UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

"Studio ed implementazione dei meccanismi di minting NFT su blockchain"

Relatore: Prof. MAURO MIGLIARDI

Laureando/a: FRANCESCO CRISCI

Correlatore: Dott. FABIO PALLARO

ANNO ACCADEMICO 2021 – 2022

Data di laurea 20-07-2022

Astratto

In questo documento viene illustrata l'implementazione della generazione di NFT e dell'automatismo di mint attraverso l'uso del framework Brownie. Le conoscenze descritte sono state acquisite durante l'esperienza svolta presso l'azienda Synclab s.r.l, sotto la supervisione dell'Ingegnere Fabio Pallaro e del professor Mauro Migliardi. Il percorso è stato strutturato in 10 settimane, suddivise in 3 settimane di preparazione teorica sugli argomenti e 7 settimane di lavoro pratico. L'obiettivo del progetto era di implementare, attraverso la combinazione dei linguaggi di programmazione python e solidity, il processo di realizzazione di NFT e l'automatismo di minting su rete Polygon di test. Come risultato si sono ottenuti una collezione di 500 immagini uniche che attraverso l'automatismo di minting, sono state rese una collezione NFT, collezione visibile sulla piattaforma openSea.

Indice

Elenco delle figure

Capitolo 1

Blockchain

Nel primo capitolo vengono illustrate le caratteristiche generali della blockchain e degli elementi che la costituiscono. Si passa successivamente all'illustrazione del problema della scalabilità di una blockchain e delle soluzioni adottate. Si arriva successivamente alla presentazione dei protocolli di consenso che governano le blockchain.

1.1 Cosa è una blockchain

"La blockchain, una tecnologia di database transazionali, è un metodo decentralizzato per gestire validazione e transazioni a prova di manomissione con consistenza attraverso un numero significativo di partecipanti, conosciuti anche come nodi"[1]. Può essere classificata come una tipologia di ledger distribuito che fornisce confidenza all'utente che l'informazione archiviata non risulti manomessa.
"Un ledger distribuito è spesso descritto come un database condiviso e distribuito a cui si accede e che viene mantenuto da un insieme di partecipanti indipendenti e possibilmente non fidati (cioè i nodi). Ogni partecipante possiede una copia identica del database delle transazioni (cioè il ledger) mantenuto su una rete peer-to-peer"[2]. Attraverso l'utilizzo di algoritmi di consenso, i partecipanti concordano ed esprimono immediatamente tutte le modifiche o le aggiunte al ledger.
Con l'evolversi di questa tecnologia, la blockchain ha ricevuto molte attenzioni a causa di caratteristiche uniche che può offrire, ovvero sicurezza, anonimato, trasparenza e decentralizzazione.
Si è dimostrato che la blockchain ha le capacità per diminuire gli stati insicuri e il dubbio fornendo la rivelazione completa delle transazioni e l'integrazione di fatti omogenei e verificati tra tutti i partecipanti alla rete. Grazie alla sua natura decentralizzata, la blockchain manca di un'autorità che sincronizzi lo stato dei processi. Si è dunque utilizzata come soluzione l'implementazione di algoritmi di consenso che sono responsabili della convalida dello stato delle transazioni propagate nella rete e del coordinamento dei nodi distribuiti. Gli algoritmi di consenso forniscono anche affidabilità, danno vita alla rete e la difendono dagli attacchi malevoli.

Questa tecnologia trova vari impieghi e può essere utilizzata per concludere accordi, tenere traccia dei movimenti sui beni, memorizzare i bilanci finanziari di singoli individui, tracciare l'assegnazione di opere d'arte, e verificare pagamenti con l'aiuto di supply chain insieme ad altri processi e procedure.

1.2 Le caratteristiche di una blockchain

"Fiducia e decentralizzazione sono le due caratteristiche principali che bisogna identificare quando si esaminano una blockchain"[1].

1.2.1 Affidabilità

La rete è governata generalmente da un protocollo di consenso chiamato Proof-of-work che rende possibile eliminare il bisogno di una terza parte di cui tutti i partecipanti si fidino per raggiungere il consenso e inscrivere nuovi blocchi nella blockchain. Grazie all'utilizzo di questo protocollo si cerca di garantire sicurezza su tutte le transazioni e beni degli utenti partecipanti alla rete. Risulta difficile, ma non impossibile, costruire back-door all'interno del sistema. Infatti del codice scritto da un qualsiasi utente può essere utilizzato all'interno della blockchain fintantoché non vengano effettuate operazioni che violino i protocolli che governano la rete.

1.2.2 Decentralizzazione

"Tra le caratteristiche maggiori, la decentralizzazione è la caratteristica predominante della tecnologia blokchain"[1]. Grazie a questo aspetto si ottengono immutabilità e resistenza a censure. La decentralizzazione è gestita da protocolli di consenso, tra i quali la proof-of-work e la proof-of-stake. Attraverso il loro utilizzo si riescono a coordinare i nodi della rete e si evita l'utilizzo di organi centralizzati come terze parti. Questa caratteristica fornisce molta sicurezza da attacchi cyber-terroristici, non escludendo, però, una possibile minaccia da parte di organizzazioni di grandi dimensioni come il governo di una nazione.

1.3 Struttura della blockchain e funzionamento di un blocco

Il termine blockchain deriva dalla sua struttura, ovvero una struttura lineare concatenata di blocchi. Come visibile in figura 1.1, ogni nodo della rete cerca di estrarre un blocco, che verrà successivamente inserito in cima alla pila dei blocchi estratti precedentemente.

L'unità fondamentale che compone i blocchi sono le transazioni, ovvero trasferimento di valuta, come nel caso di Bitcoin, o come nel caso di Ethereum, sia di valuta che di beni quali opere d'arte.

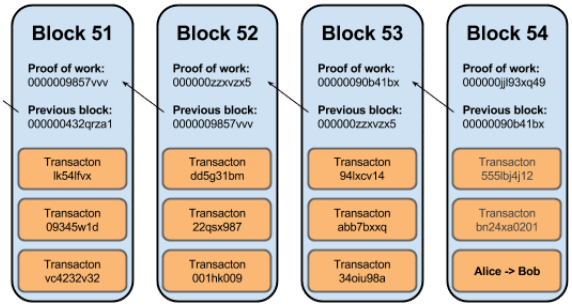

Figura 1.1: Struttura della blockchain

Grazie alla sua struttura, permette di sincronizzare gli utenti sullo stato e sulla storia delle transazioni avvenute. Attraverso questa sincronia, i nodi sono sempre in accordo sull'esatto numero di blocchi e la loro storia, mentre continuano a lavorare per crearne di nuovi. Questo meccanismo è possibile data la natura ordinata che li contraddistingue. Ogni blocco generato contiene un riferimento al precedente. Quando ne viene creato uno con successo, viene distribuito nella rete dove ognuno dei partecipanti lo aggiungerà alla propria copia privata della blockchain.
Riportiamo ora la struttura di un blocco all'interno della blockchain Ethereum, struttura che presenta molte similarità con le strutture dei blocchi di altre blockchain. Si è preferito illustrare questa struttura in quanto la rete utilizzata per il progetto in questione è la rete Polygon di test che usa, per il suo funzionamento, la rete Ethereum. La struttura di un blocco è la seguente (vedere figura 1.2):

- timestamp: l'istante temporale nella quale è stato estratto il blocco.
- numero di blocco: un numero intero che rappresenta la lunghezza della blockchain. Si usa come unità di misura il blocco.
- baseFeePerGas: la tassa minima richiesta, in gas, per far sì che una transazione venga inclusa in un blocco.
- difficoltà: la quantità di lavoro necessaria richiesta per estrarre un blocco.
- mixHash: un identificatore unico del blocco.
- parentHash: un identificatore univoco che punta al blocco precedente all'interno della catena. Questo parametro permette l'ordinamento dei blocchi e connette i blocchi della catena.
- transazioni: le transazioni che compongono il blocco.
- stateRoot: l'intero stato del sistema. Include i bilanci degli utenti, il codice del contratto che lo compone e il nonce degli utenti.
- nonce: Parte della stringa di cui si calcola l'hash. Quando l'hash calcolato rispetta determinate condizioni significa che il blocco ha superato la proof-of-work[3].

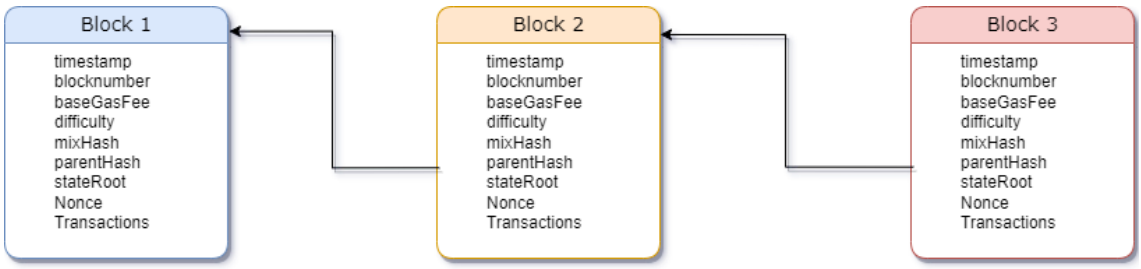

Figura 1.2: Immagine raffigurante la struttura dei blocchi e come essi sono connessi fra loro.

1.4 Tipologia di una blockchain

Le blockchain sono suddivise generalmente in tre macrocategorie:

- pubbliche, ogni utente o organizzazione può diventare parte della rete o lasciarla senza costi. Ogni partecipante possiede un insieme di diritti e privilegi simili tra loro.
- private, presentano una figura centrale che governa tramite protocolli di consenso la rete. Solamente il corpo principale che governa la rete può prendere decisioni e controllare il processo di validazione.
- consortium, solo un gruppo ristretto di utenti ha la possibilità di validare le transazioni.

1.5 I layer della blockchain

Il termine layer, nell'ambito delle blockchain, deriva dall'architettura che governa la rete. Usando questo approccio, abbiamo due categorie di layer, ovvero i layer di tipo 1 e di tipo 2.

1.5.1 Layer di tipo 1

Quando parliamo di layer di tipo 1, facciamo riferimento a reti di base, tra le quali Ethereum e Bitcoin. I layer di tipo 1 sono dunque delle blockchain basilari, dove si ha un solo canale principale. Queste reti non permettono, però, di processare un numero alto di transazioni, in quanto la rete potrebbe andare in contro a una congestione. Una possibile congestione della rete è dovuta dalla struttura dei blocchi oppure dal protocollo di consenso che si sta adottando. Infatti, un blocco può contenere un numero fisso di transazioni, e considerando che l'approvazione di un blocco richiede un intervallo compreso tra 2 e 30 minuti (bisogna considerare che in media un blocco viene considerato approvato dopo che circa 6 blocchi sono stati costruiti sopra a esso, e che in media un blocco viene aggiunto alla catena dopo un determinato lasso di tempo compreso tra i 15 secondi e i 5 minuti), processare molte transazioni può risultare difficile, se non impossibile. Come possibili soluzioni si è pensato di utilizzare un blocco di dimensione maggiore, e che quindi può contenere

più transazioni, o cambiare il protocollo di consenso. Sulla base dell'ultimo punto, la rete Ethereum sta passando da una proof-of-work a una proof-of-stake, protocolli che vedremo successivamente in u altro paragrafo. Tutte queste tematiche sono convolute in un nuovo problema: la scalabilità della blockchain. Per scalabilità della blockchain si intende l'evolversi, e dunque il crescere, della rete con il tempo. Di fatto, più la rete cresce, più transazioni verranno emesse e dovranno essere processate, rendendo quindi più alta la probabilità che la rete si congestioni. Una soluzione alternativa per migliorare la scalabilità della blockchain è stata l'adozione di blockchain frammentate. Questo è un approccio sperimentale, che mira alla suddivisione della blockchain in catene più piccole, rendendo così la gestione della rete più efficiente e facile. Come ultima soluzione, si è pensato di adottare layer di tipo 2, della quale parleremo nel prossimo paragrafo.

1.5.2 Layer di tipo 2

Per layer di tipo 2 si intendono i framework, o i protocolli, che hanno alla loro base blockchain layer di tipo 1. In questo progetto viene utilizzata la rete con layer di tipo 2 Mumbai, conosciuta anche come rete di test Polygon. Grazie all'utilizzo di layer di tipo 2, si riescono a processare centinaia, se non migliaia di transazioni in un secondo. Questo risulta essere un grande passo in avanti per risolvere il problema della scalabilità delle blockchain con layer di tipo 1. Infatti, la maggior parte del carico lavorativo viene eseguito sul secondo layer, rendendo così più efficiente il layer primario. Usando un layer secondario, si evita di dover apportare cambi strutturali al layer primario. Questa struttura fornisce due principali vantaggi: una efficienza maggiore e una elevata sicurezza, quest'ultima derivante dalle caratteristiche del layer primario. Le principali tipologie di layer secondari sono tre, e ognuna di esse offre vantaggi differenti.

- Blockchain annidate, o nested blockchains: si utilizza un modello a cascata,dove ogni blockchain è legata da una relazione padre-figlio. Il figlio prende in carico il lavoro del genitore per poi ritornare il risultato della computazione. La chain principale non prende parte nella computazione, ma risolve solo eventuali dispute tra le varie catene sottostanti.

- Canali di stato, o state channels: strumento utilizzato per facilitare la comunicazione tra una blockchain principale e una esterna. È una risorsa sigillata, ovvero non richiede la validazione dal layer sottostante. Solamente lo stato finale del canale di stato, ovvero quando il banco delle transazioni è completo, viene consegnato e registrato dalla catena principale.

