf der Plattform wird sofort übermittelt. Es werden keine Daten zurückgehalten, aggregiert oder zeitverzögert übermittelt. Diese Anforderungen treffen keine Aussage über die Verarbeitungsdauer der Daten und haben damit keine Relevanz in Bezug auf die Wahl der technischen Basis. Somit werden die Anforderungen dieser Klasse für die weitere Analyse ignoriert.

Die sechste Level-2 Subklasse der *Software Anforderungen* unter dem Sammelbegriff der *Nicht-Funktionalen Anforderungen* heißt *Portabilität* (Portability) und beschreibt die Fähigkeit des Gesamtsystems von einer Hardware bzw. Umgebung in eine andere migriert zu werden. In diesem Kontext existiert eine Anforderung (P1.2.1) der Userstory P1, welche die automatisierte Installation und Bereitstellung der Software definiert. Es ist deutlich, dass es sich hierbei um keine Technologie-entscheidende Anforderung handelt. Demnach wird diese Anforderungsklasse in der weiteren Analyse nicht beachtet.

Die siebte Level-2 Subklasse der *Software Anforderungen* unter dem Sammelbegriff der *Nicht-Funktionalen Anforderungen* heißt *Ausfallsicherheit* (Reliability) und beschreibt wie gut ein System unter bestimmten Bedingungen die geforderten Funktionalitäten durchführen kann. Anforderung SA2.2.2 beschreibt, dass die Plattform keinen Single-Point-of-Failure (SPoF) besitzen darf. Dies beschreibt eine generelle Eigenschaft und ist im Kontext eines dezentralen Systems wie es DLT ist nicht weiter von Relevanz. Daneben fordert Anforderung SA2.2.1, dass die Plattform in der Lage ist, bis zu 10.000 Endgeräte zu verarbeiten, ohne Einbußen in der Funktionalität oder der Geschwindigkeit der Verarbeitung. Diese Anforderung muss bei der Wahl der zugrundeliegenden DLT-Lösung beachtet werden, da hierbei die Performanz und Verfügbarkeit des beeinträchtigt wird.

Die achte Level-2 Subklasse der *Software Anforderungen* unter dem Sammelbegriff der *Nicht-Funktionalen Anforderungen* heißt *Sicherheit* (Security) und befasst sich mit der Thematik rund um Software- und Datensicherheit. Diese Klasse beinhaltet 16 Anforderungen (A1.1.1, A1.2.3, A1.2.4, A1.2.5, A1.3.5, A1.3.9, M3.2.2, C2.2.1, SEC1.1.2, SEC1.2.1, SEC1.2.2, SEC3.1.1, SEC3.1.2, SEC3.2.1, SA2.1.1 und SA2.1.2) aus sechs Userstories (A1, M3, C2, SEC1, SEC3 und SA2). Die vorliegenden Anforderungen lassen sich grob ob vier Subkategorien untergliedern: Verantwortlichkeit, Authentizität, Vertraulichkeit und



Integrität. Verantwortlichkeit beinhaltet fünf Anforderungen, die unter anderem die eindeutige Identifikation der Akteure sowie die Protokollierung von Aktivitäten und deren Zuordnung zu Accounts beschreiben. Während es sich bei dem Großteil um allgemeine Richtlinien handelt und dieser nicht von der konkreten technischen Umsetzung abhängig ist, so ist die generelle Identifikation von großer Relevanz für die Umsetzung auf Basis einer DLT-Lösung. Um einem Account (im Kontext von DLT auch Wallet) eine natürliche Person oder ein Unternehmen eindeutig zuordnen zu können, bedarf es einer Instanz, die auf der Plattform agiert und diese Informationen bereitstellt. Eine mögliche Umsetzung im Kontext DLT wäre die Implementierung eines entsprechenden Oracles, welches Personen und Unternehmen einer Wallet zuordnet und umgekehrt.

Die nächste Subkategorie - Authentizität - beinhaltet eine Anforderung, die die Zugriffsbeschränkung zu der Wallets (Konten) beschreibt. Da Wallets auf einer DLT durch eine Kombination aus einem öffentlichen und einem privaten Schlüssel sind und der private der PIN-Nummer eines Kontos entspricht, besteht eine implizite Zugriffsbeschränkung; es liegt hierbei keine DLT-Relevanz vor.

Der Inhalt von Nachrichten zwischen Akteuren sowie der Vertragsinhalt abgeschlossener Verträge unterliegt der Vertraulichkeit, darf also nur von beteiligten bzw. berechtigten Akteuren eingesehen werden. Da Transaktionen auf DLT-Lösungen generell öffentlich einsehbar sind liegt hier eine DLT-Relevanz vor: Es muss dafür gesorgt werden, dass der Inhalt von Transaktionen und Smart-Contracts vertraulich behandelt werden kann.

Die Subkategorie Integrität hat den größten Anteil: Insgesamt acht Anforderungen wurden für den vorliegenden Anwendungsfall ermittelt, wobei Dateninkonsistenzen, Datenverlust und Manipulationssicherheit im Fokus stehen. Diese Eigenschaften entsprechen in etwa dem, was einen Distributed Ledger ausmacht, und sind deshalb zunächst einmal nicht weiter relevant in Bezug auf die Umsetzung mittels eines DLT. Die Abbildung eines Vertrags muss allerdings so gestaltet werden, dass der Vertragsgegenstand nicht manipuliert werden kann. Dies ist bei der Implementierung des Vertrags zu beachten und hat demnach eine Auswirkung auf die DLT-Relevanz und muss in der weiteren Analyse beachtet werden.

Die neunte Level-2 Subklasse der *Software Anforderungen* unter dem Sammelbegriff der *Nicht-Funktionalen Anforderungen* heißt *Benutzbarkeit* (Usability) und befasst sich mit dem Thema UX. Diese Klasse beinhaltet zwei Anforderungen (C3.1.2 und C3.2.1) der Userstory C3 und besitzt keine DLT-Relevanz, da es um die Schnittstelle zum Endbenutzer geht und nicht um die technologische Basis der Plattform. Somit wird diese Klasse im weiteren Vorgehen nicht beachtet.

In den vorherigen Abschnitten wurden alle DLT-relevanten Anforderungen der verschiedenen Klassen identifiziert. Die Tabelle 5.1 fasst diese zusammen.

Im Laufe der Untersuchung auf DLT-Relevanz wurde deutlich, dass die Einteilung nach Anforderungsklassen nur einen geringen Beitrag zur Iden-

ID	Inhalt	Level-1	Level-2
A1.1.1	Jeder Akteur auf der Plattform kann eindeutig identifiziert werden.	Software	Security
A1.1.3	Ein Akteur agiert immer mit einer bestimmten Rolle auf der Plattform: Manufacturer, Customer, Supplier, Service-Provider oder Gerät. Ein Akteur kann mehrere Rollen haben.	Software	Functional
A1.1.5	Ein Akteur hat eine (mehrere) verifizierte Rolle(n).	Software	Functional
A1.2.5	Akteure können nur den Inhalt ihrer eigenen Nachrichten einsehen.	Software	Security
A1.3.1	Akteure schließen Verträge über die Plattform ab.	Software	Functional
A1.3.2	Verträge sind rechtlich bindend.	Software	Functional
A1.3.5	Der Vertragsgegenstand kann nicht durch Dritte manipuliert werden.	Software	Security
A1.3.7	Akteure können komplexe Vertragskonstrukte umsetzen. Verträge haben einen Status. Diese können "Aktiv", "In Erzeugung" oder "Inaktiv" sein.	Software	Functional
A1.3.8	Akteure können ihre Verträge im Nachhinein ändern.	Software	Functional
A1.3.9	Akteure können nur ihre eigenen Verträge ändern.	Software	Security
A1.3.10	Eine Vertragsänderung bedarf der Zustimmung aller beteiligten Akteure.	Software	Functional
A1.3.11	Erbrachte Leistungen werden kostenpflichtig verrechnet.	Software	Functional
M1.1.3	Geräte sind nicht immer mit der Plattform verbunden.	Software	Functional
BD4.2.1	Ein Abrechnungsmodell wird in einem Vertrag abgebildet.	System	Business
SA2.2.1	Die Plattform ist in der Lage, die Kommunikation und Datenverarbeitung bei bis zu 100.000 Endgeräten durchzuführen.	Software	Reliability

Tabelle 5.1: DLT-relevante Anforderungen



tifikation der relevanten Anforderungen beitragen konnte. Selbst die Einteilung in verschiedene Themenbereiche konnte nur einen kleinen Mehrwert liefern, indem die Übersichtlichkeit - trotz der hohen Anzahl an Anforderungen - gewahrt werden konnte. Mögliche Gründe und Auswirkungen werden später in Kapitel 8 aufgezeigt und diskutiert.

## 5.5 ANFORDERUNGSTRANSFER AUF DLT

Die Gesamtmenge der 84 Anforderungen wurde eingehend analysiert; dabei wurden 15 DLT-relevante Anforderungen identifiziert. Um geeignete DLTs auszuwählen, die für eine prototypische Verprobung in Frage kommen, werden die relevanten Anforderungen im Folgenden in DLT-Eigenschaften übersetzt.

**SMART-CONTRACTS** Acht der DLT-relevanten Anforderungen (A1.3.1, A1.3.2, A1.3.5, A1.3.7, A1.3.8, A1.3.9, A1.3.10 und BD4.2.1) beschreiben das Erstellen, das Ändern und den Aufbau von Verträgen. Damit DLTs Vertragsregeln prüfen und entsprechende Aktionen in die Wege leiten können, setzt dies voraus, dass die Implementierung Smart-Contracts unterstützen.

**ORACLE-SERVICES** Drei der DLT-relevanten Anforderungen (A1.1.1, A1.1.3, A1.1.5) beschreiben die Identifizierung und Verifizierung von Stakeholdern und deren Rollen. Diese Informationen können auf DLTs mittels Oracle-Services publiziert, gepflegt und genutzt werden. Daher ist es notwendig, dass die zugrundeliegende Implementierung Oracle-Services implementiert.

**ZAHLUNGSMITTEL** Die Anforderung A1.3.11 beschreibt die kostenpflichtige Abrechnung erbrachter Leistungen. Übersetzt in die DLT-Welt bedeutet das, dass die Implementierung ein Zahlungsmittel bereitstellen muss, damit erbrachte Leistungen gemäß des Pay-As-You-Use Modells in Echtzeit abgerechnet werden können.

**ASYNCHRONITÄT** Die Anforderung M1.1.3 definiert, dass Endgeräte, bedingt durch Verbindungsprobleme oder Ähnlichem, nicht dauerhaft mit der Plattform verbunden sind. Da eine erbrachte Dienstleistung, wie zum Beispiel eine erzeugte Tasse Kaffee, sofort abgerechnet und bei Nicht-Vorhandensein von Guthaben gar nicht erst erzeugt werden soll, muss die Implementierung eine Form von Asynchronität implementieren. Das bedeutet, dass Transaktionen auch außerhalb des Netzwerkes manipulationssicher und korrekt durchgeführt werden müssen. Dies kann durch die Implementierung von State-Channels erfolgen.

**PERFORMANZ** Anforderung SA2.2.1 definiert die Performanz des Systems nach folgenden Annahmen: Bei durchschnittlich 3 Tassen Kaffee pro Tag und Mitarbeiter und einer Verteilung von 40 Mitarbeitern auf eine

Kaffeemaschine ergibt sich bei einer Auslastung von 10.000 vermieteten Kaffeemaschinen eine benötigte Performanz von 37 Transaktionen pro Sekunde ( $3 * 40 * 10.000 / 32.400 = 37$ ).

**VERSCHLÜSSELUNG** Anforderung A1.2.5 fordert, dass Akteure nur den Inhalt ihrer eigenen Nachrichten einsehen können. Um dies zu ermöglichen, muss der Payload von Transaktionen sowie der Inhalt von Smart-Contracts verschlüsselt und damit nur für beteiligte Parteien einsehbar sein.

## AUSWAHL RELEVANTER DLTS

---

In den vorherigen Kapiteln wurden der beispielhafte Anwendungsfall sowie dessen **DLT**-relevante Anforderungen aufgezeigt. In diesem Kapitel wird darauf aufbauend eine Marktübersicht möglicher **DLTs** gegeben und auf die Erfüllung der Anforderungen überprüft.

