

bloqs.at

Stefan Huber

Version 0.0.2

Inhalt

Einführung	1
Bitcoin	1
Ethereum	2
Blockchain Anwendungen	4
Blockchain Anwendungsarchitektur	4
Szenario 1	5
Einführung: Eine Sportwette	5
Smart Contracts	6
Implementierung mit Solidity	7
Nutzung des Smart Contracts	11
Szenario 2	12
Problemstellung: Domain Name System	12
Demonstration einer Subdomain Registry	12
Smart Contract	13
Szenario 3	17
Einführung: Initial Coin Offerings	17
Crowdsale Contracts	22
Implementierung des Demo ICO	23
Anhang A: Ethereum Entwicklungsumgebung	29
Mist	29
geth	29
testrpc	29
truffle	29
Referenzen	30

Einführung

Bitcoin

Im Jahr 2008 wurde von unbekannten Autoren mit dem Pseudonym *Satoshi Nakamoto* eine Publikation mit dem Titel "Bitcoin: A Peer-to-Peer Electronic Cash System" [\[Bit2008\]](#) innerhalb einer Mailingliste [\[Mai2008\]](#) veröffentlicht. Die Publikation beschreibt auf 9 Seiten konzeptionell ein Transfersystem von virtuellen Tokens (Bitcoins), welche über das Internet ausgetauscht werden können. Neben dieser konzeptionellen Beschreibung lieferte Satoshi Nakamoto auch eine Open Source Implementierung von Bitcoin.

Bitcoin ist als verteiltes Peer-to-Peer Netzwerk organisiert, welches keinen zentralen Dienst erfordert. Somit kann jeder Internetnutzer die Bitcoin Software installieren und Teil des Netzwerks werden. Das Herzstück von Bitcoin stellt eine verteilte Datenstruktur dar (Blockchain), welche eine Liste mit allen Transaktionen zwischen den Netzwerkteilnehmern enthält. Veränderungen in dieser Datenstruktur können nur über einen Prozess der sich Mining nennt durchgeführt werden. Als Belohnung für ihre Arbeit bekommen Mining-Knoten Bitcoin-Token ausgeschüttet. Dies ermöglicht es neue Bitcoins im System zu erschaffen.

Zusammengefasst besteht Bitcoin aus 4 Bestandteilen bzw. Schlüsselinnovationen, welche in Kombination Bitcoin ausmachen:

- Ein dezentrales Peer-to-Peer Netzwerk: Das Bitcoin Protokoll
- Ein öffentliches Transaktionsbuch: Die Bitcoin Blockchain
- Ein Bündel an Regeln für die Transaktionsverifizierung und Tokenausgabe: Die Konsensus Regeln
- Ein Mechanismus zur dezentralen Konsenermittlung über den Status der Blockchain: Proof-of-Work Algorithmus

Bitcoin Adressen

Das Eigentum von Bitcoins wird über digitale Signaturen sichergestellt. Dabei fungieren Bitcoin Adressen als öffentlich austauschbare Schlüssel, welche weitergegeben werden können um eingehende Bitcoin Transaktion zu erhalten. Zu jeder Bitcoin Adresse gehört ein privater Schlüssel. Dieser Schlüssel kann vom Eigentümer der Bitcoin Adresse verwendet werden um Transaktionen zu signieren und damit die Zustimmung zu geben, dass von der zugehörigen Bitcoin Adresse eine Abbuchung vorgenommen werden darf. Jeder Netzwerkteilnehmer kann sich beliebig viele Bitcoin Adressen dezentral generieren. Aus Gründen der Anonymität sollte für jede Bitcoin Transaktion eine eigene Adresse generiert werden.

Bitcoin Transaktionen

Transaktionen stellen den wichtigsten Teil innerhalb des Bitcoin Netzwerks dar. Alle anderen Bestandteile des Bitcoin Systems ermöglichen den Austausch von Transaktionen zwischen den einzelnen Netzwerkteilnehmern. Jede valide Transaktion wird von Mining-Knoten über die Propagation eines Blockes in die Blockchain aufgenommen. Diese Aufnahme in die Blockchain

führt zu einer Veränderung der Beträge hinter den Bitcoin Adressen und ermöglicht somit die Werttransfers.

Bitcoin Mining

Der Prozess zur Generierung eines neuen gültigen Blocks und damit die Veränderung der Blockchain wird Mining genannt.

Ethereum

Ethereum ist als verteiltes Rechnernetzwerk konzipiert, sodass jegliches Programm (Turing-Completeness) innerhalb einer eigenen isolierten Ablaufumgebung (Ethereum Virtual Machine) deterministisch und reproduzierbar ausgeführt werden kann.

Das Ethereum Protokoll ist in ständiger Weiterentwicklung. Es wurden jedoch Design Prinzipien [\[Eth2017\]](#) festgelegt, welche die zukünftige Weiterentwicklung leiten. Im folgenden werden diese Prinzipien dargelegt:

- **Einfachheit:** Das grundlegende Protokoll soll so einfach wie möglich sein, sodass ein Programmierer in der Lage sein soll die Spezifikation gänzlich zu verstehen und nötigenfalls zu implementieren. Komplexität durch Optimierungen soll nicht vermieden werden sonder in eine Mittelschicht zwischen Basisprotokoll und Anwendung platziert werden.
- **Freiheiten:** In Analogie zum Konzept der *Net Neutrality* soll auch im Ethereum Protokoll keinerlei Bevorzugung oder Benachteiligung von bestimmten Transaktionen oder Contracts geschehen können.
- **Generalisierung:** Die Konzepte im Ethereum Protokoll sollen auf dem niedrigsten Level dargestellt werden, um sie dadurch beliebig miteinander kombinierbar zu machen.
- **Ohne Features:** Als folge der Generalisierung enthält das Ethereum Protokoll keinerlei Features welchen häufige Anwendungsfälle zugrundeliegen. Jegliche Funktionen entstehen aus der Kombination bzw. Nutzung der generalisierten Konzepte.
- **Risikobereitschaft:** Ist okay ein höheres Risiko mit einer Implementierung einzugehen, wenn dies nachhaltige Vorteile bringt.

Ethereum Konto (Account)

Ethereum Konten werden über eine 20 byte Adresse identifiziert (zB 0x829BD824B016326A401d083B33D092293333A830) und können genutzt werden um die Ethereum eigene Kryptowährung Ether über Transaktionen auszutauschen. Jedes Konto hat somit einen Kontostand, welcher in Ether geführt wird. In Ethereum können prinzipiell zwei Arten von Konten unterschieden werden:

- **Externe Konten:** Diese Konten werden von einem privaten Schlüssel kontrolliert. Über den privaten Schlüssel können Transaktionen signiert werden und somit der Kontostand verändert werden.
- **Vertragskonten:** Auf einem Vertragskonto findet sich der ausführbare Programmcode eines Smart Contract, welcher über Transaktionen aktiviert werden kann. Vertragskonten besitzen

ebenfalls einen Speicher, welcher den Zustand des Smart Contracts repräsentiert.

Ethereum Transaktionen

Zwischen den Ethereum Konten können Nachrichten bzw. Transaktionen ausgetauscht werden. Eine Transaktion enthält primär folgende Daten:

- Die Menge an Ether, welche vom Sender zum Empfänger übertragen werden soll.
- Falls der Empfänger ein Smart Contract ist, können auch Daten übertragen werden, welche als Eingabewerte für den Programmcode gelten.
- Ein sog. Gaspreis, welcher angibt wieviel die Transaktionsausführung maximal Kosten darf.

Neben Transaktionen zwischen bestehenden Ethereum Konten, gibt es eine spezielle Transaktion, welche zur Erstellung eines neuen Vertragskontos verwendet wird. Falls der Empfänger der Transaktion die leere Adresse ("0") ist, dann wird innerhalb des Netzwerkes ein neues Vertragskonto erstellt mit den mitgeführten Daten als Programmcode. Die Adresse des Vertragskontos wird aus der Absender Adresse abgeleitet.

Ethereum Zustandsübergänge und Codeausführung

Jede Transaktion führt bei Erfolg zu einer Zustandsänderung der Blockchain und wird innerhalb einer isolierten Instanz der Ethereum Virtual Machine ausgeführt. Jeder Schritt dieser Ausführung ist mit einer definierten Menge an GAS als Kosten bepreist. Jeder Transaktion muss daher genügend Ether als GAS mitgegeben werden, sodass die Ausführung finanziert werden kann. GAS wird somit als effizientes Mittel gegen Denial-of-Service Attacken im Ethereum Netzwerk eingesetzt. Folgendes muss für eine erfolgreiche Transaktionsausführung gelten:

- Die Transaktion muss wohl geformt und vom Absender korrekt signiert sein
- Es muss genügend Ether (für GAS und den Werttransfer) am Konto des Absenders vorhanden sein
- Es muss genügend GAS freigegeben werden um die Transaktion durchzuführen

Falls eine Transaktion fehlschlägt, werden alle Zustandsänderungen revidiert und das gesendete Ether wieder an den Absender rückgeführt. Bei einer erfolgreichen Transaktion wird die Zustandsänderung innerhalb eines Blockes gespeichert und ist nach Validierung der Miner innerhalb der Blockchain verfügbar.

