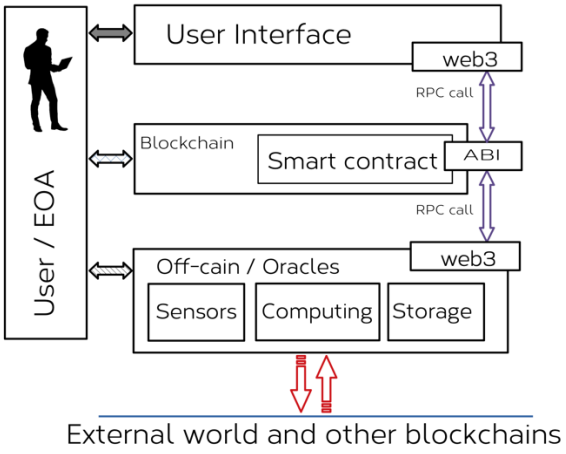
Decentralized Application

Le dApp sono dei sistemi software che sono state implementate grazie all'ausilio della tecnologia blockchain.
Ciò significa che una dApp si caratterizza per avere una sua logica basato su Smart Contract (o programmi) eseguiti all'interno di una blockchain. Questo rende l'applicazione decentralizzata e capace di ereditare le caratteristiche della blockchain.
Le dApp quindi è un sistema composto da molteplici componenti, alcuni dei quali possono non esse decentralizzati, come ad esempio lo storage e la parte di calculation. Tuttavia, oggi giorno esistono varie soluzione per il calcolo decentralizzato(nuNet) e per la memorizzazione decentralizzata(FileCoin).



Uno degli aspetti più rilevanti delle dApps concerne il metodo di comunicazione tra i vari componenti, in generale accesso alla componente blockchain tramite off-chain dipende da:

- Conoscenza dei contratti dell'ABI
- La creazione per la connessione verso il nodo/provider
- Uso delle librerie

WEB3

Tutte le BlockChain programmabili hanno un set di API disponibile con il quale le dApp possono interagire. Per differenti linguaggio di programmazione, esistono delle librerie web3 con la quale possono interagire da remoto con i nodi della blockchain.
Parliamo di web3 per indicare la terza evoluzione di internet. Le librerie di web3 facilitano la connessione e la comunicazione con i nodi.

- **Definizione di web3:**
  Il termine "Web3" si riferisce alla visione di un'Internet decentralizzata e interoperabile, costruita su blockchain e tecnologie distribuite, che mira a ridurre la dipendenza dagli intermediari centrali e a restituire il controllo e la proprietà dei dati agli utenti.

Web3 include vari metodi, ovvero:

- Gossip method: per sapere i dati riguardanti alla cima della blockchain e per le transazioni future

- State methods: per interrogare lo stato della BC e degli Smart contract
- History Methods: per interrogare i dati, per imparare i blocchi, le transazioni ecc..

Local Development Tools

Lo sviluppo di applicazioni decentralizzate può avvenire completamente in locale, prima della migrazione, il componente on-chain da blockchain pubbliche o private. Ciò è facilitato grazie all'utilizzo di strumenti e ambienti specifici.
Le suite più utilizzate sono Truffle+Ganache e Hardhat.

- Entrambi consentono la creazione di una rete blockchain locale. Lo sviluppo e compilazione di contratti intelligenti e test Solidity.
- Entrambi sono basati sulla riga di comando (CLI)
- Sono presenti anche dashboard per il monitoraggio (ganache ne ha una propria interfaccia grafica).

WEB3 in Python

La libreria web3.py è disponibile su Python che ti consente di farlo comunicare con un nodo compatibile.

- La documentazione è disponibile a questo link;
  - https://web3py.readthedocs.io/en/latest/
- L'installazione viene eseguita con:
  - $ pip installa web3

```
from web3 import Web3
w3 = Web3(Web3.HTTPProvider('URL provider and port'))

Provider Truffle develop:       http://127.0.0.1:9545/
Provider Sepolia:               https://rpc.sepolia.org/  (currently in read only)
Provider Sepolia (Third parts): https://endpoints.omniatech.io/v1/eth/sepolia/public

w3 = Web3(EthereumTesterProvider())  // dummy connection for test You need to import
                                        EthereumTesterProvider from web3.
```

Il codice fornisce un esempio di come utilizzare la libreria web3 in Python per creare una connessione a un nodo Ethereum utilizzando diversi tipi di provider, consentendo di interagire con la rete Ethereum per eseguire operazioni come il recupero di dati, l'esecuzione di transazioni e altro ancora.

WEB3 Account

L'invio di transazioni richiede il controllo completo di almeno un account e quindi della sua chiave privata. La gestione dell'account è fornita da classe web3.eth.account.
La gestione dell'account su web3 può essere eseguita in tre modi:

- **Generazione dell'account**: è possibile creare un account casuale basato su un valore di entropia a tua scelta.
  - Nel contesto dell'informatica e della crittografia, l'entropia è spesso utilizzata per valutare la complessità di una chiave crittografica o di una sequenza di dati. Una chiave crittografica ad alta entropia è considerata più sicura perché è più difficile da indovinare o decifrare. Allo stesso modo, una sequenza di dati con alta entropia è considerata più casuale e meno predicibile.