### 6.1 VORGEHEN

Es muss ein geeignetes Vorgehen erarbeitet werden, um aus der unüberschaubaren Menge an Blockchain-Lösungen eine passende Untermenge auszuwählen und dabei keine wichtigen Implementierung zu übersehen. Im Folgenden werden vier mögliche Kriterien vorgestellt, nach denen Blockchain-Lösungen bewertet werden können:

**BUSINESS RELEVANCE** Mitte April 2019 veröffentlichte das Wirtschaftsmagazin Forbes einen Artikel [5], der die eingesetzten Blockchain-Technologien großer, internationaler Unternehmen auflistet. Die daraus abzuleitende Relevanz für das internationale Business-Umfeld im Bereich Blockchain stellt einen Indikator da, der bei der Auswahl einer Blockchain-Lösung berücksichtigt werden muss. Es wurden alle Blockchain-Implementierungen berücksichtigt, die mehr als zwei große Unternehmen betreffen.

**GITHUB ACTIVITY** Die Mehrheit der Blockchain-Entwicklungsprojekte sind open-source Projekte; der Quellcode ist meist frei verfügbar und auf Github öffentlich einsehbar. Ein wichtiger Indikator für Auswahl einer **DLT**-Lösung ist die Aktivität der Entwickler auf Github: Es wird bewertet, wie regelmäßig Weiterentwicklungen stattfinden und wie interessiert die Community die Änderungen verfolgen. Je größer die Aktivität und Beliebtheit, desto wahrscheinlicher werden aktuelle Forschungsergebnisse und Neuerungen in die Lösung eingebaut. Um diese Werte vergleichen zu können, wurde sich der Quellen [www.coincodecap.de](http://www.coincodecap.de) und [www.cryptomismo.de](http://www.cryptomismo.de) bedient: Die Anzahl an Commits, aktiven Entwicklern, interessierten Beobachtern und weiteren Kennzahlen wurde gewichtet und bewertet. Das Ergebnis ist ein Ranking der aktivsten Blockchain-Projekte auf Github.

**BLOCKCHAIN ACTIVITY** Die Website [www.blocktivity.info](http://www.blocktivity.info) misst die Aktivität einer Blockchain-Plattform, indem es die Anzahl an Operationen (Transaktionen, Votes, Blogposts, etc.) verschiedener Zeitpunkte mit der Marktkapitalisierung in Relation setzt. Darüber hinaus werden Angaben über die tatsächlich verwendete und die noch verfügbare Kapazitäten der Lösungen gemacht. Diese Angaben helfen, verschiedene Lösungen gegenüberzustellen und Kapazitäten abzuschätzen. Für die

Relevanz im Kontext der Marktübersicht wurden die besten zehn Lösungen hinsichtlich Blockchain-Aktivität berücksichtigt.

**IOT SUITABILITY** Da diese Arbeit von der Synergie zwischen **DLT** und **IOT** handelt, bietet es sich an, das Themenfeld **IOT** bei der Auswahl zu berücksichtigen. Dazu wurden die zehn wertvollsten (nach Marktkapitalisierung) Blockchain-Lösungen, die als **IOT**-Lösung bei der Informationsplattform [www.cryptoslate.com](http://www.cryptoslate.com) geführt werden, berücksichtigt.

Darüber hinaus fokussiert sich diese Arbeit auf eine Auswahl eigenständiger Blockchains; ERC-Token<sup>1</sup> oder ähnliche Implementierungen (Blockchains basierend auf anderen Blockchains) wurden nicht weiter beachtet.

## 6.2 MARKTÜBERSICHT DLTS

Nach den im vorherigen Abschnitt vorgestellten Kriterien wurden 30 Blockchain-Lösungen identifiziert. Eine detaillierte Übersicht der Einzelnachweise ist im Anhang zu finden. Die Kriterien wurden folgendermaßen gewichtet: Die Business-Relevanz wird als wichtigstes Kriterium bewertet. Die IOT-Zugehörigkeit wird wichtiger als die Aktualität aber etwas geringer als die Business-Relevanz bewertet. Die Aktualität der Blockchain unterteilt sich in Blockchain-Aktivität und Github-Aktivität; diese werden gleich allerdings geringer als die IOT-Zugehörigkeit gewichtet.

Alle Lösungen überprüft, ob sie ein oder mehrere Kriterien erfüllen. Für jedes erfüllte Kriterium wurden der Lösung entsprechend der Gewichtungen Punkte angerechnet. Die Tabelle 6.1 listet die Erstausswahl gemäß der genannten Kriterien auf.

Die Blockchain-Lösung Ethereum führt die Rangliste mit vier Punkten an; das Schlusslicht stellt unter Anderem Zcash mit einem Punkt dar. Um für die weitere Betrachtung von Interesse zu sein, muss eine Lösung zu den Top10 (nach Punkten) gehören; dies entspricht 2 Punkten. Dadurch ergeben sich 11 mögliche Kandidaten (mit mind. 2 Punkten) für die Untersuchung, inwieweit diese die **DLT**-relevanten Anforderungen (vgl. Tabelle 5.1) erfüllen.

## 6.3 ANFORDERUNGSERFÜLLUNG

In Kapitel 5.5 wurden die zuvor als **DLT**-relevant eingestuftten Anforderungen in den Kontext **DLT** übersetzt. Dabei ergaben sich sechs Funktionalitäten, die eine Blockchain-Lösung implementieren muss, um für den Einsatz im vorliegenden Anwendungsfall geeignet zu sein. Diese waren Smart-Contracts, Zahlungsmittel, Oracle-Services, Asynchronität, Performanz und Verschlüsselung. Die Tabelle 6.2 zeigt die 11 übrigen Kandidaten und deren Anforderungserfüllung alphabetisch geordnet auf. Dabei wird bewertet, ob eine Anforderung erfüllt ist [*yes*] oder nicht [*no*]. Kann eine Anforderung nur teilweise, sehr schwierig oder nur unter bestimmten Voraussetzungen

<sup>1</sup> Blockchains basierend auf einem Ethereum-Standard; keine eigenständige Blockchain.

	Business Relevance	Github Activity	Blockchain Activity	IOT	TOTAL
	2	1	1	1,5	
Bitcoin	x	x			3
Bitcoin SV			x		1
BitcoinCash	x	x			3
Cardano		x			1
Corda	x				2
Cosmos		x			1
EOS		x	x		2
Ethereum	x	x	x		4
Hyperledger	x				2
INT Chain				x	1,5
IOST			x		1
IoT Chain				x	1,5
IOTA		x		x	2,5
KIN			x		1
LBRY Credits		x			1
Lisk		x			1
Monero		x			1
Nano			x		1
Particl		x			1
Quorum	x				2
Ripple	x				2
Ruff				x	1,5
SDChain				x	1,5
Steem			x		1
Stellar		x	x		2
Syscoin		x			1
Telos			x		1
Tron		x	x		2
WAVES		x			1
Zcash		x			1
Tron		x	x		2
WAVES		x			1
Zcash		x			1

Tabelle 6.1: Erstauswahl von DLT-Lösungen

	Smart-Contracts	Payment	Oracle-Services	Perf (TPS)	Async.	Tx Encrypt.
Bitcoin	(yes)	yes	(yes)	<10	yes	no
BitcoinCash	(yes)	yes	(yes)	<100	no	no
Corda	yes	(yes)	yes	~1000	yes	yes
EOS	yes	yes	yes	>1000	yes	no
Ethereum	yes	yes	yes	>10	yes	yes
Hyperledger	yes	(yes)	yes	-	yes	yes
IOTA	no	yes	no	>100	yes	yes
Quorum	yes	yes	yes	<1000	yes	yes
Ripple	no	yes	no	>1000	yes	no
Stellar	yes	yes	(yes)	>1000	yes	no
Tron	yes	yes	yes	<1000	yes	no

Tabelle 6.2: Erfüllung der DLT-relevanten Anforderungen

erfüllt werden, so wurde diese Anforderung entsprechend mit *[(yes)]* bewertet. Die Informationen der Tabelle wurden den technischen Spezifikationen oder den offiziellen Internetseiten der jeweiligen Lösungen entnommen.

Die Performanz bei privaten Blockchains (vgl. Kapitel 2.1.4) wurde ausgeklammert oder nicht angegeben, da diese sehr stark von der zugrundeliegenden Hardware abhängig ist. Darüber hinaus handelt es sich bei den Performanz-Werten um grobe Richtwerte, die teilweise nur unter Laborbedingungen erreicht werden können. Die tatsächlichen Tageswerte weichen zum Teil deutlich davon ab, daher sind diese Werte als Richtwerte zu verstehen und in der folgenden Bewertung als solche zu handhaben.

#### 6.4 BEWERTUNG, RANKING & AUSWAHL

Tabelle 6.2 zeigt mögliche DLT-Kandidaten und deren Erfüllung der übersetzten, DLT-relevanten Anforderungen auf. Die zentrale Anforderung ist die Ermöglichung von asynchronen Transaktionen bei Verbindungsverlust von Endgeräten. Lediglich Bitcoin-Cash bietet diese Funktionalität nicht, womit es als möglicher Kandidat ausscheidet. Eine weitere, entscheidende Anforderung ist die Bereitstellung von Smart-Contract Funktionalität. Dies ist sowohl bei IOTA als auch bei Ripple nicht gegeben, wodurch beide im weiteren Verlauf nicht mehr betrachtet werden. Die Möglichkeit, Transaktionen oder Smart-Contracts zu verschlüsseln, sodass nur beteiligte Parteien den Inhalt einsehen können, ist bei Bitcoin, Bitcoin-Cash, EOS, Ripple, Stellar und Tron nicht gegeben, weshalb diese im weiteren Verlauf vernachlässigt werden. Übrig bleiben Corda, Ethereum, Hyperledger und Quorum. Corda und Hyperledger bieten beide keine native Währung an; es besteht allerdings die Möglichkeit, beispielsweise mittels Smart-Contracts eine Token-Lösung

umzusetzen. Corda, Quorum und Hyperledger sind für den Einsatz in privaten Blockchains vorgesehen, Ethereum kann darüber hinaus auch als Public-Blockchain zum Einsatz kommen. Bezüglich der Performanz steht Ethereum mit etwa 20 Transaktionen pro Sekunde vergleichsweise schlecht dar.

Es zeichnet sich ab, dass die vier Lösungen Corda, Hyperledger, Ethereum und Quorum mögliche Kandidaten für eine prototypische Verprobung des vorliegenden Anwendungsfalls sind. Quorum ist in der Lage, sämtliche Anforderungen zu erfüllen. Es handelt sich um eine von der US-Bank JPMorgan entwickelte Blockchain basierend auf Ethereum. Dabei werden neue Ethereum-Releases mit Mechanismen für erhöhte Privätsphäre und alternativen Konsensmechanismen und als private und permissioned Blockchain bereitgestellt. Hierbei liegt allerdings auch die Problematik, die ebenfalls Corda und Hyperledger betreffen: Die zentrale Trustless-Eigenschaft geht in privaten und permissioned Blockchains verloren und die Transparenz wird verringert. Darüber hinaus werden die Eintrittshürden für die Teilnahme am Blockchain-Netzwerk stark erhöht. Aufgrund dessen und der Tatsache, dass Quorum zum größten Teil aus Ethereum-Quellcode besteht, fällt für die praktische Verprobung in dieser Arbeit die Wahl der Blockchain-Lösung auf Ethereum. Die nicht optimalen Transaktionsraten werden an dieser Stelle vernachlässigt; aktuelle Entwicklungen bezüglich der Skalierungsansätze von Blockchains (vgl. Kapitel [2.1.7](#)) lassen darauf hoffen, mittelfristig eine deutlich verbesserte Performanz erzielen zu können.

## UMSETZUNG

---

### 7.1 AUSWAHL DER ANWENDUNGSANFORDERUNGEN

In Tabelle 5.1 wurden alle DLT-relevanten Anforderungen aufgezeigt und in der Implementierung dieser Arbeit umgesetzt. Anforderungen, die nicht umgesetzt werden konnten oder für die eine Alternativ-Lösung gefunden wurde, werden im Folgenden genannt:

Damit ein Smart-Contract rechtskonform agiert und um die Vertragslogik nach geltendem deutschen Recht korrekt abzubilden, bedarf es ausführlicher rechtlicher Beratung durch entsprechende Experten und Anwälte. Dieser zeitaufwändige Prozess wurde im Rahmen dieser Arbeit nicht berücksichtigt. Bei einer Implementierung, die über die Machbarkeitsstudie hinausgeht und produktiv eingesetzt werden soll, muss dieser Punkt berücksichtigt werden. Damit wurde Anforderung A1.3.2 nicht erfüllt.