Ethereum Blockchain und Mining

Die Ethereum Blockchain ist der Bitcoin Blockchain sehr ähnlich. Jegliche Zustandsänderung eines Kontos, sei es am Kontostand oder am Speicher, muss über einen Konsens im Netzwerk geschehen. Das Ethereum Netzwerk basiert ebenfalls auf einem Verbund von vielen und weltweit verteilten Rechnern. Die Mining Knoten im Netzwerk validieren Transaktionen über den oben genannten Prozess und propagieren die Transaktion bei Erfolg innerhalb eines Blockes. Das Ethereum Netzwerk hat eine sehr schnelle Block Validierungs Zeit von ungefähr 17 Sekunden, sodass eine gültige Transaktion relativ rasch integriert wird.

Blockchain Anwendungen

Die Blockchain Technologie wird nach Swan[\[Swa2015\]](#) als hoch disruptive Technologie eingestuft und in 3 Kategorien eingeteilt:

- **Blockchain 1.0 - Währungen:** Mit Bitcoin haben Blockchains ihren Ursprung genommen. Bitcoin oder ähnliche Altcoins (zB Litecoin, Dash oder Monero) sind sog. Kryptowährungen und können für Werttransfers über das Internet genutzt werden.
- **Blockchain 2.0 - Verträge:** In diese Kategorie würden alle Anwendungen fallen die mehrstufige Handlungsmuster umfassen, welche durch Regeln oder Wenn-Dann Beziehungen verknüpft sind. Dies könnten Crowdfunding Verträge, Initial Coin Offerings (ICOs), jegliche Formen von Wertpapieren bis hin zu Dezentralen Autonomen Organisationen (DAOs) sein.
- **Blockchain 3.0 - Dezentrale Anwendungen:** Hierin fallen ein breites Spektrum von Anwendungen, welche Einfluss auf politische Prozesse und den Rechtsstaat nehmen könnten. Diese Kategorie ist eher als Ausblick zu verstehen, folgendes könnte dies betreffen: Demokratische Prozesse und Wahlen, Ausstellung von Identitätsdokumenten, Notariat bis Hin zur Abbildung des Gesundheitswesens.

Blockchain Anwendungsarchitektur

Klassische Softwareanwendungen sind nach einem Client-Server Paradigma strukturiert. Teile der Anwendung finden sich dabei im Client und andere Teile im Server. Dies resultiert in einer sog. Schichtenarchitektur (auch Multitier Architektur genannt). Dabei kann eine Anwendung innerhalb von 3 Schichten betrachtet werden:

- **Präsentationsschicht:** Bei dieser Schicht spricht man vom Frontend oder auch Client. Diese ist für die Präsentation der Anwendungsdaten an den Benutzer bzw. für die Aufnahme von Eingaben vom Benutzer verantwortlich. In Web-Anwendungen würde sich die Präsentationsschicht im Web-Browser finden.
- **Anwendungsschicht:** Innerhalb dieser Schicht findet sich die Geschäftslogik der Anwendung. Dies sind alle Regeln einer Domäne, welche die Anwendung abbildet. Gängig ist der sog. Thin Client Ansatz, in dem sich diese Schicht gänzlich am Server befindet. Eine sog. Fat Client Variante wäre ebenfalls möglich, in der sich diese Schicht im Client befindet.
- **Datenhaltungsschicht:** Alle Daten welche langfristig innerhalb der Anwendung verfügbar sein müssen, werden über diese Schicht verwaltet. Die Schicht findet sich grundsätzlich am Server, da alle Benutzer der Anwendung mit den Daten interagieren müssen.

Jede Schicht bietet zur anderen eine Kommunikationsschnittstelle, welche als Application Programming Interface (API) bezeichnet wird. Die Anwendungsschicht könnte zB eine Restful API anbieten, welche der Client nutzen kann. Die Datenschnittstelle könnte eine SQL API anbieten über diese könnte die Anwendungsschicht mit der Datenbank kommunizieren.

Blockchains bieten ähnlich wie klassische Datenbanken Kommunikationsschnittstellen. Eine Anwendung kann diese Schnittstellen nutzen um lesenden oder schreibenden Zugriff auf die Blockchain bekommt.

Szenario 1

In diesem Kapitel soll anhand eines Beispiels das Konzept der Smart Contracts eingeführt und erklärt werden. Im ersten Abschnitt ([Einführung: Eine Sportwette](#)) wird das zugrunde liegende Beispiel einer Sportwette erklärt. Ein weiterer Abschnitt ([Smart Contracts](#)) soll weitere Details zu Smart Contracts erläutern. Gefolgt wird dies ([Implementierung mit Solidity](#)) von einer Beschreibung zur Implementierung mit der Programmiersprache Solidity. Abschließend wird erörtert wie Smart Contracts genutzt werden können ([Nutzung des Smart Contracts](#)).

Einführung: Eine Sportwette

Am Beispiel einer Sportwette sollen die Funktionsweise und die Eigenschaften eines Smart Contracts aufgezeigt werden. Dazu wird die Sportwette auf das Ergebnis eines Fußballspiels betrachtet. Ein Fußballspiel zwischen zwei Mannschaften A und B kann generell drei Ergebnisse bzw. Endzustände aufweisen:

- Mannschaft A gewinnt und Mannschaft B verliert
- Mannschaft B gewinnt und Mannschaft A verliert
- das Spiel geht unentschieden aus

Um nun eine Wette auf eines der möglichen Ergebnisse abzugeben, würde normalerweise ein sog. Buchmacher konsultiert. Dieser würde Wetten von unterschiedlichen Teilnehmern, mit unterschiedlichen Wetteinsätzen und natürlich unterschiedlichen Tipps entgegennehmen. Dies sollte klarerweise alles vor dem Start eines Fußballspiels geschehen. Aus den gesamten abgegebenen Wetten kann dann die sog. Gewinnquote für die jeweiligen Ergebnisse berechnet werden. Diese Quote gibt an mit welchem Faktor der Wetteinsatz im Falle eines richtigen Tipps multipliziert wird. Aus der Quote errechnet sich somit der mögliche Gewinn für die Wettteilnehmer. In Tabelle 1 wird ein Beispiel eines möglichen Wettverlaufs dargestellt. In diesem vereinfachten Beispiel ist davon auszugehen, dass keine Gebühren für den Buchmacher bzw. keine Steuern erhoben werden.

Tipp	Ergebnis	kumulierte Einsätze	Prozent	Gewinnquote
Tipp 1	Mannschaft A gewinnt	€ 20.000	25%	4
Tipp 2	Mannschaft B gewinnt	€ 50.000	62,5%	1,6
Tipp 3	unentschieden	€ 10.000	12,5%	8
	gesamt	€ 80.000	100%	

Die Wettquote berechnet sich somit konkret über den gesamten Wetteinsatz dividiert durch die kumulierten Wetteinsätze aller Teilnehmer für den jeweiligen Tipp. Die Gewinnquote bedeutet somit für einen Wettteilnehmer, dass für jeden Euro der investiert wird das 4-, 1,6- bzw. 8-fache als Gewinn verbucht werden kann. An den Beispielzahlen ist zu erkennen, dass eine Gewinnquote immer größer als 1 sein muss, andernfalls würde die Teilnahme an einer Wette keinen Sinn machen. Zusätzlich ist zu sehen, dass der Tipp mit dem kleinsten kumulierten Wetteinsatz die höchste Gewinnchance bietet. Eine Wette auf das Ergebnis mit der höchsten Gewinnchance ist oft auch mit dem größten Verlustrisiko verbunden.



Wetten könnten auch als Vorhersagen betrachtet werden und die kummulierten Wetteinsätze als Eintrittswahrscheinlichkeit gedeutet werden. Im Beispiel der Fußballwette könnte dies so interpretiert werden, dass mit einer Wahrscheinlichkeit von 12,5% ein Unentschieden gespielt wird. Blockchainstartups wie **Augur** [4: Augur: <https://augur.net>], **Gnosis** [5: Gnosis: <https://gnosis.pm/>] oder **Stox** [6: Stox: <https://www.stox.com>] machen sich genau diesen Umstand zu nutze und bietet Vorhersagen der "kollektiven Intelligenz" als Dienstleistung an. Dies ist auch unter den Bezeichnungen **crowd sourcing** oder **prediction markets** bekannt.

Smart Contracts

Üblicherweise würde eine Sportwette wie oben beschrieben von einem Buchmacher abgewickelt. Dieser Buchmacher würde sich um die ordnungsmäßige Abwicklung der Wette kümmern. Je nach Vertrauen gegenüber dem Buchmacher, würden Wettteilnehmer dort Wetten abschließen oder auch nicht. Je nach Einsicht der Wettteilnehmer könnte ein Buchmacher Gewinnquoten zu seinen Gunsten manipulieren oder hohe Gebühren für Gewinne erheben.

Die ordnungsmäßige bzw. korrekte Abwicklung einer Sportwette kann jedoch formal wie in einem Algorithmus bzw. einem Computerprogramm beschrieben werden. Diesen Umstand haben sich bereits etliche online Sportwettanbieter zu nutze gemacht. Anstelle einer vertrauenswürdigen Person, welche als Buchmacher Wetten entgegennimmt, kann dies auch über eine Software Anwendung geschehen. Hierbei wird jedoch nur die Abwicklung an sich digitalisiert, eine Manipulation ist dabei noch nicht ausgeschlossen. Gemeinhin ist es so, dass Software Anwendungen zentral verwaltet sind und es ist nicht möglich den Programmcode einzusehen. Es wäre einem Wettteilnehmer also nicht möglich zu prüfen, ob die Software des Sportwettanbieters so agiert wie angepriesen. Es könnte durchaus passieren, dass auch nach einer gewonnen Wette, der Wetteinsatz nicht ordnungsgemäß ausbezahlt wird.

