Buduj Dappy za pomocą Angulara: część II

W poprzedniej części zacząłeś rozwijać swój dapp. W szczególności dowiedziałeś się o klasyfikacjach i projektach dapp i że możesz podzielić swój własny projekt dapp na pięć kroków. Następnie zastanawiałeś się, dlaczego warto korzystać z Angulara i jego zalet. Następnie utworzyłeś projekt Angular, najpierw upewniając się, że zainstalowano wymagania wstępne, a następnie instalując Angular CLI. Przyjrzałeś się elementom tworzącym Angular, takim jak komponenty, moduły i dyrektywy. Nauczyłeś się również, jak stylizować dapp, rozumiejąc architekturę w stylu Angular i pracując z materiałem Angular Material. Zacząłeś budować własne niestandardowe komponenty i tworzyć treści; podzieliłeś swoją aplikację na stopkę, nagłówek i treść oraz utworzyłeś niestandardowy składnik transferu, którego będziesz używać w tej części. W tej części omówię:

• Tworzenie inteligentnego kontraktu dapp z Truffle

• Integracja inteligentnego kontraktu z projektem Angular Twojego dapp

• Łączenie i łączenie aplikacji dapp z siecią Ethereum

Będziesz korzystać z narzędzi, które omówiłem do tej pory: Angular CLI, Truffle, ganache-cli i MetaMask. Stworzysz inteligentną umowę, której użyjesz do swojego dappa z Truffle, a następnie użyjesz biblioteki web3, aby połączyć się z lokalną siecią Ethereum i wywołać funkcje i zdarzenia inteligentnej umowy. MetaMask będzie używany do zarządzania Twoim kontem i łączenia się z nim.

Przenieś inteligentny kontrakt

Masz już logikę frontonu do przesyłania tokenów w swojej aplikacji z poprzedniej części; jednak nie masz inteligentnej umowy na interakcję z blockchainem. Inteligentne kontrakty można tworzyć przed częścią front-endową, po lub równolegle (jeśli pracujesz z zespołem programistów). Utworzyłeś już inteligentną umowę Ethereum w części 5, więc kroki opisane w tej sekcji powinny być Ci znajome. Zapraszam do ponownego odwiedzenia części 5, aby odświeżyć pamięć, ponieważ nie będę wdawać się w szczegóły dotyczące narzędzi i poleceń używanych w tej części. Na początek utworzysz nowy folder w swoim projekcie ethdapp do przechowywania projektu Truffle. W rzeczywistych projektach z wieloma programistami inteligentny kontrakt może być osobnym projektem. Dla uproszczenia uwzględnisz go w swoim projekcie, aby móc wykorzystać dolną kartę okna WebStorm Terminal do uruchamiania poleceń. Zacznij od utworzenia folderu o nazwie truffle w swoim projekcie i zainicjuj Truffle, aby utworzyć projekt.

> mkdir ethdapp/truffle

> cd truffle

> truffle init

Wskazówka: jeśli pojawią się błędy, takie jak „Error: Truffle Box”, odinstaluj Truffle, a następnie zainstaluj go ponownie i spróbuj ponownie. Aby ponownie zainstalować truffle w przypadku komunikatów o błędach, usuń ją globalnie i zainstaluj ją ponownie.

> npm uistall -g truffle

Jeśli nie masz zainstalowanego Truffle lub musisz ponownie zainstalować Truffle globalnie, uruchom polecenie install.

> npm install -g truffle

Po ponownej instalacji lub wykonaniu nowej instalacji uruchom ponownie polecenie truffle init i upewnij się, że uruchamiasz test w nowym oknie Terminala, aby upewnić się, że zmiany zostały zastosowane.

> truffle compile

> truffle migrate

> truffle test

Utwórz inteligentny kontrakt

Stworzysz inteligentną umowę i nazwiesz ją Transfer.sol; umieść to tutaj: trufla/kontrakty/Transfer.sol. Umowa pozwoli Ci na przeniesienie środków z jednego konta na drugie. Najpierw przejdź do lokalizacji kontraktów w Truffle i użyj edytora, aby utworzyć nowy plik.