Anforderung A1.2.5 beschreibt die Verschlüsselung von Transaktionsdaten, sodass Inhalte nur vom Eigentümer oder dem Adressaten einsehbar sind. Diese Funktionalität ist für das Zeigen der Machbarkeit einerseits nicht notwendig, für die Implementierung für ein Live-System jedoch essentiell. Gängige Methoden, um den Payload einer Transaktion zu verschlüsseln ist der Einsatz sogenannter Zero-Knowledge-Proofs (siehe ??). Mit Hilfe dieses Ansatzes, ein Geheimnis zu beweisen ohne den Inhalt dabei preiszugeben, können in Kombination mit der Blockchain-Technologie Anwendungen entwickelt werden, die die Benutzerdaten vollständig schützen, ohne dabei an Funktionalität einzubüßen. Diese Anforderung wurde im Rahmen dieser Arbeit nicht umgesetzt.

Anforderung A1.3.8 beschreibt, dass Verträge im Nachhinein aktualisierbar sein müssen. Die Vertragslogik wurde mittels Smart-Contract umgesetzt: Einmal in der Blockchain gespeichert, sind Daten nicht mehr veränderlich. Es gibt dennoch Möglichkeiten, einen benötigten Mechanismus einzubauen, um die Verträge im Nachhinein anzupassen. Dazu kann man sich sogenannter Proxy-Smart-Contracts bedienen. Die Grundidee dabei ist, wie in einem Netzwerk ein zentrales Eingangstor zu haben, welches fest definiert ist und sich später nicht mehr ändert. Dieser Proxy verweist dann auf die jeweils aktuelle Version des benötigten Smart-Contracts. Ändert sich der Code durch Vertragsänderungen oder sonstige Updates, kann die aktualisierte Version in die Blockchain transferiert werden. Anschließend wird dem Proxy die Adresse des neuen Contracts mitgeteilt. Bei Anfragen an den Proxy leitet dieser den Anfragenden fortan an den neuen Vertrag weiter (der alte Vertrag bleibt dennoch weiterhin bestehen). Diese Umsetzung wurde im Zuge dieser Masterarbeit nicht geleistet, da die Umsetzung dieses Ansatzes viel Zeit und Aufwand benötigt, um die gewünschten Funktionalitäten zuverlässig



sig zu implementieren. Da diese Arbeit als Machbarkeitsstudie aufgebaut wurde, wurde an dieser Stelle auf die Implementierung verzichtet. Es sei auf X und Y verwiesen; dort wird ein einfacher (allerdings nicht ausgereifter) Proxy-Smart-Contract vorgestellt. Die Nicht-Umsetzung dieser Anforderung verhindert nicht das Aufzeigen der generellen Machbarkeit.

Anforderung SA2.2.1 bezieht sich auf die Performanz, welche bei Ethereum mit etwa 20 Transaktionen pro Sekunde nicht ausreichend ist. Eine kurze Hochrechnung verdeutlicht die Problematik: Bei durchschnittlich drei Tassen Kaffee pro Mitarbeiter und Tag (9 Stunden von 08:00 Uhr bis 17:00 Uhr) und etwa 40 Mitarbeiter pro Kaffeemaschine (und Büro) ergibt sich bei 10.000 Maschinen eine benötigte Verarbeitungsgeschwindigkeit von 37 Tassen Kaffee pro Sekunde<sup>1</sup>. Damit wäre die Ethereum-Blockchain deutlich überlastet. Deshalb wurde sich hierbei einer alternativen Lösung bedient: Payment-Channel (vgl. Kapitel 2.1.7) werden eingesetzt, um das beschriebene Performanz-Problem zu lösen. Der Ansatz ist dabei wie folgt: Bei Vertragseröffnung wird gleichzeitig zwischen Kaffeemaschine und Benutzer ein Payment-Channel eröffnet. Dieser wird mit einem Betrag aufgeladen (Pre-paid; 10€), sodass schon vor der ersten Benutzung der Maschine ein Betrag zur Verbuchung zur Verfügung steht. Möchte der Benutzer einen Kaffee von der Maschine erhalten, so erstellt diese eine Quittung und überträgt diese per NFC! (NFC!) an das Smartphone des Benutzers. Dieser kann anschließend mit seinem auf dem Smartphone befindlichen Private-Key die Quittung signieren und der Kaffeemaschine damit bestätigen, diesen Kauf durchgeführt zu haben. Die Kaffeemaschine bekommt anschließend die signierte Quittung per NFC! zurückgesendet und kann die Signatur überprüfen. Ist diese korrekt, wird der Kaffee ausgegeben. Dieser Vorgang passiert offchain und wird nicht per Transaktion an die Blockchain übermittelt. Dieser Vorgang kann nun so oft wiederholt werden, bis das Guthaben im Payment-Channel aufgebraucht ist. Ist dies der Fall, so muss der Benutzer zunächst sein Guthaben neu aufladen. Die Kaffeemaschine hält die zuletzt signierte Quittung so lange vor, bis ein definierter Zustand eintritt<sup>2</sup>. Anschließend schickt sie diese Quittung per Transaktion an die Blockchain und erhält das entsprechend signierte Guthaben aus dem Payment-Channel. Dadurch kann die Zahl von etwa 1,2 Mio. Transaktionen pro Tag auf ein Bruchteil reduziert werden.

#### 7.1.1 Architektur und Aufbau

Web-Applikation in React Wallet als Browser-Plugin (Metamask) Smart-Contracts für Mietverträge, Zahlungsabwicklung und Identitätsmanagement (Oracle) auf Kovan Testnet Kaffeemaschine simuliert mittels Raspberry-Pi, angeschlossenes NFC-Entwicklungsboard Smartphone als Wallet und zum Kommunizieren mit der Kaffeemaschine

<sup>1</sup> Unter der Annahme, dass eine Tasse Kaffee mit einer Transaktion versendet wird.

<sup>2</sup> Da eine Kaffeemaschine in der Regel nicht in der Nacht benutzt wird, könnte im Zeitraum zwischen 6 Uhr abends und 6 Uhr Morgens die Synchronisation stattfinden, sofern das Prepaid-Guthaben unter ein festgelegtes Minimum sinkt.

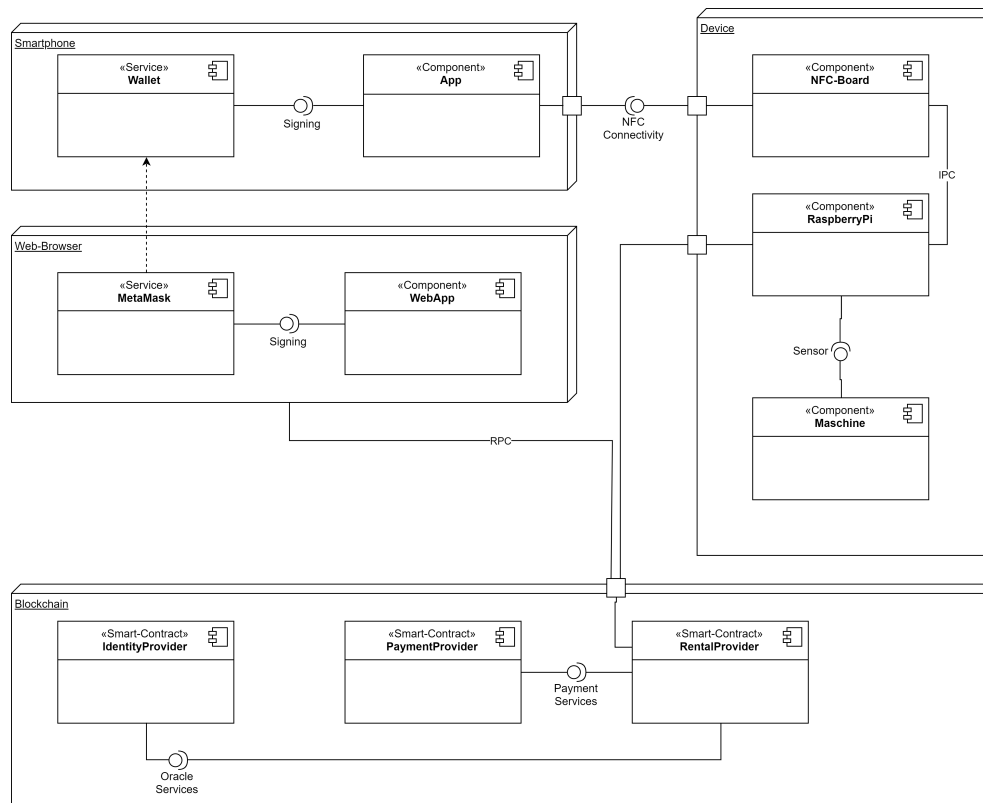


Abbildung 7.1: Architektur als UML-Komponentendiagramm

Smart-Contract	Testnet-Adresse
Rental-Provider	0xf25F1F12630158d963a9609950C3e379c65617dA
Identity-Provider	0xA17351BfA16dAE1FD285e11AcC00882Eb459C7cb
Payment-Provider	0x7d17DA28604A7bB2E121D634012Co86A6BbF6E2e

Tabelle 7.1: Adressen der Smart-Contracts auf dem öffentlichen Kovan Testnet

### 7.1.2 Deployment

## 7.2 TESTAUFBAU

Die Vertragslogiken der Mietverträge wurden als Smart-Contracts in der Programmiersprache Solidity auf der Ethereum-Blockchain entwickelt. Für Smart-Contracts auf Ethereum-Basis existiert die Truffle-Suite, die ein umfangreiches Toolset für die Entwicklung und das Testen von Smart-Contracts bereitstellt. Der Entwicklungsprozess wird durch eine lokale Blockchain-Umgebung vereinfacht, die das Deployment und Testing beschleunigt. Mittels Unit-Tests wurde sichergestellt, dass die Funktionalität gemäß den Anforderungen gegeben ist. Die Testabdeckung wurde durch das Tool solidity-coverage überprüft, die Ergebnisse werden in Abbildung 7.2 dargestellt. Die

Grafik zeigt die prozentuale und absolute Abdeckung von Statements, Funktionen und Code-Zeilen an.





File		Statements		Functions		Lines	
Ownable.sol		100%	5/5	100%	3/3	100%	6/6
RentalProvider.sol		96.64%	144/149	96.15%	25/26	96.79%	151/156
IdentityProvider.sol		96.36%	53/55	100%	17/17	95.83%	46/48
PaymentProvider.sol		93.55%	58/62	86.67%	13/15	93.75%	60/64

Abbildung 7.2: Testabdeckung der Solidity Smart-Contracts

**E2E!** (EzE!)-Tests wurden manuell durchgeführt. Dafür wurden die Smart-Contracts auf ein öffentliches Ethereum-Testnet deployed, die Hochrechnung Skalierungsansatz

### 7.3 DATENSCHUTZ & PRIVATSPHÄRE

Untersuchung Datenschutz und Privatsphäre



## DISKUSSION

---

- 9.1 WIEDERAUFNAHME THESE TEIL 1: EIGNUNG FÜR IOT?
- 9.2 WIEDERAUFNAHME THESE TEIL 2: TECHNISCHE ANFORDERUNGEN  
IMMER GLEICH?

