# Verantwoording ‘JorisCoin Manager’

30-03-2020 - NOVI Hogeschool - Praktijk 2 - Joris van Laar (800009581)

## De applicatie

De JorisCoin Manager is een webapplicatie waarmee een gebruiker zijn fictieve cryptovaluta (JorisCoins) en de bijbehorende blockchain kan beheren. De backend is opgezet in Python (Flask) en de frontend bestaat uit een webpagina die is opgebouwd uit HTML en JavaScript (Vue.js).

Er is getracht zo dicht mogelijk bij de realiteit te blijven t.a.v. de technische opzet van de blockchain. Al ontbreekt er (vooralsnog) wel een netwerk aan ‘nodes’, oftewel computers die gezamenlijk de correctheid van de blockchain valideren. Deze applicatie draait alleen lokaal op de computer van de gebruiker, waardoor de gebruiker alleen JorisCoins kan versturen naar andere fictieve gebruikers, maar niets kan ontvangen van andere gebruikers.

## Specificaties

- Gebruikte programmeertalen:
  - Python 3.8.2
  - HTML + JavaScript
- IDE: Visual Studio Code 1.43.1
- Versiebeheer: GitHub + Sourcetree -> https://github.com/jorisvanlaar/blockchain-manager
- OS: Windows 10
- Getest in: Google Chrome 80.0.3987.149

## Setup

Om de applicatie foutloos te laten compileren dienen de volgende stappen te worden ondernomen:

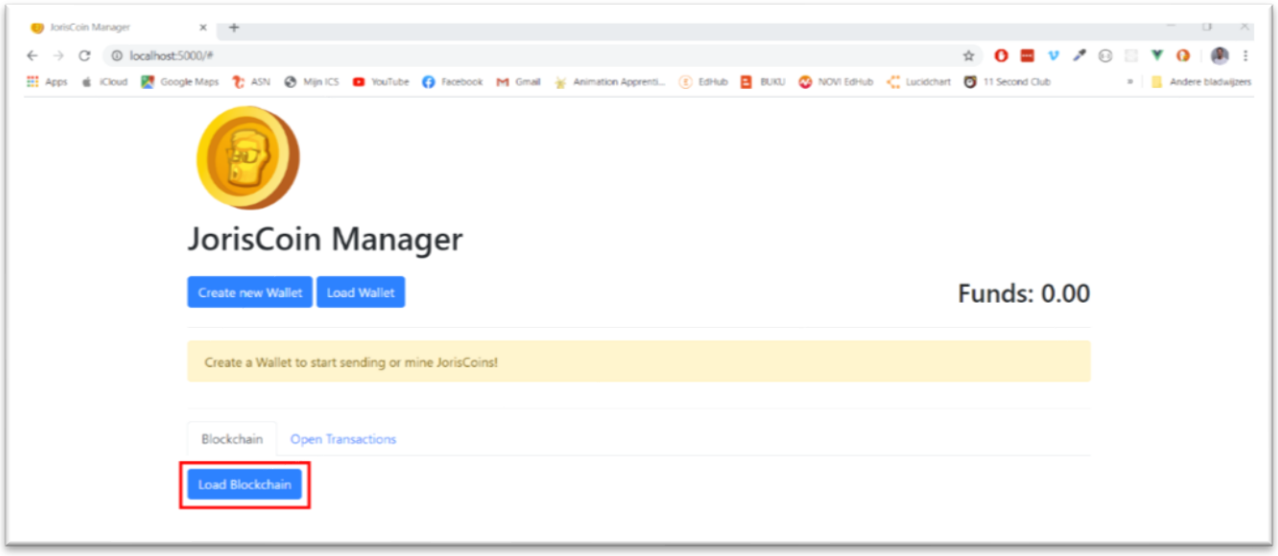
1. Controleer bij het uitpakken van de .zip ervoor dat er maar één folderniveau is uitgepakt, met de naam *blockchain-manager*. Sommige unzipping software (zoals Winrar) heeft de neiging om soms een extra folder aan te maken, waardoor er twee folderniveaus van dezelfde naam ontstaan. In dit geval zou dat resulteren in een *blockchain-manager* folder IN een *blockchain-manager* folder. Dit zorgt voor issues in de webapplicatie bij het inladen van data!
2. Installeren benodigde third-party Python packages m.b.v. twee terminal commands:
   a. *python -m pip install pycryptodome*
   b. *python -m pip install Flask*

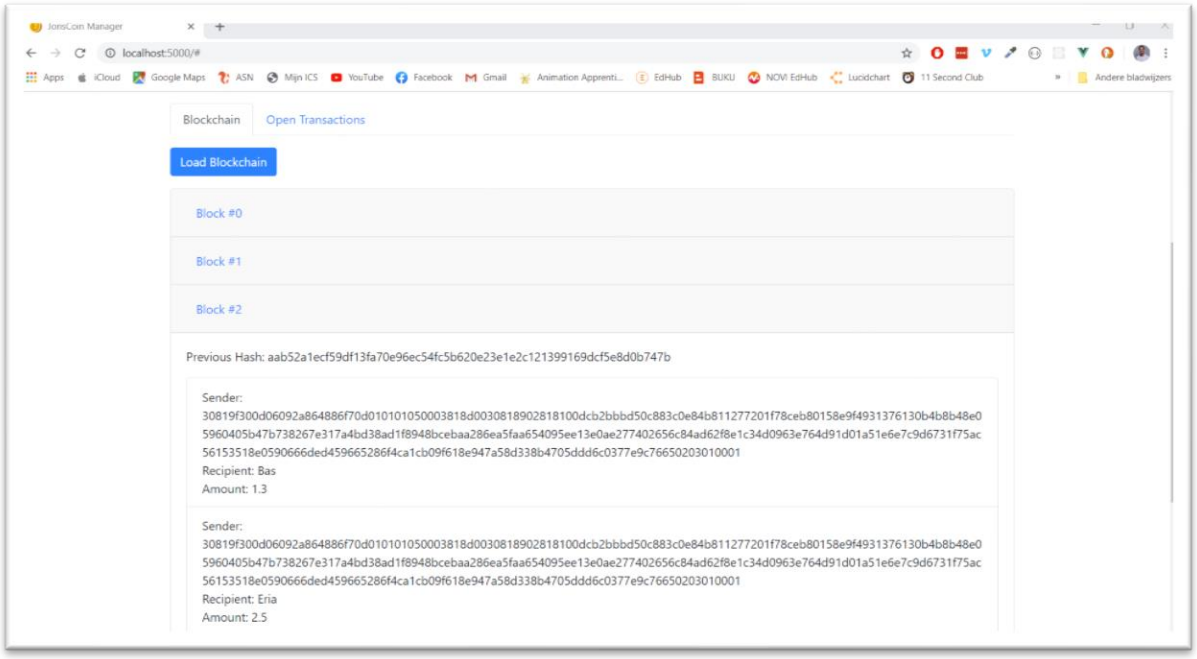
PyCryptodome wordt gebruikt om de public en private keys voor de wallet te genereren.
Flask biedt een vrijwel kant-en-klare Python server. Hierdoor wordt het makkelijk gemaakt om HTTP endpoints (routes) open te stellen op de lokale server, waar de client requests naar kan versturen.