- Catene ausiliari, o sidechains: catene a sé stanti che hanno il compito di processare un numero elevato di transazioni. Ognuna di queste catene può adottare un qualsiasi protocollo di consenso, che varia in base ai requisiti richiesti. Il layer principale ha il compito di mantenere la rete sicura, di risolvere dispute e di confermare le transazioni. Tutte le violazioni sulla sicurezza che si possono verificare all'interno di una catena ausiliaria, non hanno ripercussioni sulle altre sidechains o, in particolar modo, sul layer primario.

1.5.3 Layer 1 vs Layer 2

L'utilizzo di un layer di tipo 2 nasce dalla richiesta, sempre maggiore, di approvare un numero crescente di transazioni. Ad esempio, se si vuole far girare un videogioco su un layer di tipo 1, si riscontrerebbero enormi difficoltà, in quanto processare una singola transazione può richiedere un considerevole lasso di tempo, rendendo così sgradevole l'esperienza di un utente. Ma cosa succede se questo videogioco vuole usufruire comunque della sicurezza offerta dai layer di tipo 1? Si utilizzano layer di tipo 2, che offrono una velocità computazionale maggiore e la sicurezza richiesta dall'utente. In generale, più che una alternativa al layer di tipo 1, il layer di tipo 2 è un upgrade che permette di superare i limiti fisici e tecnologici che limitano le blockchain layer 1.
Come spiegato nel paragrafo precedente, un layer di tipo 2 ha come obbiettivo la suddivisione del lavoro, in modo che la rete principale, quelle di layer 1, non vada incontro a un sovraccarico di lavoro e che quindi non si congestioni. Un layer di tipo 2 permettere di processare molte transazioni attraverso l'esecuzione in parallelo di tutte le operazioni necessarie affinché queste ultime possano essere processate correttamente. Usando meccanismi di parallelismo e suddivisione del lavoro, tutte le reti di layer 2, che fanno riferimento al layer primario, permettono di processare un numero maggiore di transazioni rispetto ad una rete che utilizza solamente un layer di tipo 1.

1.6 Protocolli di consenso

I protocolli di consenso sono le leggi che governano e regolano le comunicazioni tra i nodi della rete. In questo paragrafo vedremo i due protocolli principali e più utilizzati dalle blockchain attuali, ovvero la proof-of-work e la proof-of-stake.

1.6.1 Proof-of-work

"La proof-of-work richiede che i partecipanti estraggano un blocco e richiede che risolvano problemi di difficoltà crescente per accertarsi che il blocco sia valido"[4]. Ogni volta che un blocco viene estratto con successo, una ricompensa viene conferita al nodo che ha completato il blocco. Questa ricompensa viene calcolata in base al numero di partecipanti presenti nella rete e in base alla difficoltà assegnata a quel blocco. Più grande sarà la difficoltà del blocco, maggiore sarà la ricompensa del miner e viceversa. Uno dei principali problemi riguardo a questo protocollo è lo spreco delle risorse. Di fatto, tutti i nodi nella rete competono per risolvere il prima possibile il nuovo blocco. Quindi tutti i nodi lavorano continuamente e non hanno mai la certezza che il blocco che il loro blocco venga effettivamente aggiunto in cima alla catena. Un altro lato negativo di questo protocollo è il continuo crescere dei requisiti minimi dei dispositivi utilizzabili. Con il passare del tempo, i nodi necessitano di apparecchiature sempre più potenti e performanti per riuscire a risolvere un blocco. Richiedono anche una sorgente di corrente continua in modo da rendere il ledger

più robusto contro manomissioni da parte di altri nodi. L'ultimo aspetto negativo che affrontiamo in questo paragrafo è l'attacco al 51% sulla rete. Questa tipologia di attacco richiede che un utente, o anzi, una organizzazione, controlli almeno il 51% della rete, riuscendo così a controllare lo stato della rete e aggiudicarsi una autorità che non dovrebbe essere possibile all'interno della blockchain. Chi controlla la rete con questo genere di attacco, controlla anche la storia dei blocchi e delle transazioni. Mette a rischio, dunque, tutti gli utenti che hanno preso parte alla rete, che hanno speso tempo e risorse per estrarre blocchi. Blocchi che verranno "rimossi" dalla blockchain per venir rimpiazzati dai blocchi estratti dagli utenti esecutori di questo attacco. Un attacco al 51% risulta essere, però, poco fattibile, in quanto controllare il 51% della rete risulta essere più dispendioso del guadagno effettivo e molto difficile da raggiungere considerando che la rete è molto vasta. "Una delle assunzioni più grandi da bitcoin è che la maggior parte dei minatori sono affidabili. Questo però, non è un qualcosa che può essere provato ma più una supposizione"[4].

1.6.2 Proof-of-stake

"La proof-of-stake fornisce la possibilità di lavorare al nodo che presenta nel proprio stake (numero di token che il nodo possiede nella rete) il maggior numero di token"[4]. Così facendo si assicura che il nodo in questione sia credibile e non ha intenzione di manipolare la rete, rendendo così la rete sicura. Il nodo designato per lavorare usa il proprio stake come garanzia. Infatti, se il nodo cerca in un qualsiasi modo di manomettere la rete, perde il proprio stake. Per cui, la proof-of-stake incentiva i nodi a lavorare correttamente attraverso un meccanismo punitivo nei confronti di chi ha secondi fini. Questo protocollo di consenso, come affermato da [4], si è dimostrato essere più adatto a reti con un setup privato, in quanto meno scalabile rispetto a reti che adottano la proof-of-work, anche se esistono algoritmi più performanti come, ad esempio, quelli basati sulla BFT[1]. Tuttavia, nonostante queste considerazioni, la rete Ethereum sta implementando come rimpiazzo alla proof-of-work il protocollo proof-of-stake.

1.6.3 Proof-of-work vs proof-of-stake

Vediamo ora le differenze tra i due protocolli di consenso e cosa rende un protocollo migliore dell'altro. Sotto un aspetto di dispendio energetico, la proof-of-work utilizza il livello più alto di energia possibile per tutto il periodo di attività della rete. La proof-of-stake risulta essere invece meno dispendiosa sotto questo punto di vista, anche se la quantità di energia richiesta e le risorse impiegate sono molte. Considerando, invece, la correttezza della rete, la proof-of-work risulta essere molto più adeguata ed "onesta" con i miner. A livello di distribuzione di premi dovuti al mining del blocco, la proof-of-work premia chi ha risolto il blocco più rapidamente, mentre la proof-of-stake tiene conto solamente di chi ha guadagnato più token, di cui parleremo più avanti, all'interno della

[1] Byzantine fault-tolerant è un algoritmo di consenso che coordina la rete in caso di guasti arbitrari

rete. L'ultimo aspetto da considerare è l'affidabilità del sistema. Mentre la proof-of-stake è meno affidabile, in quanto tende molto a una struttura "oligarchica", la proof-of-work risulta essere molto affidabile.

1.7 Cosa è uno smart contract

"Gli smart contract sono solitamente programmi creati da utenti, protocolli o algoritmi che possono essere usati per verificare, validare o rendere irreversibili transazioni"[5]. Sono memorizzati all'interno in un indirizzo[2] specifico all'interno della blockchain. "Possono essere interpretati come contratti che sono scritti in codice e che sono automaticamente eseguiti sulla blockchain"[6]. Gli smart contract sono immutabili, una volta spediti nella rete non possono essere modificati e vengono scritti attraverso l'uso di linguaggi di programmazione. In questo progetto si è fatto riferimento al linguaggio di programmazione Solidity, un linguaggio orientato agli oggetti, dove le classi prendono il nome di contratti, utilizzato all'interno della blockchain Ethereum, Polygon principale e di test. Tutti i metodi della classe che modificano il contenuto del contratto hanno un costo espresso in wei, e vengono chiamati ed eseguiti attraverso l'uso di transazioni. Il wei è l'unita di misura per il costo delle transazioni. Un Ether, ovvero una moneta della valuta Ethereum, risulta essere 10^{18} wei, così come 10^{18} wei possono essere convertiti in un Matic, valuta della rete Polygon. Alcune strutture degli smart contract, con il passare del tempo, sono diventate degli standard da seguire per gli utenti della rete. In particolar modo, gli standard ERC20, ERC721 e ERC1155 sono i più utilizzati e "famosi" attualmente. In questo capitolo verranno elencati tutti e 3 per capirne le differenze, soffermandosi in modo particolare sullo standard ERC721, standard sulla quale il progetto discusso si basa.

[2] un indirizzo è una stringa univoca di testo che identifica un contratto, una transazione o un utente all'interno della blockchain.

Capitolo 2

Tipologie di token

In questo capitolo vediamo in dettaglio cosa è lo standard ERC721 e come esso permette di rappresentare token non fungibili. Viene mostrato l'interfaccia degli smart contract degli standard ERC20, ERC721 e ERC1155, elencandone i metodi che li rendono importanti all'interno della blockchain.

2.1 Standard ERC20

"Ogni contratto ERC20 è identificato da un nome e un simbolo che può essere impostato solo una volta durante la costruzione del contratto e fornisce funzionalità per la gestione del token, come trasferimenti o approvazione"[7]. Anche se uno standard, lo ERC20 si appoggia su librerie esterne, come SafeMath, per eseguire operazioni sui token. Tra le caratteristiche principali ci sono un mapping[1] chiamato _balances che riporta il bilancio in token di uno specifico indirizzo. Sono presenti anche ulteriori metodi e variabili, che sono riportati in figura 2.1. "È presente un ulteriore mapping annidato, ovvero un doppio mapping, chiamato _allowance che connette il proprietario con un delegato e il delegato con il numero massimo di token che è autorizzato a spendere"[7].
Un token ERC20 può essere divisibile, ovvero rappresenta un oggetto non unico, intercambiabile e che più persone possono avere contemporaneamente. Un'analogia al token ERC20 è sicuramente una qualsiasi valuta. Chiunque può possedere dei dollari, o degli euro, aventi tutti lo stesso valore per ogni persona e con la proprietà di essere divisi in unità più piccole.

2.2 Standard ERC721

"Lo standard ERC721 ha introdotto il concetto di token non fungibili. I token non fungibili, o NFT, sono una tipologia speciale di token che permette loro d'identificare qualcosa di unico, diventando così non interscambiabili, come token ERC20 e altre cripto-valute"[8]. "Gli NFT permettono di di-

[1] Il mapping è un dizionario solidity, dove si possono creare coppie chiave-valore di un qualsiasi tipo supportato da solidity. In questo specifico caso, la chiave è di tipo indirizzo.

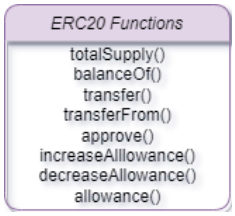

Figura 2.1: Metodi che compongono lo smart contract dello standard ERC20

mostrare la proprietà di un token digitale senza l'intervento di un'istituzione centralizzata, così che il token può essere usato in vari spazi sociali, tipicamente associate al contesto digitale"[9]. "Questo standard può essere definito come lo standard che descrive come creare token indistruttibili o inseparabili nella blockchain Ethereum"[10]. È composto da varie funzioni, sia ereditate da altri standard che proprie dello standard stesso (vedere figura 2.2). Le funzioni presenti all'interno di questo standard sono lasciate in bianco, e devono essere implementate dall'utente. Richiede, dunque, una conoscenza abbastanza importante del linguaggio di programmazione solidity, di come gli smart contract funzionano e in che modo possono essere efficienti e privi di ambiguità. Per l'implementazione del contratto in sé, sono richieste delle interfacce[2], metodi e altre funzionalità proprie di altri contratti. Fortunatamente ci sono organizzazioni come Openzeppelin che forniscono agli utenti le librerie necessarie e la firma del contratto ERC721 gratuitamente, in modo che anche i programmatori novizi possano implementare il proprio token ERC721. Ogni insieme di NFT costituisce una collezione, che può essere una collezione d'immagini, una collezione di contratti d'affitto, una collezione di biglietti aerei o un qualsiasi gruppi di elementi rappresentabili non intercambiabili. Ad esempio, un biglietto aereo potrebbe essere tramutato in un NFT, in quanto esiste un unico biglietto per un posto per un volo che ha una specifica data e luogo. "Più in generale, un NFT è usato per identificare univocamente qualcosa o qualcuno"[9]. Lo standard ERC721 permette, quindi, di trasferire dei token da un indirizzo a un altro, con la peculiarità di fornire anche i metadati necessari a interpretare in modo corretto l'NFT ricevuto. Inoltre, tutti i token creati, sono unici, in quanto i metadati che li compongono e i valori dei parametri assegnati a essi sono differenti. L'insieme di questi fattori costituisce gli elementi di input per la funzione di hash che genererà il dna del token. Il dna del token è l'hash che lo compone, ovvero la stringa univoca che indentificherà univocamente il token in questione.

2.2.1 I metodi dello standard ERC721

Vediamo ora in dettaglio i metodi più importanti che compongono lo standard ERC721 così da ottenere una visione generale di ciò che lo smart contract permette all'utente di fare.

- name() : ritorna come valore la stringa contenente il nome del token.

[2]Una interfaccia in solidity è un contratto che definisce solo la firma dei metodi, senza specificarne il corpo.

Figura 2.2: I metodi che compongono lo smart contract dello standard ERC721

- totalSupply() : ritorna il numero totale di token creati dallo smart contract.
- balanceOf() : ritorna il numero di token posseduti da un utente. Accetta come parametro l'indirizzo dell'utente della quale si vuole sapere il bilancio.
- ownerOf() : accetta come parametro un intero che rappresenta l'id del token che si vuole ispezionare. Ritorna l'indirizzo del proprietario del token specificato.
- setApproval() : permette di autorizzare un altro utente a gestire uno o più token. I token autorizzati possono essere trasferiti o distrutti dal proprietario e da tutti gli utenti autorizzati.
- transfer(): permette di trasferire un token da un utente a un altro. Questa operazione può essere eseguita solamente dal proprietario o da un indirizzo approvato tramite il metodo setApproval().
- tokMetadata() : ritorna i metadati che compongono il token in questione. I metadati contengono solitamente le caratteristiche che definiscono il token, e nel caso di un'immagine, contiene anche l'indirizzo su dove l'immagine e memorizzata nella rete.