## AUSBLICK

---

Optimierungspotential: - dezentrale Datenspeicherung in IPFS, um Kosten zu senken - Einsetzen von Zero-Knowledge Proofs, um Datenschutz und Privatsphäre sicherzustellen - Proxy-Smart-Contracts umsetzen - Erweiterung um Service- und Lieferverträge - Anwendungsfall ausbauen

Teil II

APPENDIX

## APPENDIX: ANFORDERUNGEN

---



Story	Description	Task	Description	Requirement	Description	Class Level-1	Class Level-2	Non-Functional	Non-Functional Subcategory
Story A1 "Agieren auf Plattform"	"Als Akteur möchte ich in meiner Rolle als [X] auf der Plattform agieren."	Task A1.1	Akteure können sich an der Plattform registrieren und gemäß ihrer Rolle miteinander agieren.	Requirement A1.1.1	Jeder Akteur auf der Plattform kann eindeutig identifiziert werden.	Software	Nonfunctional	Security	Accountability
				Requirement A1.1.2	Ein Akteur registriert sich und meldet sich auf der Plattform an, bevor er dort agieren kann.	Software	Functional		
				Requirement A1.1.3	Ein Akteur agiert immer mit einer bestimmten Rolle auf der Plattform: Manufacturer, Customer, Supplier, Service-Provider oder Gerät. Ein Akteur kann mehrere Rollen haben.	Software	Functional		
				Requirement A1.1.4	Es existiert eine Oberfläche, auf die jeder Akteur Zugriff hat. Dort kann er sich registrieren und anmelden.	Software	Functional		
				Requirement A1.1.5	Ein Akteur hat eine (mehrere) verifizierte Rolle(n).	Software	Functional		
		Task A1.2	Akteure können über die Plattform miteinander kommunizieren.	Requirement A1.2.1	Akteure kommunizieren über die Plattform.	Software	Functional		
				Requirement A1.2.2	Die Kommunikation der beteiligten Akteure wird sofort übermittelt.	Software	Nonfunctional	PerformanceEfficiency	Time behavior
				Requirement A1.2.3	Die Kommunikation zwischen den Akteuren ist nachvollziehbar und eindeutig zuordenbar.	Software	Nonfunctional	Security	Accountability
				Requirement A1.2.4	Die Kommunikation zwischen den Akteuren kann nicht gelöscht oder manipuliert werden.	Software	Nonfunctional	Security	Integrity
				Requirement A1.2.5	Akteure können nur den Inhalt ihrer eigenen Nachrichten einsehen.	Software	Nonfunctional	Security	Confidentiality
		Task A1.3	Akteure können über die Plattform Verträge miteinander abschließen.	Requirement A1.3.1	Akteure schließen Verträge über die Plattform ab.	Software	Functional		
				Requirement A1.3.2	Verträge sind rechtlich bindend.	Software	Functional		
				Requirement A1.3.3	Akteure können Verträge ablehnen oder annehmen.	Software	Functional		
				Requirement A1.3.4	Es existiert eine Oberfläche, auf die jeder Akteur Zugriff hat. Dort kann er Vertragsanfragen erstellen. Auf der Empfängerseite muss eine Oberfläche existieren, die diese Anfragen anzeigt.	Software	Functional		
				Requirement A1.3.5	Der Vertragsgegenstand kann nicht durch Dritte manipuliert werden.	Software	Nonfunctional	Security	Integrity
				Requirement A1.3.6	Akteure haben Zugriff auf alle Vertragsinformationen.	Software	Functional		
				Requirement A1.3.7	Akteure können komplexe Vertragskonstrukte umsetzen. Verträge haben einen Status. Diese können "Aktiv", "In Erzeugung" oder "Inaktiv" sein.	Software	Functional		
				Requirement A1.3.8	Akteure können ihre Verträge im Nachhinein ändern.	Software	Functional		
				Requirement A1.3.9	Akteure können nur ihre eigenen Verträge ändern.	Software	Nonfunctional	Security	Confidentiality
				Requirement A1.3.10	Eine Vertragsänderung bedarf der Zustimmung aller beteiligten Akteure.	Software	Functional		
				Requirement A1.3.11	Erbrachte Leistungen werden kostenpflichtig verrechnet.	Software	Functional		
		Task A1.4	Akteure benötigen grafische Oberflächen zum Agieren auf der Plattform.	Requirement A1.4.1	Es existiert eine Oberfläche, auf die der Akteur Zugriff hat. Dort hat er eine Übersicht über alle seiner Verträge sowie aggregierte Informationen wie Anzahl aller Verträge, Kontostand, etc. Es werden ebenfalls angebotene Verträge angezeigt, die angenommen oder abgelehnt werden können.	Software	Functional		
				Requirement A1.4.2	Es existiert eine Oberfläche, auf die der Akteur Zugriff hat. Dort hat er eine detaillierte Übersicht über einen seiner Verträge und kann sich Detailinformationen dazu ansehen. Außerdem sieht er eine Übersicht über alle Nachrichten, die mit diesem Vertrag in Verbindung stehen und den Vertrag bearbeiten.	Software	Functional		
Story M1 "Geräte vermieten"	"Als Manufacturer möchte ich meine Geräte über die Plattform vermieten können, um meinen Umsatz zu steigern."	Task M1.1	Die Geräte können auf der Plattform vermietet werden.	Requirement M1.1.1	Ein Gerät ist Eigentum eines Manufacturers.	Software	Functional		
				Requirement M1.1.1	Ein Manufacturer kann beliebig viele Geräte besitzen und über die Plattform vermieten.	Software	Functional		
		Task M1.2	Die Geräte benötigen Sensoren, um Fehler und Defekte zu detektieren.	Requirement M1.2.1	Geräte können Fehlerzustände detektieren. Tritt ein Fehler auf, wird dieser dem Customer über ein Display angezeigt.	Software	Functional		
				Requirement M1.2.2	Geräte können Defekte detektieren. Tritt ein Fehler auf, wird dieser über die Plattform an den Service-Provider (Service-Vertrag) gemeldet.	Software	Functional		
				Requirement M1.2.3	Ein Defekt (Defektes Mahlwerk, undichte Anschlüsse, etc.) hindert das Gerät am Durchführen seiner Tätigkeit und muss durch einen Service-Provider behoben werden.	Software	Functional		
				Requirement M1.2.4	Ein Fehlerzustand (Leerer Wasserbehälter, geöffnete Abdeckung, etc.) hindert das Gerät am Durchführen seiner Tätigkeit und kann meistens durch den Customer behoben werden.	Software	Functional		
Story M2 "Verträge erzeugen"	"Als Manufacturer möchte ich in der Lage sein, Verträge anzulegen, um meine Geräte über die Plattform vermieten zu können."	Task M2.1	Ein Manufacturer kann Verträge erzeugen.	Requirement M2.1.1	Es existiert eine Oberfläche, auf die der Manufacturer Zugriff hat. Dort kann er Mietverträge erzeugen und als Antwort auf seine Mietanfrage an den Customer senden.	Software	Functional		
				Requirement M2.1.2	Es existiert eine Oberfläche, auf die der Manufacturer Zugriff hat. Dort kann er alle Service-Provider und deren Dienstleistungen einsehen. Service-Verträge erzeugen und an einen Service-Provider senden.	Software	Functional		
				Requirement M2.1.3	Es existiert eine Oberfläche, auf die der Manufacturer Zugriff hat. Dort kann er alle Supplier einsehen, Lieferverträge erzeugen und an den Supplier senden.	Software	Functional		
Story M3 "Verträge abrechnen"	"Als Manufacturer benötige ich eine korrekte, nutzungabhängige und automatische Abrechnung der vermieteten Geräte, die regelmäßig aktualisiert wird, sowie der erbrachten Dienstleistungen, um den Umsatz aufrecht zu erhalten."	Task M3.1	Die Geräte müssen den Verbrauch detektieren.	Requirement M3.1.1	Geräte detektieren den Verbrauch (Kaffeemaschine: Anzahl Kaffees) und senden diesen an die Plattform.	Software	Functional		
		Task M3.2	Die Geräte müssen die Verbrauchsdaten an die Plattform melden.	Requirement M3.2.1	Geräte senden den Verbrauch nach Fertigstellung des Produktes sofort an die Plattform.	Software	Nonfunctional	PerformanceEfficiency	Time behavior
				Requirement M3.2.2	Die Verbrauchsdaten der Geräte können nicht manipuliert werden.	Software	Nonfunctional	Security	Integrity
Story C1 "Ansicht verfügbarer Geräte"	"Als Customer möchte ich verfügbare Haushaltsgeräte angezeigt bekommen, um das passende Gerät mieten zu können."	Task C1.1	Es muss eine Auflistung aller verfügbarer (mietbarer) Geräte existieren.	Requirement C1.1.1	Es existiert eine Oberfläche, auf die der Customer Zugriff hat. Dort hat er die Möglichkeit, alle verfügbaren Geräte aufzulisten.	Software	Functional		
		Task C1.2	Es müssen alle für den Customer relevanten Informationen über das Gerät vorhanden und einsehbar sein.	Requirement C1.2.1	Der Customer hat Zugriff auf eine detaillierte Beschreibung des Geräts.	Software	Functional		
		Task C2.1	Das Gerät muss die Reinigung / Wartung durch den Customer detektieren können.	Requirement C2.1.1	Geräte detektieren eine Reinigung (Produktbehälter leeren, etc.).	Software	Functional		
				Requirement C2.1.2	Geräte detektieren eine Wartung (Entkalken, etc.).	Software	Functional		

