Volume 18 entwickler spezial

BLOCKCHAIN

DIGITAL, DEZENTRAL, DISRUPTIV

KONZEPTE
Von den Grundlagen bis zum Smart Contract

TOOLS
Ethereum, Hyperledger Fabric, Solidity und Co.

PRAXIS
Blockchain-Anwendungen selbst entwickeln

Sonderdruck für
www.maibornwolff.de

Maiborn Wolff
Mensch IT

© Venomous Vector/Shutterstock.com

www.entwickler-magazin.de

+++ Smart Contracts +++ Ethereum +++ Hyperledger +++ Kryptowährungen +++

CI für DApps

Dezentrale Apps zentral testen mit CI

DApps (Decentralized Apps) sind verteilt laufende Anwendungen, die in der Regel auf einer Blockchain implementiert werden. Ihr Vorteil: Sie sind schwer angreifbar und sehr manipulationssicher. Das macht DApps für einige Anwendungsfälle zu einer interessanten Alternative zu Client-Server-Anwendungen. Als Entwickler sind wir jedoch an einige Arbeitsabläufe gewöhnt, die Frameworks für Blockchain-Entwicklung noch nicht umsetzen. Eine CI-Pipeline kann aber durchaus eingerichtet werden.

von Jacek Varky

Der Backend-Code einer DApp wird dezentral auf einem Peer-to-Peer-Netzwerk aus vielen Rechnern betrieben. Die verteilten Anwendungen kommen damit ohne zentrale Server aus. Das macht sie schwer angreifbar, daher stammt die Bezeichnung „unstoppable Apps“. Hinzu kommt: Wenn die Programme richtig programmiert sind, skalieren sie automatisch mit der Anzahl der Nutzer. Mehr Benutzer bedeuten auch mehr Nodes, die an der Weiterverteilung des Programms teilnehmen.

DApps müssen nicht zwingend auf einer Blockchain betrieben werden; die dezentrale Anlage der Blockchain macht eine Verknüpfung jedoch naheliegend. Zusätzlich kommen DApps so auch in den Genuss der Vorteile der Blockchain, etwa von Smart Contracts (Kasten: „Was sind Smart Contracts?“) und der Unveränderbarkeit der Blockchain.

Vor- und Nachteile von DApps

Wie jede Technologie, hat auch die Blockchain Vor- und Nachteile. Es ist wichtig, diese zu kennen, bevor man entscheiden kann, ob man ein Projekt als DApp umsetzten möchte. Die Vorteile der Technologie sind eine hohe Transparenz, Manipulations- und Ausfallsicherheit sowie geringere Transaktionskosten [1]:

- Transparenz: Nutzer und Entwickler können mit Hilfe von Etherscan [2] Nachrichten aus der Blockchain auslesen und validieren. Damit ist das System transparenter als eine Client-Server-Anwendung.

 © Software & Support Media GmbH

- Manipulationssicherheit: Die abgelegten Daten sind durch die Verwendung von Kryptografie unveränderlich und dadurch manipulations- und revisionssicher.
- Ausfallsicherheit: Die Blockchain bietet Ausfallsicherheit, da die Informationen nicht auf einem zentralen Server, sondern verteilt auf vielen Servern auf der gesamten Welt gespeichert werden. Wenn einer dieser Server nicht mehr erreichbar ist oder durch einen Cyberangriff kompromittiert wurde, gehen die Informationen der Blockchain trotzdem nicht verloren.
- Transaktionskosten: Die Blockchain erlaubt Transaktionen zwischen zwei Parteien, ohne eine dritte Partei, etwa eine Bank, einzuschalten. Wenn der unabhängige Dritte nicht für Dienste entlohnt werden muss, sinken die Transaktionskosten. Die Betriebskosten für die Blockchain bleiben natürlich; eine andere Art der Applikation würde jedoch auch Kosten für Infrastruktur und Betrieb verursachen.