3. Server (*node.py*) opstarten met de terminal command: *python node.py*
4. Client opstarten in browser: http://localhost:5000/

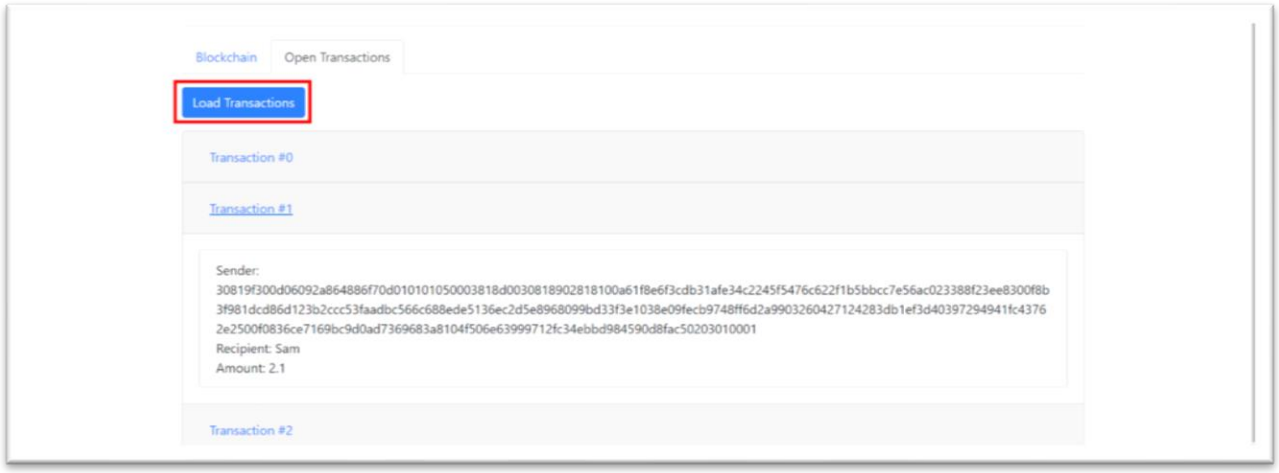
## Gebruiksaanwijzing

1. Laad de blockchain in door op de *Load Blockchain* knop te klikken. Er verschijnt nu een lijst aan blocks, waarbij elk block de volgende data bevat:
    a. Een *previous hash*, die wordt gebruikt om het voorgaande block in de chain te valideren.
    b. Eén of meerdere *transacties*, bestaande uit een sender, recipient en amount.
        i. Het eerste block in de chain is leeg, dat komt omdat dit het 'genesis block' is.
        ii. Elk block bevat altijd één transactie met als sender 'MINING' en een amount van 10. Een miner ontvangt namelijk vanuit de blockchain 10 JorisCoins als beloning voor het minen van een nieuw block.
        iii. Het id voor de sender is de public key van diens wallet.