> cd ethapp/truffle/contracts

> vim Transfer.sol

Pełny kod Transfer.sol znajduje się tutaj:

```
pragma solidity ^0.5.0;

contract Transfer {

address payable from;

address payable to;

constructor() public {

from = msg.sender;

}

event Pay(address _to, address _from, uint amt);

function pay( address payable _to ) public payable returns

(bool) {

to = _to;

to.transfer(msg.value);

emit Pay(to, from, msg.value);

return true;

}

}
```

Przejdźmy przez kod. Najpierw musisz zdefiniować wersję solidności, której będziesz używać, oraz nazwę kontraktu.

```
pragma solidity ^0.5.0;

contract Transfer {
```

Następnie zdefiniuj adresy od i do oraz konstruktor.

address payable from;

address payable to;

constructor() public {

from = msg.sender;

}

Będziesz używać zdarzenia Pay, które zostanie wywołane po użyciu funkcji płatności.

event Pay(adres _to, adres _from, uint amt);

Funkcja płatności wykorzystuje zdarzenie Pay do interakcji z siecią i wykonania rzeczywistego transferu.

function pay(address payable _to) public payable returns

(bool) {

to = _to;

to.transfer(msg.value);

emit Pay(to, from, msg.value);

return true;

}

}

Otóż to. Zachowałeś prostotę i prostotę z tylko jednym wydarzeniem i jedną funkcją.

Stwórz sieć rozwoju Truffle

Następnym krokiem jest zastąpienie pliku truffle/truffle-config.js następującą konfiguracją:

module.exports = {

networks: {

development: {

host: "127.0.0.1",

port: 8545,

network_id: "*",

gas: 5000000,

gasPrice: 100000000000

}

}

};

Zwróć uwagę, że wskazujesz na port 8545, który pomoże ci podczas uruchamiania MetaMask w dalszej części tej części.

Wdróż inteligentny kontrakt

Innym potrzebnym plikiem konfiguracyjnym jest plik kontraktu wdrażania. Utwórz plik wdrożenia i nazwij go truffle/migrations 2_deploy_contracts.js. W tym pliku konfiguracyjnym wystarczy wskazać utworzony przez siebie kod SOL inteligentnego kontraktu Transfer.

var Transfer = artifacts.require("./Transfer.sol");

module.exports = function(deployer) {

deployer.deploy(Transfer);

};

Teraz jesteś gotowy do stworzenia swojej sieci na porcie 8545 za pomocą Ganache, więc przejdź do projektu Truffle i uruchom to polecenie:

> cd ethdapp/truffrle

> ganache-cli -p 8545

Wskazówka: jeśli pojawią się błędy, takie jak „Niezgodność NODE_MODULE_VERSION”, odinstaluj i ponownie zainstaluj ganache-cli. Następnie otwórz nowe okno Terminala i upewnij się, że działa poprawnie. Aby w razie potrzeby ponownie zainstalować ganache-cli, uruchom to:

> npm uninstall -g ganache-cli

> npm install -g ganache-cli

Aby upewnić się, że działa poprawnie, uruchom to:

> ganache-cli help

Następnie w nowym oknie Terminala skompilujmy i wdrożmy Twój kontrakt, gdy ganache jest nadal uruchomiony.

> truffle compile

Dane wyjściowe kompilacji powinny zapewnić sukces, tworząc kontrakt w folderze Contract.

Compiling ./contracts/Transfer.sol…

Writing artifacts to ./build/contracts

Utworzony plik to Transfer.json, którego będziesz używać w swoim dappie do interakcji z siecią. Następnie wdrożysz swoją umowę za pomocą polecenia migracji.

> truffle migrate --network development

Wyniki powinny potwierdzać, że kontrakt został przeniesiony do sieci. Podsumowanie wyników powinno również pokazać, że wdrożenie przebiegło dobrze i opłata.