Ein Smart Contract hingegen ist ähnlich einem Computerprogramm, das nachvollziehbar genau das macht, was beschrieben ist und dies unmanipulierbar durchführt. Im Englischen wird dies auch mit dem Ausspruch "Code is Law" bekräftigt. Dies deutet darauf hin, dass so wie es im Quelltext des Smart Contracts beschrieben ist, so gilt es auch (ähnlich wie ein Gesetz). Konzeptionell ist ein Smart Contract einem Computerprogramm sehr ähnlich, jedoch gibt es einige unterscheidende Merkmale:

- Smart Contracts sind autonom in ihrer Ausführung. Falls ein Smart Contract einmal im Netzwerk bereitgestellt ist, steht dieser dort für immer bereit. Die programmierten Regeln sind somit gültig und können nicht mehr abgeschaltet oder verändert werden.
- Die Kommunikation mit Smart Contracts wird über Transaktionen realisiert. Dabei kann eine Transaktion sowohl Daten als auch Werte, abgebildet als virtueller Token, enthalten. Mit der Transaktion wird dem Smart Contract die Hoheit über die übermittelten Werte übertragen.
- Für die Ausführung eines Smart Contracts muss bezahlt werden. Die Mining-Knoten, welche das Netzwerk betreiben, bekommen eine Gebühr (Mining Fee) als Entschädigung für ihre Arbeitsleistung. Jede Transaktion, welche mit dem Smart Contract interagiert, muss auch genügend Mittel bereitstellen, damit die programmierten Regeln ausgeführt werden.

Konkret ist die Ausführung des Smart Contracts in Abschnitte gegliedert. Jeder Abschnitt wird durch eine Transaktion bzw. durch ein Ereignis gestartet. Am Beispiel der Sportwette können 3 Abschnitte identifiziert werden:

- Abschnitt 1: Innerhalb dieses Abschnitts können Wettteilnehmer ihre Wetten platzieren. Diese Phase würde solange andauern, bis das Fußballspiel gestartet wird. Nach diesem Ereignis können keine Wetten mehr platziert werden.
- Abschnitt 2: Dieser Abschnitt würde während des Fußballspiels ablaufen. Der Smart Contract wäre im sog. Leerlauf und würde nur auf das Ereignis "Ende des Fußballspiels" warten.
- Abschnitt 3: Der letzte Abschnitt, nachdem das Spielergebnis feststeht, wäre die Auszahlung. Die Wettteilnehmer, welche die Wette zu ihren Gunsten platziert haben, können sich nun ihre Gewinne auszahlen.

Implementierung mit Solidity

Der Solidity Programmcode ist innerhalb einem Verzeichnisses auf github [7: Szenario 1 auf github: <https://github.com/getbloqs/scenario01>] hinterlegt und einsehbar. In diesem Abschnitt werden einige Ausschnitte des Programmcodes angeführt und erläutert. Dies soll dazu dienen den Smart Contract zu dokumentieren und wesentliche Konzepte der Programmiersprache Solidity einzuführen.

Listing 1

```
contract SportsBet {  
}
```

Jeder Smart Contract wird wie oben veranschaulicht (Listing 1) über das Schlüsselwort `contract` deklariert. Dies hat eine starke Ähnlichkeit zur Deklaration von Klassen in Objektorientierten Programmiersprachen (OOP). Im allgemeinen gibt es eine Vielzahl Ähnlichkeiten mit OOP, sodass für Softwareentwickler das Erlernen von Solidity eine geringe Lernkurve aufweist.

Listing 2

```
contract SportsBet is Owned {  
}
```

Solidity unterstützt auch das Konzept der Vererbung (Listing 2). Dies ist hilfreich um die Wiederverwendbarkeit von Programmcode zu erhöhen. Über das Schlüsselwort `is` erbt `SportsBet` von `Owned`. Alle Funktionen, welche in `Owned` definiert sind, sind somit auch Teil von `SportsBet`. Neben der einfachen Vererbung werden auch `Interfaces` bzw. `abstrakte Methoden` unterstützt, dies wird in einem späteren Abschnitt noch thematisiert.

```

contract SportsBet is Owned {

    // unique identifier of sports game
    string public game;

    function SportsBet(string _game) {
        game = _game;
    }
}

```

Eine wesentliche und wichtige Eigenschaft von Smart Contracts ist es Zustände zu speichern. Dazu können Attribute bzw. Zustandsvariablen deklariert werden. In Listing 3 wird dazu eine Variable mit Namen `game` deklariert. Diese Variable hat den Datentyp `string` und kann Zeichenketten enthalten. Solidity unterstützen unterschiedlichste Datentypen, eine umfassende Liste ist der Dokumentation [8: Liste aller Solidity Datentypen: <http://solidity.readthedocs.io/en/latest/types.html>] zu entnehmen. Mit der Angabe `public` wird die Sichtbarkeit der Variable für andere Smart Contracts angegeben. Somit kann die Variable `game` von anderen Smart Contracts abgefragt werden. Neben `public` gibt es weitere Sichtbarkeitsdeklarationen, welche ebenfalls für Funktionen gelten und weiter unten eingeführt werden.



Sichtbarkeit von Zuständen

Alle Zustände, welche innerhalb eines Smart Contracts hinterlegt werden, sind grundsätzlich über die Blockchain öffentlich einsehbar. Die Sichtbarkeit einer Variable innerhalb eines Smart Contracts (`public`, `private`, `internal` oder `external`) bezieht sich dabei nur auf die programmatischen Zugriffsmöglichkeiten anderer Smart Contracts. Es gibt keine Möglichkeit Daten innerhalb einer Blockchain "nicht-öffentlich" zu speichern. Somit sind alle Daten die von einem Smart Contract zur Bearbeitung benötigt werden öffentlich.

Ein Smart Contract kann des Weiteren über einen sog. Konstruktor verfügen. Listing 3 enthält ebenfalls einen Konstruktor. Dieser ist generell nichts anderes als eine normale Funktion, mit dem Unterschied, dass der Funktionsname mit dem Namen des Smart Contracts übereinstimmen muss. Eine Funktionsdeklaration wird über das Schlüsselwort `function` durchgeführt. Die Funktion kann Übergabeparameter definieren, welche zur Konstruktion des Smart Contracts mitgegeben werden müssen.

Listing 4

```
contract SportsBet is Owned {

    struct Bet {
        uint tip;
        uint amount;
    }

    mapping (address => Bet) bets;

    function bet(uint tip) public payable {
        if (tip < 1) {
            tip = 1;
        } else if (tip > 3) {
            tip = 3;
        }

        if (bets[msg.sender].tip == 0) {
            bets[msg.sender].tip = tip;
        }
        bets[msg.sender].amount += msg.value;
    }

}
```

Neben einfachen Datentypen wie in [Listing 3](#) die Variable `game` können auch komplexere Datentypen selbst definiert werden. Um eine Wette zu repräsentieren wird in [Listing 4](#) der komplexe Datentyp `Bet` eingeführt. Dieser enthält einen Zahlenwert `tip` für den Wettipp und einen weiteren Zahlenwert `amount` für die Höhe des Einsatzes deklariert. `uint` deklariert dabei einen sog. unsigned Integer, also einen Zahlenwert der nur positiv sein kann. Der komplexe Datentyp wird über das Schlüsselwort `struct` deklariert.

Wie bereits erwähnt besitzen Smart Contracts Funktionen. In [Listing 4](#) werden eine Funktionen des Smart Contracts `SportsBet` implementiert. Generell stellen Funktionen (je nach Sichtbarkeit) die Schnittstelle des Smart Contracts nach Außen dar. Diese Schnittstelle wird über die sog. Signatur der Funktion definiert. Die Signatur setzt sich aus unterschiedlichen Bestandteilen zusammen:

- Der Name der Funktion
- Den spezifizierten Übergabeparametern
- Den Rückgabewerten, falls diese definiert sind
- Sichtbarkeits- bzw. sonstigen Modifikatoren

Neben der Signatur, welche auch als Funktionskopf bezeichnet werden kann, gibt es einen Funktionskörper. Der Funktionskörper wird von zwei geschwungenen Klammern umschlossen `{ }`. Innerhalb dieses Körpers wird die Logik der Funktion implementiert. Dazu werden unterschiedliche Konstrukte der Programmierung angewandt. In der Funktion `bet` aus [Listing 4](#) werden sog. Kontrollstrukturen eingesetzt, um den übergebenen Parameter zu überprüfen. Die

verfügbaren Kontrollstrukturen in Solidity können in der Dokumentation [9: Kontrollstrukturen in Solidity: <https://solidity.readthedocs.io/en/latest/control-structures.html#control-structures>] eingesehen werden. Falls die Signatur Rückgabewerte definiert, müssen diese über das Schlüsselwort `return` übergeben werden. Dieses Schlüsselwort beendet ebenfalls die Ausführung der Funktion.

Sichtbarkeitsmodifikatoren

Funktionen und Zustandsvariablen besitzen eine Sichtbarkeit innerhalb des Smart Contracts bzw. nach Außen zum Netzwerk. Solidity bietet 4 verschiedene Sichtbarkeitsmodifikatoren.