Questi metodi sono i metodi riportati nell'articolo[10], ma sono solamente i metodi più importanti che costituiscono lo standard ERC721.
Ecco una lista di altri metodi che compongono lo standard: constructor(), supportsInerface(), symbol(), tokenURI(), _baseURI(), approve(), getApprove(), setApprovalForAll(), isApprovedForAll(), transferFrom(), safeTransferFrom(), _safeTransfer(), _exists(), _isApprovedOrOwner(), _safeMint(), _mint(), _transfer(), _approve(), _setApprovalForAll(), _beforeTokenTransfer() e _afterTokenTransfer(). I metodi appena elencati sono stati presi dalla pagina ufficiale docs.openzeppelin.com e alcuni di essi saranno trattati e spiegati più approfonditamente nel capitolo dedicato alla realizzazione del progetto in sé.

2.2.2 I metadati

I metadati sono i tratti descrittivi dei token, ovvero contengono tutte le informazioni necessarie per interpretare correttamente il token e i tratti distintivi di ogni singolo NFT. Come affermato precedentemente, i metadati sono ciò che rendono unico un token, ovvero i valori dei tratti che lo compongono. Per capire al meglio il ruolo dei metadati bisogna capire cosa sono i tratti che compongono un token. Così come una persona possiede tratti specifici che la contraddistinguono dagli altri esseri umani, così gli NFT hanno le loro caratteristiche. Prendiamo come esempio il ritratto di una persona, e supponiamo che ora questo ritratto sia il nostro NFT. I tratti che compongono il nostro quadro possono essere:

- il nome del quadro, nel nostro caso il nome del token.
- il colore dei capelli della persona, esprimibile con un valore. Ad esempio, il colore dei capelli è biondo.
- il colore degli occhi, ad esempio azzurro.

Questi riportati sono solo alcuni dei valori che un utente può impostare manualmente, o causalmente attraverso l'uso di programmi, nei metadati di un token. Ovviamente, come nella vita reale, ci possono essere dei tratti meno comuni di altri, come ad esempio colore degli occhi azzurri o capelli ricci. Questo concetto introduce un ulteriore livello di classificazione degli NFT, ovvero la rarità. La rarità ci permette di determinare quali NFT siano più difficili da possedere, in quanto più la rarità di un NFT è alta, meno sarà la probabilità che quell'NFT sia stato generato.
Come analogia possiamo prendere in esempio una qualsiasi collezione di carte collezionabili. Ovviamente ci saranno carte che sono più difficili da trovare di altre, così come ci saranno NFT con caratteristiche uniche più difficili da ottenere. Il concetto di rarità influisce anche sul prezzo di un token, infatti più un token possiederà caratteristiche rare o addirittura uniche, più il suo prezzo risulterà alto. In conclusione, i metadati giocano un ruolo importante nella compra-vendita dei token non fungibili.

2.3 Standard ERC1155

Questo standard è nato dai conseguenti limiti ed errori d'implementazione dei due standard precedenti e permette di combinare le caratteristiche dei due in un unico ed efficiente smart contract.
Permette di gestire contemporaneamente le due tipologie di token trattate precedentemente, in modo autonomo e indipendente. Infatti, i bilanci di token ERC20 e di ERC721, così come le firme dei metodi che li compongono, sono divisi e non a rischio di errori come l'interpretazione sbagliata di uno di essi.
Una delle motivazioni per la quale questo standard è diventato necessario è l'incompatibilità tra i

due tipi di standard all'interno degli applicativi che li utilizzano. Gli applicativi che fanno uso di uno dei due standard, o di entrambi, prendono il nome di Dapps.
Le Dapps hanno la capacità di gestire i due standard separatamente, ma questo risulta essere molto dispendioso e complicato per moli grandi di utenze e per periodi prolungati.
Grazie all'uso dello standard ERC1155, si sono rese le Dapps molto più efficienti e in grado di gestire efficacemente i due standard contemporaneamente.
I metodi che compongono lo standard sono gli stessi dello standard ERC721, con l'aggiunta di alcuni metodi peculiari quali balanceOfBatch(), safeBatchTransferFrom(), _safeBatchTransferFrom(), _mintBatch() e _burnBatch(). Tali metodi vengono illustrati all'interno della figura 2.3.

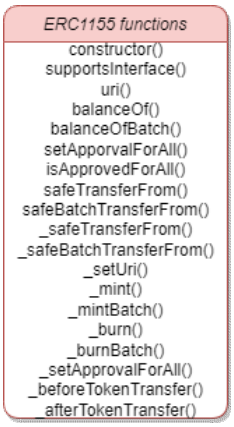

Figura 2.3: I metodi che compongono lo smart contract dello standard ERC1155

Come possiamo notare, ci sono dei metodi che si riferiscono a batch di token. Questi sono metodi molto utili in quanto permettono di mandare token multipli all'interno di una singola transazione. I batch di token vengono trattati come se fossero dei singoli token, di fatto dispongono di tutte le funzionalità che i singoli token possiedono all'interno dello standard ERC721.

Capitolo 3

Generative art

In questo capitolo vengono illustrate i meccanismi e l'implementazione della Generative art utilizzate all'interno del progetto. Vengono fornite inizialmente conoscenze teoriche riguardanti l'argomento, per poi passare alla parte implementativa e allo pseudocodice utilizzati per la realizzazione del progetto.

3.1 Cosa è la Generative art

La Generative art, o arte generativa, è il processo utilizzato per la creazione delle immagini finali che andranno a comporre la collezione di NFT. Le immagini vengono generate attraverso la scelta casuale degli elementi dei vari livelli che la compongono. I dettagli come occhi, naso, bocca, capelli, orecchie e accessori appartengono a layer distinti tra loro. Ogni layer dispone di un catalogo di scelte dalla quale il programma di generazione può prelevare un dettaglio. In generale, ogni livello deve possedere più di un singolo taglio di capelli, di una tipologia di orecchie, di occhi di colori differenti e di altre caratteristiche. Questo perché ogni NFT appartenente a una collezione deve essere diverso da tutti gli altri, ovvero deve essere unico nel complesso delle caratteristiche che lo compongono. Se per esempio un'immagine risulta avere capelli viola, occhi azzurri, orecchie a punta, tutte le altre dovranno avere almeno una di queste caratteristiche differenti. Quindi un'altra immagine avente capelli viola, occhi verdi e orecchie a punta può essere presente senza alcun tipo di problema, mentre un'altra immagine avente le stesse caratteristiche della prima non può essere presente. Questa operazione può risultare, però, più difficile di quel che sembri. Infatti, a livello informatico, la generazione casuale è un argomento particolare, in quanto i calcolatori sono macchine deterministiche, per cui risulta difficile generare casualmente ogni volta un qualcosa sempre di diverso. Ovviamente, ci sono soluzioni che permettono di rendere sempre diverso la generazione di pattern casuali, tra le quali l'uso dell'istante temporale in cui la macchina lavora, in quanto sarà sempre diverso dagli istanti che lo precedono. La generazione causale degli elementi che compon-

gono l'immagine che andrà a costituire l'NFT ha introdotto due concetti importanti, ovvero la rarità e i metadati.

3.1.1 Rarità di un NFT

In questo sotto-paragrafo vediamo più in dettaglio il concetto di rarità, concetto già introdotto nel capitolo precedente.
Il concetto di rarità è un concetto molto influente nell'ambito delle NFT relative a opere artistiche. Possiamo immaginare le NFT come delle carte collezionabili, senza la possibilità di avere duplicati.
La rarità nasce dalla natura casuale delle caratteristiche che compongono ogni singolo NFT. Se durante la fase di generazione si generano un numero immagini minore del numero totale di combinazioni possibili, ci saranno delle caratteristiche che compariranno meno di altre. Questo accade a causa della natura della casualità, e comporta la naturale generazione del concetto di rarità. Se su 500 immagini generate, solo 20 presentano occhi azzurri, ovviamente quelle 20 NFT saranno più difficili da ottenere e trovare rispetto a tutte le altre che hanno occhi marroni o verdi. Ovviamente, una NFT è composta da più di una singola caratteristica, quindi ci possono essere combinazioni di più tratti meno frequentemente di altre, rendendo il calcolo della rarità più difficile da calcolare.
La rarità viene generalmente calcolata attraverso l'uso di applicativi online nati con lo specifico scopo di attribuire un valore di rarità alle singole componenti di una collezione NFT.
Questi applicativi rendono, ovviamente, il calcolo della rarità molto più semplice e alla portata di utenti meno esperti nell'ambito.
Come ultima considerazione riguardo la rarità, si vuole riportare l'influenza che essa ha nella compravendita delle NFT, e in che modo si può rendere equiprobabile il ricevere uno qualsiasi degli NFT.
La rarità influenza ovviamente il prezzo di compravendita delle NFT. Come naturale che sia, NFT più rare avranno un valore maggiore rispetto a NFT con tratti, o combinazione di essi, più comuni. Per ovviare a questo problema si è introdotto il concetto di revealing, ovvero il rivelare il contenuto delle NFT solo dopo che tutte, o una gran parte di esse, è stata mintata dagli utenti (vedremo il concetto di minting più avanti nella tesi). In questo modo, tutti gli utenti che sono interessati a comprare un NFT della collezione, non sanno cosa hanno ottenuto fino al giorno della rivelazione di tuti gli NFT. Solitamente, per attrarre l'attenzione degli utenti, si imposta una immagine place holder, ovvero un segna posto, che raffigura in maniera vaga e non dettagliata la natura delle singole NFT.

3.1.2 Generazione dei metadati

Parliamo ora in maniera più dettagliata dei metadati, concetto anch'esso introdotto precedentemente nel capitolo 2.
Come affermato, i valori dei metadati devono rispecchiare ciò che effettivamente compone l'immagine. Se per esempio il nostro NFT ha i capelli viola, gli occhi arancio e le orecchie con un piercing, il rispettivo file dei metadati dovrà avere in corrispondenza dei campi capelli, occhi e orecchie i valori viola, arancio e piercing. Solo in questo modo, il file che descrive i metadati è coerente con ciò che è rappresentato nell'immagine.
I metadati vengono quindi generati assieme alle immagini, in modo da configurarli con i valori giusti e sono rappresentati con la formattazione json[1] in modo che un negozio online, o market place, come Opensea possa interpretare i dati di un NFT correttamente.
I metadati sono, dunque, fondamentali e senza di essi molte caratteristiche tipiche degli NFT non possono esistere. Durante la fase di generazione, il file .json associato viene aggiornato ogni qual volta che un tratto di un layer viene scelto, così che non ci siano errori o ambiguità all'interno del file. Nella seguente figura (Fig. 3.1) riportiamo un esempio di struttura di un file json relativo a un NFT generato. Nell'esempio riportato, si può vedere come caratteristiche tipo name, description,

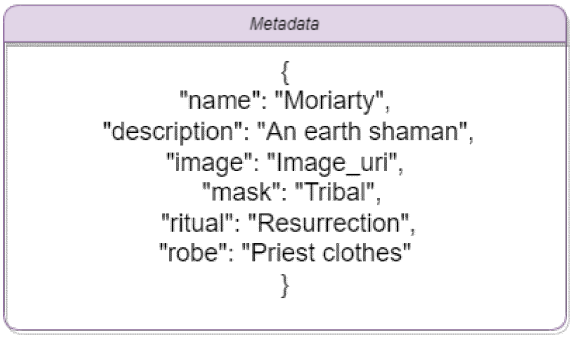

Figura 3.1: Immagine raffigurante un esempio di file json con i relativi metadati.

mask, ritual e robe abbiano ognuno dei valori specifici, come Moriarty o Tribal. Il campo "image" viene aggiornato successivamente all'upload delle immagini su IPFS[2], che vedremo più avanti nella parte di codice.
Questa è solo una delle possibili configurazioni che possono essere presenti all'interno dei metadati, in quanto anche un solo valore differente è accettabile, rendendo così unico nel suo genere uno specifico NFT.
I metadati generati così, verranno successivamente interpretati dal market place, nel nostro caso Opensea, più precisamente la sua rete di test, e grazie al campo "image", saprà quale immagine mostrare, mentre nel campo description verranno riportati i vari valori degli altri campi. In fine, a

[1] Il json è uno standard dedito alla rappresentazione testuale di strutture dati.
[2] IPFS, o interplanetary file system, è una rete di nodi alla pari, decentralizzata, dove è possibile memorizzare dati in modo che una risorsa sia sempre disponibile in ogni luogo ed in ogni momento.

destra dell'immagine verrà riportato il nome del NFT in questione.

3.2 GenerativeArt.py

In questa sezione vedremo in maniera più dettagliata il codice python implementato sotto forma di pseudo-codice, in particolare verranno elencati i metodi che lo compongono e le loro funzionalità.