Summary

=======

> Total deployments: 2

> Final cost: 0.0525573 ETH

Konsola Truffle

Teraz, gdy masz już skompilowany i wdrożony kontrakt, aby wejść w interakcję z siecią, uruchom konsolę, jak pokazano tutaj:

> truffle console --network development

Konta

Jeśli uruchomisz getAccounts, otrzymasz listę kont powiązanych z Twoim portfelem.

truffle(development)> web3.eth.getAccounts()

['0x1eFf25A40C82EA65BC88E45d02368897EC922FEf',

'0xC135058b33d5df78636Cf14b74F281f95c4a407c',

'0xe682300Ef633F7d4f0d8Cb07c1bAD5d9B4eaE974'

....]

Następnie możesz zdefiniować adres1 i adres2 jako pierwsze i drugie konto.

truffle(development)> web3.eth.getAccounts().then(function(a)

{address1=a[0]})

undefined

truffle(development)> web3.eth.getAccounts().then(function(a)

{address2=a[1]})

undefined

Teraz, gdy są zdefiniowane, możesz je wywołać i uzyskać pierwszy i drugi rachunek w wyniku.

truffle(development)> address1

'0x1eFf25A40C82EA65BC88E45d02368897EC922FEf'

truffle(development)> address2

'0xC135058b33d5df78636Cf14b74F281f95c4a407c'

Możesz również użyć getBalance, aby uzyskać saldo, które masz pod tymi adresami.

truffle(development)> web3.eth.getBalance(address1)

'99942134400000000000'

truffle(development)> web3.eth.getBalance(address2)

'100000000000000000000'

Przetestuj transfer inteligentnego kontraktu

Teraz, gdy już zdefiniowałeś dwa adresy i znasz saldo na tych rachunkach, możesz zdefiniować swoją umowę i przekazywać część środków pomiędzy rachunkami. Aby to zrobić, najpierw zdefiniuj umowę i nazwij ją transferSmartContract.

truffle(development)> Transfer.deployed().

then(function(instance){transferSmartContract = instance;})

undefined

Następnie uruchom zmienną transferSmartContract, którą zdefiniowałeś, aby upewnić się, że pracował i pokazać wartość obiektu.

> transferSmartContract

Teraz możesz przesyłać środki za pomocą inteligentnej umowy między dwoma kontami. Konto 2 zawiera ładną okrągłą liczbę, więc przeniesiesz 5 eth.

> transferSmartContract.pay(address2, {from: address1, value: 5});

Dane wyjściowe polecenia zawierają informacje o transakcji i wydobywaniu. Teraz możesz zobaczyć zaktualizowane saldo.

> web3.eth.getBalance(adres1);

'999421343999999999995'

> web3.eth.getBalance(adres2);

'100000000000000000005'

Jak widzisz saldo się zmieniło i udało Ci się dokonać przelewu token między dwoma adresami.

Połącz z siecią Ethereum

Twoja umowa pracuje w Terminalu; następnym krokiem jest interakcja Twojego dappa z umową. Odbywa się to za pośrednictwem web3.js, który jest zbiorem bibliotek umożliwiających interakcję z lokalnym lub zdalnym węzłem Ethereum przy użyciu połączenia HTTP lub IPC. Najpierw przejdź z powrotem do folderu projektu Angular, a następnie zainstaluj web3.js z flagą --save, aby zapisać instalowaną bibliotekę.

> cd ethdapp/

> npm install web3 –save

+ web3@1.0.0-beta.55

Jeśli instalacja przebiegła pomyślnie, w danych wyjściowych zobaczysz, że wersja została zainstalowana. W chwili pisania tego tekstu web3 jest w wersji 1.0.0-beta55. Musisz również zainstalować umowę truflową, która zapewnia kod opakowania, który ułatwia interakcję z umową.

> npm install truffle-contract –save

+ truffle-contract@4.0.15

Wskazówka : web3 wersja 1.0.0 beta i wersja kontraktu na trufle 4.0.15 to najnowsze wersje i są kompatybilne z Angular 7.3.x. Jednak może się to zmienić, więc obserwuj wersję, którą instalujesz, aby upewnić się, że jest kompatybilna i aby uniknąć błędów. Zainstaluj ponownie z dokładną [wersja], na przykład @4.0.15, jeśli napotkasz problemy ze zgodnością.