- **external**: Externe Funktionen bilden u.a. die Schnittstelle nach Außen eines Smart Contracts. Externe Funktionen können nur über Transaktionen, von anderen Smart Contracts oder über einen Message Call aufgerufen werden. Ein interner Aufruf der Funktion ist nur über `this` möglich. Zustandsvariablen können nicht als **external** deklariert werden.
- **public**: Funktionen und Zustandsvariablen können intern oder extern aufgerufen werden. Für Zustandsvariablen, welche als **public** deklariert wurden, wird automatisch eine sog. Getter-Funktion erzeugt.
- **internal**: Ein Zugriff auf **internal** Funktionen oder Zustandsvariablen ist nur vom deklarierten oder vererbten Smart Contract möglich.
- **private**: Zustandsvariablen bzw. Funktionen dieser Sichtbarkeit sind nur innerhalb des Smart Contracts ansprechbar.

Die Defaultsichtbarkeit von Funktionen ist **public** und die von Zustandsvariablen ist **internal**.

Listing 4 enthält auch ein **mapping**. Mappings sind sehr wichtige und effiziente Strukturen zur Speicherung von Zuständen in Smart Contracts. In anderen Programmiersprachen werden Mappings auch assoziativ Speicher oder Hashtabellen genannt. Ein mapping ist demnach eine Datenstruktur um Werte anhand einen Schlüssels zu speichern bzw. abzufragen. Das deklarierte **mapping** in **Listing 4** speichert Wetten (**Bet**) anhand der Adresse (**address**) des Wetteteilnehmers.

Die Funktion **bet** in **Listing 4** speichert den übergebenen Parameter **tip** als Wette (**Bet**) innerhalb des **mapping** (**bets**). Die Funktion prüft zuerst ob ein valider Tipp (Tipp 1, Tipp 2 oder Tipp 3) abgegeben wurde. Dannach wird überprüft ob der Absender der Transaktion und somit der Aufrufer der Funktion **bet** bereits eine Wette abgegeben hat. Der Absender der Transaktion wird über die spezielle Variable **msg.sender** abgefragt. Falls noch kein Tipp abgegeben wurde, wird dieser für den Absender gesetzt. Dazu wird der Tip über das **mapping** mit dem Schlüssel **msg.sender** (**msg.sender** ist vom Datentyp **address**) in der Wette (**Bet**) gesetzt. In jedem Fall wird der Wetteinsatz erhöht. Dazu wird die Höhe des gesendeten Ethers zum Wetteinsatz (**amount**) addiert.

Die Transaktion, welche die Funktion des Smart Contracts aufruft, sendet Ether als Wetteinsatz mit. Dieses Ether wird dem Smart Contract bereitgestellt und dieser verfügt nun darüber.



Spezielle Variablen: Block bzw. Transaktions Eigenschaften

Funktionen haben Zugriff auf spezielle Variablen, welche wichtige Informationen über den aktuellen Ausführungskontext enthalten. So kann zB über die Variable `msg` auf den Absender der Transaktion (`msg.sender`) den gesendeten Betrag an Ether (`msg.value`) oder das noch verfügbare Gas (`msg.gas`) zugegriffen werden. Alle verfügbaren bzw. speziellen Variablen sind in der Dokumentation [11: Spezielle Variablen in Solidity: <https://solidity.readthedocs.io/en/latest/units-and-global-variables.html#special-variables-and-functions>] einsehbar.

Nutzung des Smart Contracts

Der Smart Contract wurde mit der Programmiersprache Solidity entwickelt, welche in einen Ethereum Smart Contract kompiliert werden kann. Das Ergebnis dieser Kompilierung ist letztlich nichts anderes als Bytecode, welcher auf der Ethereum Virtual Machine (EVM) ausgeführt werden kann.

Der kompilierte Smart Contract kann über eine Ethereum Wallet im Netzwerk bereitgestellt werden. Dies erfordert eine Transaktion an eine leere Adresse, welche als Nachricht den kompilierten Smart Contract enthält. Für diese Bereitstellung muss der Transaktion natürlich genügend Gas als Gebühr mitgegeben werden. Diese Bereitstellung führt ebenfalls den Konstruktor des Smart Contracts aus, das bereitgestellte Gas muss auch dafür reichen.

Nachdem der Smart Contract bereitgestellt wurde, kann über Transaktionen mit seiner öffentlichen Schnittstelle (`external` oder `public` Funktionen) kommuniziert werden. Der Smart Contract besitzt eine eindeutige Adresse und kann darüber identifiziert und angesprochen werden. Nutzer können Transaktionen an diese Adresse senden. Innerhalb der Nachricht können die gewünschten Funktionen bzw. Funktionsparameter angegeben werden. Am Beispiel der Sportwette würde innerhalb der Nachricht ein Tipp für das Ergebnis angegeben und die Funktion `bet` aufgerufen.

Szenario 2

Problemstellung: Domain Name System

Das Internet und alle darauf basierenden Dienste nutzen sog. IP-Adressen um Rechner im Netzwerk zu identifizieren. IP-Adressen sind mehrstellige Zahlenkombinationen und für Menschen schwer zu merken, sodass Domain-Namen eingeführt wurden um dies zu erleichtern. Ein Domain-Name ist ein leicht merkbarer Bezeichner, welcher quasi als Alias für eine IP-Adresse verwendet wird.

Das System, welches die Namensauflösung, also die Umwandlung von Domain-Name in eine IP-Adresse, ist das Domain Name System (DNS). DNS ist zwar grundsätzlich eine dezentrale Datenbank unterliegt jedoch zentraler Kontrolle. An der obersten Hierarchieebene steht dabei die Internet Corporation for Assigned Names and Numbers (ICANN), welche Top Level Domains (TLD) vergibt und die Verwaltung dieser an die jeweiligen regionalen Verwaltungen weiterreicht. Die regionalen Verwaltungen vertreiben die Domain-Namen dann über akkreditierte Registrare an Domain Käufer.

DNS basiert auf einer Serverinfrastruktur, welche Anfragen für die Übersetzung von Domain-Namen zu IP-Adressen vornehmen. Über die Kontrolle der DNS-Server können Domain-Namen in DNS-Zonen manipuliert und damit zB zensiert werden. So können zB Webseiten wie Wikipedia in Ländern wie der Türkei gesperrt werden.

Ein auf Blockchain-Technologie basierendes Domain Name System würde es ermöglichen nicht nur die Datenhaltung zu dezentralisieren, sondern auch die Kontrolle über die DNS-Einträge. So würde zB der DNS-Eintrag für Wikipedia nur vom Betreiber selbst verwaltet und verändert werden können.

Demonstration einer Subdomain Registry

Als Prototyp einer dezentralen Domain Registrierung wird dies anhand einer Demonstration mittels von Subdomain Redirects realisiert. Das "Eigentum" von Subdomains einer Toplevel Domain werden über einen Smart Contract verwaltet. Folgende Funktionen sind verfügbar:

- Es können für einen Gegenwert, welcher in Ether bezahlt wird, Subdomains für einen Zeitraum registriert werden.
- Der Eigentümer der Subdomain kann einen beliebigen Redirect Link hinterlegen. Bei Aufruf der Subdomain über einen Browser soll ein Redirect zum hinterlegten Link durchgeführt werden.
- Nach einer gewissen Dauer soll die Registrierung ablaufen. Der Eigentümer hat jedoch während des Registrierungszeitraums die Möglichkeit der Verlängerung der Registrierung.
- Der Ersteller des Smart Contracts soll die Möglichkeit haben unsachgemäße Registrierungen zu löschen.

Smart Contract

Im folgenden soll der entsprechende Smart Contract zur Registrierung von Subdomain Redirects besprochen werden. Dabei werden Ausschnitte des gesamten Smart Contracs einzeln angeführt.

Listing 2.1

```
contract SubdomainRedirect {  
  
    struct SubdomainEntry {  
        string redirect;  
        address owner;  
        uint registeredUntil;  
    }  
  
    uint public registrationFee = 5 finney;  
    uint public registrations = 0;  
    mapping (string => SubdomainEntry) entries;  
    mapping (uint => string) names;  
  
    // ...  
  
}
```

In [Listing 2.1](#) werden die generischen Smart Contract Attribute aufgeführt. Als wesentlicher Bestandteil ist die Datenstruktur `SubdomainEntry` zu nennen, welche eine Registrierung repräsentiert. Folgende Attribute sind im `SubdomainEntry` enthalten:

- `redirect` ist eine Zeichenkette, welche den hinterlegten Redirect Link enthält.
- `owner` repräsentiert den Eigentümer der Registrierung als Ethereum adresse.
- `registeredUntil` gibt den Zeitstempel an, an dem die Registrierung abläuft.

Im Smart Contract ist des weiteren eine `registrationFee` hinterlegt, welche die Kosten für eine Registrierung angibt. Im Beispiel werden 5 Finneys angegeben. `registrations` ist ein Zähler, welcher die Anzahl von Registrierungen angibt. `entries` enthält alle Registrierungen und sind über den Namen der Subdomain indexiert. Ein weiterer Index `names` ermöglicht die Abfrage des Namens der registrierten Subdomain anhand einer Ganzzahl.