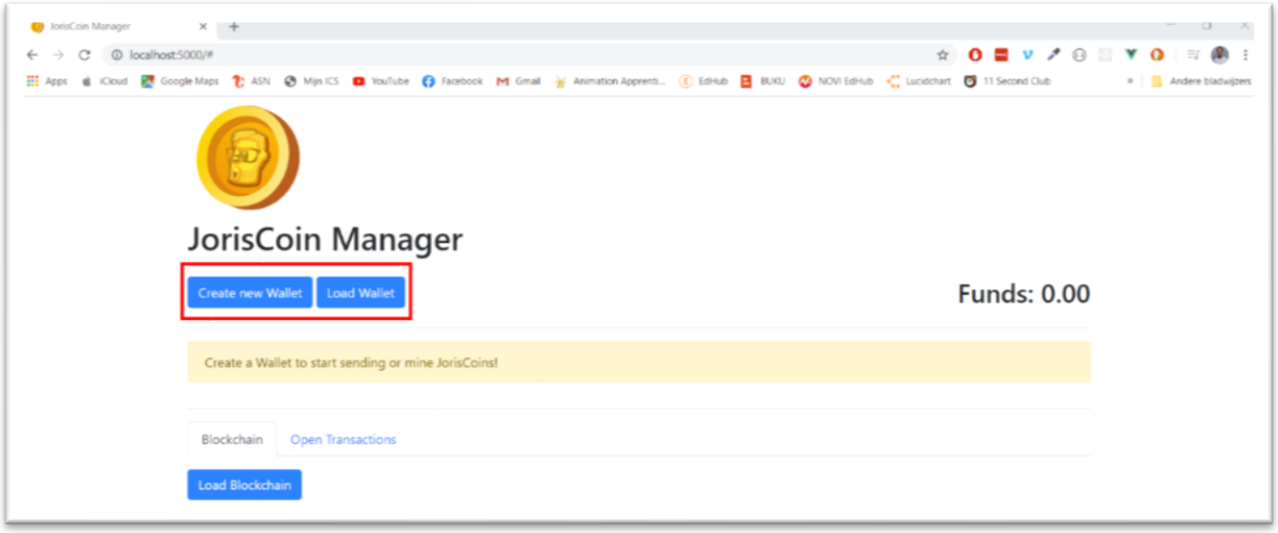


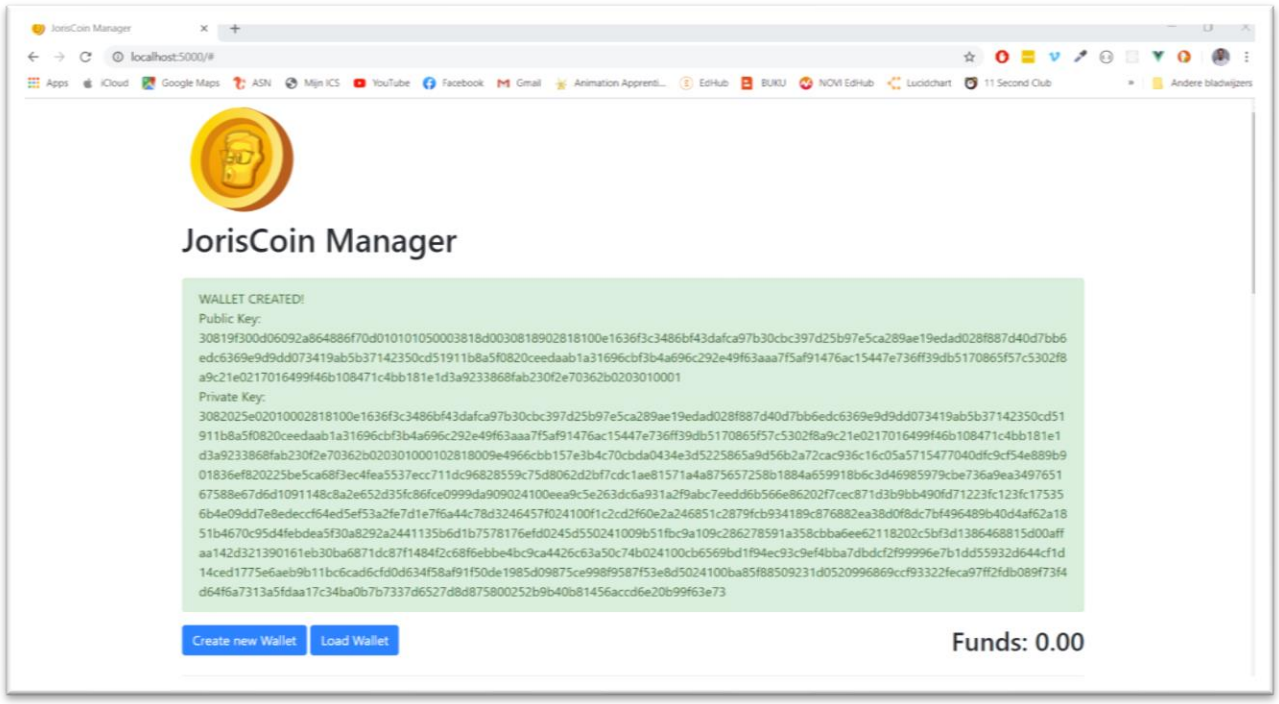
2. Bekijk de openstaande transacties door het *Open Transactions* tab te openen, en te klikken op de *Load Transactions* knop.
   Er verschijnt een lijst aan transacties die wel al verzonden zijn, maar nog niet zijn gemined, en dus nog niet zijn toegevoegd aan de blockchain.



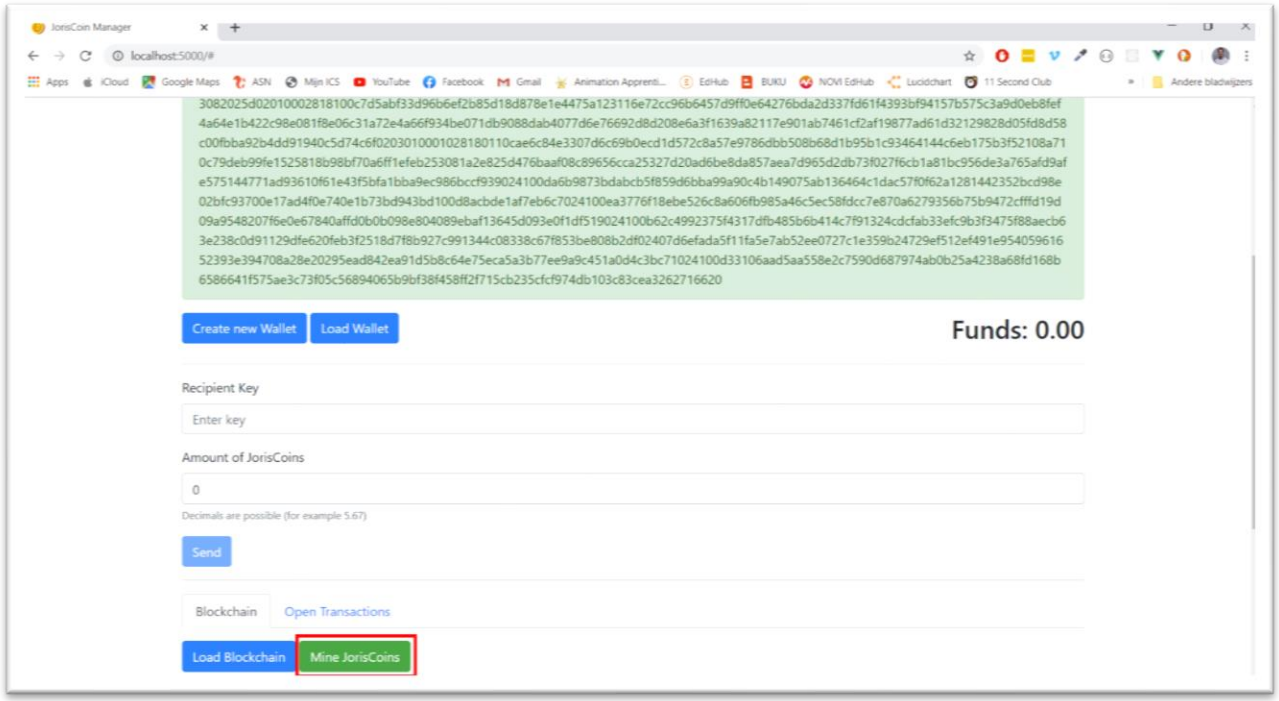

3. Laten we daarom gaan minen! Hiervoor is het wel nodig dat je een wallet aanmaakt, of een bestaande wallet inlaadt. Beiden opties zijn mogelijk, aangezien er voor het gemak al een wallet is opgeslagen in de *wallet.txt* file.



a. Je ziet nu een bevestiging dat je wallet is aangemaakt/ingeladen. De public en private key van je wallet worden getoond, en eventuele bijbehorende funds zijn ook ingeladen.