Usługa transferu

Teraz, gdy masz już zainstalowane biblioteki, możesz kontynuować. W tej sekcji utworzysz i napiszesz klasę usług. Klasa usług będzie twoją środkową warstwą frontonu do interakcji z web3. Aby rozpocząć, możesz użyć flagi ng s, co oznacza „usługę”.

> ng g s services/transfer --module=app.module

CREATE src/app/services/transfer.service.spec.ts

CREATE src/app/services/ transfer.service.ts

Zastąpisz początkowy kod klasy usługi logiką do interakcji z web3. Najpierw zdefiniujesz biblioteki, których będziesz używać, czyli rdzeń Angulara oraz zainstalowane biblioteki truffle-contract i web3.

import { Injectable } from '@angular/core';

const Web3 = require('web3');

import * as TruffleContract from 'truffle-contract';

Następnie zdefiniujesz trzy zmienne, których będziesz później używać: require, window i tokenAbi. Zauważ, że tokenAbi wskazuje na plik ABI skompilowany z pliku SOL kontraktu.

declare let require: any;

declare let window: any;

const tokenAbi = require('../../../truffle/build/contracts/

Transfer.json');

Aby móc współpracować z web3, potrzebujesz dostępu do roota, więc musisz go wstrzyknąć do swojego projektu.

@Injectable({

providedIn: 'root'

})

Następnie zdefiniuj definicję klasy, zmienne account i web3 oraz init web3.

export class TransferService {

private _account: any = null;

private readonly _web3: any;

constructor() {

if (typeof window.web3 !== 'undefined') {

this._web3 = window.web3.currentProvider;

```
} else {
this._web3 = new Web3.providers.HttpProvider('http://
localhost:8545');
}
window.web3 = new Web3(this._web3);
console.log('transfer.service :: this._web3');
console.log(this._web3);
}
```

Zwróć uwagę, że otoczyłeś komunikaty console.log wokół kodu, dzięki czemu możesz zobaczyć komunikaty w sekcji komunikatów konsoli przeglądarki w trybie narzędzia programisty, aby pomóc Ci zrozumieć, co się dzieje. Aby to zrobić, otwórz przeglądarkę w trybie narzędzi programistycznych. W przeglądarce Chrome wybierz Widok Widok programisty -> Programista -> Narzędzia programistyczne.

Potrzebujesz metody asynchronicznej, aby uzyskać adres i saldo konta, więc możesz użyć funkcji obietnicy. Jeśli Twoje konto nie zostało wcześniej pobrane, wywołasz web3.eth.getAccounts, tak jak w Terminalu, aby pobrać dane. Potrzebujesz również kodu błędu, jeśli coś pójdzie nie tak.

```
private async getAccount(): Promise<any> {
console.log('transfer.service :: getAccount :: start');
if (this._account == null) {
this._account = await new Promise((resolve, reject) => {
console.log('transfer.service :: getAccount :: eth');
console.log(window.web3.eth);
window.web3.eth.getAccounts((err, retAccount) => {
console.log('transfer.service :: getAccount: retAccount');
console.log(retAccount);
if (retAccount.length > 0) {
this._account = retAccount[0];
resolve(this._account);
} else {
alert('transfer.service :: getAccount :: no
accounts found.');
reject('No accounts found.');
}
if (err != null) {
```

```
alert('transfer.service :: getAccount :: error

retrieving account');

reject('Error retrieving account');

}

});

}) as Promise< any >;

}

return Promise.resolve(this._account);

}
```

Podobnie potrzebujesz metody obsługi do interakcji i uzyskania salda konta. Używasz web3.eth.getBalance tak samo, jak w Terminalu i owijasz sprawdzanie błędów. Ty też składasz to jako obietnicę. Powodem, dla którego potrzebujesz obietnicy, jest to, że te wywołania są asynchroniczne, a JavaScript nie.

```
public async getUserBalance(): Promise< any > {

const account = await this.getAccount();

console.log('transfer.service :: getUserBalance :: account');

console.log(account);

return new Promise((resolve, reject) => {

window.web3.eth.getBalance(account, function(err, balance) {

console.log('transfer.service :: getUserBalance ::

getBalance');

console.log(balance);

if (!err) {

const retVal = {account: account, balance: balance};

console.log('transfer.service :: getUserBalance ::

getBalance :: retVal');

console.log(retVal);

resolve(retVal);

} else {

reject({account: 'error', balance: 0});

}

});
```

```
}) as Promise< any >;

}
```