Story C2 "Geräte warten"	"Als Customer möchte ich die gemieteten Geräte reinigen und warten können, um dafür vom Hersteller eine Gutschrift auf mein Vertragskonto zu erhalten."	Task C2.2	Es muss sichergestellt werden, dass eine Reinigung / Reparatur dem Gerät bzw. dessen Sensoren nicht vorgespült werden kann.	Requirement C2.2.1	Detektierte Reinigungen / Wartungen können nicht manipuliert oder dem Gerät vorgespült werden.	Software	Nonfunctional	Security	Integrity
		Task C2.3	Der Customer muss vom Gerät eindeutig identifiziert werden können.	Requirement C2.3.1	Das Gerät kann den Customer identifizieren.	Software	Functional		
		Task C2.4	Die Reinigung / Wartung muss an die Plattform übertragen werden und gemäß des Vertrages abgerechnet werden.	Requirement C2.4.1	Das Gerät sendet detektierte Reinigungen / Wartungen an die Plattform.	Software	Functional		
				Requirement C2.4.2	Das Gerät sendet detektierte Reinigungen / Wartungen nach Abschluss direkt an die Plattform.	Software	Nonfunctional	PerformanceEfficiency	Time behavior
Story C3 "Einfache Bedienbarkeit der Plattform"	"Als Customer möchte ich eine intuitive, einfach zu bedienende Oberfläche, um mich gut auf der Plattform zurechtzufinden."	Task C3.1	Ein Customer muss zunächst mit der Plattform bekannt gemacht werden.	Requirement C3.1.1	Ein Customer erhält eine initiale Einführung über die Plattform bei der ersten Anmeldung.	System	Transition		
				Requirement C3.1.2	Die Funktionen der Plattform sind für den Customer schnell erlernbar und leicht verständlich.	Software	Nonfunctional	Usability	Learnability
		Task C3.2	Die Plattform muss optisch ansprechend sein.	Requirement C3.2.1	Die Plattform ist für den Customer optisch ansprechend.	Software	Nonfunctional	Usability	User interface aesthetics
Story SP1 "Dienstleistungen bereitstellen"	"Als Service-Provider möchte ich meine Angebotspalette auf der Plattform anbieten können."	Task SP1.1	Ein Service-Provider muss seine angebotenen Dienstleistungen auf der Plattform eingeben können.	Requirement SP1.1.1	Es existiert eine Oberfläche, auf die der Service-Provider Zugriff hat. Dort kann er alle Dienstleistungen, die er anbietet, sowie Detailinformationen, wie zum Beispiel Kosten der Dienstleistung, eintragen und damit auf der Plattform verfügbar machen.	Software	Functional		
				Requirement SP1.1.2	Es existiert eine Oberfläche, auf die der Service-Provider Zugriff hat. Dort kann er bereits angebotene Dienstleistungen editieren oder löschen.	Software	Functional		
Story SP2 "Service-Aufträge abschließen"	"Als Service-Provider möchte ich nach der Durchführung der Wartung diese mit meinem Smartphone am Gerät bestätigen, um den Service-Auftrag abzuschließen."	Task SP2.1	Der Service-Provider kann mit dem Gerät per Smartphone kontaktlos kommunizieren.	Requirement SP2.1.1	Gerät und Service-Provider können miteinander kommunizieren.	Software	Functional		
				Requirement SP2.1.2	Ein Gerät kann den Service-Provider eindeutig identifizieren.	Software	Functional		
		Task SP2.2	Es wird eine App benötigt, mit der der Service-Provider seine Identität und die durchgeführte Wartung am Gerät bestätigen kann.	Requirement SP2.2.1	Es existiert eine Smartphone-Oberfläche, auf die der Service-Provider Zugriff hat. Darüber kann er Wartungen abschließen.	Software	Functional		
				Requirement SP2.2.2	Ein Service-Provider kann einen Service-Auftrag starten und beenden, wenn er sich in der Nähe des Gerät befindet.	Software	Functional		
Story SEC1 "Sichere Zahlungsabwicklung"	"Als IT-Security-Beauftragter möchte ich sichergestellt wissen, dass die Zahlungsabwicklung auf der Plattform sicher und voll funktionsfähig ist."	Task SEC1.1	Es muss sichergestellt werden, dass Geldtransfers vom Sender an den Empfänger durchgeführt werden.	Requirement SEC1.1.1	Geldtransfers werden auf der Plattform geloggt.	Software	Functional		
				Requirement SEC1.1.2	Bei Nicht-Ausführung von Geld- und Nachrichtentransfers werden die Parteien benachrichtigt.	Software	Nonfunctional	Security	Integrity
				Requirement SEC1.1.3	Die Plattform prüft Geldtransfers vor Ausführung.	Software	Functional		
		Task SEC1.2	Es muss sichergestellt werden, dass Sender und Empfänger des Geldes eindeutig identifizierbar sind.	Requirement SEC1.2.1	Jeder Akteur besitzt eine (mehrere) eindeutige Kontonummer(n).	Software	Nonfunctional	Security	Accountability
				Requirement SEC1.2.2	Konten sind zugriffsgeschützt.	Software	Nonfunctional	Security	Authenticity
Story SEC2 "Sichere Kommunikation und signierte Nachrichten"	"Als IT-Security-Beauftragter möchte ich eine verschlüsselte Kommunikation mit der Plattform, damit meine Daten nicht in die Hände von Dritten gelangen."	Task SEC2.1	Die Kommunikation zwischen Akteuren und der Plattform muss verschlüsselt werden.	Requirement SEC2.1.1	Sämtliche Verbindungen sind per SSL/TLS zu verschlüsseln.	System	Quality		
		Task SEC2.2	Aktuelle Sicherheitsstandards müssen verwendet werden.	Requirement SEC2.2.1	Es können Passwortregeln hinterlegt werden. Die Einhaltung dieser Regeln wird überprüft.	System	Quality		
Story SEC3 "Manipulationssicherheit"	"Als IT-Security-Beauftragter möchte ich eine manipulationssichere Plattform, um die Integrität und Echtheit der Daten zu gewährleisten."	Task SEC3.1	Es müssen Vorkehrungen gegen Manipulationen getroffen werden.	Requirement SEC3.1.1	Der Zugang zu den Backend-Systemen wird protokolliert und nur Berechtigten gestattet.	Software	Nonfunctional	Security	Accountability
				Requirement SEC3.1.2	Manipulationen werden durch den Einsatz kryptographischer Methoden verhindert.	Software	Nonfunctional	Security	Integrity
		Task SEC3.2	Es ist nachvollziehbar, wer wann auf die Plattform zugegriffen hat.	Requirement SEC3.2.1	Alle Aktivitäten auf der Plattform werden geloggt.	Software	Nonfunctional	Security	Accountability
Story BD1 "Geschäftsmodell"	"Als Business-Developer möchte ich in Zukunft eine Plattform schaffen, auf der Hersteller unterschiedlicher Branchen ihre Produkte nach dem Pay-As-You-Use Prinzip vermieten können, um den Kunden eine breitere Produktpalette zu bieten."	Task BD1.1	Das Pay-As-You-Use Abrechnungsmodell muss in einem Vertrag abgebildet werden können.	Requirement BD1.1.1	Vermietete Geräte werden nach dem Pay-As-You-Use Prinzip abgerechnet.	System	Business		
				Requirement BD1.2.1	Customer bezahlen jede verbrauchte Einheit (z.B. pro Tasse Kaffee) des gemieteten Geräts an den Manufacturer. Die genauen Kosten sind vom Gerät abhängig und werden durch den Manufacturer festgelegt.	Software	Functional		
		Task BD1.2	Es ist vertraglich festgelegt, welcher Akteur für welche Dienstleistung wie viel Geld bezahlt bzw. erhält.	Requirement BD1.2.2	Manufacturer bezahlen Customer für jede durchgeführte Wartung des vermieteten Geräts. Die Höhe der Zahlung ist im Mietvertrag geregelt.	Software	Functional		
				Requirement BD1.2.3	Manufacturer bezahlen Service-Provider für jede erbrachte Service-Leistung. Die Höhe der Zahlung ist im Service-Vertrag geregelt.	Software	Functional		
				Requirement BD1.2.4	Verträge sind individuell gestaltbar.	Software	Functional		
				Requirement BD1.2.5	Manufacturer bezahlen Supplier für jede erbrachte Lieferung. Die Höhe der Zahlung ist im Liefervertrag geregelt.	Software	Functional		
Story BD2 "Plattform für Partner"	"Als Business-Developer möchte ich eine Plattform, die für Partner wie Service-Provider oder Supplier leicht zugänglich ist."	Task BD2.1	Partner agieren auf der Plattform.	Requirement BD2.1.1	Auf der Plattform agieren Geschäftspartner (Service-Provider, Supplier, ect.).	System	Business		
				Requirement BD2.1.2	Die Nutzung der Plattform ist intuitiv und schnell erlernbar. Umfangreiche Mitarbeiterschulungen zur Benutzung der Plattform sind nicht notwendig.	System	Transition		
		Task BD2.2	Partner müssen als solche identifiziert sein.	Requirement BD2.2.1	Der Prozess zur Prüfung, dass es sich z.B. bei einem Service-Provider auch tatsächlich um einen solchen handelt, ist benutzerfreundlich und so schnell und einfach wie möglich umsetzbar.	System	Stakeholder		
Story BD3 "Hersteller-übergreifende Plattform"	"Als Business-Developer möchte ich in Zukunft eine Plattform schaffen, auf der Hersteller unterschiedlicher Branchen ihre Produkte nach dem Pay-As-You-Use Prinzip vermieten können, um den Kunden eine breitere Produktpalette zu bieten."	Task BD3.1	Die Plattform muss zukünftig weitere Hersteller zulassen, um ein ganzes Ökosystem von Geräten aller Art dem Customer zugänglich zu machen.	Requirement BD3.1.1	Es können in Zukunft weitere Hersteller auf der Plattform ihre Geräte zur Miete anbieten.	System	Business		
		Task BD3.2	Ein Gerät muss auf der Plattform generisch repräsentiert werden und darf nicht von einem bestimmten Produkttyp oder Hersteller abhängig sein.	Requirement BD3.2.1	Die Plattform kann Geräte unterschiedlicher Art anbinden und abrechnen.	Software	Nonfunctional	Compatibility	Interoperability
		Task BD3.3	Ein Vertrag muss auf der Plattform generisch repräsentiert werden und darf nicht von einem bestimmten Produkttyp oder Hersteller abhängig sein.	Requirement BD3.3.1	Die Plattform kann unterschiedliche Vertragsarten abbilden und abrechnen.	Software	Nonfunctional	Compatibility	Interoperability
Story BD4 "Vertragsgestaltung"	"Als Business-Developer möchte ich die Plattform dazu nutzen, künftig andere Vertragsarten umzusetzen, um weitere Geschäftsfelder und Kunden zu gewinnen."	Task BD4.1	Die Einbindung von Verträgen sowie deren Struktur muss möglichst modular und unabhängig geschehen, damit später andere Verträge leicht integriert werden können.	Requirement BD4.2.1	Ein Abrechnungsmodell wird in einem Vertrag abgebildet.	System	Business		
				Requirement BD4.2.2	Einzelne Bestandteile der Plattform beeinflussen sich gegenseitig nicht und können leicht ausgetauscht werden.	Software	Nonfunctional	Maintainability	Modularity
Story SA1 "Modularer Aufbau"	"Als System-Architekt möchte ich eine modular aufgebaute Plattform, damit diese später modifiziert werden kann."	Task SA1.1	Funktionalitäten werden in Modulen gekapselt und als Service bereitgestellt.	Requirement SA1.1.1	Ein Software-Modul wird unabhängig von anderen Modulen bereitgestellt und gewartet.	Software	Process		
Story SA2	"Als System-Architekt möchte ich SPOFs und Datenverlust vermeiden."	Task SA2.1	Die Plattform kann im Falle eines Crashes alle Daten konsistent halten.	Requirement SA2.1.1	Bei Ausfall einzelner Komponenten gehen keine Daten verloren.	Software	Nonfunctional	Security	Integrity
				Requirement SA2.1.2	Bei Ausfall einzelner Komponenten entstehen keine Dateninkonsistenzen.	Software	Nonfunctional	Security	Integrity

"Redundanz"	damit das System auch im Fehlerfall weiter funktionsfähig ist.	Task SA2.2	Die Plattform darf keinen SPOF haben, damit im Falle eines Crashes das Gesamtsystem weiterhin lauffähig ist.	Requirement SA2.2.1	Die Plattform ist in der Lage, die Kommunikation und Datenverarbeitung bei bis zu 10.000 Endgeräten durchzuführen.	Software	Nonfunctional	Reliability	Availability
				Requirement SA2.2.2	Die Plattform ist nicht von einer einzelnen Komponente (SPOF) abhängig.	Software	Nonfunctional	Reliability	Availability
Story P1 "Deployment & Testing"	"Als Betreiber der Plattform möchte ich eine automatisierte und korrekte Bereitstellung der Plattform, um meinen Aufwand zu reduzieren."	Task P1.1	Es wird nur getestete Software bereitgestellt.	Requirement P1.1.1	Für alle Software-Komponenten existieren hinreichende Tests, die Test-Abdeckung beträgt 80%.	Software	Nonfunctional	Maintainability	Testability
		Task P1.2	Die Bereitstellung ist automatisiert.	Requirement P1.2.1	Die Software wird automatisiert durch Skripte installiert und bereitgestellt.	Software	Nonfunctional	Portability	Installability

APPENDIX: CODE

---

```
pragma solidity >= 0.5.0 < 0.7.0;

import "./IdentityProvider.sol";
import "./PaymentProvider.sol";
import "./Ownable.sol";

contract RentalProvider is Ownable {
    //possible states of an agreement
    enum AgreementState {
        Pending,
        Active,
        Terminated
    }

    RentalRequest[] public requests;
    RentalAgreement[] public agreements;
    address[] public rentableDevices;

    struct RentalRequest {
        address tenant;
        address lessor;
        address device;
        uint256 contractTerm;
    }

    struct RentalAgreement {
        address payable tenant;
        bytes tenantSignature;
        address payable lessor;
        bytes lessorSignature;
        address device;
        uint usageFee;           //
        uint contractTerm;      //unix timestamp
        uint creation;          //unix timestamp
        AgreementState state;
        bytes32 paymentAgreementHash;
    }

    address public oracle_addr;
    address public paymentProvider_addr;

    constructor () public {
        owner = msg.sender;
    }
}
```

```

function init() public onlyOwner {
    IdentityProvider oracle = IdentityProvider(oracle_addr);
    rentableDevices = oracle.getKnownDevices();
}

function destroy() public {
    require(msg.sender == owner, "Only owner can call this function.");
    selfdestruct(owner);
}

function getRentableDevices() public view returns (address[] memory){
    return rentableDevices;
}

function deviceIsRentable(address _device) public view returns (bool)
{
    bool found = false;
    for(uint i = 0; i < rentableDevices.length; i++) {
        if(_device == rentableDevices[i]) {
            found = true;
        }
    }
    return found;
}

function createRequest(address _device, address _lessor, uint256 _term
) public {
    require(isKnownParticipant(msg.sender, 2));
    require(isKnownParticipant(_device, 5));
    require(isKnownParticipant(_lessor, 1));
    require(deviceIsRentable(_device));
    IdentityProvider oracle = IdentityProvider(oracle_addr);
    require(_lessor == oracle.getIdentityOwner(_device));
    require(!agreementExists(msg.sender, _device));
    require(!requestExists(msg.sender, _lessor, _device, _term));
    requests.push(RentalRequest(msg.sender, _lessor, _device, _term));
    removeDeviceFromRentableList(getRentableDeviceListIndex(_device));
}

function removeRequest(address _tenant, address _lessor, address _
device, uint256 _term) public {
    require(isKnownParticipant(msg.sender, 1));
    require(msg.sender == _lessor);
    require(requestExists(_tenant, _lessor, _device, _term));
    uint index = 0;
    for(uint i = 0; i < requests.length; i++) {
        if(requests[i].tenant == _tenant && requests[i].lessor == _lessor
&& requests[i].device == _device && requests[i].contractTerm
== _term) {
            index = i;
        }
    }
}