Listing 2.2

```
contract SubdomainRedirect {  
  
    // ...  
  
    function createRegistration(string name, string redirect) external payable {  
        require(  
            msg.value >= registrationFee &&  
            ( entries[name].owner == address(0) ||  
              entries[name].registeredUntil < now ) &&  
            bytes(redirect).length > 0  
        );  
  
        entries[name].redirect = redirect;  
        entries[name].owner = msg.sender;  
        entries[name].registeredUntil = now + 1 years + 30 days;  
        names[registrations] = name;  
        registrations += 1;  
    }  
  
    // ...  
  
}
```

Eine Registrierung kann erstellt werden mittels der Funktion `createRegistration`, welche in [Listing 2.2](#) dargestellt ist. Die Registrierung benötigt zwei Parameter `name` und `redirect`, welche den zu registrierenden Subdomain Namen und andererseits den zu registrierenden Redirect Link spezifizieren.

Im ersten Schritt wird überprüft ob die Registrierung durchgeführt werden kann. Dazu wird überprüft ob die Gebühr enthalten ist und keine gültige Registrierung für den gewünschten Namen vorherrscht. Bei einer erfolgreichen Überprüfung wird der Sender der Transaktion als Besitzer der Registrierung angegeben. Weiters wird das Ablaufdatum mit einem Jahr und 30 Tagen gesetzt.

Listing 2.3

```
contract SubdomainRedirect {

    // ...

    function renewRegistration(string name) external payable {
        require(
            msg.value >= registrationFee &&
            entries[name].owner == msg.sender
        );

        entries[name].registeredUntil = now + 1 years + 30 days;
    }

    function updateRegistration(string name, string redirect) external {
        require(
            entries[name].owner == msg.sender &&
            entries[name].registeredUntil > now
        );

        entries[name].redirect = redirect;
    }

    // ...

}
```

Neben der Erstellung einer neuen Registrierung (siehe [Listing 2.2](#)) können bestehende Registrierungen (1) erneuert werden bzw. (2) editiert werden. In [Listing 2.3](#) werden die entsprechenden Funktionen `renewRegistration` und `updateRegistration` dargestellt:

- Die Erneuerung der Registrierung kann über die Funktion `renewRegistration` durchgeführt werden. Als Parameter benötigt die Funktion den registrierten Namen `name`. Falls die Registrierung bereits für den Sender (`msg.sender`) vorliegt und die entsprechende Gebühr (`msg.value`) enthalten ist, kann die Erneuerung für 1 Jahr und 30 Tage durchgeführt werden.
- Eine bestehende Registrierung kann über die Funktion `updateRegistration` durchgeführt werden. Dabei sieht die Signatur der Funktion die Parameter `name` und `redirect` vor, welche den Namen der Subdomain bzw. den neuen Redirect Link enthalten sollen. Falls die Registrierung auf den Sender (`msg.sender`) der Transaktion registriert ist, kann die Editierung durchgeführt werden.

Listing 2.4

```
contract SubdomainRedirect {  
  
    // ...  
  
    function clearRegistration(string name, bool full) ownerOnly external {  
        entries[name].redirect = "";  
  
        if (full) {  
            entries[name].owner = address(0);  
            entries[name].registeredUntil = 0;  
        }  
    }  
  
    // ...  
  
}
```

Für den Ersteller des Smart Contracts wird die Funktion bereitgestellt eine Registrierung aufzulösen. Dabei besteht die Möglichkeit einer schwachen Auflösung (quasi als Warnung) und einer vollen Auflösung. Die Funktion `clearRegistration` stellt dies bereit. Als Parameter wird der Name der Registrierung (`name`) und die Angabe ob eine volle Auflösung stattfinden soll (`full`).

Szenario 3

Einführung: Initial Coin Offerings

Initial Coin Offerings (ICOs) sind eine Möglichkeit für Startups bzw. Unternehmen ein sog. Crowdfunding mittels Einsatz der Blockchaintechnologie durchzuführen. Dabei werden innerhalb eines vorgegebenen Investitionszeitraums eigens geschaffene Token gegen Investitionen in Kryptowährungen ausgegeben. Die ausgegebenen Token können unterschiedlichste Funktionen bzw. Nutzen für den Investor haben. Ebenfalls, können sie wie eine Kryptowährung genutzt werden und zB auf Börsen gehandelt werden.

Mögliche Funktionen eines ICO Tokens:

- **Utility Token:** Die Unternehmen, welche Tokens ausgeben, integrieren den Token in ihr Geschäftsmodell. Der Token kann Vielfach als Gutschein betrachtet werden und wird benötigt um die Dienstleistung bzw. das Produkt des Unternehmens zu nutzen. Filecoin als erfolgreichster ICO im Jahre 2017 beispielsweise führt den Token Filecoin ein um Speicherplatz zu handeln. Nutzer des dezentralen Speicherdienstes können gegen die Bezahlung von Filecoins Daten im Netzwerk speichern. Die Anbieter von Speicherplatz können im Gegenzug Filecoins verdienen.
- **Security Token:** Security Tokens können als eine Form von Unternehmensbeteiligung betrachtet werden. Diese Form von ICO Tokens ist rechtlich bzw. regulatorisch eher problematisch einzustufen. Trotzdem gibt es erfolgreiche ICOs von Security Tokens. So wurde der PAY Token als Security Token ausgegeben und soll zukünftig Eigentümern des Tokens eine Gewinnausschüttung in Ether ermöglichen, welche direkt an den Tokenbesitzer (über den Smart Contract) gehen.

ICOs werden vorrangig auf der Ethereum Blockchain abgewickelt. Die Investitionen in den ICO werden demnach vorrangig in Ether getätigt. Der ERC-20 Standard hat sich als Token Standard herausgebildet um ICOs abzuwickeln. Es gibt kaum ICOs die nicht als ERC-20 Token ausgegeben werden. Demnach soll in Abschnitt [ERC-20 Token Standard](#) der ERC-20 Token Standard im Detail beschrieben werden.

ERC-20 Token Standard

Der ERC-20 Token Standard wurde von der Ethereum Community definiert um Token auf der Ethereum Blockchain zu repräsentieren. Ein gemeinsamer und akzeptierter Standard hat viele Vorteile für eine aufstrebende Technologie. Unterschiedliche Anwendungsmöglichkeiten können entstehen und den Standard als Basis nutzen. Einige ausgewählte Beispiele:

- **ERC-20 Wallets:** Wallets welche ERC-20 kompatible Token unterstützen, sind prinzipiell für alle Token, welche den Standard implementieren zugänglich. So können neue Token wie ein Plugin in der Wallet geladen werden. Das aufwändige Programmieren einer eigene Wallet für einen neuen Token würde somit entfallen.
- **ERC-20 Börsen:** Kryptowährungen und Tokens werden auf Börsen gehandelt, durch das Zusammentreffen von Angebot und Nachfrage, wird unter anderem der Marktpreis des Tokens ermittelt. Die Aufnahme eines neuen Token, welcher an der Börse gehandelt werden kann, ist

technisch einfach realisierbar. Für alle ERC-20 kompatiblen Tokens kann es eine Implementierung geben.

- ERC-20 Token Explorer: Alle Transaktionen von Kryptowährungen und Token sind generell öffentlich innerhalb der jeweiligen Blockchains einsehbar. Sog. Blockexplorer sind Benutzerfreundliche Web-Anwendungen, welche Blockchains einsehbar machen. Für ERC-20 Token kann ein Explorer entwickelt werden, welcher neue Tokens einfach integrieren kann.

Generische Token Repräsentierung

Tokens werden innerhalb eines Ethereum Smart Contracts abgebildet. Der Token Contract enthält eine einfache Bilanz, welcher die Tokenstände der einzelnen Konten aufführt. Im folgenden wird eine einfache Darstellung dieser Bilanz als Tabelle repräsentiert. Es werden 3 Token Besitzer **A**, **B** und **C** und deren Token Kontostände abgebildet. Zusätzlich wird die Gesamtsumme aller Token angezeigt.

Besitzer	Token Summe
A	640
B	50
C	100
...	...
Gesamtsumme	10.000

Alle Kontobewegungen innerhalb dieses Tokens werden über diese einfache Bilanz abgebildet. Ein Transfer von 40 Token von Besitzer **A** zu Besitzer **B** würde zu einer Transformation der Bilanz führen. Das Konto von **A** würde sich um 40 Token verringern und das Konto von **B** würde sich um 40 Token erhöhen. Das Ergebnis würde sich wie folgt darstellen:

Besitzer	Token Summe
A	600
B	90
C	100
...	...
Gesamtsumme	10.000

Intern hat jeder ERC-20 Token eine Datenstruktur, welche die Kontostände in dieser einfachen Form abbildet.