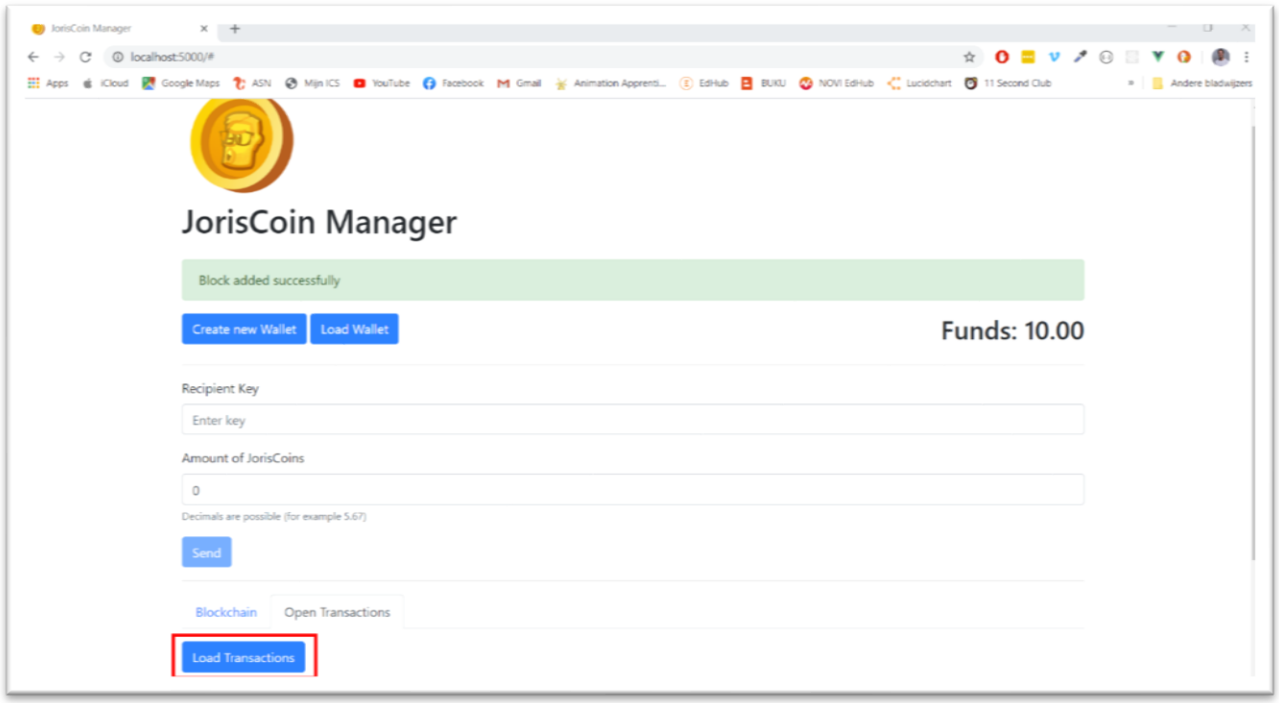



b. Open de *Blockchain* tab en klik op de *Mine JorisCoins* knop om te gaan minen.



4. Tijdens het minen worden alle openstaande transacties gebundeld in een block, en dit block wordt vervolgens aan de blockchain toegevoegd. Ook zijn je funds nu met 10 JorisCoins vermeerderd als beloning voor het minen.

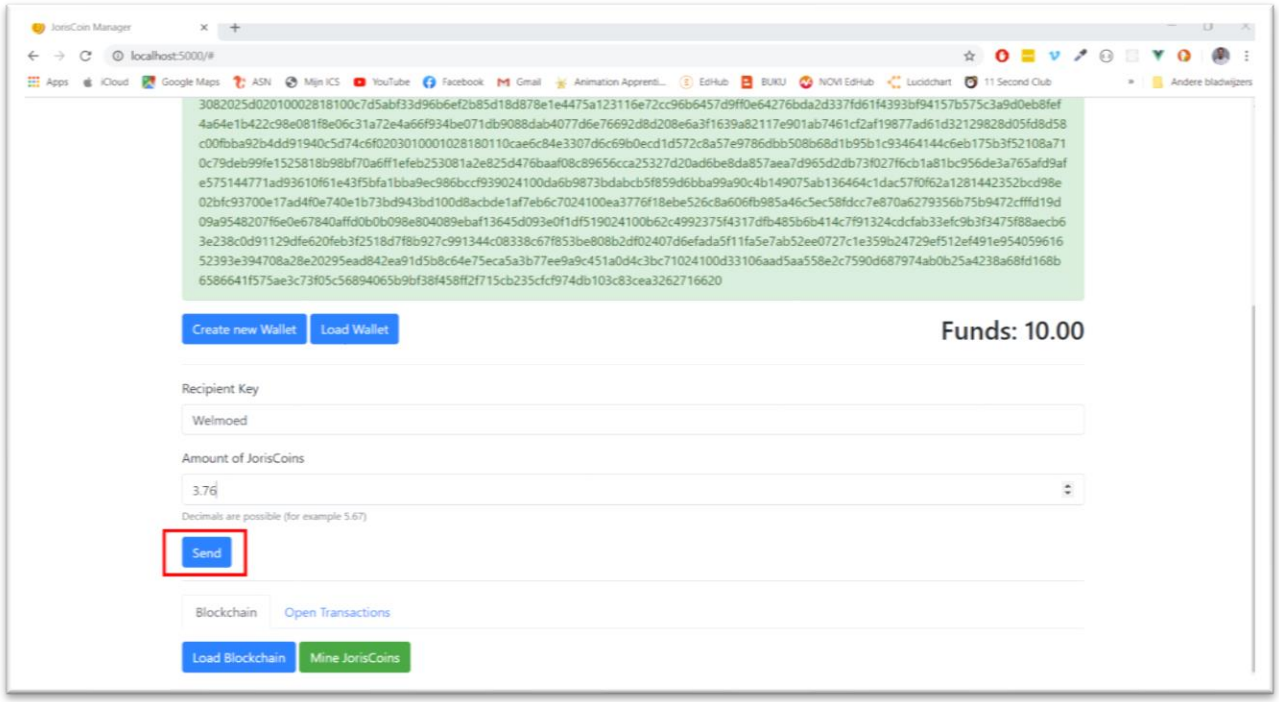
   In eerste instantie lijkt het alsof de lijst met openstaande transacties nog steeds is gevuld. Maar herlaadt de openstaande transacties door nogmaals op de *Load Transactions* knop te klikken, en je ziet dat de lijst nu leeg is.