Die Blockchain hat aber auch einige Nachteile, die auf Anwendungen durchschlagen:

- Skalierbarkeit: Die Blockchain selbst ist noch nicht skalierbar und im Vergleich zu einer zentralen Datenbank daher langsam. Eine DApp, die auf einer langsamen Blockchain betrieben wird, erbt somit deren Performanceprobleme.
- Transparenz: Sind Daten einmal in der Blockchain hinterlegt, können sie von jedem Nutzer und Entwickler zu jeder Zeit eingesehen werden. Was im einen Fall ein Vorteil ist, kann auch zum Nachteil werden.
- Speicherbedarf: Mit der Zeit wächst die Datenmenge in der Blockchain an, und jeder Teilnehmer, der die ganze Blockchain bei sich lokal speichert, muss diese Speichermenge vorhalten. Im Fall von Bitcoin sind das mittlerweile über 180 GB [3]; auch die Ethereum-Blockchain kommt auf eine für Privatleute schlecht handelbare Größe.

Es gibt einige Anwendungsfälle, die mehr Vorteile als Nachteile mit sich bringen. Wichtig ist, nicht dem „Shiny-New-Toy“-Syndrom zu verfallen und blind (für die Nachteile) jeden Anwendungsfall, in dem es eine neutrale Vermittlerrolle gibt oder der von Dezentralität profitieren könnte, auf Basis von Blockchain-Technologie lösen zu wollen. Eine genaue Abwägung tut hier Not.

Wenn ein Anwendungsfall deutlich von den Vorteilen der Blockchain profitiert, steht der Entwicklung einer DApp nichts mehr im Wege. Als Frameworks für die DApp-Entwicklung kommen Truffle [4] oder Embark [5] in Frage.

Embark: DApps entwickeln und testen

Truffle ist eine weit verbreitete Entwicklungs- und Testumgebung für die DApp-Entwicklung auf Ethereum. Das Framework unterstützt Tests in Solidity. Dadurch ist es möglich, Test Contracts zu schreiben, die mit anderen Smart Contracts interagieren. Wir nutzen Embark für viele unserer Prototyping-Projekte und haben deswegen nach einer Möglichkeit gesucht, eine CI-Pipeline dafür aufzusetzen.

Auch Embark ist eine Entwicklungs- und Testumgebung für DApps auf Ethereum. Mit Embark können entwickelte Smart Contracts auf einer privaten oder simulierten Blockchain getestet werden, bevor sie dann für alle Benutzer im öffentlichen Netzwerk bereitgestellt werden. Entwickler können also lokal auf dem eigenen Rechner Smart Contracts entwickeln und testen, ohne mit den Ethereum-(Test-)Netzwerken verbunden zu sein.

Hat man einen Smart Contract entwickelt, ermöglicht Embark das automatisierte Deployment beziehungsweise Redeployment nach Änderungen am Smart Contract. Embark unterstützt außerdem Test-driven Development [6]. So kann man mit JavaScript Testfälle für Smart Contracts schreiben und diese dann mit Embark automatisch ausführen. Embark ist sehr flexibel und bietet dem Entwickler die Möglichkeit, viele Einstellungen anzupassen. So kann jeder Entwickler für sich selbst entscheiden, wie sich Embark verhalten soll.

Embark ist das erste Tool in unserer CI-Pipeline. Als zweite Komponente wollen wir das Codeanalysetool Mythril einbinden.

Smart Contracts testen

Eine Blockchain ist im Prinzip unveränderlich. Das bedeutet auch: Nach dem Deployment ist es nicht mehr direkt möglich, Schwachstellen oder Bugs zu beheben. Dafür müsste man den Programmcode aktualisieren – und das geht eben nicht. Stattdessen muss man den aktualisierten Smart Contract neu deployen. Das kostet Zeit und birgt Risiken; beispielsweise müssen alle Referenzen auf den neuen Smart Contract aktualisiert werden. Deswegen ist es sehr wichtig, Smart Contracts vor dem Deployen gründlich zu testen. Das Testen von Smart Contracts kann in drei Kategorien unterteilt werden:

1. **Interaktion mit Smart Contract aus der Web3-Perspektive:** Um Smart Contracts aus der Web3-Perspektive zu testen, schreiben wir Unit-Tests mit

Was sind Smart Contracts?