Na koniec potrzebujesz metody, aby przekazać wartości z formularza i przelać płatność z jednego konta na drugie. Użyj metody płatności wynikającej z umowy i zakończ sprawdzanie błędów.

```
transferEther(value) {

const that = this;

console.log('transfer.service :: transferEther to: ' +

value.transferAddress + ', from: ' + that._account + ',

amount: ' + value.amount);

return new Promise((resolve, reject) => {

console.log('transfer.service :: transferEther :: tokenAbi');

console.log(tokenAbi);

const transferContract = TruffleContract(tokenAbi);

transferContract.setProvider(that._web3);

console.log('transfer.service :: transferEther ::

transferContract');

console.log(transferContract);

transferContract.deployed().then(function(instance) {

return instance.pay(

value.transferAddress,

{

from: that._account,

value: value.amount

});

}).then(function(status) {

if (status) {

return resolve({status: true});

}

}).catch(function(error) {

console.log(error);

return reject('transfer.service error');
```

});

});

}

}

Teraz, gdy masz ukończoną usługę transferu, możesz się połączyć transfer.component, aby uzyskać adres i saldo konta użytkownika oraz móc przelać środki po wypełnieniu formularza. Najpierw musisz zdefiniować komponent usługi, który utworzyłeś. Otwórz src/app/component/transfer/transfer.component.ts i dodaj import oświadczenie u góry dokumentu.

import {TransferService} from '../../services/transfer.service;

Dla definicji komponentu dodaj TransferService jako dostawcę.

@Component({

..

providers: [TransferService]

})

Dodaj także TransferService do konstruktora, aby móc go używać w swojej klasie.

constructor(private fb: FormBuilder,

private transferService: TransferService) { }

Następnie zaktualizuj metodę getAccountAndBalance, aby zawierała wywołanie klasę usługi i pobrać rzeczywiste konto użytkownika i saldo.

getAccountAndBalance = () => {

const that = this;

this.transferService.getUserBalance().

then(function(retAccount: any) {

that.user.address = retAccount.account;

that.user.balance = retAccount.balance;

console.log('transfer.components :: getAccountAndBalance

:: that.user');

console.log(that.user);

}).catch(function(error) {

console.log(error);

});

}

Na koniec zaktualizuj submitForm, aby wywołać transferEther, aby przelać i zapłacić. Zastąp wyświetlone tutaj komentarze submitForm TODO wywołaniami połączeń serwisowych:

// TODO: service call

Następnie przekaż dane przesłane przez użytkownika:

this.transferService.transferEther(this.userForm.value).

then(function() {

}).catch(function(error) {

console.log(error);

});

});

Połącz z MetaMaską

W tym momencie twój kod dapp jest gotowy. Jeśli jednak przetestujesz teraz swój dapp, web3 nie będzie mógł połączyć się z kontem. Musisz połączyć się z MetaMask. Istnieje problem prywatności związany z aplikacjami, w których złośliwe witryny internetowe są w stanie wstrzyknąć kod, aby wyświetlić działania użytkowników i adresy Ethereum, a następnie znaleźć saldo, historię transakcji i dane osobowe. Te złośliwe witryny mogą następnie inicjować niechciane transakcje w imieniu użytkownika, a użytkownik może przypadkowo zatwierdzić nieautoryzowaną transakcję i stracić środki. Aby uniknąć tych problemów i połączyć się z usługą Angular, połączysz przeglądarkę z siecią przez MetaMask. Korzystałeś już z MetaMask, więc powinieneś ją zainstalować. Cofnijmy się na chwilę. Jak pamiętasz, założyłeś sieć przez ganache-cli na porcie 8545.