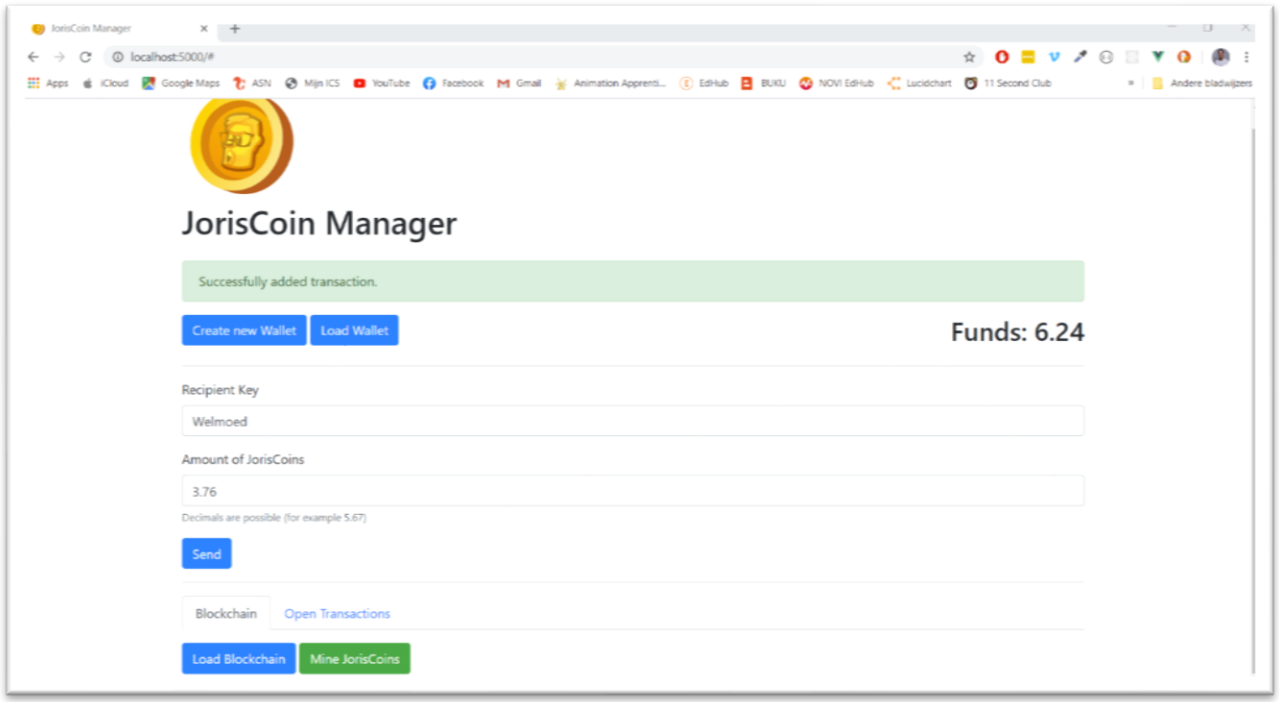

   Herlaadt ook de blockchain door in de *Blockchain* tab op de *Load Blockchain* knop te klikken. Je ziet nu dat er een nieuw block is toegevoegd aan het einde van de chain. Dit block bevat alle transacties die je zojuist hebt gemined, plus de 'vergoedings-transactie' als beloning voor het minen.

5. Tot slot kun je JorisCoins naar een andere (fictieve) gebruiker versturen. Vul de public key (een naam is ook goed) van de ontvanger in en de hoeveelheid JorisCoins die je naar deze gebruiker wilt versturen. Klik op de *Send* knop.

   Zowel het aanmaken van een transactie als het minen, zijn functionaliteiten die alleen beschikbaar zijn als er een wallet is aangemaakt/ingeladen.




6. Je ziet een bevestiging dat je transactie is toegevoegd aan de lijst met openstaande transacties, en je funds zijn verminderd met het verzonden bedrag.



# Requirements

## 1. Het MVC-patroon wordt toegepast.

**Model**
De *blockchain.txt* en *wallet.txt* bestanden vormen de daadwerkelijke data van de webapp. De blockchain-data is opgebouwd uit een list aan blocks en een list aan openstaande transacties. De wallet-data bestaat uit een public en private key.
De *node.py* file manipuleert deze data en stuurt een server-respons met de gemodificeerde data terug naar de controller.

**View**:
De *node.html* file vormt de presentatie-laag van de applicatie en is verantwoordelijk voor het tonen van een UI op een webpagina. Interactie van de gebruiker wordt door de view afgehandeld door events te versturen naar de controller.

**Controller**:
Ik koppel een Vue.js instance aan de webpagina (*node.html*). Deze Vue instance doet dienst als controller en luistert naar events die worden afgevuurd vanuit de view. Bij het ontvangen van een event stuurt de controller vervolgens een HTTP request naar het model. Wanneer er vanuit het model een respons wordt teruggestuurd, zorgt de controller ervoor dat de view worden geüpdatet met de ontvangen data.