Die meisten von uns kennen die Funktionsweise eines Smart Contracts bereits aus früher Kindheit: vom Kaugummiautomat. Wenn wir eine passende Münze in den Automaten eingeworfen haben, ließ sich der Knopf drehen. Je nach Fabrikat gab der Automat auch sofort eine Handvoll bunter, grauslich schmeckender Kaugummis frei. Was wir damals möglicherweise noch nicht reflektiert haben: Wir haben es mit einer Wenn-dann-Beziehung zu tun. Der Automat gibt die Ware nur aus, wenn die eingeworfene Münze bestimmte Bedingungen erfüllt. Dasselbe gilt für einen Smart Contract: Er ist eine Wenn-dann-Funktion – allerdings nicht hardwaregebunden am Bahnhof oder neben dem Briefkasten, sondern als Computerprogramm auf der Blockchain.

© Software & Support Media GmbH

Abb. 1: Embark testet den SimpleStorage Smart Contract

```
SimpleStorage
  ✓ should set constructor value
  ✓ set storage value (86ms)


2 passing (276ms)

> All tests passed
```

Listing 1: Embark im Docker-Container

```
FROM node:latest

USER node

#install ganache, embark and mocha
RUN npm install -g ganache-cli && npm
                    install -g embark

#create project dir
RUN mkdir /home/node/project

#set workdir
WORKDIR /home/node/project
```

Listing 2: Myrthil im Docker-Container

```
FROM ubuntu:bionic

RUN apt-get update \
 && apt-get install -y
        software-properties-common \
   python3-setuptools \
 && add-apt-repository -y
            ppa:ethereum/ethereum \
 && apt-get update \
 && apt-get -f install \
 && apt-get install -y \
  solc \
  libssl-dev \
  python3-dev \
  pandoc \
  git \
  python3-pip \
  python3-dev \
 && ln -s /usr/bin/python3
           /usr/local/bin/python \
 && pip3 install mythril \
 && mkdir /home/test/ \
 && cd /home/test

WORKDIR /home/test/
```

Listing 3: „.gitlab-ci.yml"-Datei

```
stages:
  - buildImage

variables:
  DOCKER_REGISTRY: "docker.
              maibornwolff.de"

buildEmbark:
  stage: buildImage
  image: docker:latest
  services:
    - docker:dind
  script:
    - docker build -t embark_ci -f
            Dockerfile_Embark .
    - docker push
      "$DOCKER_ARTIFACT_URL_EMBARK"

buildMythril:
  stage: buildImage
  image: docker:latest
  services:
    - docker:dind
  script:
    - docker build -t mythril_ci -f
            Dockerfile_Mythril .
    - docker push
      "$DOCKER_ARTIFACT_URL_MYTHRIL"
```

JavaScript. Embark und Truffle haben jeweils einen */test*-Ordner, in dem die JavaScript-Dateien mit den Tests abgelegt werden. Sobald man *embark test* oder *truffle test* in der Konsole eingibt, werden die Modultests aus dem Order */test* schrittweise ausgeführt. Wenn alle Testfälle erfolgreich ausgeführt worden sind, sieht die Ausgabe wie in **Abbildung 1** aus.

2. **Interaktion zwischen Smart Contracts:** Um die Interaktion zwischen Smart Contracts zu testen, schreiben wir die Testfälle nicht mit JavaScript, sondern mit Solidity selbst. Das erlaubt uns, die Interaktion zwischen Smart Contracts zu testen, die über Transaktionen kommunizieren.
3. **Schwachstellen in Smart Contracts:** Die vorherigen Tests überprüfen, ob sich Smart Contracts verhalten wie gewünscht. Sie decken jedoch keine Schwachstellen auf. Da Smart Contracts auf einer dezentralen Architektur ausgeführt werden, ergeben sich andere Schwachstellen als bei Applikationen auf einer zentralen Architektur. Um diesen Aspekt zu überprüfen, nutzen wir Mythril [7]. Mythril testet die Codequalität von Smart Contracts mit einer statischen Codeanalyse. Im Gegensatz zu Truffle oder Embark schreiben wir also keine Testfälle, sondern prüfen den Code auf bekannte Schwachstellen. Mythril ist Open Source. Es findet die gängigsten Schwachstellen, etwa die Top-10-Schwachstellen für Smart Contracts und DApps [8].