```

```

    }
    for (uint i = index; i < requests.length-1; i++){
        requests[i] = requests[i+1];
    }
    requests.pop();
    //delete requests[requests.length-1];
    //requests.length--;
}

function getRequestsAsLessor() public view returns(address[] memory,
    address[] memory, address[] memory, uint256[] memory) {
    require(isKnownParticipant(msg.sender, 1));
    uint count = 0;
    for(uint i = 0; i < requests.length; i++) {
        if(requests[i].lessor == msg.sender) {
            count++;
        }
    }
    address[] memory tenants = new address[](count);
    address[] memory lessors = new address[](count);
    address[] memory devices = new address[](count);
    uint256[] memory terms = new uint256[](count);
    uint index = 0;
    for(uint i = 0; i < requests.length; i++) {
        if(requests[i].lessor == msg.sender) {
            tenants[index] = requests[i].tenant;
            lessors[index] = requests[i].lessor;
            devices[index] = requests[i].device;
            terms[index] = requests[i].contractTerm;
            index++;
        }
    }
    return (tenants, lessors, devices, terms);
}

function getRequestsAsTenant() public view returns(address[] memory,
    address[] memory, address[] memory, uint256[] memory) {
    require(isKnownParticipant(msg.sender, 2));
    uint count = 0;
    for(uint i = 0; i < requests.length; i++) {
        if(requests[i].tenant == msg.sender) {
            count++;
        }
    }
    address[] memory tenants = new address[](count);
    address[] memory lessors = new address[](count);
    address[] memory devices = new address[](count);
    uint256[] memory terms = new uint256[](count);
    uint index = 0;
    for(uint i = 0; i < requests.length; i++) {
        if(requests[i].tenant == msg.sender) {
            tenants[index] = requests[i].tenant;

```

```

        lessors[index] = requests[i].lessor;
        devices[index] = requests[i].device;
        terms[index] = requests[i].contractTerm;
        index++;
    }
}
return (tenants, lessors, devices, terms);
}

function requestExists(address _tenant, address _lessor, address _
    device, uint256 _term) public view returns (bool){
    for(uint i = 0; i < requests.length; i++) {
        if(requests[i].tenant == _tenant && requests[i].lessor == _lessor
            && requests[i].device == _device && requests[i].contractTerm
                == _term ) {
            return true;
        }
    }
    return false;
}

function removeDeviceFromRentableList(uint index) public {
    require(index < rentableDevices.length);
    require(index >= 0);
    for (uint i = index; i<rentableDevices.length-1; i++){
        rentableDevices[i] = rentableDevices[i+1];
    }
    rentableDevices.pop();
    //delete rentableDevices[rentableDevices.length-1];
    //rentableDevices.length--;
}

function getRentableDeviceListIndex(address _device) public view
    returns (uint) {
    for(uint i = 0; i < rentableDevices.length; i++) {
        if(_device == rentableDevices[i]) {
            return i;
        }
    }
    revert();
}

/// @notice asks registered oracle whether the identity is known or
    not
/// @param _addr the address of the identity address
/// @param _roleID the role of the identity
/// @return a boolean whether or not the identity is known
function isKnownParticipant(address _addr, uint _roleID) public view
    returns(bool) {
    IdentityProvider oracle = IdentityProvider(oracle_addr);
    if(oracle.identityExists(_addr) && oracle.getIdentityRole(_addr)
        == _roleID) {

```

```

        return true;
    } else {
        return false;
    }
}

/// @notice sets the oracle address
/// @dev only callable by owner
/// @param _addr the address of the oracle
function registerIdentityProvider(address _addr) public onlyOwner {
    oracle_addr = _addr;
}

/// @notice sets the PaymentProvider address
/// @dev only callable by owner
/// @param _addr the address of the PaymentProvider
function registerPaymentProvider(address _addr) public onlyOwner {
    paymentProvider_addr = _addr;
}

function verify(address _tenant, address _lessor, address _device,
    uint256 _fee, uint256 _term, bytes memory _sig, address _signer)
    public pure returns (bool) {
    (uint8 v, bytes32 r, bytes32 s) = splitSignature(_sig);
    bytes32 hashCalc = keccak256(abi.encodePacked(_tenant, _lessor, _
        device, _fee, _term));
    bytes memory prefix = "\x19Ethereum Signed Message:\n32";
    return ecrecover(keccak256(abi.encodePacked(prefix, hashCalc)), v, r
        , s) == _signer;
}

/// @notice creates a rentalAgreement
/// @param _tenant address of the tenant
/// @param _lessorSignature signature of lessor of rentalAgreement
/// @param _device address of device
/// @param _usageFee fee for usage in wei
/// @param _contractTerm timestamp, when rentalAgreement times out
function createRenting(address _tenant, bytes memory _lessorSignature,
    address _device, uint _usageFee, uint _contractTerm) public {
    require(verify(_tenant, msg.sender, _device, _usageFee, _
        contractTerm, _lessorSignature, msg.sender));
    //check if sender is Manufacturer, tenant is Customer and device is
    Device
    require(isKnownParticipant(msg.sender, 1));
    require(isKnownParticipant(_tenant, 2));
    require(isKnownParticipant(_device, 5));
    //check contractTerm to be in future!
    require(_contractTerm > now);
    //check that no rentalAgreement with same parameters exists
    require(!agreementExists(_lessorSignature));
}

```



```

//already removed by request!
//removeDeviceFromRentableList(getRentableDeviceListIndex(_device));

removeRequest(_tenant, msg.sender, _device, _contractTerm);

bytes32 paymentAgreementHash = keccak256(abi.encodePacked(now, _
    tenant, msg.sender));
PaymentProvider paymentProvider = PaymentProvider(paymentProvider_
    addr);
paymentProvider.addPaymentAgreement(paymentAgreementHash, msg.sender
    , address(uint160(_tenant)), _device);

agreements.push(RentalAgreement(address(uint160(_tenant)), new bytes
    (65), msg.sender, _lessorSignature, _device, _usageFee, _
    contractTerm, now, AgreementState.Pending, paymentAgreementHash)
    );
}

function agreementExists(bytes memory _lessorSignature) public view
    returns (bool) {
    for(uint i = 0; i < agreements.length; i++) {
        if(keccak256(agreements[i].lessorSignature) == keccak256(_
            lessorSignature)) {
            return true;
        }
    }
    return false;
}

function agreementExists(address _tenant, address _device) public view
    returns (bool) {
    require(isKnownParticipant(_tenant, 2));
    require(isKnownParticipant(_device, 5));
    for(uint i = 0; i < agreements.length; i++) {
        if(agreements[i].device == _device && (agreements[i].state ==
            AgreementState.Pending || agreements[i].state ==
            AgreementState.Active)) {
            return true;
        }
    }
    return false;
}

function getIDs(uint _stateID) public view returns (uint[] memory) {
    uint count = 0;
    for(uint i = 0; i < agreements.length; i++) {
        if((agreements[i].tenant == msg.sender || agreements[i].lessor ==
            msg.sender) && agreements[i].state == AgreementState(_stateID)
            ) {
            count++;
        }
    }
}

```

```

uint[] memory rentalAgreementIDs = new uint[](count);
uint index = 0;
for(uint i = 0; i < agreements.length; i++) {
    if((agreements[i].tenant == msg.sender || agreements[i].lessor ==
        msg.sender) && agreements[i].state == AgreementState(_stateID)
        ) {
        rentalAgreementIDs[index] = i;
        index++;
    }
}
return rentalAgreementIDs;
}

function getByID(uint _id) public view returns (address, bytes memory,
    address, bytes memory, address, uint, uint, uint, uint, bytes32)
{
    require(_id < agreements.length);
    RentalAgreement memory agreement = agreements[_id];
    require(msg.sender == agreement.tenant || msg.sender == agreement.
        lessor);
    return (agreement.tenant, agreement.tenantSignature, agreement.
        lessor, agreement.lessorSignature, agreement.device, agreement.
        usageFee, agreement.contractTerm, agreement.creation, uint256(
        agreement.state), agreement.paymentAgreementHash);
}

function accept(uint _id, bytes memory _signature) public payable {
    require(_id < agreements.length);
    require(now < agreements[_id].contractTerm);
    require(agreements[_id].state == AgreementState.Pending);
    require(agreements[_id].tenant == msg.sender);
    require(recoverSigner(getRentalAgreementHash(_id), _signature) ==
        msg.sender);
    require(msg.value > 0);
    agreements[_id].tenantSignature = _signature;
    agreements[_id].state = AgreementState(1);
    PaymentProvider paymentProvider = PaymentProvider(paymentProvider_
        addr);
    paymentProvider.charge.value(msg.value)(agreements[_id].
        paymentAgreementHash);
}

function splitSignature(bytes memory _sig) internal pure returns (uint
    8 v, bytes32 r, bytes32 s) {
    require(_sig.length == 65);
    assembly {
        // first 32 bytes, after the length prefix.
        r := mload(add(_sig, 32))
        // second 32 bytes.
        s := mload(add(_sig, 64))
        // final byte (first byte of the next 32 bytes).
        v := byte(0, mload(add(_sig, 96)))
    }
}

```

```

    }
    if (v < 27) {
        v += 27;
    }

    require (v == 27 || v == 28);
    return (v, r, s);
}

function recoverSigner(bytes32 _message, bytes memory _sig) internal
    pure returns (address) {
    (uint8 v, bytes32 r, bytes32 s) = splitSignature(_sig);
    return ecrecover(_message, v, r, s);
}

// function payForUsage(uint _id, uint _timestamp) payable public {
//     require(_id < agreements.length);
//     //payment must be in the past
//     require(_timestamp <= now);
//     // the contract must not be outdated
//     require(now < agreements[_id].contractTerm);
//     RentalAgreement memory rentalAgreement = agreements[_id];
//     //rentalAgreement must be accepted (and therefore signed) by
//     sender
//     require(recoverSigner(getRentalAgreementHash(_id),
//         rentalAgreement.tenantSignature) == msg.sender);
//     require(msg.sender == rentalAgreement.tenant);
//     require(msg.value == rentalAgreement.usageFee);
//     require(rentalAgreement.state == AgreementState.Active);
//     rentalAgreement.lessor.transfer(msg.value);
//     bytes32 agreementHash = getRentalAgreementHash(_id);
//     bytes32 paymentHash = getPaymentHash(_timestamp, rentalAgreement.
//         device, msg.sender, rentalAgreement.lessor, msg.value,
//         agreementHash);
//     payments[paymentHash] = Payment(_timestamp, rentalAgreement.
//         device, msg.sender, rentalAgreement.lessor, msg.value,
//         agreementHash);
//     paymentsByPayer[msg.sender].push(paymentHash);
//     paymentsByReceiver[rentalAgreement.lessor].push(paymentHash);
// }

function getRentalAgreementHash(uint _id) public view returns (bytes
    32) {
    require(_id < agreements.length);
    bytes32 message = keccak256(abi.encodePacked(agreements[_id].tenant,
        agreements[_id].lessor, agreements[_id].device, agreements[_id]
        ].usageFee, agreements[_id].contractTerm));
    bytes memory prefix = "\x19Ethereum Signed Message:\n32";
    return keccak256(abi.encodePacked(prefix, message));
}