Voor de setup van de controller heb ik gekeken naar het MVC-voorbeeld uit de officiële documentatie van het Vue.js framework:

- https://vuejs.org/v2/examples/todomvc.html
- https://codesandbox.io/s/github/vuejs/vuejs.org/tree/master/src/v2/examples/vue-20-todomvc?from-embed

*CODE:*

- *node.py*
- *node.html*

## 2. De blockchain kan worden ingeladen van de harde schijf. De individuele blocks waaruit de blockchain bestaat kunnen worden bekeken, en zullen de hash van het vorige block en de transacties waar het block uit bestaat (verzender, ontvanger en de hoeveelheid JorisCoins) tonen.

Bij opstart van de Flask server wordt direct een instance van de blockchain aangemaakt, waarbij de chain eerst alleen bestaat uit een genesis block. Maar de constructor van de Blockchain class voert ook meteen een *load_data()* method uit. Dit zorgt ervoor dat de data van de *blockchain.txt* file wordt ingelezen in de blockchain instance.

De view luistert naar een *onLoadData* click event, die wordt afgevuurd wanneer de gebruiker op de *Load Blockchain* button klikt. De controller stuurt vervolgens, m.b.v. de Axios library, een GET request naar de *'/chain'* route op de server. De server haalt de chain data op, maar deze chain is opgebouwd uit Block objecten. En een Block object is op zijn beurt weer opgebouwd uit Transaction objecten. Aangezien objecten nooit zijn te converten naar JSON data, worden eerst alle Block en Transaction objecten in de chain geconvert naar dictionaries. En vervolgens wordt de gehele chain dan geconvert naar JSON data. Deze data wordt door de server als respons teruggestuurd.

Zodra de respons van de server is ontvangen wordt de promise van de controller uitgevoerd, wat zoveel betekent als dat de data van de blockchain dynamisch wordt ingeladen op de webpagina.

*CODE:*

- *node.py – line 13, 180 t/m 205*
- *blockchain.py – line 14 t/m 19, 50 t/m 80*
- *node.html – line 154, 345 t/m 366*

## 3. De user kan een nieuwe wallet aanmaken of inladen van de harde schijf, zodat hij in staat is om transacties te versturen of blocks te minen. In de wallet wordt een balans aan JorisCoins bijgehouden.

Zoals gezegd wordt bij opstart van de server direct een instance van de blockchain aangemaakt, maar óók een wallet instance. De public en private keys van dit wallet object zijn in eerste instantie op *None* gezet, omdat de gebruiker dan nog moet aangeven of hij een wallet met nieuwe keys wil aanmaken, of een opgeslagen wallet met bestaande keys wil inladen.
De public key van het wallet object is onderdeel van de constructor voor de blockchain class, omdat de public key dienst doet als id voor de computer (*node)* waarop de instance van de blockchain draait. Hoewel die key in eerste instantie *None* is, kan de gebruiker op deze manier wel alvast de inhoud van de blockchain in de app opvragen.

Voor het aanmaken van een wallet luistert de view naar een *onCreateWallet* click event, die wordt afgevuurd wanneer de gebruiker op de *Create New Wallet* button klikt. De controller stuurt vervolgens een POST request naar de *'/wallet'* route op de server. De server genereert m.b.v. de PyCryptodome package een public en private key voor de wallet, en slaat die op in de *wallet.txt* file. De blockchain wordt geherinitialiseerd, om diens id (die gebaseerd is op de public key van de wallet) te wijzigen van *None* in de nu gegenereerde public key. Als respons stuurt de server de public key, private key en de funds (bij nieuwe wallet altijd 0) als JSON data terug. De controller ontvangt deze data, en zorgt ervoor dat in de view een bevestiging wordt getoond, de funds worden geüpdatet, en ontsluit (o.b.v. conditional rendering) de mine en transactie functionaliteit van de app.

Voor het inladen van een bestaande wallet luister de view naar een *onLoadWallet* clickevent, die wordt afgevuurd wanneer de gebruiker op de *Load Wallet* button klikt. De controller stuurt vervolgens een GET request naar de *'/wallet'* route op de server. De server leest de public en private key in vanaf de *wallet.txt* file, en herinitialiseert de blockchain met de ingeladen public key als diens nieuwe id. Als respons stuurt de server de public key, private key en de funds als JSON data terug.
De controller ontvangt deze data, en zorgt ervoor dat in de view een bevestiging wordt getoond, de

funds worden geüpdatet, en ontsluit (o.b.v. conditional rendering) de mine en transactie functionaliteit van de app.

*CODE:*

- *node.py – line 12-13, 42 t/m 78*
- *node.html – line 81 t/m 138, 250 t/m 302*
- *wallet.py – line 9 t/m 53*
- *blockchain.py - line 111 t/m 145*

**4. De user kan transacties versturen door de naam van de ontvanger en het bedrag in te voeren. De hoeveelheid verstuurde JorisCoins wordt afgetrokken van de balans van de wallet. De transactie wordt gesigned met een private key die bij de wallet van de user hoort. Een user kan nooit meer JorisCoins versturen dan hij in zijn wallet heeft.**

De view luistert naar een *onSendTx* form submit event, die wordt afgevuurd wanneer de gebruiker op de *Send* button heeft geklikt. Deze button wordt pas actief wanneer de gebruiker een key/naam voor de ontvanger heeft ingevuld en een bedrag aan JorisCoins heeft ingevuld dat hoger is dan 0.
De controller stuurt vervolgens een POST request naar de *'/transaction'* route op de server. Met dit request stuurt de controller ook de vereiste recipient en amount data mee naar de server. De server onttrekt vervolgens deze transactie-data uit het binnenkomende request.

Een wallet bestaat simpelweg uit een Public Key/Private Key paar. De public key doet dienst als id voor de gebruiker om JorisCoins te versturen/ontvangen. De private key van de gebruiker signeert de transactie met een signature (een hash) die is gegenereerd o.b.v. de transactie-data (sender, recipient, amount). Dit signeren van transacties voorkomt manipulatie van transacties, omdat de public key kan verifiëren dat een signature van een transactie gesigneerd is door de private key. Op het moment dat iemand een transactie aanpast, zal de signature niet meer kloppen wanneer deze geverifieerd wordt door de public key. En de signature zelf is ook niet te manipuleren door iemand met kwade bedoelingen, omdat dit alleen kan als hij in het bezit is van de bijbehorende private key.