3.2.1 Il metodo NFT_generator()

Questo è il metodo principale dedito alla generazione delle immagini e dei metadati degli NFT. Viene riportato di seguito lo pseudocodice e successivamente verrà spiegato il suo funzionamento.

while $num\_NFT < tot\_NFT$ **do**
  $iterations+ = 1$
  $metadata = sample$
  $metadata\_file\_name = path$
  $randomness()$
  if $isUnique(pattern)$ **then**
    $save(pattern)$
  else
    $ritorna\ all'inizio\ del\ ciclo$
  end if
  $imposta\ il\ layer\ di\ base$
  $aggiungi\ gli\ altri\ layer$
  $image\_generator()$
  $aggiorna\ la\ descrizione\ del\ file\ json$
  $ritorna\ i\ metadati$
end while

Questo codice crea NFT fintanto che non si arriva al numero desiderato. A ogni iterazione, viene aumentato un contatore, che verrà poi utilizzato alla fine dell'esecuzione totale del programma per indicare il numero d'iterazioni necessarie per generare tutti gli NFT richiesti. La variabile metadata contiene il template json che verrà riempito successivamente assieme alla fusione dei layer attraverso il metodo image_generator(). La variabile metadata_file_name memorizzerà successivamente il path dove i metadati verranno salvati e il loro relativo nome. Successivamente viene evocata la funzione randomness(), che vedremo in seguito nello specifico, e che si occupa della scelta casuale degli elementi di ogni layer, e salva temporaneamente il pattern appena creato. Se il pattern creato non è univoco, ovvero è stato creato precedentemente, si torna all'inizio del ciclo, e si cerca un

Figura 3.2: Layer 0

Figura 3.3: Layer 1

Figura 3.4: Layer 2

Figura 3.5: Layer 3

Figura 3.6: Layer 4

nuovo pattern, altrimenti viene memorizzato per confronti futuri. Infine vengono impostati il layer di base e tutti gli altri layer, che verranno presi in carico dalla funzione image_generator() che si occupa della generazione delle immagini, così come della creazione dei metadati.
La situazione temporanea che si viene a creare all'interno della variabile layer è la seguente riportata nelle figure 3.2, 3.3, 3.4, 3.5 e 3.6: Questa situazione è solamente una delle tante possibili che l'algoritmo può generare. Per una migliore comprensione di cosa sta effettivamente succedendo all'interno dell'algoritmo vediamo i metodi randomness() e image_generator().

3.2.2 Il metodo randomness()

Vediamo ora in dettaglio il metodo randomness() che si occupa della scelta casuale degli elementi per ogni layer.
Il metodo randomness() esegue una serie di processi per ogni singolo layer che deve comporre l'immagine. Viene riportato di seguito lo pseudocodice del metodo:

for $each\ layer$ **do**
$rando\ =\ random.randint(1, nElementsInLayer)$
$randoms.append(rand)$
$metadata\_field\ =\ get\_metadatafield()$
$metadata[metadata\_field]\ =\ list\_of\_values[rand]$
end for

Questo metodo sceglie casualmente uno degli elementi per ogni layer, e aggiorna i campi dei metadati relativi a essi. L'array randoms memorizza i valori generati casualmente che corrisponderanno alla corrispettiva immagine all'interno del set. Si può notare il metodo get_metadatafield(), il quale scopo è quello di ottenere il valore del campo dei metadati che si vuole aggiornare correlato al layer che stiamo considerando nella specifica iterazione.
Come ultimo passaggio del metodo, viene aggiornato il campo dei metadati con il rispettivo valore, che corrisponde al valore in posizione "rand" dell'array list_of_values, array che contiene tutti i valori possibili per uno specifico livello.
In versioni successive del codice è stata introdotta una forzatura all'interno della scelta dei para-

metri. Si sono aggiunti dei pesi ai vari elementi, in modo che alcuni elementi vengano scelti più frequentemente di altri. Così facendo, si può controllare il tratto o i tratti che si vogliono rendere più o meno rari. Lo pseudocodice modificato è stato riportato nell'appendice della tesi(vedere A.1).

3.2.3 Il metodo image_generator()

In questo paragrafo vediamo il metodo che unisce gli elementi di ogni layer per comporre l'immagine finale. Riportiamo ora lo pseudo-codice del metodo e conseguentemente la spiegazione di esso.

$baseImage = Image.open(Layer[0])$
for $eachLinLayer$ **do**
  $temp = Image.open(L)$
  $baseImage.paste(temp)$
end for
$baseImage.save(Path)$

baseImage è la variabile dedita a salvare l'immagine di base in modo che successivamente si possano fondere gli elementi dei livelli successivi su di essa. Il ciclo successivo si occupa d'incollare gli elementi sull'immagine di base. Alla fine di ogni iterazione l'immagine finale viene salvata nel percorso scelto.
È stata usata la libreria image di Python per consentire di aprire le immagini correttamente, ed è stata usata la libreria Pillow per manipolare le immagini.
Di seguito viene riportato il risultato finale (vedi Fig 3.7) dopo l'esecuzione del metodo image_generator().

Figura 3.7: Immagine finale dopo l'esecuzione di image_generator()

Capitolo 4

NFTContract e IPFS

In questo capitolo vediamo lo smart contract realizzato usando lo standard ERC721 e l'aggiornamento dei metadati attraverso l'upload su IPFS. Viene fornita una spiegazione su cosa è l'ipfs e sul suo utilizzo nel progetto.

4.1 Cosa è l'IPFS

"L'ipfs è un sistema distribuito per memorizzare e accedere a file, siti web, applicazioni e dati" [11]. Grazie a questa tecnologia è possibile accedere a risorse, che geograficamente potrebbero trovarsi altrove, molto più efficientemente e rapidamente. Di fatto, una risorsa può trovarsi in uno qualsiasi dei nodi che compongono la rete. IPFS è una rete decentralizzata di nodi alla pari dove ogni risorsa è condivisa egualmente tra essi.
Come metodo d'indirizzamento, al posto di disporre di un indirizzo dove cercare il contenuto, IPFS valuta di più il contenuto. Di fatto, viene generato un hash del contenuto della risorsa caricata sulla rete, che verrà utilizzato per comporre l'uri per accedere alla risorsa.
L'upload su IPFS è stato utilizzato per caricare le immagini, e i metadati realtivi a esso, in modo da ottenere poi l'uri che il market place Opensea può utilizzare. I metadati, prima dell'upload su IPFS vengono aggiornati con gli uri delle immagini relativi a esso.
Così facendo, la risorsa una volta immessa nella rete, sarà sempre disponibile anche se il nodo proprietario dovesse smettere di funzionare. Questo accade perché essendo IPFS una rete decentralizzata di nodi alla pari, e più nello specifico un file system interplanetario (da cui la sigla IPFS), permette di distribuire una risorsa sull'intera rete, in modo che un qualunque nodo ne possa usufruire o distribuire agli altri utenti.
Più in generale, si è caricato sulla rete IPFS una cartella contenente tutte le immagini generate, in modo da disporre di un hash unico per accedere alle immagini molto più efficientemente. Disponendo di un uri base uguale per tutti, risulta più facile per i software del calcolo della rarità a trovare i valori effettivi e le ricorrenze delle caratteristiche. Un uri di una cartella su ipfs ha una sintassi del

tipo *"https://ipfs.io/ipfs/hash_cartella"*, mente una qualunque delle n immagini che compongono la collezione avrà un uri così strutturato: *"https://ipfs.io/ipfs/hash_cartella/n.png"*.

4.2 MetadataUpdate.py

Il programma MetadataUpdate è stato realizzato per aggiornare il campo image dei metadati dopo che è stato effettuato l'upload su IPFS della cartella contenete tutte le immagini generate attraverso l'uso del programma GenerativeArt.py.
Questo programma compone l'uri necessario a Opensea per interpretare i metadati correlati a una determinata immagine.
L'uri dell'immagine viene calcolato attraverso l'hash della cartella e l'id dell'immagine considerata. L'uri finito è nella stessa forma vista nel paragrafo precedente, e una volta calcolato viene immesso all'interno del campo image dei metadati. In figura 4.1 è mostrata la situazione finale dei metadati di una delle immagini all'interno della collezione.

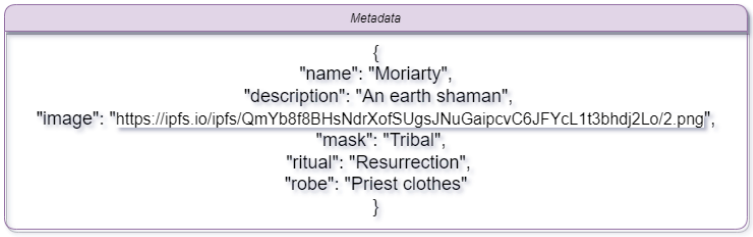

Figura 4.1: Campo image dei metadati aggiornati con il rispettivo uri.

Dopo l'aggiornamento dei metadati, tutti i file json all'interno della cartella metadata vengono caricati su rete IPFS, e l'hash che viene ricevuto come risposta viene memorizzato in modo che poi possa essere generato l'uri necessario allo smart contract. L'uri utilizzato all'interno smart contract verrà usato successivamente da Opensea per leggere i metadati e trovare l'immagine associato a esso.
Finita l'esecuzione di metadataUpdate.py si ha tutto il necessario affinché lo smart contract possa funzionare correttamente e le nostre immagini diventare NFT.
L'aggiornamento dei metadati risulta, quindi, fondamentale affinché lo smart contract possa comunicare correttamente i dati al market place. Analogamente, l'ordine di caricamento del materiale su IPFS è molto importante, in quanto se si caricassero i metadati contemporaneamente assieme alle immagini, o addirittura prima, si avrebbero degli uri che portano a dei metadati privi d'informazioni vitali. Queste informazioni non calcolate darebbero vita a degli NFT con delle caratteristiche associate a nessuna immagine, ovvero a dei semplici metadati privi di significato.
Aggiornare i metadati dopo che sono stati caricati sulla rete IPFS risulterebbe impossibile data la sua natura. Soprattutto, dato che cambiando anche di un singolo bit il contenuto dei metadati, l'hash ri-

sultante sarebbe completamente diverso. Questo accade perché, come detto precedentemente, IPFS è una rete che cerca risorse in base al loro contenuto e non al loro indirizzo.

4.3 NFTContract.sol

In questo paragrafo parliamo della struttura e del funzionamento dello smart contract NFTContract realizzato affinché la nostra collezione d'immagini generate precedentemente diventino ufficialmente degli NFT.
Il contratto eredita tutti i metodi dello standard ERC721 e del contratto Ownable. Quest'ultima tipologia di contratto non viene trattata, in quanto permette semplicemente al contratto di avere un proprietario. Permette, dunque, di gestire le autenticazioni e le autorizzazioni necessarie.

4.3.1 Variabili

Il contratto presenta le seguenti variabili:

- uriPrefix: permette d'impostare l'uri di base del gruppo di collezionabili.
- uriSuffix: impostato automaticamente con l'estensione ".json", in quanto il contratto si occuperà di trattare i metadati.
- hiddenMetadataUri: contiene l'uri dei metadati relativi all'immagine di base che verrà mostrata prima che si faccia il reveal degli NFT. Il reveal verrà trattato alla fine di questo capitolo, in modo da avere chiara gli stati del contratto.
- cost: contiene il costo in Ether per coniare un singolo NFT. Nel nostro caso parte da un valore di 0.01 Ether.
- maxSupply: permette di conoscere il numero massimo di collezionabili che è possibile rendere NFT. Dato che lo scopo del progetto è di creare 500 NFT unici, si è impostato masSupply a questo valore.
- maxMintAmountPerTx: indica il numero di NFT che è possibile coniare con un'unica transazione. Ha un valore di base di 5, ma può essere modificato.
- paused: variabile che indica lo stato del contratto. Se risulta vera non è possibile coniare nuovi NFT. È impostata alla creazione del contratto su vero.
- revealed: variabile che indica se gli NFT sono stati rivelati o meno. È impostata alla creazione del contratto su falso.

Come è possibile notare, risultano fondamentali le variabili uriPrefix e maxSupply, in quanto governano il funzionamento centrale del contratto. Senza la variabile uriPrefix si avrebbero degli NFT vuoti, ovvero privi di metadati e delle immagini che li raffigurano. D'altro canto, senza la variabile maxSupply si potrebbero coniare più NFT di quanti metadati ci siano, arrivando dopo molti mint a trovarsi in una situazione come la precedente.

4.3.2 Metodi

In questo paragrafo vediamo i metodi che compongono lo smart contract. Alcuni dei metodi riportati sono ereditati dallo standard ERC721 mentre altri sono stati implementati da me. I contratti necessari per l'ereditarietà sono stati presi dal repository online di Openzeppelin su un file definito come *libraries*. Successivamente è stato importato il file *libraries* nel codice dello smart contract. I metodi che compongono lo smart contract sono:

- constructor() ERC721(name, symbol): questo metodo è il costruttore del contratto ed eredita il costruttore dello standard ERC721. Accetta due parametri, name e symbol che corrispondono al nome della collezione NFT e al simbolo associato. All'interno di questo metodo viene impostato manualmente l'hiddenMetadataUri, in modo che i singoli token abbiano tutti la stessa immagine di copertina.

- totalSupply(): metodo ereditato. Fornisce il numero di token coniati fino a quel momento.

- mint(mintAmount): funzione che permette di mintare, ovvero coniare, un numero di NFT pari a mintAmount. Richiede che la variabile paused sia falsa, altrimenti non è possibile mintare gli NFT. Richiede, anche, che il prezzo che l'utente ha intenzione di pagare deve essere maggiore o uguale al costo fissato per mintAmount. Se le condizioni sono soddisfatte, viene chiamato il metodo _mintLoop che permette di coniare il numero desiderato di NFT. Un ulteriore controllo sul mintAmount viene effettuato da una porzione di codice che in solidity prende il nome di modifier. Questo modifier controlla che mintAmount sia un numero più grande di zero e minore o uguale al limite imposto da maxMintPerTx. Controlla, inoltre, che il numero di NFT coniati sommato al mintAmount non superi il valore di maxSupply. Se una di queste due condizioni è soddisfatta, il metodo che utilizza il modifier lancia un messaggio di errore.

- mintForAddress(mintAmount, receiver): permette di coniare degli NFT per uno specifico indirizzo[1]. Questo metodo può essere eseguito solamente dal proprietario del contratto e chiama successivamente il metodo _mintLoop.

[1]Un indirizzo è una stringa alfanumerica che indica univocamente un portafoglio elettronico sulla blockchain Ethereum e Polygon.