```

```

// function getPaymentHash(uint _timestamp, address _device, address _
    payer, address _receiver, uint _amount, bytes32 _agreementHash)
    public view returns (bytes32) {
//     bytes32 message = keccak256(abi.encodePacked(_timestamp, _device,
//         _payer, _receiver, _amount, _agreementHash));
//     // bytes memory prefix = "\x19Ethereum Signed Message:\n32";
//     // return keccak256(abi.encodePacked(prefix, message));
//     return message;
// }

// function getPayment(bytes32 _paymentHash) public view returns (uint
    256, address, address, address, uint256, bytes32){
//     require(msg.sender == payments[_paymentHash].device || msg.sender
//         == payments[_paymentHash].payer || msg.sender == payments[_
//             paymentHash].receiver);
//     return (payments[_paymentHash].timestamp, payments[_paymentHash].
//         device, payments[_paymentHash].payer, payments[_paymentHash].
//         receiver, payments[_paymentHash].amount, payments[_paymentHash].
//         agreementHash);
// }

// function getIncomingPaymentHashes() public view returns (bytes32[]
    memory) {
//     return paymentsByReceiver[msg.sender];
// }
//
// function getOutgoingPaymentHashes() public view returns (bytes32[]
    memory) {
//     return paymentsByPayer[msg.sender];
// }

function terminate(uint _agreementID) public {
    require(_agreementID < agreements.length);
    RentalAgreement memory rentalAgreement = agreements[_agreementID];
    require(rentalAgreement.state == AgreementState.Active);
    require(msg.sender == rentalAgreement.tenant || msg.sender ==
        rentalAgreement.lessor);
    agreements[_agreementID].state = AgreementState.Terminated;

    PaymentProvider paymentProvider = PaymentProvider(paymentProvider_
        addr);
    paymentProvider.empty(agreements[_agreementID].paymentAgreementHash)
        ;

    rentableDevices.push(rentalAgreement.device);
}
}

```

```

// struct EIP712Domain {
//     string name;
//     string version;
//     uint256 chainId;
//     address verifyingContract;
//     bytes32 salt;
// }
//
// struct RentalSignatureObject {
//     address tenant;
//     address lessor;
//     address device;
//     uint256 fee;
//     uint256 term;
// }
//
// bytes32 constant public EIP712DOMAIN_TYPEHASH = keccak256(
//     "EIP712Domain(string name,string version,uint256 chainId,
//         address verifyingContract,bytes32 salt)"
// );
//
// bytes32 constant public RENTALSIGNATUREOBJECT_TYPEHASH = keccak256(
//     "RentalSignatureObject(address tenant,address lessor,address
//         device,uint256 fee,uint256 term)"
// );
//
// bytes32 public DOMAIN_SEPARATOR;

// constructor () public {
//     DOMAIN_SEPARATOR = hash(EIP712Domain({
//         name: "Device Rental",
//         version: '1',
//         chainId: 1579447585291,
//         verifyingContract: this
//         verifyingContract: address(this),
//         salt: bytes32(0xf2d857f4a3edcb9b78b4d503bfe733db1e3f6cdc2b
//             7971ee739626c97e86a558)
//     }));
// }
//
// function hash(EIP712Domain memory eip712Domain) internal pure
//     returns (bytes32) {
//     return keccak256(abi.encode(
//         EIP712DOMAIN_TYPEHASH,
//         keccak256(abi.encodePacked(eip712Domain.name)),
//         keccak256(abi.encodePacked(eip712Domain.version)),
//         eip712Domain.chainId,
//         eip712Domain.verifyingContract,
//         eip712Domain.salt

```

```

//    });
// }
//
// function hash(RentalSignatureObject memory _sigObj) internal pure
//     returns (bytes32) {
//     return keccak256(abi.encode(
//         RENTALSIGNATUREOBJECT_TYPEHASH,
//         keccak256(abi.encode(_sigObj.tenant)),
//         keccak256(abi.encode(_sigObj.lessee)),
//         keccak256(abi.encode(_sigObj.device)),
//         keccak256(abi.encode(_sigObj.fee)),
//         keccak256(abi.encode(_sigObj.term))
//         // keccak256(bytes(mail.contents))
//     ));
// }

//
// function verify(address _tenant, address _lessee, address _device,
//     uint256 _fee, uint256 _term, bytes memory _sig, address _signer)
//     public view returns (bool) {
//     (uint8 v, bytes32 r, bytes32 s) = splitSignature(_sig);
//     // Note: we need to use 'encodePacked' here instead of 'encode'.
//     RentalSignatureObject memory sigObj = RentalSignatureObject(_
//         tenant, _lessee, _device, _fee, _term);
//     bytes32 digest = keccak256(abi.encodePacked(
//         "\x19\x01",
//         DOMAIN_SEPARATOR,
//         hash(sigObj)
//     ));
//     return ecrecover(digest, v, r, s) == _signer;
// }

pragma solidity >= 0.5.0 < 0.7.0;
import "./Ownable.sol";

/// @notice https://programtheblockchain.com/posts/2018/02/23/writing-a-simple-payment-channel/

contract PaymentProvider is Ownable {

    struct PaymentAgreement {
        address payable sender;
        address payable receiver;
        address device;
        uint256 balance;
        uint256 numPayments;
        mapping(bytes32 => bool) usedSignatures;
        mapping(uint256 => Usage) history;
    }

    struct Usage {
        uint256 timestampStart;
        uint256 timestampEnd;
    }

```

```

    uint256 units;
    uint256 cost;
}

mapping(bytes32 => PaymentAgreement) public paymentAgreements;

address private rentalProvider;

/// @notice sets the rentalProvider address
/// @dev only callable by owner
/// @param _addr the address of the rentalProvider
function registerRentalProvider(address _addr) public onlyOwner {
    rentalProvider = _addr;
}

function addPaymentAgreement(bytes32 _hash, address payable _receiver,
    address payable _sender, address _device) public {
    PaymentAgreement memory tmp;
    tmp.sender = _sender;
    tmp.receiver = _receiver;
    tmp.device = _device;
    tmp.balance = 0;
    tmp.numPayments = 0;
    paymentAgreements[_hash] = tmp;
}

// The sender can charge the channel
function charge(bytes32 _hash) public payable {
    require(msg.sender == paymentAgreements[_hash].sender || msg.sender
        == rentalProvider);
    paymentAgreements[_hash].balance = paymentAgreements[_hash].balance
        + msg.value;
}

function getSender(bytes32 _hash) public view returns (address) {
    require(msg.sender == paymentAgreements[_hash].sender || msg.sender
        == paymentAgreements[_hash].receiver);
    return paymentAgreements[_hash].sender;
}

function getReceiver(bytes32 _hash) public view returns (address) {
    require(msg.sender == paymentAgreements[_hash].sender || msg.sender
        == paymentAgreements[_hash].receiver);
    return paymentAgreements[_hash].receiver;
}

function getDevice(bytes32 _hash) public view returns (address) {
    require(msg.sender == paymentAgreements[_hash].sender || msg.sender
        == paymentAgreements[_hash].receiver);
    return paymentAgreements[_hash].device;
}

```

```

function getBalance(bytes32 _hash) public view returns (uint256) {
    require(msg.sender == paymentAgreements[_hash].sender || msg.sender
        == paymentAgreements[_hash].receiver);
    return paymentAgreements[_hash].balance;
}

function getPaymentHistory(bytes32 _hash) public view returns(uint
    256[] memory, uint256[] memory, uint256[] memory, uint256[] memory
    ) {
    uint count = paymentAgreements[_hash].numPayments;
    uint256[] memory timestampsStart = new uint256[](count);
    uint256[] memory timestampEnd = new uint256[](count);
    uint256[] memory units = new uint256[](count);
    uint256[] memory costs = new uint256[](count);
    for (uint i = 0; i<count; i++){
        timestampsStart[i] = paymentAgreements[_hash].history[i].
            timestampStart;
        timestampEnd[i] = paymentAgreements[_hash].history[i].
            timestampEnd;
        units[i] = paymentAgreements[_hash].history[i].units;
        costs[i] = paymentAgreements[_hash].history[i].cost;
    }
    return (timestampsStart, timestampEnd, units, costs);
}

function empty(bytes32 _hash) public {
    require(msg.sender == paymentAgreements[_hash].receiver || msg.
        sender == rentalProvider);
    paymentAgreements[_hash].sender.transfer(paymentAgreements[_hash].
        balance);
    paymentAgreements[_hash].balance = 0;
}

// The receiver can pay out his balance by presenting a signed
// amount from the sender. The receiver will be sent that amount but
// the channel will stay open.
// The rest of the balance is kept in the channel and not payed out
// back to the sender.
function redeem(bytes32 _hash, uint256 _timestampStart, uint256 _
    timestampEnd, uint256 _units, uint256 _cost, bytes memory _
    signature, address _device) public {
    bytes32 paymentHash = keccak256(abi.encodePacked(_hash, _
        timestampStart, _timestampEnd, _units, _cost, _signature, _
        device));
    require(paymentAgreements[_hash].usedSignatures[paymentHash] ==
        false, "Signature already used!");
    require(_timestampStart < _timestampEnd, "timestampStart lower than
        timestampEnd!");
    require(_device == paymentAgreements[_hash].device, "Device invalid
        !");
}

```



```

require(_cost <= paymentAgreements[_hash].balance, "Not enough money
!");
require(msg.sender == paymentAgreements[_hash].receiver, "You must
be the receiver in order to redeem a payment!");
require(isValidSignature(paymentAgreements[_hash].sender, _
timestampStart, _timestampEnd, _units, _cost, _device, _
signature), "Signature invalid!");
uint256 num = paymentAgreements[_hash].numPayments;
paymentAgreements[_hash].history[num] = Usage(_timestampStart, _
timestampEnd, _units, _cost);
paymentAgreements[_hash].numPayments++;

uint256 totalCosts = _cost * _units;

paymentAgreements[_hash].receiver.transfer(totalCosts);
paymentAgreements[_hash].usedSignatures[paymentHash] = true;
paymentAgreements[_hash].balance = paymentAgreements[_hash].balance
- totalCosts;
}

// The sender can extend the expiration at any time.
// function extend(uint256 newExpiration) public {
//     require(msg.sender == sender);
//     require(newExpiration > expiration);
// }
// expiration = newExpiration;
// }

// If the timeout is reached without the receiver closing the channel,
// then
// the ether is released back to the sender.
// function claimTimeout() public {
//     require(now >= expiration);
//     selfdestruct(sender);
// }

#####Helper#####

/// @notice gets the address of this contract
function getAddress() public view returns (address) {
    return address(this);
}

function isValidSignature(address _sender, uint256 _timestampStart,
uint256 _timestampEnd, uint256 _units, uint256 _cost, address _
device, bytes memory _signature) public view returns (bool) {
    bytes32 message = prefixed(keccak256(abi.encodePacked(this, _
timestampStart, _timestampEnd, _units, _cost, _device)));

    // Check that the signature is from the payment sender.
    return recoverSigner(message, _signature) == _sender;

```

```

}

function splitSignature(bytes memory sig) public pure returns (uint8,
    bytes32, bytes32) {
    require(sig.length == 65);

    bytes32 r;
    bytes32 s;
    uint8 v;

    assembly {
        // first 32 bytes, after the length prefix
        r := mload(add(sig, 32))
        // second 32 bytes
        s := mload(add(sig, 64))
        // final byte (first byte of the next 32 bytes)
        v := byte(0, mload(add(sig, 96)))
    }
    if (v < 27) {
        v += 27;
    }

    require (v == 27 || v == 28);

    return (v, r, s);
}

function recoverSigner(bytes32 message, bytes memory sig) public pure
    returns (address) {
    uint8 v;
    bytes32 r;
    bytes32 s;

    (v, r, s) = splitSignature(sig);

    return ecrecover(message, v, r, s);
}

// Builds a prefixed hash to mimic the behavior of eth_sign.
function prefixed(bytes32 hash) public pure returns (bytes32) {
    return keccak256(abi.encodePacked("\x19Ethereum Signed Message:\n
        32", hash));
}
}

pragma solidity >= 0.5.0 < 0.7.0;

import "./Ownable.sol";
// import "./DateLib.sol";

/// @title IdentityProvider
/// @author Sebastian Kanz

```

```

/// @notice Taken from: https://github.com/jrkosinski/oracle-example/
blob/part2-step1/oracle/contracts/BoxingOracle.sol
/// @notice Collects and provides information on identities
contract IdentityProvider is Ownable {
    mapping(address => Identity) identities;

    address[] knownDevices;
    address[] knownManufacturers;
    address[] knownServiceProviders;
    address[] knownSuppliers;

    // using DateLib for DateLib.DateTime;

    //defines an identity
    struct Identity {
        string name;
        uint256 date;           //GMT timestamp of date of registration
        Role role;
        address ownedBy;
        bool exists;
    }

    //possible roles of identities
    enum Role {
        None,
        Manufacturer,
        Customer,
        ServiceProvider,
        Supplier,
        Device
    }

    constructor () public {
        owner = msg.sender;
        // addTestData();
    }

    /// @notice checks whether the identity is known or not
    /// @param _identity the identity address to analyze
    /// @return returns true if the identity is known
    function identityExists(address _identity) public view returns (bool)
    {
        if(identities[_identity].exists) {
            return true;
        } else {
            return false;
        }
    }

    /// @notice gets the role of the identity
    /// @param _identity the identity address to analyze