> ganache-cli -p 8545

I podłączyłeś Truffle do sieci.

> truffle migrate --network development

Następnie mogłeś połączyć się na porcie 8545 i uruchamiać polecenia w Terminalu. Możesz teraz połączyć MetaMask w przeglądarce. Aby się połączyć, wybierz MetaMask i wybierz Localhost 8545 z menu rozwijanego. Zauważ, że wybrałeś port 8545 wcześniej w tym rozdziale. Jest to domyślny port w MetaMask, więc można łatwo połączyć się z siecią prywatną, wybierając pozycję menu rozwijanego zamiast wskazywać niestandardowy port. Jednak po sprawdzeniu listy kont nie widać żadnych kont. Powodem, dla którego nie widzisz kont, jest to, że za każdym razem, gdy uruchamiasz sieć, musisz zaktualizować konta. Istnieją dwa sposoby na zaktualizowanie MetaMask listą kont.

Opcja 1: Po uruchomieniu Ganache użyj flagi m, aby przekazać mnemonik reprezentujący klucze prywatne, które miałeś w Ganache. Na przykład polecenie będzie wyglądać tak:

> ganache-cli -p 8545 -m 'journey badge medal slender behind junk develop produce spy enemy transfer room'

Opcja 2: Po uruchomieniu ganache-cli zobaczysz listę kont,

klucze prywatne i mnemoniki.

> ganache-cli -p 8545

Poszukaj tego wyjścia i skopiuj mnemonik.

HD Wallet

==========

Mnemonic: journey badge medal slender behind junk develop

produce spy enemy transfer room

Base HD Path: m/44'/60'/0'/0/{account_index}HD Wallet

==========

Mnemonic: journey badge medal slender behind junk develop

produce spy enemy transfer room

Base HD Path: m/44'/60'/0'/0/{account_index}

Następnie wyloguj się z MetaMask i wklej mnemonik ręcznie. Kliknij prawym przyciskiem i wybierz „Wyloguj.

Po wylogowaniu pojawi się ekran powitalny z linkiem pod nim, który mówi „Importuj za pomocą frazy początkowej konta". Kliknij ten link. Teraz możesz wkleić mnemonik, wybierając hasło i klikając Przywróć.

Przetestuj swoją funkcjonalność Dapp

Teraz jesteś w końcu gotowy do przetestowania swojego dappa. Po odświeżeniu przeglądarki zobaczysz adres i saldo. Następnie wypełnij formularz i zainicjuj przelew. Zauważ, że MetaMask otwiera się, aby potwierdzić transfer. Jest to dodatkowa miara bezpieczeństwa zapewniająca, że tylko autoryzowane przelewy zostaną zatwierdzone.

Dokąd się udać?

Kontynuuj pracę i ulepszanie utworzonego dappa. Na przykład możesz wykonać następujące czynności:

- Utwórz klasę usług użytkownika i współdzieloną klasę usług, aby przechowywać informacje o użytkownikach i współdzielone informacje

- Utwórz usługę logowania/wylogowania

- Utwórz opcję przełączania się między kontami

- Utwórz menu boczne, aby lepiej poruszać się po aplikacji

- Zaktualizuj inteligentną umowę i dodaj więcej metod i wydarzeń

Streszczenie

W tej części utworzyłeś inteligentny kontrakt transferu i projekt rozwoju Truffle, a także połączyłeś się z lokalną siecią programistyczną Ganache. Nauczyłeś się, jak pracować z siecią Ethereum za

pośrednictwem Truffle i jak przetestować swój inteligentny kontrakt. Testujesz transfer środków za pomocą inteligentnej umowy za pomocą wiersza poleceń. Na koniec połączyłeś swój dapp z siecią Ethereum za pomocą utworzonego przez siebie komponentu Angular TransferService. Korzystając z biblioteki web3 wykonałeś kilka połączeń serwisowych. Wreszcie połączyłeś się z MetaMask, aby zarządzać swoimi kontami. W następnym rozdziale dowiesz się o bezpieczeństwie i zgodności blockchain.