- walletOfOwner(owner): questo metodo permette di controllare il portafoglio di uno specifico indirizzo. Restituisce un array contenente gli id dei token che l'indirizzo owner possiede.
- tokenURI(id): questo metodo richiede che il token specificato da id sia stato già coniato. Se non è così, restituisce un messaggio di errore. Controlla successivamente se il valore di revealed è vero, ovvero se gli NFT sono stati rivelati. Se così non fosse, verrebbe restituito il valore di hiddenMetadataUri. Infine, se tutti i controlli sono stati superati, viene restituito un uri composto dalla concatenazione di uriPrefix, id e uriSuffix.
- _mintLoop(receiver, mintAmount): Esegue un ciclo di minting fintantoché non vengono coniati tutti gli NFT richiesti attraverso mintAmount. Gli NFT coniati appartengono all'indirizzo receiver. A ogni iterazione del ciclo viene incrementato il valore di supply, in modo che si tenga la variabile sempre in uno stato coerente.

Successivamente a questi metodi si trovano tutti i metodi dediti alla modifica e alla restituzione delle variabili di esemplare del contratto. Ovviamente tutti i metodi che modificano lo stato del contratto possono essere eseguiti solamente dal proprietario, mentre quelli dediti alla semplice visione del valore di queste ultime può essere eseguito da chiunque.
I metodi non riportati sono stati inseriti all'interno nella sezione A.2 dell'appendice.

4.3.3 Revealing del contratto

Il seguente paragrafo è dedicato al concetto di revealing del contratto. Si è deciso d'inserirlo in questo capitolo piuttosto che nel capitolo 2 in quanto è un concetto non appartenente allo standard ERC721.
Il revealing di un contratto deriva dalla rarità. Quando si vogliono vendere le proprie NFT al pubblico, ovviamente, non si vuole che vengano vendute solo quelle con rarità maggiore.
Per ovviare a questo problema, è stato introdotto il concetto di revealing, ovvero di rivelare quali NFT sono stati coniati solamente dopo che una certa quantità, o tutti, sono stati ottenuti tramite minting da altri utenti.
Basta pensare al revealing come a un pacchetto di carte collezionabili. Prima di sapere cosa è contenuto all'interno della bustina di carte, l'utente vede solo ed esclusivamente il design riportato sulla facciata frontale della bustina. Il revealing funziona in modo analogo, si sceglie un'immagine segna posto che viene mostrata agli utenti quando coniano degli NFT. Solamente dopo, il proprietario del contratto rivelerà il contenuto degli NFT agli acquirenti.
Così facendo, le NFT più rare non possono essere coniate esclusivamente trascurando tutte le altre.
Riportiamo ora, in figura 4.2, un esempio di segna posto che può essere utilizzato per NFT che hanno un design come quello illustrato nel capitolo 3.
Il place holder, ossia il segna posto, deve avere un significato. Ovvero deve essere in linea con gli NFT che si vogliono vendere. Non è buona norma mettere un design che non ha un nesso logico

Figura 4.2: Esempio di place holder NFT.

con ciò che si vuole proporre agli utenti della rete.
Nel nostro caso, si è usato semplicemente lo stesso design utilizzato per il Layer 1 per la parte di generative art. Come detto precedentemente, si sarebbe potuto utilizzare anche un design alternativo, ma pur sempre inerente al prodotto finale. Si è optato per questa scelta per semplicità nell'illustrare il concetto.

Capitolo 5

Automatismo di minting

In questo capitolo parliamo del framework Brownie che permette di scrivere codice python in grado di gestire gli smart contract. Verranno successivamente descritti i vari pezzi di codice sorgenti che permettono di automatizzare il processo di minting.

5.1 Il framework Brownie

Il framework Brownie permette di testare sulla Ethereum Virtual Machine gli smart contract attraverso uno sviluppo basato sul linguaggio Python.
Brownie permette di scrivere all'interno di esso smart contract, interfacce per contratti, script e casi di test in python. I casi di test verranno discussi in maniera dettagliata all'interno del capitolo 6.
Grazie alla sua natura, Brownie permette di sviluppare parallelamente smart contract e script python dediti alla gestione e al comportamento di quest'ultimi. Offre compatibilità con il linguaggio Vyper e dispone di una console nativa che permette interazioni rapide con il progetto.
Mette a disposizione molte reti sulla quale è possibile lavorare, tra le quali Ethereum mainnet, con le relative reti di test, Polygon e Mumbai. Sfortunatamente, la maggioranza di queste reti richiede l'uso di un applicativo esterno, ovvero Infuria, della quale non tratteremo. Per ovviare a questo problema, si è inserita manualmente la rete di test Ploygon così che è stato possibile lavorare senza riscontrare difficoltà.
Per utilizzare al meglio Brownie occorre impostare le variabili di ambiente, in particolar modo la chiave privata[1] del portafoglio elettronico che si vuole usare. Per fare ciò occorre semplicemente eseguire il comando "export PRIVATE_KEY = my_private_key" all'interno del framework. Dopo questa operazione è possibile inviare contratti sulla rete blockchain a partire dal nostro portafoglio elettronico.
Nel caso non si voglia utilizzare un proprio account, data la dipendenza dal software ganache (soft-

[1] La chiave privata è una sequenza alfanumerica che identifica univocamente un account all'interno della blockchain. Deve essere conosciuta solo al proprietario del portafoglio e permette di spostare e/o spendere i token presenti in esso.

ware che permette di simulare una piccola blockchain a livello locale), Brownie mette a disposizione una serie di 5 account di prova che l'utente può utilizzare a proprio piacimento.
Affinché il framework funzioni correttamente, bisogna impostare un file di configurazione, in modo tale che tutto funzioni correttamente. In particolar modo, bisogna impostare il campo wallets specificando la chiave privata utilizzata all'interno delle variabili di ambiente e bisogna impostare i parametri necessari affinché si utilizzi correttamente la rete desiderata.
Bisogna dunque configurare il campo networks specificando i valori di caratteristica della rete Polygon di test, valori che riportiamo in Fig 5.1. Vediamo ora in figura 5.2 un esempio di file di

```
vrf_coordinator: '0x8C7382F9D8f56b33781fE506E897a4F1e2d17255'
link_token: '0x326C977E6efc84E512bB9C30f76E30c160eD06FB'
keyhash: '0x6e75b569a01ef56d18cab6a8e71e6600d6ce853834d4a5748b720d06f878b3a4'
```

Figura 5.1: Campi e valori specifici della rete Polygon di test.

configurazione completo, in modo da avere chiara la sua struttura.

```
wallets:
  from_key: ${PRIVATE_KEY}
networks:
  mumbaiTesnnet:
    vrf_coordinator: '0x8C7382F9D8f56b33781fE506E897a4F1e2d17255'
    link_token: '0x326C977E6efc84E512bB9C30f76E30c160eD06FB'
    keyhash: '0x6e75b569a01ef56d18cab6a8e71e6600d6ce853834d4a5748b720d06f878b3a4'
```

Figura 5.2: Esempio di file di configurazione completo

5.2 Deploy.py

Parliamo ora dello script che si occupa del deploy del contratto NFTContract. Riportiamo prima lo pseudo codice dl contratto per poi commentarlo:

$$dev = wallet\_used\_for\_deploy$$
$$contract = NFTContract.deploy()$$
$$contract.setPaused(False)$$
$$contract.setRevealed(True)$$
$$contract.setMaxSupply(500)$$

Lo script viene riportato in maniera dettagliata all'interno dell'appendice nella sezione A.3.
Il primo passaggio effettuato è memorizzare l'indirizzo dell'account che si ha intenzione di utilizzare per interagire con il contratto. Nel nostro caso, useremo il wallet[2] corrispondente alla chiave

[2]Un wallet, o portafoglio, attraverso l'uso di chiave pubblica e privata, permette di memorizzare token fungibili e non. Permette di effettuare e ricevere transazioni. Può essere un supporto fisico o un applicativo software.

privata impostata nel paragrafo precedente.
Successivamente, viene memorizzato il contratto NFTContract che viene immesso sulla rete attraverso il metodo deploy. Una volta memorizzato il contratto all'interno della variabile contract possiamo utilizzare i metodi del contratto che più ci risultano utili.
I principali metodi che vengono chiamati sono setPaused(False), che ci permette di cambiare lo stato del contratto. Dopo l'esecuzione di questo metodo risulterà possibile coniare gli NFT. Per motivi di semplicità, si è preferito inserire questo comando all'interno del contratto di deploy.
Successivamente si esegue in maniera analoga il metodo setREvealed(True), facendo così risultare gli NFT rivelati subito dal momento del minting.
Come ultimo metodo, viene eseguito setMaxSupply(500), che ci permette d'impostare il numero massimo di NFT che possono essere coniati. Nello pseudo-codice viene riportato esplicitamente il numero 500 solo a fini didattici. Nel codice completo, viene passato un argomento che indica il numero massimo di NFT, numero che l'utente può impostare prima del deploy del contratto stesso.
È stato omesso un parametro all'interno di tutti i metodi chiamati, incluso il metodo deploy del contratto. Il parametro in questione è "from":dev, che specifica il wallet che il framework deve utilizzare per eseguire i metodi. Nel nostro caso, il wallet è wallet_used_for_deploy.
Risulta necessario utilizzare un wallet in quanto tutti i metodi che cambiano lo stato del contratto, deploy incluso, richiedono un pagamento da parte dell'utente che opera. Questo vale per una qualsiasi transazione all'interno di qualunque blockchain. È la tassa che si paga per effettuare una transazione.

5.3 Mint.py

Vediamo ora lo script dedito al mint delle NFT. Questo script permette di coniare un numero arbitrario di NFT, numero che viene fornito in standard input al momento dell'esecuzione. Vediamo ora lo pseudo-codice relativo allo script. Verrà riportato nella sezione A.4 dell'appendice il codice completo.

$dev = wallet\_used\_for\_deploy$
$contract = last_deployed_contract$
$s = int(input())$
for $i\ in\ range\ (s)$ **do**
  $contract.mint(1)$
  $sleep(60)$
end for

Come si può notare, viene coniato solamente una NFT per iterazione. Si è adottato questo approccio per gestire meglio errori ed eccezioni lanciati in fase di esecuzione.
La variabile contract contiene l'ultima istanza del contratto NFTContract che è stata immessa nella

rete, in modo che le NFT mintate siano coerenti con il contratto a cui appartengono.
È stato fissato anche un tempo di riposo tra i vari mint, di un minuto, in modo da fornire il tempo necessario affinché la transazione venga processata.
Per una gestione più completa degli errori, è stato inserito un blocco try-except[3] in modo da notificare all'utente finale il caso limite nella quale tutti gli sono stati già coniati. Si è inserito anche un ulteriore controllo sulla natura del numero immesso. Questo in quanto numeri negativi non possono essere accettati. Quindi, se il numero fornito è negativo, o una stringa al posto di un numero, viene lanciato un messaggio di errore e si interrompe l'esecuzione dello script.
Un tratto distintivo dei parametri passati nella funzione di mint all'interno dello script python è la presenza del campo "value" seguito da un valore espresso in wei. Più precisamente, quando viene fornito l'account dalla quale si vuole coniare un nuovo NFT, viene specificato anche la quantità di denaro che l'utente è disposto a pagare.
Nel nostro caso, questo valore equivale a $50 * 10^{15}$ wei, o più semplicemente a 0.05 Ether, nel nostro caso Matic. Questo valore di 0.05 Matic deriva dal semplice prodotto tra il valore della variabile cost, espressa in Ether, moltiplicata per il numero di NFT che possono essere coniati per singola transazione.
Il pagamento di una determinata somma deriva dalla natura del metodo mint. Questo metodo risulta essere payable, ovvero richiede un prezzo da pagare per la sua esecuzione oltre alle sole tasse di transazione.
Dopo l'esecuzione di questo script, se le transazioni sono andate a buon fine, è possibile verificare la presenza degli NFT all'interno del market place Opensea, più precisamente nell'aria dedicato alle reti di test.

5.4 setUri.py

Vediamo ora lo script dedito alla configurazione dell'uri che permetterà al market place di accedere ai metadati e all'immagine relativi al corrispettivo token.
È importante che prima dell'esecuzione dello script, le immagini siano state caricate su IPFS, i metadati aggiornati attraverso lo script metadataUpdate.py, i metadati caricati si IPFS e l'uri di base composto dall'hash della cartella dei metadati creato.
Una volta soddisfatte le condizioni qui sopra riportate, lo script può essere eseguito senza problemi. Come per gli altri script, il codice completo è stato riportato all'interno dell'appendice. Questo in particolare all'interno della sezione A.5.
Vediamo ora lo pseudo codice relativo a questo script:

$$dev = wallet\_used\_for\_deploy$$

[3]Il blocco try-except è un costrutto dedito alla gestine di eccezioni all'interno di una porzione di codice. Il programma prova a eseguire il codice presente all'interno del blocco try, se questo codice lancia una eccezione, viene eseguito il codice all'interno del blocco except. Così facendo, il programma continua ad essere eseguito.

$contract = last_deployed_contract$
$uriPrefix = "https://ipfs.io.ipfs/hash/"$
$contract.setUriPrefix(uriPrefix)$

Conseguentemente all'esecuzione di questo script, si ha a disposizione una collezione di NFT perfettamente visibili e completi.
Nell'ultimo paragrafo di questo capitolo vene riportato lo stato finale degli NFT e come è possibile visualizzarli all'interno del negozio online Opensea.
Come per gli altri metodi, anche setUriPrefix richiede un parametro aggiuntivo che corrisponde a dev, ovvero il wallet dalla quale bisogna far partire le transazioni.
Ovviamente, se si dovesse cambiare in un qualsiasi momento l'uri delle NFT si andrebbe in contro a problemi, come la scomparsa delle immagini e dei dati relativi all'NFT, o, in situazioni meno fortunate, al visualizzare immagini e metadati relativi ad altri NFT.