Allgemeine ERC-20 Token Schnittstelle

Der ERC-20 Tokenstandard definiert eine Schnittstelle für Ethereum Smart Contracts. Diese Schnittstellenbeschreibung besteht aus 4 wichtigen Attributen. Diese Attribute sind **name**, **symbol**, **decimals** und **totalSupply**. **name**, **symbol** und **decimals** sind zwar optional definiert im Standard, es sind jedoch essentielle Attribute für die Darstellung in Applikationen. Im folgenden sollen die Attribute kurz beschrieben werden: * **name**: Der Name bzw. die Bezeichnung des Tokens wird als Zeichenkette angegeben. Es ist keine maximale Länge für den Namen definiert, es sollte jedoch in Betracht der Darstellung in Benutzerschnittstellen auf die Länge geachtet werden. * **symbol**:

Jeder Token kann über ein Symbol verfügen, welches als kurze und prägnante Identifikation gilt. Dies ist vergleichbar mit Währungssymbolen. Generell gibt es keine Beschränkung der Symbollänge, gängig sind in jedem Fall 3-stellige Zeichenketten (zB *PAY* Repräsentiert TenX Tokens). Es ist zu erwähnen, dass es keine zentrale Registrierung für Symbole gibt, sodass durchaus ein Symbol mehrfach in Verwendung sein kann. Bei der Erstellung eines neuen Tokens sollte eine genaue Recherche vorangehen. * *decimals*: Die Angabe des Attribut *decimals* gibt die Teilbarkeit des Tokens an. Ein Token mit dem Wert 5 für das Attribut bedeutet das ein Token 5 Dezimalstellen hat: *1,00000*. Ein Kontostand wird immer in der kleinsten Stückelung eines Tokens geführt und als Ganzzahl angegeben (siehe dazu *totalSupply*). * *totalSupply*: Das Attribut *totalSupply* gibt die Gesamtmenge der Tokens an, welche sich in Umlauf befinden. Die Angabe ist dabei eine Ganzzahl, welche in der kleinste Stückelung der Tokens angegeben ist. Das Attribut *totalSupply* errechnet sich mit folgender Formel $totalSupply = (Anzahl\ Tokens) * 10^{decimals}$.

Neben diese 4 Attributen werden im Token Standard noch Funktionen definiert für den Transfer von Token zwischen unterschiedlichen Konten, bzw. auch zur Abfrage von Kontoständen. Die Funktion zum Transfer von Tokens hat folgende Signatur:

Listing 3.1

```
function transfer(address _to, uint256 _value) returns (bool success)
```

Die Funktion *transfer* ist verantwortlich für den Eigentumsübergang des angegebenen Tokenbetrags von einem Kontoeigentümer zu einem anderen. Dabei fungiert der Aufrufer der Funktion (*msg.sender*) als Quelle eines Tokenbetrages (*_value*), welcher auf dem Konto eines Empfängers (*_to*) gutgeschrieben wird. Die *transfer* Funktion gibt als Resultat immer einen boolschen Wert für den Erfolg des Transfers zurück. Es könnte beispielsweise versucht werden, dass der Sender mehr Token transferieren will als sich in seinem Besitz befinden. Dies würde mit einer Rückgabe von *false* resultieren.

Die andere wichtige Funktion *balanceOf* gibt den Kontostand eines angegebenen Kontos (*_owner*) zurück. Der Kontostand wird dabei als Ganzzahl repräsentiert, welche in der Einheit der kleinste Tokenstückelung angegeben ist. Die Signatur dieser Funktion sieht folgendermaßen aus:

Listing 3.2

```
function balanceOf(address _owner) constant returns (uint256 balance)
```

Zusammenfassend wird in Listing 3.3 ein vollständiger Smart Contract gezeigt, welcher die bisher besprochene ERC-20 Token Schnittstelle implementiert. Die Bezeichnung (*name*) des Tokens lautet *Example Token* und das Token Symbol (*symbol*) wird als *EXP* angegeben. Die Anzahl der gesamten verfügbaren Token wird mit *10.000* angegeben. Die Anzahl der Dezimalstelle wird auf *18* fixiert, somit ergeben sich $10.000 * 10^{18}$ Token in der kleinsten Stückelung. Es wird ebenfalls ein Konstruktor angegeben, welcher die definierte Gesamtmenge der Tokens auf dem Konto des Smart Contract Erstellers gutschreibt.

In der Funktion *transfer* wurden Überprüfungen eingeführt, welche einerseits ausschließen das Tokens an die *address(0)* gesendet werden. Zu dieser Adresse ist kein privater Schlüssel verfügbar und wird somit unbrauchbar. Ein Tokentransfer an die Adresse *address(0)* kommt einer Zerstörung

der jeweiligen Tokens gleich (sog. Burning von Token). Andererseits wird überprüft ob der Sender (`msg.sender`) überhaupt genügend Token zur Verfügung hat.

Listing 3.3

```
contract BasicERC20TokenExample {

    string public constant name = "Example Token";
    string public constant symbol = "EXP";
    uint8 public constant decimals = 18;
    uint256 public constant totalSupply = 10000 * (10 ** uint256(decimals));

    mapping(address => uint256) balances;

    function BasicERC20TokenExample() public {
        balances[msg.sender] = totalSupply;
    }

    function transfer(address _to, uint256 _value) public returns (bool) {
        require(_to != address(0));
        require(_value <= balances[msg.sender]);
        balances[msg.sender] = balances[msg.sender] - _value;
        balances[_to] = balances[_to] + _value;
        Transfer(msg.sender, _to, _value);
        return true;
    }

    function balanceOf(address _owner) public view returns (uint256 balance) {
        return balances[_owner];
    }

}
```

Weitere ERC-20 Token Funktionen

Neben den generischen Funktionen zum Transfer von Tokens (`transfer`) und zur Abfrage der Kontostände (`balanceOf`) von Token Besitzern, werden von der ERC-20 Token Schnittstelle noch Möglichkeit zur Vergabe von Abbuchungslimits für andere Tokenbesitzer bereitgestellt. So kann beispielsweise Token Besitzer **A** an Token Besitzer **B** ein Limit über die Verfügung von 100 seiner Token geben. **B** kann damit bis zur Ausschöpfung des Limits Transfers von Konto **A** tätigen. Dafür werden die Funktionen `approve`, `transferFrom` und `allowance` in der Schnittstelle spezifiziert.

Listing 3.4

```
function approve(address _spender, uint256 _value) returns (bool success)
```

Die Funktion `approve` ermöglicht es einem Kontoinhaber ein Limit für einen anderen Kontoinhaber zu setzen. In [Listing 3.4](#) wird dem `_spender` ein Betrag von Tokens (`_value`) als Limit es Kontoinhabers (`msg.sender`) gesetzt.

Listing 3.5

```
function transferFrom(address _from, address _to, uint256 _value) returns (bool success)
```

Das Limit das über die Funktion `approve` gesetzt wurde, kann über die Funktion `transferFrom` genutzt werden. Die Signatur der Funktion `transferFrom` findet sich in [Listing 3.5](#). So kann der Kontoinhaber dem ein Limit für ein anderes Konto (`_from`) gegeben wurde eine Anzahl von Tokens (`_value`) an einen dritten Kontoinhaber (`_to`) weitergeben. Die Funktion `transferFrom` kann für das entsprechende Konto (`_from`) solange aufgerufen werden bis das gesetzte Limit ausgeschöpft wurde. Jeder Aufruf der Funktion `transferFrom` minimiert das gesetzte Limit entsprechend um die Anzahl der Tokens (`_value`).

Ähnlich der Funktion `balanceOf` gibt es auch ein Äquivalent `allowance` zur Abfrage des gesetzten Limits das ein Kontoinhaber (`_owner`) einem Anderen (`_spender`) zur Verfügung stellt. In [Listing 3.6](#) wird die Signatur dieser Funktion dargestellt.

Listing 3.6

```
function allowance(address _owner, address _spender) constant returns (uint256 remaining)
```

Für den ERC-20 Token sind zusätzlich 2 Ereignisse definiert (`Transfer` und `Approval`), welche an entsprechender Stelle erzeugt werden sollten. So wird das `Transfer` Ereignis innerhalb einer erfolgreichen Ausführung der `transfer` bzw. `transferFrom` Funktion ausgeführt. Das `Approval` Ereignis innerhalb einer erfolgreichen Ausführung der `approve` Funktion. Anschließend an [Listing 3.3](#) soll nun in [Listing 3.7](#) eine vollständige ERC-20 Token Implementierung angeführt sein.

```

contract FullERC20TokenExample is BasicERC20TokenExample {

    mapping (address => mapping (address => uint256)) internal allowed;

    function transferFrom(address _from, address _to, uint256 _value) public returns
(bool) {
        require(_to != address(0));
        require(_value <= balances[_from]);
        require(_value <= allowed[_from][msg.sender]);

        balances[_from] = balances[_from] - _value;
        balances[_to] = balances[_to] + _value;
        allowed[_from][msg.sender] = allowed[_from][msg.sender] - _value;
        Transfer(_from, _to, _value);
        return true;
    }

    function approve(address _spender, uint256 _value) public returns (bool) {
        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
        return true;
    }

    function allowance(address _owner, address _spender) public view returns (uint256) {
        return allowed[_owner][_spender];
    }
}

```

Crowdsale Contracts

Neben dem ERC-20 Token Contract, welcher den Token repräsentiert der durch einen ICO ausgegeben wird, wird die Ausgabe selbst über einen Smart Contract getätigt. Crowdsale Contracts können dabei unterschiedlichste Formen annehmen und bieten dem Unternehmen das den ICO durchführt sehr viele Gestaltungsmöglichkeiten.