```

```

/// @return a string representative of the Role. If the identity is
    unknown it returns the default (Role.None)
function getIdentityRoleName(address _identity) public view returns (
    string memory) {
    Role role = identities[_identity].role;
    if(role == Role.Manufacturer) {
        return "Manufacturer";
    } else if(role == Role.Customer) {
        return "Customer";
    } else if(role == Role.ServiceProvider) {
        return "ServiceProvider";
    } else if(role == Role.Supplier) {
        return "Supplier";
    } else if(role == Role.Device) {
        return "Device";
    } else {
        return "None";
    }
}

/// @notice gets the owner of the identity (if it is a device)
/// @param _identity the identity address to analyze
/// @return an address. If the identity is owned by no one it returns
    the default address (0x0..)
function getIdentityOwner(address _identity) public view returns (
    address) {
    return identities[_identity].ownedBy;
}

/// @notice gets the role of the identity
/// @param _identity the identity address to analyze
/// @return a uint. If the identity is unknown it returns the default
    (0)
function getIdentityRole(address _identity) public view returns (uint)
{
    return uint(identities[_identity].role);
}

/// @notice gets the creation time of the identity
/// @param _identity the identity address to analyze
/// @return a unix timestamp. If the identity is unknown it returns
    the default (0)
function getIdentityCreationTime(address _identity) public view
    returns (uint256) {
    return identities[_identity].date;
}

/// @notice gets the name of the identity
/// @param _identity the identity address to analyze
/// @return a unix string. If the identity is unknown it returns the
    default (empty string)

```

```

function getIdentityName(address _identity) public view returns (
    string memory) {
    return identities[_identity].name;
}

/// @notice adds a new identity
/// @dev only callable by owner of smart contract
/// @param _name the name of the identity
/// @param _roleID the role of the identity as integer representative
/// @param _identity the identity address to analyze
/// @param _owner the owner of the identity
function addIdentityOwnedBy(string memory _name, uint _roleID, address
    _identity, address _owner) onlyOwner public {
    require(!identityExists(_identity));
    require(identityExists(_owner));
    require(_roleID == 5);
    Identity memory newIdentity = Identity(_name, now, Role(_roleID), _
        owner, true);
    identities[_identity] = newIdentity;
    knownDevices.push(_identity);
}

/// @notice adds a new identity
/// @dev only callable by owner of smart contract
/// @param _name the name of the identity
/// @param _roleID the role of the identity as integer representative
/// @param _identity the identity address to analyze
function addIdentity(string memory _name, uint _roleID, address _
    identity) onlyOwner public {
    require(!identityExists(_identity));
    require(_roleID > 0 && _roleID != 5);
    Identity memory newIdentity = Identity(_name, now, Role(_roleID), _
        identity, true);
    identities[_identity] = newIdentity;
    if(Role(_roleID) == Role.Manufacturer) {
        knownManufacturers.push(_identity);
    } else if(Role(_roleID) == Role.ServiceProvider) {
        knownServiceProviders.push(_identity);
    } else if(Role(_roleID) == Role.Supplier) {
        knownSuppliers.push(_identity);
    } else if(Role(_roleID) != Role.Customer) {
        revert("This should not happen.");
    }
}

/// @notice deletes an identity
/// @dev only callable by owner of smart contract
/// @param _identity the identity address to analyze
/// @return a boolean whether or not deleting was successful
function deleteIdentity(address _identity) onlyOwner public returns (
    bool) {

```

```

    if(identityExists(_identity)) {
        identities[_identity] = Identity("", 0, Role.None, address(0x0),
            false);
        return true;
    } else {
        return false;
    }
}

/// @notice tests the connection to the smart contract
/// @dev only for testing purposes
/// @return always true
function testConnection() public pure returns (bool) {
    return true;
}

/// @notice gets the address of this contract
function getAddress() public view returns (address) {
    return address(this);
}

/// @notice adds test identities
/// @dev only callable by owner of smart contract
function addTestData() onlyOwner public {
    addIdentity("Manufacturer", uint(Role.Manufacturer), address(0xe3CBf
        35cD6FadEA91C4b9A19882Af976725753dD));
    addIdentity("Customer1", uint(Role.Customer), address(0x37848e7bDDbF
        872093D346f89be9B507eDf6462b));
    addIdentity("Customer2", uint(Role.Customer), address(0x8Ef5906034be
        23248EF49C38fEf9f17265ab044F));
    addIdentityOwnedBy("Device1", uint(Role.Device), address(0xf00C699c
        96E05f919EE1C57B824a29cC6216352E), address(0xe3CBf35cD6FadEA91C4
        b9A19882Af976725753dD));
    addIdentityOwnedBy("Device2", uint(Role.Device), address(0xCF5f7aa
        0103662Ef67cF453F7d4A6bFDfF04057e), address(0xe3CBf35cD6FadEA91C
        4b9A19882Af976725753dD));
    addIdentityOwnedBy("Device3", uint(Role.Device), address(0x4A21C369a
        3B7f4C56eC4DcBAB706C30897Cb1845), address(0xe3CBf35cD6FadEA91C4b
        9A19882Af976725753dD));
    addIdentityOwnedBy("Device4", uint(Role.Device), address(0xEa05a
        247562D4865F6e2a5d9347EeB61C966A214), address(0xe3CBf35cD6FadEA
        91C4b9A19882Af976725753dD));
}

function getKnownManufacturers() public view returns(address[] memory)
{
    return knownManufacturers;
}

function getKnownServiceProviders() public view returns(address[]
    memory){
    return knownServiceProviders;
}

```

```
}

function getKnownSuppliers() public view returns(address[] memory){
    return knownSuppliers;
}

function getKnownDevices() public view returns(address[] memory){
    return knownDevices;
}
}
```

## LITERATUR

---

- [1] Alain Abran und James W. Moore, Hrsg. *Guide to the Software Engineering Body of Knowledge: 2004 Version SWEBOK*. Los Alamitos, CA: IEEE Computer Society Press, 2005. ISBN: 0-7695-2330-7. URL: <http://www2.computer.org/portal/web/swebok/2004guide>.
- [2] Mohammed Alani. *Guide to OSI and TCP/IP Models*. Jan. 2014. ISBN: 978-3-319-05152-9. DOI: [10.1007/978-3-319-05152-9](https://doi.org/10.1007/978-3-319-05152-9).
- [3] Kariappa Bheemaiah. "Block Chain 2.0: The Renaissance of Money". In: *wired* (2015). URL: <https://www.wired.com/insights/2015/01/block-chain-2-0/>.
- [4] Binance. *Die Geschichte der Blockchain*. 2019. URL: <https://www.binance.vision/de/blockchain/history-of-blockchain>.
- [5] Michael del Castillo. "Blockchain 50: Billion Dollar Babies". In: *Forbes* (2019). URL: <https://www.forbes.com/sites/michaeldelcastillo/2019/04/16/blockchain-50-billion-dollar-babies/#6dfb0c0657cc>.
- [6] R.J. Cloutier (Editor in Chief). *The Guide to the Systems Engineering Body of Knowledge (SEBoK)*. v.2.0. BKCASE. The Trustees of the Stevens Institute of Technology, International Council on Systems Engineering, Institute of Electrical and Electronics Engineers Computer Society, 2019.
- [7] K. Christidis und M. Devetsikiotis. "Blockchains and Smart Contracts for the Internet of Things". In: *IEEE Access* 4 (2016), S. 2292–2303. ISSN: 2169-3536. DOI: [10.1109/ACCESS.2016.2566339](https://doi.org/10.1109/ACCESS.2016.2566339).
- [8] Cisco. *Internet of Things*. 2016. URL: <https://www.cisco.com/c/dam/en/us/products/collateral/se/internet-of-things/at-a-glance-c45-731471.pdf>.
- [9] Arnold Daniels. *The rise of private permissionless blockchains*. 2018. URL: <https://medium.com/ltonetwork/the-rise-of-private-permissionless-blockchains-part-2-62553256953d>.
- [10] T. M. Fernández-Caramés und P. Fraga-Lamas. "A Review on the Use of Blockchain for the Internet of Things". In: *IEEE Access* 6 (2018), S. 32979–33001. ISSN: 2169-3536. DOI: [10.1109/ACCESS.2018.2842685](https://doi.org/10.1109/ACCESS.2018.2842685).
- [11] Forbes. *Blockchain's Secret 1,000 Year History*. 2018. URL: <https://www.forbes.com/sites/oliversmith/2018/03/23/blockchains-secret-1000-year-history/#6871020e18d2>.
- [12] R. Han, V. Gramoli und X. Xu. "Evaluating Blockchains for IoT". In: *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. 2018, S. 1–5. DOI: [10.1109/NTMS.2018.8328736](https://doi.org/10.1109/NTMS.2018.8328736).
- [13] IEEE. "Systems and software engineering – Life cycle processes – Requirements engineering". In: *ISO/IEC/IEEE 29148:2011(E)* (2011).



- [14] ISO/IEC. *ISO/IEC 25010 System and software quality models*. Techn. Ber. 2010.
- [15] Iiba. *Babok: A Guide to the Business Analysis Body of Knowledge*. Bd. 3. International Institute of Business Analysis, 2015. ISBN: 9781927584026. URL: <https://books.google.de/books?id=ogxTrgEACAAJ>.
- [16] Project Management Institute. *A Guide to the Project Management Body of Knowledge(PMBOK Guide)*. 4th. PMI global standard. Project Management Institute, 2010. ISBN: 9781933890661.
- [17] M Macdonald, Lisa Liu-Thorrold und R Julien. *The Blockchain: A Comparison of Platforms and Their Uses Beyond Bitcoin*. Feb. 2017. DOI: [10.13140/RG.2.2.23274.52164](https://doi.org/10.13140/RG.2.2.23274.52164).
- [18] Mohammad Maroufi, Reza Abdolee und Behzad Mozaffari Tazehkand. "On the Convergence of Blockchain and Internet of Things (IoT) Technologies". In: *CoRR abs/1904.01936* (2019). arXiv: [1904.01936](https://arxiv.org/abs/1904.01936). URL: <http://arxiv.org/abs/1904.01936>.
- [19] Mehrdad Salimitari und Mainak Chatterjee. "A Survey on Consensus Protocols in Blockchain for IoT Networks". In: 2018.
- [20] Milan Sallaba, Dirk Siegel und Sebastian Becker. *IoT powered by Blockchain - How Blockchains facilitate the application of digital twins in IoT*. Techn. Ber. Deloitte, Blockchain Institute, 2018.
- [21] M. Samaniego und R. Deters. "Blockchain as a Service for IoT". In: *2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCoM) and IEEE Smart Data (SmartData)*. 2016, S. 433–436. DOI: [10.1109/iThings - GreenCom - CPSCoM - SmartData.2016.102](https://doi.org/10.1109/iThings-GreenCom-CPSCoM-SmartData.2016.102).
- [22] W3C. *Decentralized Identifier*. 2019. URL: <https://www.w3.org/TR/did-core/>.
- [23] Z. Zheng, S. Xie, H. Dai, X. Chen und H. Wang. "An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends". In: *2017 IEEE International Congress on Big Data (BigData Congress)*. 2017, S. 557–564.
- [24] H. Zimmermann. "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection". In: *IEEE Transactions on Communications* 28.4 (1980), S. 425–432. ISSN: 1558-0857. DOI: [10.1109/TCOM.1980.1094702](https://doi.org/10.1109/TCOM.1980.1094702).
- [25] Sorin Zoican, Marius Vochin, Roxana Zoican und Dan Galatchi. "Blockchain and Consensus Algorithms in Internet of Things". In: Nov. 2018. DOI: [10.1109/ISETC.2018.8583923](https://doi.org/10.1109/ISETC.2018.8583923).