5.5 Script Utili

Parliamo ora di altri due script python che sono stati realizzati col fine di rendere l'interazione con il contratto più intuitiva. Questi due script non sono di vitale importanza per l'esecuzione del progetto, e sono stati realizzati principalmente con fini di test.
I due script sono:

- reveal.py: realizzato con lo scopo di rivelare le NFT in un momento diverso rispetto al deploy dello smart contract. Presenta del codice molto semplice, codice che viene riportato per intero qui di seguito.

  $dev = accounts.add(config['wallets']['from\_key'])$
  $contract = NFTContract[len(NFTContract) - 1]$
  $contract.setRevealed(True, \{"from" : dev\})$

  Come si può notare, le prime due righe di codice sono le stesse usate in tutti gli altri script. Hanno lo scopo di definire il portafoglio dalla quale si vuole effettuare la transazione e l'istanza del contratto che si vuole utilizzare.
  L'ultima riga permette di effettuare il revealing del contratto come programmato. Come anticipato, attraverso questo contratto si può effettuare il revealing in un qualsiasi momento diverso dal deploy del contratto. Così facendo, si ha il controllo sul quando rivelare le NFT agli utenti che ne hanno mintati alcuni.
  In un ambiente di sviluppo reale, il revealing deve essere gestito separatamente dal deploy del contratto. Attraverso questa idea, si è deciso di implementare uno script dedito solo al revealing.

- tokenURI.py: realizzato con lo scopo di ritornare l'uri di uno specifico token che è stato coniato. Nonostante tutti i metadati sono contenuti all'interno della stessa cartella, è stato implementato comunque un metodo che restituisce l'uri dei singoli NFT. Il tutto è dovuto dal fatto che lo standard ERC721 prevede un metodo che ritorni l'uri delle singole NFT. Riportiamo ora il codice completo dello script:

  $dev = accounts.add(config['wallets']['from\_key'])$
  $contract = NFTContract[len(NFTContract) - 1]$
  $s = int(input())$
  $try:$
  $\quad print(contract.tokenURI(s, \{"from" : dev\}))$
  $except:$
  $\quad print("token\ not\ minted...")$

  Anche qui sono presenti le righe di codice riguardanti il wallet e l'ultima istanza del contratto. Come si può notare, viene richiesto all'utente un intero che permette di restituire l'uri del token la quale l'id è stato specificato.
  La chiamata al metodo tokenURI del contratto viene eseguito all'interno di un blocco try-except. Questo perché il metodo tokenURI può restituire solo l'uri dei token che sono stati coniati. Se si provasse a interrogare il contratto con un id di un token non ancora coniato, o addirittura con un id di un token che è maggiore del numero possibile di token coniabili, viene restituito un messaggio di errore da parte del contratto. Per questo motivo, per gestire l'errore, si è inserito la chiamata al metodo tokenURI all'interno del blocco try-except.

5.6 Visualizzazione degli NFT su Opensea

Vediamo ora come è possibile visualizzare gli NFT all'interno del market place Opensea.
Una volta che un NFT risulta essere coniato correttamente, esso appartiene a un indirizzo specifico. Nel caso della chiamata al metodo mint, il proprietario risulta essere l'indirizzo che ha effettuato l'esecuzione del metodo. Nel caso della chiamata al metodo mintForAddress, il proprietario degli NFT richiesti risulta essere l'indirizzo specificato all'interno.
L'indirizzo che possiede gli NFT, che siano 1 o molteplici, ha la possibilità di visualizzarli attraverso l'utilizzo di tool online, o market places, come Opensea.
Per procedere alla visualizzazione degli NFT all'interno del market place, occorre collegare il proprio wallet, ovvero il proprio account al sito.
Nel nostro caso useremo il wallet online chiamato MetaMask, una estensione per browser web, che offre tutte le funzionalità necessarie affinché sia possibile effettuare transazioni e possedere token sulla blockchain.
Una volta connesso il wallett con il market place, sarà possibile visualizzare le NFT che si possie-

dono, come mostrato nella figura 5.3. Come si può notare, subito dopo la coniazione del token, non

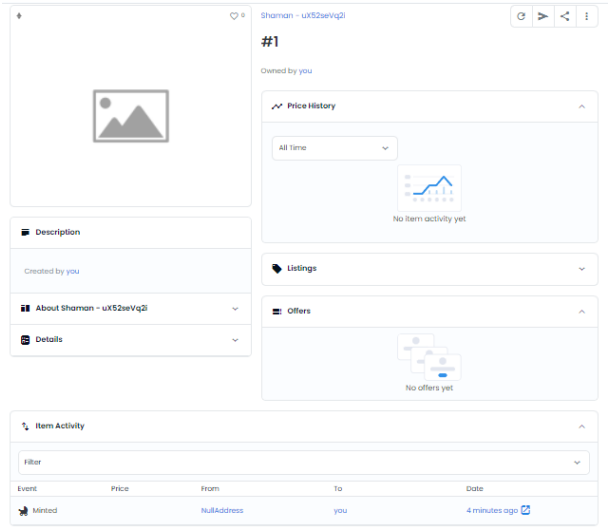

Figura 5.3: Schermata di Opensea di una NFT appena coniata

è presente l'immagine associata o i metadati. Questo accade in quanto c'è bisogno di un tempo di attesa per la quale la transazione viene finalizzata e il market place riesce a recepire e interpretare i metadati correttamente.
Aspettato un lasso temporale adeguato, solitamente intorno ai 20 minuti, è possibile ricaricare i metadati attraverso l'apposito pulsante presente su Opensea, e visualizzare correttamente immagine e metadati, come mostrato all'interno della figura 5.4.

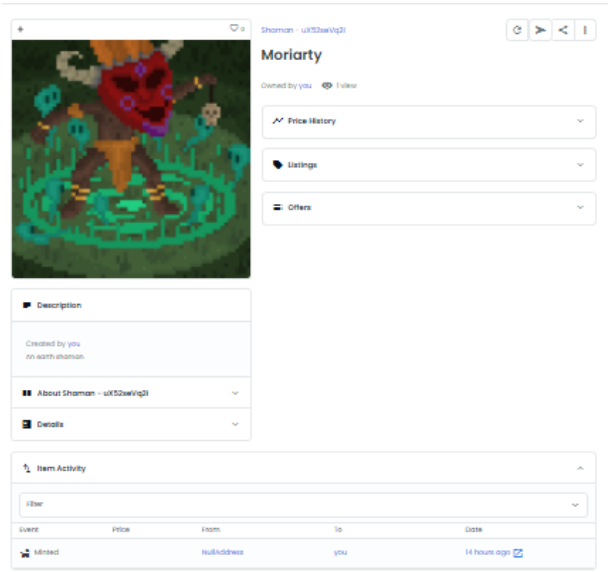

Figura 5.4: Schermata di Opensea di una NFT dopo un determinato intervallo di tempo

Come si può notare, il tempo di attesa affinché tutti i metadati e l'immagine del token fossero vi-

sibili è di 14 ore circa. Come anticipato, c'è bisogno di aspettare un certo lasso di tempo affinchè tutti i metadati possano essere visibili all'utente esterno. Questo tempo varia in base allo stato della rete e ad altri fattori esterni ad essa.

Capitolo 6

Testing

In questo capitolo vediamo i test che sono stati fatti attraverso l'utilizzo della libreria pytest del linguaggio python. I test sono stati eseguiti attraverso l'utilizzo del framework Brownie.
Sono stati creati 3 script di testing con scopi differenti, il primo permette di provare a eseguire tutti i metodi elencati nel contratto per verificarne la correttezza logica per dei casi di base. Il secondo è stato ideato per controllare i casi limite della funzione di mint, mentre il terzo, e ultimo script di test, è stato ideato per testare i vari casi limite sul prezzo per la quale la funzione di mint esegua correttamente il suo lavoro.

6.1 Test_1.py

Come anticipato nell'introduzione del capitolo, questo script di test permette di verificare la correttezza di tutti i singoli metodi del contratto NFTContract sviluppato nel capitolo 4.
Come prima cosa, ogni test deve appartenere a un modulo e deve contenere la parola test all'interno del nome proprio del file.
Successivamente bisogna definire una funzione che ritorna l'istanza del contratto a partire da un account di prova.
Come detto precedentemente, Brownie mette a disposizione cinque account di prova grazie alla sua dipendenza dal software ganache.
Una volta chiari questi punti, le prime righe di codice all'interno dello script, in generale all'interno di ogni script di test, sono le seguenti:

$@pytest.fixture(scope = \text{"}module\text{"})$
$def\ NFT() :$
    $return\ NFTContract.deploy(\{\text{"}from\text{"} : accounts[0]\})$

La prima riga definisce il module alla quale facciamo riferimento per tutti gli script di test e la seguente funzione crea un oggetto che fa riferimento al contratto, oggetto usabile dalle funzioni di test per eseguire i vari metodi del contratto.
Successivamente a questo primo blocco di codice, vengono definite tutte le altre funzioni dedite alla verifica del comportamento del contratto.
Un dettaglio importante è la presenza del nome test all'interno del nome di tutte le funzioni, la quale devono svolgere attività di verifica.
La presenza della stringa test all'interno del nome del file e delle funzioni è di vitale importanza, senza di essi il framework Brownie non riconoscerebbe le funzioni e quindi i test non verrebbero eseguiti.
Prima di presentare le funzioni all'interno del modulo, bisogna parlare della funzione assert, funzione che permette di verificare una eguaglianza tra due elementi.
Assert accetta due parametri all'interno di essa, e controlla che i due valori siano identici. Se la condizione è soddisfatta ritorna che il test effettuato è andato a buon fine, altrimenti restituisce che il test è fallito.
Passiamo ora a illustrare i vari casi di test di questo modulo:

- test_setPaused(NFT): il suo obbiettivo è testare lo stato del contratto. Viene controllato che il valore della variabile paused restituito dalla funzione isPaused è vero. Successivamente si usa il metodo setPaused per impostare lo stato del contratto a non in pausa, ovvero paused ha valore falso. Una volta eseguito il metodo, si controlla che il valore della variabile risulta ora falso. Vediamo ora il codice relativo a esso:

  $$def\ test\_setPaused(NFT)$$
  $$\quad assert\ NFT.isPaused()\ ==\ True$$
  $$\quad NFT.setPaused(False, \{\text{“}from\text{”} : acounts[0]\})$$
  $$\quad assert\ NFT.isPaused()\ ==\ False$$

  Come si può notare, per la verifica viene usato la funzione assert, funzione che viene fornita dalla libreria pytest del linguaggio python.

- test_name(NFT): il suo obbiettivo è verificare che il nome del token è stato impostato correttamente in fase di costruzione del contratto. Nel nostro caso, verifica che il nome è Shaman. Il codice è:

  $$def\ test\_name(NFT):$$
  $$\quad assert\ NFT.name()\ ==\ \text{“}Shaman\text{”}$$

- test_symbol(NFT): l'obbiettivo di questa funzione è controllare che il simbolo relativo al token sia stato impostato correttamente in fase di costruzione. Nel nostro caso, il simbolo deve essere SHMN. Vediamone ora il codice:

$def\ test\_symbol(NFT):$
$assert\ NFT.symbol()\ ==\ \text{“}SHMN\text{”}$

- test_balancing(NFT): questa funzione è stata scritta con lo scopo di verificare che il totalSupply() del contratto, ovvero il numero totale di NFT coniati, assume un valore coerente. Come primo passo, viene impostato il contratto come non in pausa, poi viene eseguita la funzione di mint per un singolo NFT. Infine, viene controllato che il valore di totalSupply() equivale ad 1. Il codice che esegue questo test è il seguente:

  $def\ test\_balancing(NFT):$
  $NFT.setPaused(False, \{\text{“}from\text{”} : acounts[0]\})$
  $NFT.mint(1, \{\text{“}from\text{”} : accounts[0], \text{”}value\text{”} : 50 * 10 * *15\})$
  $assert\ (NFT.totalSupply()\ ==\ 1)$

- test_walletOfOwner(NFT): in questa funzione di test viene verificato che il portafoglio di un account sia stato aggiornato correttamente dopo che viene chiamata la funzione di mint. Dato che la funzione di mint è stata già eseguita all'interno del test precedente, il codice diviene il seguente:

  $def\ test\_walletOfOwner(NFT):$
  $assert\ NFT.walletOfOwner(accounts[0])\ ==\ [1]$

  Come si può notare, viene controllato che il prtafoglio dell'account di test utilizzato sia uguale a una lista, o array, contenete un intero. L'intero, presente all'interno della lista, rappresenta l'identificatore del token. In questo caso, l'unico token che è stato coniato è quello con identificatore 1.

- test_tokenURI(NFT):il seguente test è stato scritto con l'obbiettivo di controllare che l'uri di base del token sia la stringa vuota. Viene utilizzato il metodo tokenURI(id) del contratto. Il codice è il seguente:

  $def\ test\_tokenURI(NFT):$
  $uri\ =\ NFT.tokenURI(1)$
  $assert\ uri\ ==\ \text{””}$

- test_reveal(NFT): con il seguente test, si è voluto controllare lo stato di revealing del contratto subito dopo il deploy e dopo averne cambiato il valore. Il valore atteso al primo controllo è il valore falso, in quanto è il valore di base impostato. Si è implementato il seguente codice:

  $def\ test\_reveal(NFT):$
  $assertNFT.isRevealed()\ ==\ False$
  $NFT.setRevealed(True, \{\text{”}from\text{”} : account[0]\})$
  $assertNFT.isRevealed()\ ==\ True$

Come si può notare, si è usato l'ausilio del metodo isRevealed del contratto. Così facendo è possibile accertarsi dell'incapsulamento delle variabili e dell'effettivo funzionamento del metodo.