Als de gebruiker over voldoende funds beschikt en de signature is geverifieerd, zal vervolgens de transactie aan de list van open transactions worden toegevoegd en wordt de aangepaste blockchain data (incl. open transactions) opgeslagen in de *blockchain.txt* file. De server stuurt dan als respons de details van de succesvol toegevoegde transactie terug in de vorm van JSON data. De controller op zijn beurt zorgt ervoor dat de view een bevestiging toont en de funds worden geüpdatet (vermindering met het verstuurde bedrag aan JorisCoins).

*CODE:*

- *node.html – line 119 t/m 136, 303 t/m 328*
- *node.py – line 100 t/m 144*
- *wallet.py – line 60 t/m 83*
- *blockchain.py – line 155 t/m 184*
- *verification.py – line 49 t/m 57*

5. **Een transactie die nog niet onderdeel uitmaakt van de blockchain, wordt gezien als een 'open transactie'. Een lijst aan open transacties is in te zien in de applicatie. Zodra de user JorisCoins mined, zullen de open transacties worden gebundeld in een block, dit block wordt aan de blockchain toegevoegd, en de lijst aan open transacties wordt leeggemaakt. De user ontvangt als miner een vastgestelde vergoeding aan JorisCoins waarmee de balans van zijn wallet wordt vermeerderd. Wanneer de user opnieuw de blockchain inlaadt, zal het nieuw toegevoegde block worden getoond.**

Wanneer de server wordt opgestart worden in eerste instantie de open transacties van de blockchain geïnitialiseerd met een lege list. Maar vervolgens worden deze open transacties direct geüpdatet met data die wordt ingelezen van de *blockchain.txt* file. De view luistert naar een *onLoadData* click event, waarbij er rekening wordt gehouden of de *Blockchain* tab of de *Open Transactions* tab actief is. Het event wordt afgevuurd wanneer de gebruiker op de *Load Transactions* button klikt.
De controller stuurt vervolgens een GET request naar de *'/transactions'* route op de server. De server op zijn beurt vraagt de open transactions op. Dit is in eerste instantie een list, maar dat is niet te converten naar JSON data. Daarom wordt de list eerst geconvert naar een dictionary, en die wordt weer geconvert naar JSON data. Deze JSON data wordt als respons door de server teruggestuurd. De controller zorgt ervoor dat de view wordt geüpdatet met de teruggestuurde open transactions data.

Voor het minen (toevoegen van een nieuw block aan de blockchain) wordt door de view geluisterd naar een *onMine* click event, die wordt afgevuurd wanneer de gebruiker op de *Mine JorisCoins* button klikt. De controller stuurt vervolgens een POST request naar de *'/mine'* route op de server. De server vraagt het huidige laatste block in de blockchain op en genereert een 'previous hash', gebaseerd op de inhoud van dit laatste block. Vervolgens wordt een Proof of Work nummer voor het nieuwe block gegenereerd. Van elke transactie in het toe te voegen block wordt de signature geverifieerd. Dan wordt er een belonings-transactie van 10 JorisCoins toegevoegd aan het block, als beloning voor het minen.

Nadat dit alles is gebeurd kan dan eindelijk een block instance worden aangemaakt die is opgebouwd uit een indexnummer, 'previous hash', alle openstaande transacties, Proof of Work nummer, en een timestamp. Dit nieuwe block wordt toegevoegd aan de blockchain, de lijst met open transacties wordt leeggehaald, en de blockchain data (incl. open transacties) wordt opgeslagen in de *blockchain.txt* file. De server stuurt als respons het toegevoegde block terug naar de client. De controller update de view met een bevestigings-alert en update de funds (vermeerdert met 10 als beloning voor het minen).

*CODE:*

- *blockchain.py – line 17-18, 50 t/m 80, 187 t/m 225*
- *blockchain.html – line 154-155, 345 t/m 382*
- *node.py – line 148 t/m 175*
- *hash_util.py*
- *verification.py – line 11 t/m 16*
- *transaction.py – line 19 t/m 22*
- *wallet.py – line 70 t/m 83*
- *block.py*

## 6. Implementeren van beveiliging door het toepassen van 'previous hash' en Proof-of-Work, waarmee wordt gevalideerd dat de blocks in de blockchain niet zijn gemanipuleerd.

Tijdens het minen van een block wordt het huidige laatste block in de chain opgehaald. Vervolgens wordt een 'previous hash' gegenereerd die wordt opgeslagen in het nieuwe block, zodat er een link ontstaat met het voorgaande block. Het genereren van de previous hash gaat als volgt:

a) Om het huidige laatste block ongewijzigd te laten, wordt hier een copy van gemaakt. En die kopie wordt geconvert naar een dictionary.
b) Dan worden de transaction objecten waar het block uit bestaat, geconvert naar OrderedDict's. Er is specifiek gekozen voor converten naar OrderedDict en niet een gewone dictionary, om te waarborgen dat de volgorde van de keys van de transaction objecten vaststaat. Deze volgorde is namelijk van invloed op de hash die wordt gegenereerd.
   Wanneer de volgorde van de keys niet vast zou staan, bestaat er het risico dat er verschillende hashes voor eenzelfde block worden gegenereerd. Terwijl je juist wilt dat steeds dezelfde hash voor hetzelfde block wordt gegenereerd (als de inhoud gelijk is gebleven). Wanneer dit niet het geval is, kan het anders gebeuren dat een correct block toch als incorrect wordt gezien, wanneer diens hash wordt geverifieerd.
c) Vervolgens moet het block worden geconvert naar JSON data, maar het block zelf is echter een dictionary. Om een eenduidige hash voor het block te garanderen moet dus ook de volgorde van de keys van deze dictionary worden vastgezet. Dit gebeurt m.b.v. van het *sort_keys* argument in de *json.dumps()* method.
d) Het block wordt ge-encode naar een string, wat nodig is om het te kunnen hashen.
e) Met behulp van de *sha256()* method wordt voor deze string een hash gegenereerd. De *sha256()* method is een algoritme dat een hash van 64 karakters genereert o.b.v. een ingegeven string, waarbij het algoritme ervoor zorgt dat dezelfde input altijd tot eenzelfde hash als output leidt.
f) Tot slot wordt m.b.v. de *hexdigest()* method de bytehash die is gegenereerd door *sha256()* omgezet naar een gewone, leesbare string.