Im folgenden sollen gängige Parameter eines Crowdsale Contracts besprochen werden:

- *Pre-ICO Phasen:* Um den Verkauf von Token attraktiver zu machen, kann als Instrument ein Pre-ICO stattfinden. Vor dem Start des tatsächlichen ICO, können Tokens zu verringerten Preisen angeboten werden. Als Regel für den Smart Contract könnte ein Zeitraum definiert werden, welcher mit einer maximalen Anzahl von Pre-ICO Token angeboten wird. Beispiel: Eine Woche vor dem ICO Start werden 100.000 Token mit einem 10% Rabatt ausgegeben.
- *Investitionsschwelle:* Im ICO Contract kann eine Mindestinvestmentschwelle definiert werden. Dies würde bedeuten, dass bei einer Unterfinanzierung die Investoren ihr Investment wieder rückfordern können. Der Smart Contract würde dafür eine eigene Methode bereitstellen. Beispiel: Ein ICO wird mit einer mind. Investitionsschwelle von 20.000 Token definiert. Werden

nun bis Ende des ICO nur 19.000 Token verkauft, können die Investoren ihr Investment über den Smart Contract wieder rückfordern. Der Ersteller des ICO Contracts hat keine Möglichkeit vor Überschreiten der Investmentschwelle auf die Investitionen zuzugreifen.

- *Airdrops*: Dies eine spezielle Form um die Attraktivität des ICO zu steigern. So kann nach einem gewissen Auswahl-schema an Konten vor ICO Start Tokens verschenkt werden. Dies kann dann auch an eine Regel geknüpft sein, dass diese Token nur abgerufen werden können, wenn ein bestimmtes Investment gemacht wird. Beispiel: Für einen neuen ICO werden allen Investoren eines komplementären vorhergehenden ICO 10 Token gutgeschrieben.
- *Spezielle Incentives*: Es könnten Regeln im ICO definiert sein die nach unterschiedlichen Gegebenheiten Tokens an Investoren verschenken. Beispiel: Jeder zehnte Investor bekommt einen Token geschenkt.
- *Bounties*: Für die Unterstützung während des ICO können Token verschenkt werden. Dies kann zB bei der Übersetzung des ICO Whitepapers in unterschiedliche Sprachen passieren oder für spezielle Promotionen in Social Media Netzwerken. Beispiel: Für die Übersetzung des Whitepapers in die Sprache Deutsch werden 100 Token vergeben.

Implementierung des Demo ICO

Zur technischen Realisierung eines ICO werden einerseits die entsprechenden Smart Contracts (Token, Crowdsale) und andererseits eine ICO Web Anwendung benötigt. In [Crowdsale Smart Contract](#) wird der implementierte Crowdsale Contract besprochen und in [ICO Web Anwendung](#) wird auf die implementierte ICO Web Anwendung eingegangen.

Crowdsale Smart Contract

Im folgenden soll ein sehr einfacher Crowdsale Smart Contract besprochen werden. Die Implementierung findet sich in [Listing 3.8](#) aufgeführt.

- Der `ExampleCrowdsale` Contract enthält einen leicht modifizierten `ExampleToken` aus [Listing 3.6](#) und [Listing 3.7](#) als Attribut. Innerhalb des Konstruktors wird eine Instanz des `ExampleToken` erstellt und dem Attribut zugewiesen. Dies führt zur Erstellung eines eigenen Smart Contracts mit eigener Adresse, welcher den ERC-20 Token repräsentiert.
- Dem Konstruktor wird eine Rate (`_rate`) übergeben. Diese Rate bestimmt sozusagen den Wechselkurs zwischen Ether und dem auszuschüttenden ERC-20 Token.
- Dem Konstruktor wird ebenfalls eine maximale Investitionshöhe (`_cap`) übergeben. Dies bestimmt die maximale Menge an Tokens, welche ausgeschüttet werden können über den Crowdsale. Falls die investierten Weis (kleinste Einheit von Ethereum) das Maximum erreichen, können keine weiteren Tokens mehr generiert werden.
- Es gibt eine default Funktion (anonyme Funktion), welche standardmäßig bei Transaktionen an den Crowdsale Contract ausgeführt wird. Diese default Funktion ist sehr wichtig, da somit alle gängigen Ethereum Wallets unterstützt werden können. Nur wenige Wallets können zusätzliche Parameter, wie zB den aufzurufenden Funktionsnamen, übergeben.
- Die wichtige Funktion `buyTokens` generiert für den übergebenen Kontoinhaber (`beneficiary`) neue Tokens. Als wichtige Vorarbeit wird vorerst überprüft, ob es sich um eine valide Transaktion handelt und überhaupt neue Tokens generiert werden können. Dazu wird

überprüft, ob das Maximum (`cap`) noch nicht überschritten wurde und ob überhaupt eine Investition vorliegt (`wei > 0`). Ebenfalls wird geprüft ob es sich nicht um die `address(0)` handelt, welche die Transaktion initiiert hat. Dies würde nämlich zu einem sog. Token burning führen. Falls die Überprüfung positiv verläuft, werden anhand der Rate die Anzahl der neuen Tokens berechnet und im Token Contract erzeugt. Dazu wird die Funktion `mint` verwendet, welche in [Listing 3.9](#) aufgeführt ist.

- Die Funktion `mint` aus [Listing 3.9](#) zeigt auf, dass der Erzeuger bzw. Owner (`onlyOwner`) die Befähigung hat neue Tokens zu schöpfen. Im Beispiel würde der Erzeuger des `ExampleToken` der Crowdsale Contract sein und somit die einzige Instanz, welche neue Tokens schöpfen könnte.

```

contract ExampleCrowdsale {

    ExampleToken public token;
    address public wallet;
    uint256 public cap;
    uint256 public rate;
    uint256 public weiRaised;

    function ExampleCrowdsale(uint256 _rate, uint256 _cap) public {
        require(_rate > 0 && _cap > 0);

        token = new ExampleToken();
        rate = _rate;
        cap = _cap;
        wallet = msg.sender;
    }

    function () external payable {
        buyTokens(msg.sender);
    }

    function buyTokens(address beneficiary) public payable {
        require(beneficiary != address(0));
        require(validPurchase());

        uint256 weiAmount = msg.value;
        uint256 tokens = weiAmount * rate;
        weiRaised = weiRaised + weiAmount;

        token.mint(beneficiary, tokens);
    }

    function withdrawFunds() internal {
        wallet.transfer(this.balance);
    }

    function validPurchase() internal view returns (bool) {
        bool nonZeroPurchase = msg.value != 0;
        bool withinCap = (weiRaised + msg.value) <= cap;

        return nonZeroPurchase && withinCap;
    }
}

```

```
contract MintableToken extends BasicERC20TokenExample {  
  
    function mint(address _to, uint256 _amount) onlyOwner public returns (bool) {  
        totalSupply = totalSupply + _amount;  
        balances[_to] = balances[_to] + _amount;  
        return true;  
    }  
  
}
```

ICO Web Anwendung

Die zugehörige Web Anwendung zu einem ICO verfolgt primär die Aufgabe einer Investorin den ICO zu vermarkten. So kommt es vorrangig auf eine entsprechende Gestaltung an. Einige gängige Funktionen, welche jede ICO Web Anwendung haben soll, wurden im Beispiel implementiert und im Folgenden erläutert.

In [ICO Web Anwendung](#) findet sich ein Screenshot der implementierten Web Anwendung. Folgende Aspekte finden sich in der Web Anwendung:

- Die Adresse des Smart Contracts, welcher Investments entgegen nimmt, wird klar ersichtlich. Die Investorin würde diese Adresse in ihrer Wallet öffnen und würde somit die Zieladresse der zu tätigenen Transaktion besitzen.
- Der Wechselkurs zu dem die Investorin Tokens gegen Ether tauschen kann.
- Es gibt die Möglichkeit eine Suche mittels der Adresse der Kontoinhaberin zu tätigen, um den entsprechenden Tokenstand abzufragen.
- In [ICO Kontostand Abfrage](#) wird das Ergebnis der Tokenstandabfrage angezeigt.