Infrastruktur für CI-Pipeline

Wir wollen mit GitLab und Docker eine simple CI-Pipeline aufbauen, die die Funktionalität und Sicherheit von Smart Contracts bei jedem Push überprüft. Damit vermeiden wir, dass wir manuell auf unseren Laptops testen müssen. Mit zwei Vorteilen: Die Tests werden automatisch bei jedem Push durchgeführt. Und wir binden unsere Entwicklungsrechner nicht für die Testzeit – bei einem Prototyp haben wir für den Test bei jedem Push mehr als eineinhalb Minuten gebraucht. Das ist auf die Dauer nicht sinnvoll. Letztlich sind wir erst mit der CI-Pipeline in der Lage, DApps nach modernen Software-Engineering-Methoden zu bauen.

Als Infrastruktur für unsere CI-Pipeline nutzen wir GitLab und Docker, beides Tools, die bei vielen Unternehmen und eben auch bei uns zur Standardausstattung von Projekten gehören. Damit GitLab-CI funktioniert, wird noch ein Runner benötigt. Ein Runner ist eine isolierte Maschine, die die definierten Testfälle aufgreift und ausführt. Außerdem haben wir noch eine Docker Registry, in der wir erstellte Docker Images intern speichern.

Docker fasst alle Konfigurationsdateien, Bibliotheken und Tools der Applikationen in einem Container zusammen. Das hat zwei signifikante Vorteile gegenüber virtuellen Maschinen: Größe und Modularität. Container isolieren Applikationen auf einem gemeinsamen Betriebssystem voneinander. Das bedeutet, dass ein Container nur die wichtigsten Dateien enthält und somit eine Größe im Bereich von 100 MB hat. Eine virtuelle

 © Software & Support Media GmbH

Maschine hingegen enthält ein gesamtes Betriebssystem und alle dazu benötigen Tools. Daraus resultiert, dass eine virtuelle Maschine mehrere Gigabyte groß ist. Ein weiterer Vorteil ist, dass ein Container ein modulares System erlaubt. Verschiedene Applikationen können in verschiedene Container gepackt und einzelne Module einfach ausgetauscht werden.

Im nächsten Schritt werden wir zwei Projekte in GitLab einrichten. Das erste Projekt benutzen wir zum Erstellen der Docker Images, das zweite Projekt enthält die Smart Contracts, die getestet werden sollen.

Container bauen

Für unsere CI-Pipeline bauen wir Docker-Container auf, die die jeweiligen Testtools installiert haben. Damit trennen wir Frameworks und Tools voneinander, und die Tests können parallel ausgeführt werden. Die erstellten Container werden im internen Docker Registry hinterlegt. Der Vorteil davon ist, dass wir nicht für jeden Test einen neuen Container erstellen müssen, sondern den vorgefertigten Container nutzen können. Wir benutzen das Docker Registry, um Docker Images abzulegen und herunterzuladen. Die Registry wird im internen Netzwerk gehostet und erlaubt die volle Kontrolle über die Images. Wer diese Kontrolle nicht benötigt, kann Docker Hub [9] verwenden.

Zuerst erstellen wir einen Docker-Container, in dem das Embark-Framework installiert ist. Das Erstellen eines Containers mit Truffle funktioniert sehr ähnlich. Das entsprechende Dockerfile ist in Listing 1 zu sehen.

Per Dockerfile wird ein Image auf Basis des Node Image erstellt, in dem Node.js bereits installiert ist. Zusätzlich werden Ganache-CLI [10] und Embark installiert. Ganache-CLI emuliert eine Blockchain, ohne einen Node zu benötigen. Außerdem wird noch ein *project*-Verzeichnis im *home*-Ordner des Nutzers *node* erstellt. Der zweiter Docker-Container enthält Mythril (Listing 2).

Das Image für Mythril basiert auf dem Ubuntu Bionic Image (Ubuntu 18.04). Im nächsten Schritt werden alle benötigten Tools installiert. Beide Dockerfiles werden im Root-Verzeichnis eines GitLab-Projekts abgelegt. Das führt aber noch nicht dazu, dass wir direkt eine Pipeline haben. Damit GitLab die Docker Images erstellt, wird noch eine sogenannte *.gitlab-ci.yml*-Datei im Root-Verzeichnis des GitLab-Projekts benötigt. In dieser Konfigurationsdatei werden sogenannte Jobs definiert, die der Runner aufgreift und ausführt (Listing 3).