- test_setCost(NFT): in questa funzione si vuole testare la funzione setCost(amount). Attraverso la verifica del valore della variabile cost, prima e dopo l'esecuzione del metodo setCost(), è possibile controllare che il valore base sia quello aspettato in fase di deploy del contratto, e che il valore sia stato modificato correttamente dal metodo. Il codice utilizzato a questo scopo è il seguente:

$$
\begin{aligned}
&def\ test\_setCost(NFT):\\
&\qquad assert NFT.cost() == 0.01 * 10 * *18\\
&\qquad NFT.setRCost(0.05 * 10 * *18, \{"from" : account[0]\})\\
&\qquad assert NFT.cost() == 0.05 * 10 * *18
\end{aligned}
$$

- test_setMaxMintAmountPerTx(NFT): attraverso questa funzione è possibile verificare il valore base, e il valore modificato attravero l'uso di setMaxMintPerTx(Amount), della variabile maxMintAmountPerTx. Il valore base che ci si aspetta è 5, mentre il valore dopo la modifica deve risultare 1. Il codice risulta essere il seguente:

$$
\begin{aligned}
&def\ test\_setMaxMintPerTx(NFT):\\
&\qquad assert\ NFT.maxMintPerTx() == 5\\
&\qquad NFT.setMaxMintPerTx(1, \{\text{“}from\text{”} : accounts[0]\})\\
&\qquad assert\ NFT.maxMintPerTx() == 1
\end{aligned}
$$

  La modifica di maxMintPerTx influenzerà anche i successivi casi di test, in quanto tutti i test in questo script fanno parte dello stesso modulo.

- test_setHiddenMetadataUri(NFT) : questo test ha lo scopo di verificare lo stato della variabile hiddenMetadataUri prima e dopo la sua modifica attraverso l'utilizzo del metodo setHiddenMetadataUri(uri). Il valore iniziale della variabile deve essere la stringa vuota, mentre il valore modificato deve risultare essere una stringa di prova. L'implementazione del test è la seguente:

$$
\begin{aligned}
&def\ test\_setHiddenMetadataUri(NFT):\\
&\qquad assert\ NFT.hiddenMetadataUri() == \text{””}\\
&\qquad NFT.setHiddenMetadataUri(\text{”}stringa\_di\_prova\text{”}, \{\text{“}from\text{”} : accounts[0]\})\\
&\qquad assert\ NFT.hiddenMetadataUri() == \text{”}stringa\_di\_prova\text{”}
\end{aligned}
$$

- test_setUriPrefix(NFT): così come il test precedente mirava a controllare lo stato della variabile hiddenMetadataUri, questo test ha lo scopo di controllare lo stato della variabile uriPrefix. La metodologia è del tutto analoga a quella del test precedente, con la differenza che si va

a modificare il valore della variabile uriPrefix. Dato che la natura di questo test è identica a quella del precedente, si è scelto di non riportare il codice di questo test.

- test_setUriSuffix(NFT): questo test, come i due precedenti, ha lo scopo di controllare lo stato della variabile uriSuffix, prima e dopo la sua modifica. Il valore base è ".json", per cui ci si aspetta che il suo valore corrisponda a quello prima della sua modifica. Successivamente si modifica il suo valore a una stringa del tipo "new_suffix" e si controlla che il valore sia stato aggiornato correttamente. Anche per questo test non è stato riportato il codice, dato che risutla essere del tutto identico al codice dei due test precedenti.

- test_allURI(NFT): in questo test si vuole controllare che l'uri finale, ovvero la combinazione di prefisso, id e suffisso, per ogni token, sia coerente con i valori impostati attraverso i metodi setUriPrefix e setUriSuffix. Questa condizione deve essere vera per tutti i token che sono stati precedentemente coniati. La sua implementazione risulta essere la seguente:

  $def\ test\_allURI(NFT):$
  for $i\ in\ range(1, NFT.totalSupply() + 1)$: **do**
  $assert\ NFT : tokenURI(i)\ ==\ f"new\_prefix\{i\}.new\_suffix"$
  end for

  dove new_prefix e new_suffix sono i valori attribuiti alle rispettive variabili nei test sull'uriSuffix e sull'uriPrefix.

- test_mintLoop(NFT): in questo test si vuole controllare che il ciclo di mint funzioni correttamente. In particolar modo, si vuole controllare che la variabile totalSupply(), il wallet dell'indirizzo e il bilancio dello stesso sono aggiornate correttamente. Successivamente, si vuole controllare che non sia possibile eseguire la funzione di mint che accetta un parametro maggiore di 1, modificare il numero massimo di NFT che possono essere coniati in una singola transazione e controllare nuovamente lo stato delle variabili dopo una chiamata a mint. Il codice che implementa questo test è il seguente:

  $temp\ =\ [1]$
  for $i\ in\ range\ (1, 15)$: **do**
  $temp.append(i + 1)$
  $NFT.mint(1, \{"from" : accounts[0], "value" : 50 * 10 * *15\}$
  $assert(NFT.totalSupply()\ ==\ i + 1)$
  $assert(NFT.walletOfOwner(accounts[0])\ ==\ temp)$
  $assert(NFT.valanceOf(accounts[0])\ ==\ i + 1)$
  end for
  try :
  $NFT.mint(3, \{"from" : accounts[0], "value" : 50 * 10 * *15\})$

$except:$
$print(\text{"except"})$
$assertFalse == False$
$NFT.setMaxMintPerTx(3, \{\text{"}from\text{"} : accounts[0]\})$
$NFT.mint(3, \{\text{"}from\text{"} : accounts[0], \text{"}value\text{"} : 50 * 10 * *15\})$
$temp.append(16)$
$temp.append(17)$
$temp.append(18)$
$assert(NFT.totalSupply() == 18)$
$assert(NFT.walletOfOwner(accounts[0]) == temp)$
$assert(NFT.valanceOf(accounts[0]) == 18)$

Come si può notare, si è utilizzato una lista come supporto al controllo del wallet dello specifico indirizzo. Ogni volta che viene coniato un nuovo NFT, deve essere aggiornato il wallet dell'indirizzo attraverso l'aggiunta dell'id del nuovo token all'interno del wallet stesso.
All'interno del blocco except è presente un assert molto banale, questo è stato oggetto di uso per verificare che se lanciata una eccezione, essa potesse essere gestita correttamente.

Questa è la lista completa delle funzioni scritte per l'esecuzione della prima serie di test. Come è possibile vedere in Fig 6.1, vengono superati tutti senza problemi.

```
============================================= test session starts ==============================================
platform linux -- Python 3.10.4, pytest-6.2.5, py-1.10.0, pluggy-1.0.0
rootdir: /mnt/c/Users/franc/Desktop/ArchLinux/Project/ERC721
plugins: eth-brownie-1.16.4, forked-1.3.0, hypothesis-6.21.6, xdist-1.34.0, web3-5.23.1
collected 14 items

Launching 'ganache-cli --accounts 10 --hardfork istanbul --gasLimit 12000000 --mnemonic brownie --port 8545'...

tests/test_1.py ..............                                                                    [100%]

============================================== 14 passed in 12.80s ==============================================
Terminating local RPC client...
```

Figura 6.1: Risultato del primo script di test.

6.2 Test_2.py

Vediamo ora in dettaglio il secondo script di test, che si occupa di testare in maniera più dettagliata la funzione di mint.
Così come nel primo file di script, anche qui sono presenti le tre linee di codice dedite alla definizione del modulo e all'istanza del contratto, per cui non verranno riportate nuovamente.

- test_alternateAccounts(NFT): in questa funzione si vogliono testare i metodi minte mintForAddress, verificando che il totalSupply(), i bilanci e i wallet degli account vengano aggiornati correttamente. All'interno della funzione si trovano le righe di codice per rivelare

il contratto e per rimuovere lo stato di pausa. Successivamente viene impostato il numero di mint per transazioni a 3. Tolte le linee di codice riguardanti la premessa (molti esempi sono stati fatti nel paragrafo precedente a riguardo), il codice della funzione diventa il seguente:

$def\ test\_alternateAccounts(NFT):$
$NFT.mint(3, \{"from" : accounts[0], "value" : 50 * 10 * *15\})$
$NFT.mintForAddress(3, accounts[1])$
$assert\ NFT.totalSupply() == 6$
$assert\ NFT.walletOfOwner(accounts[0]) == [1, 2, 3]$
$assert\ NFT.walletOfOwner(accounts[1]) == [4, 5, 6]$
$assert\ NFT.balanceOf(accounts[0]) == NFT.balanceOf(accounts[1])$
$assertNFT.balanceOf(accounts[0]) == 3$
$assertNFT.balanceOf(accounts[1]) == 3$

Come si può notare, è presente una riga che eguaglia i bilanci dei due portafogli. Questa specifica linea di test vuole dimostrare che entrambi i bilanci siano stati aggiornati correttamente e, dato che entrambi contengono tre elemente, devono risultare uguale.

- test_mintLoop(NFT): in questa funzione vengono coniati nuovi NFT per altri due indirizzi. Successivamente all'esecuzione del metodo mint, vengono verificati che tutti i bilanci, vecchi e nuovi, il totalSupply e i wallet siano in uno stato coerente. Come controllo finale, viene controllato che i bilanci di due wallet risultino diversi, in quanto un account possiede 2 token mentre l'altro 3. Il codice è il seguente:

$def\ test\_mintLoop(NFT):$
$NFT.mintForAddress(2, accounts[2])$
$NFT.mintForAddress(3, accounts[3])$
$assert\ NFT.totalSupply() == 11$
$assert\ NFT.walletOfOwner(accounts[0]) == [1, 2, 3]$
$assert\ NFT.walletOfOwner(accounts[1]) == [4, 5, 6]$
$assert\ NFT.walletOfOwner(accounts[2]) == [7, 8]$
$assert\ NFT.walletOfOwner(accounts[3]) == [9, 10, 11]$
$assert\ NFT.balanceOf(accounts[0]) == NFT.balanceOf(accounts[1])$
$assertNFT.balanceOf(accounts[0]) == 3$
$assertNFT.balanceOf(accounts[1]) == 3$
$assertNFT.balanceOf(accounts[2]) == 2$
$assertNFT.balanceOf(accounts[3]) == 3$
$assert\ NFT.balanceOf(accounts[2]) != NFT.balanceOf(accounts[3])$

- test_idOutOfIndex(NFT): attraverso l'uso di questa funzione di verifica, si vuole essere si-

curi che in caso si voglia ispezionare l'id di un token non ancora coniato, venga lanciata un'eccezione. Il codice è:

$def\ test\_test_IdOutOfIndex(NFT):$
$try:$
$\quad NFT.tokenURI(99, \{"from" : accounts[0]\})$
$except:$
$\quad print("except")$
$\quad assert\ True\ ==\ True$

L'ultima riga è un controllo "dummy" che permette di assicurasi che ci si trovi all'interno di quella porzione di codice.

- test_mintLimit(NFT): questa è la funzione di testing principale dello script. Prova a fare il mint di tutti gli NFT possibili e si assicura che una volta raggiunto il limite massimo, il valore della variabile totalSupply() risulti effettivamente il valore aspettato. Nel nostro caso, ci aspettiamo un valore di 500. La sua implementazione, per quanto semplice, è:

$def\ test\_mintLimit(NFT):$
while $True$: **do**
$\quad try:$
$\quad\quad NFT.mint(1, \{"from" : accounts[0]\})$
$\quad except:$
$\quad\quad assert\ NFT.totalSupply()\ ==\ 500$
$\quad\quad break$
end while

Come si può notare, la verifica sulla variabile totalSupply() viene effettuata all'interno del blocco except. Questo accade perché ci si aspetta che, una volta raggiunto il limite di NFT coniabili, venga lanciata una eccezione.

- test_allTokenUri(NFT): questa funzione serve ad accertarsi che il prefisso dell'uri dei token venga impostato correttamente per tutti i token disponibili. Viene effettuato un successivo controllo dopo una nuova modifica del prefisso. Si è aggiunta una variabile contatore che permette di contare che il controllo dell'uri sia stato effettuato per tutti i token coniati nel test precedente. Vediamo ora il codice:

$def\ test\_allTokenUri(NFT):$
$count\ =\ 0$
$NFT.setUriPrefix("prefix", \{"from" : accounts[0]\})$
for $i\ in\ range\ (1, NFT.totalSupply() + 1)$ **do**
$\quad count\ +=\ 1$

$assert\ NFT.tokenURI(i)\ ==\ f"prefix\{i\}.json"$

end for

$assert\ count\ ==\ 500$

$NFT.setUriPrefix("new\_prefix", \{"from" : accounts[0]\})$

$count\ =\ 0$

for $i\ in\ range\ (1, NFT.totalSupply() + 1)$ **do**

$count\ +=\ 1$

$assert\ NFT.tokenURI(i)\ ==\ f"new\_prefix\{i\}.json"$

end for

$assert\ count\ ==\ 500$

- test_wholeWallet(NFT): attraverso questa funzione si vuole accertare che i wallet degli account siano stati modificati correttamente. In particolar modo, si vuole essere sicuri che i wallet degli account, che non hanno ricevuto un nuovo token, restino invariati. Successivamente, si vuole controllare che tutti i restanti token appartengano a un account in particolare. Il codice è il seguente:

$def\ test\_wholeWallet(NFT)$:

$assert\ NFT.walletOfOwner(accounts[1])\ ==\ [4, 5, 6]$

$assert\ NFT.walletOfOwner(accounts[2])\ ==\ [7, 8]$

$assert\ NFT.walletOfOwner(accounts[3])\ ==\ [9, 10, 11]$

$temp\ =\ [1, 2, 3]$

for $i\ in\ range\ (12, NFT.totalSupply() + 1$ **do**

$temp.append(i)$

end for

$assert\ NFT.walletOfOwner(accounts[0])\ ==\ temp$

Tutti i test relativi a questo script sono i seguenti. Come è possibile vedere in fig 6.2, sono stati tutti superati senza rilevare alcun tipo di problema.