Op deze manier wordt dus steeds de hash van een block opgeslagen in het block dat erop volgt. Zodoende heeft elk block in de blockchain weet van de inhoud van het block voor hem. Wanneer dan de inhoud van een block wordt gemanipuleerd, dan zal dit bij het volgende block moeten opvallen, omdat de 'previous hash' die daar is opgeslagen niet meer overeenkomt met het block ervoor.

De Proof of Work (PoW) is belangrijk, omdat het ervoor zorgt dat het minen van nieuwe blocks uitdagender wordt. Wat eigenlijk zoveel betekent als dat de activiteit van minen meer tijd (en in de meest realistische zin ook: meer hardware resources) vereist.
Want het beveiligingsmechanisme waarbij een block wordt vergeleken met de opgeslagen 'previous hash' in het opvolgende block is op zichzelf niet genoeg. Iemand met kwade bedoelingen kan een block manipuleren en vervolgens alle opvolgende 'previous hashes' ook manipuleren, waardoor het gemanipuleerde block toch als valide wordt gezien. Maar het PoW voorkomt dit.
De Proof of Work is samen te vatten als het vinden van een nummer (de proof) dat ervoor zorgt dat er wordt voldaan aan een door de blockchain gesteld PoW-criterium.
De combinatie van de proof + de transacties in het block + de previous hash moeten samen leiden tot een PoW-hash die voldoet aan het PoW criterium. In de JorisCoin blockchain is het PoW-criterium dat de proof ervoor moet zorgen dat de PoW-hash van 64 karakters begint met twee nullen.

Op het moment dat een proof nummer wordt gevonden dat ervoor zorgt dat aan het PoW-criterium wordt voldaan, dan wordt dat proof nummer opgeslagen in het nieuwe block.
De beveiliging zit hem erin dat dit vinden van een valide proof een tijdsintensief proces is. Als iemand met kwade bedoelingen namelijk een block in de blockchain manipuleert, dan resulteert geen van de proof nummers in de opvolgende blocks nog in een valide PoW-hash. Deze persoon zou dan voor elk opvolgend block ná het gemanipuleerde block, weer opnieuw een proof nummer moeten vinden dat wel weer een valide PoW-hash genereert. Dit updaten van de blockchain kost ontzettend veel tijd. Daarentegen hoeven andere 'eerlijke' miners in de tussentijd het proof nummer voor maar één nieuw block te vinden, waardoor de blockchain overschreven zal worden. Een 'slechte' miner kan natuurlijk nooit een gedeelte van de gehele blockchain manipuleren, in de tijd die het andere miners kost om één nieuw block aan de chain toe te voegen.

In de JorisCoin blockchain wordt tijdens het minen van een nieuw block een geldig Proof of Work nummer gegenereerd. Er zijn drie elementen nodig die samen tot een geldige PoW-hash (die begint met twee nullen) moeten leiden: de open transacties, de 'previous hash' en het proof nummer. Dit gaat als volgt in zijn werk:

a) De open transacties van het nieuw block staan natuurlijk vast, dus dan zijn nog de 'previous hash' en proof nummer nodig.
b) Voor de previous hash wordt het huidige laatste block in de chain opgehaald, en voor dat block wordt een hash gegenereerd. Hoe dit genereren van de 'previous hash' gebeurt is reeds tot in detail beschreven binnen deze requirement.
c) Blijft alleen nog het vinden van een valide proof nummer over. Daarvoor wordt simpelweg begonnen door de proof op nul te zetten, en dan te checken of de combinatie open transactions + previous hash + proof leidt tot een 'guess-hash' die begint met twee nullen. Is dat niet het geval dan wordt het proof nummer met één vermeerdert en wordt opnieuw gecheckt. Net zo lang tot dit leidt tot een 'guess-hash' die wel begint met twee nullen.
d) Dit genereren van een 'guess-hash' gebeurt door eerst de open transacties, de previous hash en het proof nummer te converten naar strings. Hierbij worden de open transacties eerst geconvert naar OrderedDict's om de volgorde van de keys vast te zetten, en zodoende een consistente 'guess-hash' te waarborgen.
   De drie strings worden samengevoegd tot één string, en van deze string wordt een hash van 64 karakters gemaakt m.b.v. de *sha256()* method. De bytehash die hieruit komt wordt d.m.v. *hexidigest()* omgezet naar een leesbare string, waardoor kan worden gecheckt of deze begint met twee nullen.
e) Uiteindelijk wordt een proof nummer gevonden, die in combinatie met de open transacties en de previous hash leidt tot een 'guess-hash' die begint met twee nullen. Dit proof nummer wordt opgeslagen in het nieuwe block.

De validatie van de algehele blockchain wordt uitgevoerd wanneer een gebruiker de blockchain inlaadt door op de *Load Blockchain* button te klikken. Er wordt dan door alle blocks van de chain gegaan, en de previous hashes en Proof of Work nummers van elk block worden gecontroleerd. Op het moment dat de blockchain invalid blijkt te zijn, wordt de blockchain als leeg op de webpagina getoond en verschijnt er een error alert.

Daarnaast kan het signen van transacties met de private key (en verifiëren van deze signature met de public key) worden gezien als een derde veiligheidsmaatregel in de JorisCoin blockchain. Maar waar de previous hashes en Proof of Work zijn gericht op beveiliging tegen manipulatie van de blocks, is het signeren van transacties gericht op beveiliging van manipulatie van uitgaande transacties.

*CODE:*

- *node.py – line 148 t/m 167, 180 t/m 205*
- *blockchain.py – line 101 t/m 108, 187 t/m 225*
- *hash_util.py*
- *wallet.py – line 60 t/m 83*
- *verification.py*
- *transaction.py – line 19 t/m 22*