In der gezeigten *.gitlab-ci.yml*-Datei werden zwei Jobs definiert, *buildEmbark* und *buildMythril*. Wenn beide Jobs die gleiche *stage*-Bezeichnung haben – hier *buildImage* –, dann werden diese Jobs vom Runner parallel ausgeführt. Jeder Job greift auf das entsprechende Dockerfile zu und erstellt ein Image, das dann in das interne Docker Registry gepusht wird (**Abb. 2**).

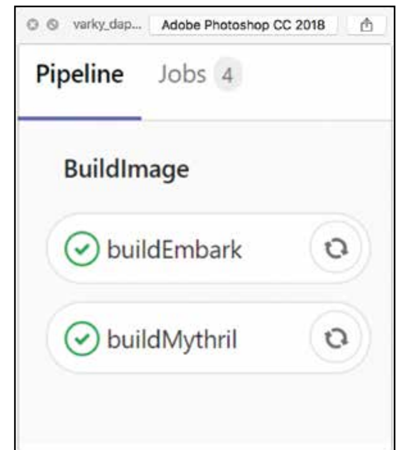

Abb. 2: Darstellung der erfolgreich ausgeführten Jobs im GitLab

Listing 4: Smart Contract

```
// contracts/SimpleStorage.sol
pragma solidity ^0.4.7;

contract SimpleStorage {
  uint public storedData;

  constructor(uint initialValue) {
    storedData = initialValue;
  }

  function set(uint x) {
    storedData = x;
  }

  function get() constant returns
                (uint retVal) {
    return storedData;
  }
}
```

Automatisiertes Testing in der CI-Pipeline

Damit haben wir die Docker Images erstellt, die wir für die Entwicklung von Smart Contracts verwenden können. Wir erstellen ein zweites GitLab-Projekt für das Dev-Projekt. Dafür nutzen wir Embark, das uns folgende Ordnerstruktur liefert:

Listing 5: JavaScript-Test

```
// test/simple_storage_spec.js
const SimpleStorage = embark.require('Embark/contracts/SimpleStorage');
let accounts;
config({
  contracts: {
    "SimpleStorage": {
      args: [100],
      onDeploy: ["SimpleStorage.methods.setRegistar(web3.eth.defaultAccount).send()"]
    }
  }
}, (err, theAccounts) => {
  accounts = theAccounts;
});

contract("SimpleStorage", function () {
  this.timeout(0);
  it("should set constructor value", async function () {
    let result = await SimpleStorage.methods.storedData().call();
    assert.strictEqual(parseInt(result, 10), 100);
  });


  it("set storage value", async function () {
    await SimpleStorage.methods.set(150).send();
    let result = await SimpleStorage.methods.get().call();
    assert.strictEqual(parseInt(result, 10), 150);
  });
});
```


© Software & Support Media GmbH

- Embark
  - *.embark*
  - *Build*
  - *contracts*
  - *test*
  - *chain.json*
  - *contracts.json*
  - *embark.json*
  - *package.json*

Im Ordner *contract* haben wir den Smart Contract aus Listing 4 hinzugefügt. Es ist ein Smart Contract mit einer simplen Funktion: Speichere einen Unsigned Integer und gib ihn zurück.

Im nächsten Schritt erstellen wir eine JavaScript-Datei (Listing 5), die den SimpleStorage Contract testet. In der JavaScript-Datei wird der Parameter für den Konstruktor beim Deployment des Contracts definiert. Sobald der Smart Contract deployt ist, wird im ersten Test überprüft, ob die *state*-Variable *storedData* den Wert 100 enthält. Im zweiten Testfall wird der Wert durch einen Aufruf der *set*-Funktion auf 150 verändert. Anschließend wird durch einen Aufruf der *get*-Funktion überprüft, ob der Wert tatsächlich 150 ist. Ist der Test erfolgreich durchlaufen, erhalten wir die Ausgabe in **Abbildung 1**: *All tests passed.*