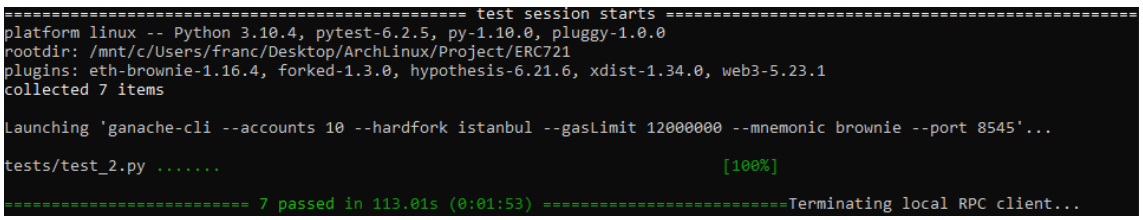

Figura 6.2: Risultato del secondo script di test.

6.3 Test_3.py

Vediamo ora l'ultimo script di testing realizzato. Il suo obbiettivo è trovare il miglio prezzo possibile per coniare correttamente degli NFT. Come constatato anche per gli altri due script di test, anche per questo script presenta le prime 3 righe per definire il modulo e l'istanza del contratto.
Presenta solamente due funzioni di test in quanto l'unico valore accettabile è 50*10^{15} wei. Questi due test sono stati implementati per dare una prova concreta che non si possono effettuare transazioni con prezzi inferiori, ma solo con il prezzo scelto dal proprietario del contratto. Le funzioni di test sono le seguenti:

- test_priceRangeToMint(NFT): lo scopo di questo test è trovare il prezzo minimo per la quale è possibile eseguire la funzione di mint. Il valore che ci si aspetta è 10^16wei, ma è un valore che può essere utilizzato una e una sola volta. Quando si proverà a coniare nuovi token nella funzione di test successiva, si noterà che l'unico valore che funziona per tutti i casi di test è solamente il valore di cost moltiplicato per il valore di maxMintPerTx, ovvero 50*10^15 wei. Il codice che implementa la ricerca del prezzo minimo è il seguente:

  $def\ test_priceRangesToMint(NFT):$
  $val\ =\ 1$
  while $True$ **do**
    $try:$
      $NFT.mint(1, \{"from" : accounts[0], "value" : val * 10 * *15\})$
      $break$
    $except:$
      $val\ +=\ 0.5$
  end while
  $assert\ val\ ==\ 10$

  L'assert su val riguarda non il prezzo della transazione in sé, ma il coefficiente moltiplicativo del valore base che risulta essere 10^{15} Come predetto a livello teorico, il valore che viene restituito per il primo mint funzionale è 10^{16} wei, valore che però scopriremo non essere adatto per l'uso totale del contratto.

- test_tryNewPriceForAll(NFT): questa funzione di test è complementare alla precedente, è serve a dimostrare che il prezzo ideale da pagare affinché il tutto funzioni è 5*10^{16} wei. L'implementazione è la seguente:

  $def\ test_tryNewPriceForAll(NFT):$
  $val\ =\ 10 * *16$
  $iterations\ =\ 1$
  while $iterations < 600$ **do**

try :

$NFT.mint(1, \{"from" : accounts[0], "value" : val\})$

$except$:

$iterations += 1$

if $iterations <= 10$ **then**

$val = int(5 * iterations * 10 * *15)$

end if

if $iterations == 600$ **then**

$break$

end if

end while

$assert\ NFT.totalSupply() == 500$

$assert\ val == 5 * 10 * *16$

La variabile iterations serve a tenere conto di quanti tentativi di mint sono stati eseguiti. Nel caso in cui si superi di 100 il limite di NFT disponibili, si esce dal ciclo e si controlla se il totalSupply e il prezzo totale siano quelli che ci si aspetta. Più nello specifico, ci si aspetta un valore di val di 5*10^{16} wei e un totalSupply di 500. È stato necessario l'utilizzo di un blocco try-except in quanto se il prezzo proposto per la transazione non è consono all'esecuzione, viene lanciata una eccezione. Quando ciò accade, viene incrementato il valore di val di una certa quantità, ovvero viene aumentato di un valore pari a 0.5 ogni iterazione. Dato che il valore che sicuramente funzioni per la transazione è 5*10^{16} wei,l'incremento della variabile val viene fermato quando raggiunge quel valore. Si è deciso di utilizzare questa scelta implementativa perché così facendo, si è sicuri che quello sia il prezzo minore per la quale la funzione di mint venga eseguita correttamente.

Tutti i test relativi a questo script sono stati eseguiti con successo, come è possibile vedere in fig 6.3. Attraverso l'uso dei casi di test è stato possibile controllare la correttezza del contratto NFTContract

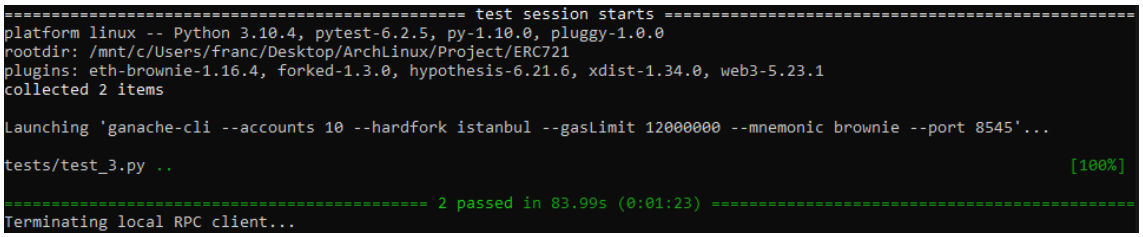

Figura 6.3: Risultato del terzo script di test.

e della sua robustezza agli errori.
Verificata la correttezza, è possibile usufruire del contratto anche su reti principali come Polygon main net o Ethereum, avendo la certezza che il contratto si comporta come pianificato.

Conclusioni

L'obbiettivo del progetto svolto in fase di tirocinio era di ottenere 500 NFT univoci attraverso l'utilizzo di python, solidity e del framework Brownie. Sono stati implementati l'algoritmo di Generative Art e di automatismo di mint in python e lo smart contract NFTContract in solidity, raggiungendo l'obbiettivo prefissato di ottenere a fine esecuzione 500 possibili token univoci.
Sono stati sviluppati dei casi di test in python che hanno permesso di verificare la correttezza del codice del contratto.
Grazie all'utilizzo del framework Brownie è stato possibile coordinare in successione la Generative Art e l'automatismo di mint, rendendo il tutto parte di un unico processo.
Alla fine delle dieci settimane, sono stati raggiunti tutti gli obbiettivi discussi all'inizio del percorso da tirocinante. Sono state acquisite nuove conoscenze e nuove competenze utili nell'ambito dell'ingegneria informatica. L'utilizzo della blockchain e degli smart contract è ancora in fase di sviluppo. È una tecnologia con molte applicazioni e possibilità che influenzerà il modo d'interagire con la rete in futuro.
Aver acquisito questa tipologia di conoscenze risulta essere importante per la crescita personale e professionale.
È disponibile l'intera repository su github al seguente indirizzo: https://github.com/cexxo/GenerativeArt.

Appendice A

A.1 Variante metodo randomness()

In questa sezione riportiamo la variante del metodo randomness(), variante che tiene conto anche dei pesi attribuiti ai singoli elementi di ogni layer.

for $each\ layer$ **do**
  for $each\ element\ in\ layer$ **do**
    $temp\_list.append(element)$
    $temp\_weight.append(weight)$
  end for
  $rand\ =\ random.choices(temp\_list, temp\_weight)$
  $randoms.append(rand)$
  $metadata\_field\ =\ get\_metadatafield()$
  $metadata[metadata\_field]\ =\ list\_of\_values[rand]$
  $temp\_list.clear()$
  $temp\_weight.clear()$
end for

A.2 Metodi secondari NFTContract

In questa sezione dell'appendice vengono riportati i metodi del contratto NFTContract non illustrati nel capitolo 4.

- setRevealed(state): permette di modificare lo stato del contratto, ovvero rendere gli NFT rivelati o meno. Si imposta il valore della variabile revealed al valore state. Operazione eseguibile solo dal proprietario del contratto

- setCost(cost): permette di modificare il costo di mint del contratto ad un valore pari a cost. Operazione eseguibile solo dal proprietario del contratto.

- setMaxMintAmountPerTx(amount): permette di modificare il numero massimo di NFT che possono essere coniati per transazione ad un valore pari a amount. Operazione eseguibile solo dal proprietario del contratto.
- setHiddenMetadataUri(uri): permette di modificare l'uri da restituire, in caso il revealing non è stato effettuato,ad un valore pari ad uri. Operazione eseguibile solo dal proprietario del contratto.
- setUriPrefix(prefix): permette di modificare il prefisso dell'uri pari ad al valore di prefix. Operazione eseguibile solo dal proprietario del contratto.
- setUriSuffix(suffix): permette di modificare il suffisso dell'uri pari ad al valore di suffix. Operazione eseguibile solo dal proprietario del contratto.
- _baseURI: permette di ottenere il valore del prefisso dell'uri. Operazione eseguibile solo internamente al contratto.
- isRevealed(): permette di conoscere lo stato di revealing del contratto. Restituisce il valore di revealed. Operazione eseguibile da chiunque.
- isPaused(): permette di conoscere lo stato di pausa del contratto. Restituisce il valore di paused. Operazione eseguibile da chiunque.
- setMaxSupply(max): permette di impostare, solo una volta, il numero massimo di NFT che possono essere coniati. Operazione eseguibile solo dal proprietario del contratto.

A.3 Codice Deploy.py

In questa sezione vediamo il codice completo relativo allo script Deploy.py.

$$
\begin{aligned}
&fp = open(\text{"}max\_mint.txt\text{"}, \text{"}r\text{"})\\
&max\_mint = int(fp.readline().strip())\\
&dev = accounts.add(config['wallets']['from\_key'])\\
&contract = NFTContract.deploy(\{\text{"}from\text{"} : dev\})\\
&contract.setPaused(False, \{\text{"}from\text{"} : dev\}\\
&contract.setRevealed(True, \{\text{"}from\text{"} : dev\})\\
&contract.setMaxSupply(max\_mint, \{\text{"}from\text{"} : dev\})
\end{aligned}
$$

A.4 Codice Mint.py

In questa sezione vediamo il codice completo relativo allo script Mint.py.

$dev = accounts.add(config['wallets']['from\_key'])$
$contract = NFTContract[len(NFTContract) - 1]$
$try:$
    $s = int(input())$
$except:$
    $pass$
for $i\ in\ range(s)$ **do**
  $try:$
      $contract.mint(1, \{"from" : dev, "value" : 50 * 10 * *15\})$
      $time.sleep(60)$
  $except:$
      $print(all\ the\ NFT\ have\ been\ minted...)$
end for

Come si può notare, si accede alle istanze del contratto come se esso fosse un array od una lista. Dato che voglio sempre accedere all'ultima istanza creata di NFTContract, prenderò in considerazione sempre il membro che si trova in posizione finale.

A.5 Codice setUri.py

In questo paragrafo viene mostrato il codice completo relativo allo script setUri.py illustrato all'interno del capitolo 5.

$dev = accounts.add(config['wallets']['from\_key'])$
$contract = NFTContract[len(NFTContract) - 1]$
$hash = get\_hash()$
$uri = "https://ipfs.io/ipfs/" + hash + "/"$
$contract.setUriPrefix(uri, \{"from" : dev\})$

Il metodo get_hash() permette di ottenere l'hash relativo alla cartella caricata su IPFS.

Bibliografia

[1] O. Ali, A. Jaradat, A. Kulakli, and A. Abuhalimeh, "A comparative study: Blockchain technology utilization benefits, challenges and functionalities," *IEEE Access*, vol. 9, pp. 12730–12749, 2021.

[2] M. Touloupou, K. Christodoulou, A. Inglezakis, E. Iosif, and M. Themistocleous, "Towards a framework for understanding the performance of blockchains," in *2021 3rd Conference on Blockchain Research Applications for Innovative Networks and Services (BRAINS)*, pp. 47–48, 2021.

[3] C. Smithlast, "Blocks," in *Ethereum.org*, 2021.

[4] P. R. Nair and D. R. Dorai, "Evaluation of performance and security of proof of work and proof of stake using blockchain," in *2021 Third International Conference on Intelligent Communication Technologies and Virtual Mobile Networks (ICICV)*, pp. 279–283, 2021.

[5] R. Sujeetha and C. A. S. Deiva Preetha, "A literature survey on smart contract testing and analysis for smart contract based blockchain application development," in *2021 2nd International Conference on Smart Electronics and Communication (ICOSEC)*, pp. 378–385, 2021.

[6] X. Ma, J. Zhou, H. Guo, and J. Wang, "Design of a stored-value card platform based on smart contract," in *2019 3rd International Conference on Circuits, System and Simulation (ICCSS)*, pp. 178–182, 2019.

[7] L. Toderean, C. Antal, M. Antal, D. Mitrea, T. Cioara, I. Anghel, and I. Salomie, "A lockable erc20 token for peer to peer energy trading," in *2021 IEEE 17th International Conference on Intelligent Computer Communication and Processing (ICCP)*, pp. 145–151, 2021.

[8] D. Chirtoaca, J. Ellul, and G. Azzopardi, "A framework for creating deployable smart contracts for non-fungible tokens on the ethereum blockchain," in *2020 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*, pp. 100–105, 2020.

[9] S. Casale-Brunet, P. Ribeca, P. Doyle, and M. Mattavelli, "Networks of ethereum non-fungible tokens: A graph-based analysis of the erc-721 ecosystem," in *2021 IEEE International Conference on Blockchain (Blockchain)*, pp. 188–195, 2021.

[10] S. Goyal, K. Sanjith, A. Sisodia, N. M. Suhaas, and S. Akram, "Transactions process in advanced applications on ethereum blockchain network," in *2020 International Conference on Recent Trends on Electronics, Information, Communication Technology (RTEICT)*, pp. 275–281, 2020.

[11] docs.ipfs.io, "Ipfs," in *https://docs.ipfs.io/concepts*, 2022.