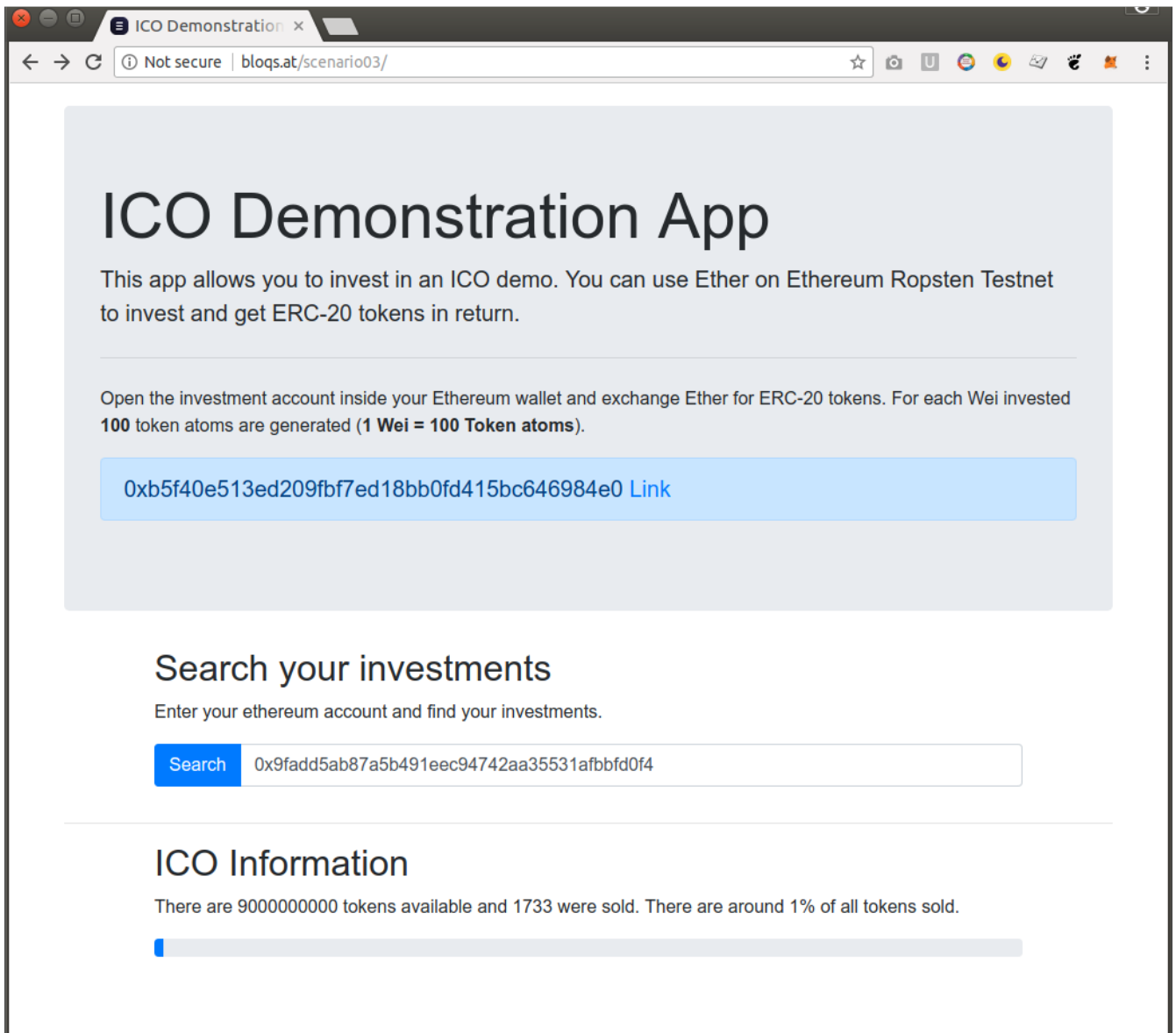


Figure 1. ICO Web Anwendung

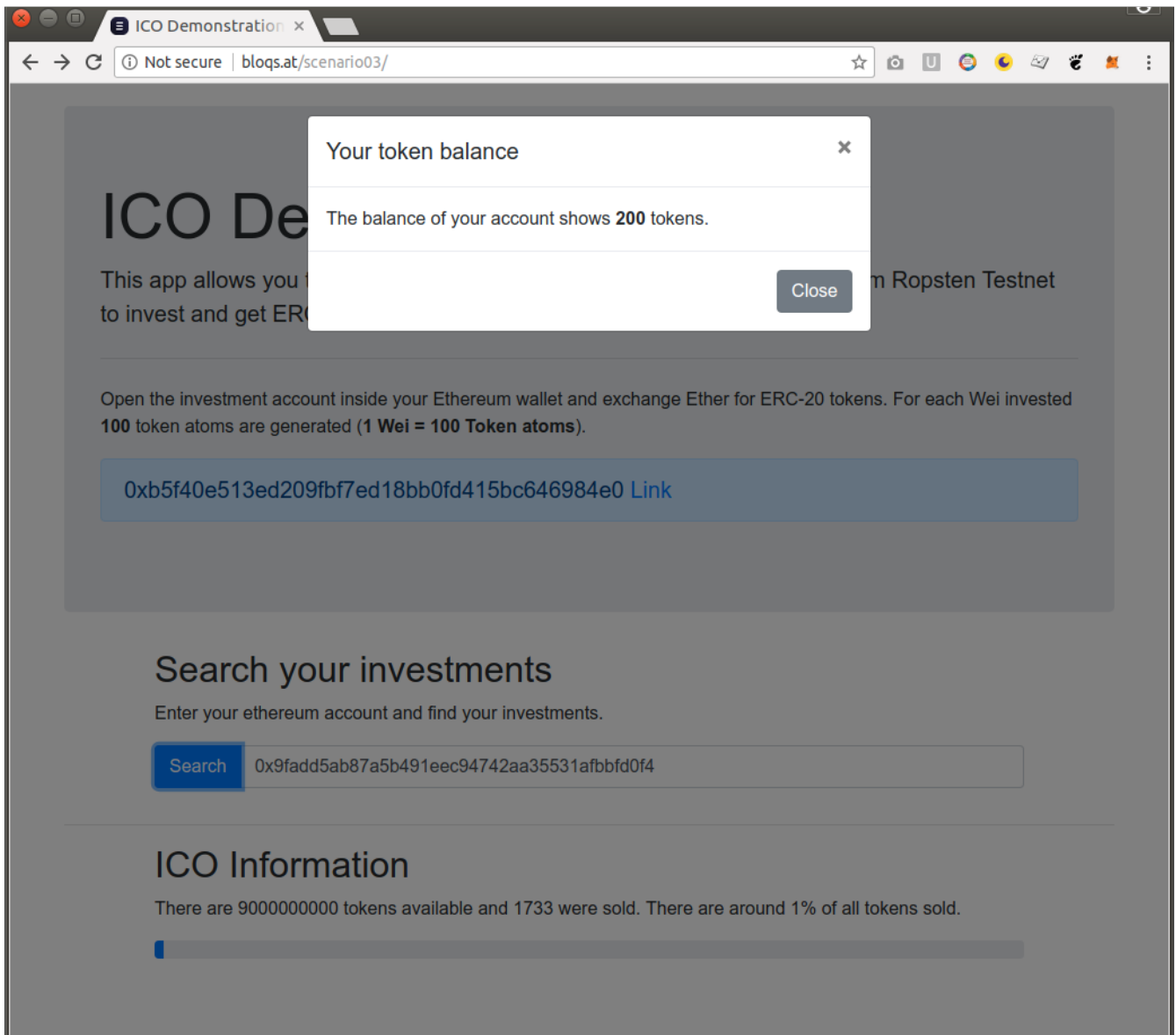


Figure 2. ICO Kontostand Abfrage

Ein wichtiger Aspekt zur Implementierung der Web Anwendung besteht darin mit dem Ethereum Peer-to-Peer Netzwerk zu kommunizieren. Eine Web Anwendung wird in einer sog. Browser Sandbox ausgeführt und kann aus Sicherheitsgründen keine Kommunikation direkt über TCP Sockets durchführen. Das Ethereum Peer-to-Peer Netzwerk verfügt über ein eigenes Protokoll, welches direkt auf der TCP Netzwerkschicht aufsetzt. Innerhalb der Browser Sandbox kann jedoch nur das HTTP-Protokoll genutzt werden.

Die ICO Web Anwendung benötigt für entsprechende Funktionen jedoch die Möglichkeit mit den Smart Contracts zu kommunizieren. Es müssen zB die Tokenstandsabfragen durchgeführt werden. In der Implementierung wurde der externe Dienst [<https://infura.io>](infura.io) genutzt, welcher einen kostenlosen HTTP Proxy für das Ethereum Netzwerk bereitstellt.

Anhang A: Ethereum Entwicklungsumgebung

Mist

geth

testrpc

Smart Contracts, welche für die Ethereum Blockchain entwickelt werden, sollten auch entsprechend getestet werden. Erst dannach sollten die Smart Contracts im Livenet bereitgestellt werden. Neben dem Livenet gibt es für Ethereum auch reale Testnetzwerke, welche von der Ethereum Community betrieben werden. Um Smart Contracts am eigenen Rechner zu entwickeln und testen, kann auf ein simuliertes Testnetzwerk zurückgegriffen werden. Die node.js Anwendung **testrpc** stellt ein solches simuliertes Netzwerk bereit.

Der Quellcode und etwas Dokumentation zu **testrpc** findet sich auf github [12: Link zu testrpc: <https://github.com/ethereumjs/testrpc>]. Die Anwendung kann mithilfe des **node package managers** installiert werden. Dazu muss folgender Befehl ausgeführt werden.

```
$ npm install -g ethereumjs-testrpc
```

Nach Installation kann das simulierte Netzwerk über die Kommandozeile gestartet werden. Defaultmäßig wird das Netzwerk auf Port **8545** ausgeführt. Dies kann jedoch über Optionen an den Startbefehl geändert werden. Details zu den unterschiedlichen Optionen ist der Dokumentation zu entnehmen.

truffle

Truffle [13: Link zu truffle: <http://truffleframework.com>] ist ein Kommandozeilenprogramm welches die Entwicklung von Smart Contract Projekten mit Solidity stark vereinfacht und erleichtert. Truffle stellt folgende Funktionen bereit:

...

Truffle ist eine node.js Anwendung und kann über den **node package manager** installiert werden. Dazu muss folgender Befehl ausgeführt werden:

```
$ npm install -g truffle
```

Referenzen

- [Bit2008] Bitcoin: A Peer-to-Peer Electronic Cash System, Satoshi Nakamoto; Link: <http://bitcoin.com/bitcoin.pdf>
- [Swa2015] Melanie Swan. Blockchain
- [Eth2017] Design Rationale, Ethereum; Link: <https://github.com/ethereum/wiki/wiki/Design-Rationale>
- [Mai2008] Mail Archive Cypherpunk Mailingliste; Link: <http://www.mail-archive.com/cryptography@metzdowd.com/msg09959.html>