Damit die Tests mit jedem Push in das Projekt automatisch ausgeführt werden, benötigen wir auch hier eine *.gitlab-ci.yml*-Datei im Root-Verzeichnis des Projekts. Den Inhalt der Konfigurationsdatei habe ich in Listing 1 gezeigt. Der Runner greift die definierten Jobs auf und führt sie parallel durch. Beim Laden der Images wird auch das gesamte GitLab-Projekt automatisch in den Docker-Container geladen. Somit stehen alle Dateien aus dem Projekt im Container direkt zur Verfügung. Im *script*-Teil der Konfigurationsdatei wird definiert, welche Befehle innerhalb des Containers ausgeführt werden sollen:

- Mit *cd embark && embark test* wird in den Embark-Ordner gewechselt und *embark test* ausgeführt.
- Bei Mythril wird *myth* mit dem Pfad zu den Contracts angegeben.

Listing 6: „gitlab-ci.yml“-Datei zum Ausführen von Embark- und Mythril-Tests

```
stages:
  - test

executeEmbark:
  stage: test
  image: embark_ci
  script:
    - cd embark && embark test

executeMythril:
  stage: test
  image: mythril_ci
  script:
    - myth -x embark/contracts/* >>
                          mythResults
    - cat mythResults
    - grep "No issues were detected. "
                          mythResults
```

Da Mythril das Ergebnis direkt auf dem Terminal ausgibt und der Exit-Code unabhängig vom Ergebnis immer 0 ist, speichern wir das Ergebnis in einer Datei und überprüfen, ob sie den String *No issues were detected.* enthält (Listing 6). Ist dieser String enthalten, wird der Job mit dem Status 0 abgeschlossen: *Job succeeded.* Falls wir diese Schritte nicht machen, wird der Job ebenfalls mit *Job succeeded* abgeschlossen, unabhängig vom Prüfergebnis.

Fazit und Ausblick

Wir haben gezeigt, wie man mit einer bereits bestehenden Infrastruktur eine Continuous-Integration-Pipeline für das Entwickeln von DApps aufbaut. Damit sind wir in der Lage, das Deployen und Testen von Smart Contracts zu automatisieren und dadurch den Entwicklungsprozess von DApps effizienter zu gestalten. Wir haben die Pipeline mit Embark und Mythril aufgebaut; das kann natürlich noch erweitert werden: Die hier vorgestellte Pipeline enthält zum Beispiel nur den Schritt „Test“. Dem würden weitere Schritte folgen, etwa das Deployment der Software. Eine Alternative für den Testschritt, den wir hier mit Embark durchgeführt haben, wäre Truffle.

Selbstverständlich ist es auch sinnvoll, aufwendigere Tests durchzuführen. Es ist jedoch wichtig anzumerken, dass die oben vorgestellten Tools nur Fehler finden, die man auch selbst testet, und Schwachstellen, die bereits bekannt sind. Soll ein Smart Contract im öffentlichen Ethereum-Netzwerk deployt werden, oder in einem Netzwerk, dessen Teilnehmern man misstraut, sollte ein unabhängiger Auditor den Smart Contract überprüfen – ähnlich einem Whitehat-Hacker.

Jacek Varky ist Blockchain Engineer bei der MaibornWolff GmbH. Schon seit seinem Masterstudium beschäftigt er sich und experimentiert mit Blockchain-Technologien. Während der Masterarbeit hat Jacek ein Off-Chain-Payment-Protokoll für Proof-of-Work-Protokolle entwickelt.

Links & Literatur

[1] Blockchain statt Datenbank: https://t3n.de/news/blockchain-statt-datenbank-diese-1063641/

[2] Etherscan: https://etherscan.io

[3] Blockchain Size: https://www.blockchain.com/de/charts/blocks-size

[4] Truffle Suite: https://truffleframework.com

[5] Embark: https://github.com/embark-framework/embark

[6] TDD: https://en.wikipedia.org/wiki/Test-driven_development

[7] Mythril: https://github.com/ConsenSys/mythril

[8] Decentralized Application Security Project: https://dasp.co/

[9] Docker Hub: https://hub.docker.com/

[10] Ganache CLI: https://github.com/trufflesuite/ganache-cli

 © Software & Support Media GmbH