- **Via nodo o provider**: il nodo (o provider come infura, oppure via bridge come metamask) ti fornisce i tuoi account e ti permette di firmare transazioni tramite i dati registrati al suo interno.
- **Tramite chiave privata**: l'account viene generato localmente utilizzando la chiave privata (opportunamente memorizzati sul tuo dispositivo).

```python
from web3 import Web3
import secrets

w3 = Web3(Web3.HTTPProvider('https://endpoints.omniatech.io/v1/eth/sepolia/public'))

# Genera una stringa casuale di 32 byte (256 bit)
entropy = secrets.token_hex(32)

new_account = w3.eth.account.create(entropy)
print(new_account)
```

- **Argomento importante:**
  In Web3, la **chiave privata** è un componente critico per l'accesso e il controllo di un account Ethereum. Ecco cosa fa la chiave privata in Web3:

  1. Identificazione dell'account: La chiave privata è utilizzata per identificare univocamente un account Ethereum sulla blockchain. Ogni account Ethereum ha una chiave privata associata, che è un numero privato unico.

  2. Firma delle transazioni: Per inviare transazioni sulla blockchain Ethereum, è necessario firmarle digitalmente utilizzando la chiave privata. La firma digitale garantisce l'autenticità e l'integrità delle transazioni, consentendo ai nodi della rete di verificare che la transazione sia stata effettivamente autorizzata dal proprietario dell'account.

  3. Accesso ai fondi e agli asset: La chiave privata è necessaria per accedere ai fondi e agli asset detenuti nell'account Ethereum. Solo chi possiede la chiave privata associata a un account può inviare fondi da quell'account o eseguire altre operazioni su di esso.

  4. Generazione della chiave pubblica: La chiave privata è utilizzata per generare la chiave pubblica corrispondente, che a sua volta è utilizzata per derivare l'indirizzo dell'account Ethereum. L'indirizzo dell'account è la forma leggibile dell'account Ethereum e viene utilizzato per identificare l'account nella rete.

In sintesi, la chiave privata è uno degli elementi fondamentali della sicurezza e della gestione degli account Ethereum in Web3. È utilizzata per firmare transazioni, autenticare l'accesso agli account e garantire la sicurezza e l'integrità delle operazioni sulla blockchain Ethereum. È estremamente importante mantenere la propria chiave privata in un luogo sicuro e confidenziale, poiché chiunque abbia accesso alla chiave privata può controllare l'account Ethereum associato.

Transazioni

Web3 fornisce un controllo completo sui parametri della transazione. Una transazione è un dizionario che ha alcune chiavi specifiche, tra cui:
to, value, gas, maxFeePerGas, maxPriorityFeePerGas, nonce, chainId.

Alcuni dei parametri della transazione possono variare in base allo stato della blockchain o della transazione stessa. Possiamo modificare o aggiungere elementi al dizionario della transazione tramite il metodo update.

Nonce: Il numero di transazioni create dall'account.
Gas: massimo gas per questa transazione. Può essere calcolato dopo aver creato il dizionario della transazione e quindi aggiornarlo.
MaxFeePerGas. Corrisponde alla massima tassa di base che desideriamo pagare ed è dettata dallo stato della rete, a seconda che l'obiettivo sia stato raggiunto o meno. Possiamo determinarlo a partire da quello dell'ultimo blocco estratto.
Gasprice: Se supportato, usare invece di maxFeePerGas.

Esempio.
- **Parte definita**: chainId, from, to, value

```
transaction = {
        'chainId':w3.eth.chain_id,
        'from':fromAccount.address,
        'to':toAddress,
        "value": value,
        }
```

-**Parte Variabile:** gas, gas_price, nonce

```
transaction.update({'gas': w3.eth.estimate_gas(transaction)})

block = w3.eth.get_block('latest')
transaction.update({'gasPrice': int(block.baseFeePerGas *1.01) })

transaction.update({"nonce": w3.eth.get_transaction_count(fromAccount.add
```

Una transazione deve essere firmata utilizzando una chiave privata. Quello che ottieni è un oggetto transazione che contiene i dati della transazione "rawTransaction", hash e elementi di firma.

Per eseguire la transazione è necessario utilizzare il metodo send_raw_transaction che prende come argomento rawTransaction.
Ora, per confermare, basta aspettare la ricevuta utilizzando il metodo appropriato.

Una volta ottenuta la ricevuta, possiamo, tra le altre cose, analizzare il gas utilizzato.

Contratti Web3

Web3 consente di interagire con i contratti in due modi:

1. **Interazione con contratti esistenti sulla blockchain**: Utilizzando l'indirizzo del contratto e l'ABI (Application Binary Interface), che descrive la struttura e le funzioni del contratto, è possibile creare un riferimento al contratto esistente sulla blockchain. Una volta creato il riferimento, è possibile chiamare le funzioni del contratto, sia quelle che modificano lo stato della blockchain (trasazioni) sia quelle che leggono lo stato esistente senza modifiche (chiamate di tipo "view" o "pure").
    - **l'ABI** è una descrizione formale dell'interfaccia del contratto, che specifica i tipi di dati utilizzati dal contratto, nonché i nomi e i tipi di input e output delle funzioni del contratto.

2. **Creazione di nuovi contratti sulla blockchain**: Utilizzando il bytecode del contratto e l'ABI, è possibile deployare un nuovo contratto sulla blockchain. Quando si effettua il deploy, viene generato un nuovo indirizzo del contratto sulla blockchain, che viene restituito come output del processo di deploy. Una volta deployato, il contratto diventa parte della blockchain e può essere utilizzato e interagito come qualsiasi altro contratto esistente.

Le operazioni che possono essere eseguite attraverso i riferimenti ai contratti includono:

- **Chiamate statiche a funzioni pure o view**: Queste chiamate non modificano lo stato della blockchain e possono essere utilizzate per ottenere informazioni o eseguire calcoli basati sullo stato esistente del contratto.

- **Transazioni per attivare altre funzioni**: Queste transazioni modificano lo stato della blockchain e possono essere utilizzate per attivare funzioni del contratto che effettuano cambiamenti nello stato o eseguono altre azioni sulla blockchain.

In sintesi, Web3 consente di interagire con i contratti sulla blockchain sia per chiamare funzioni esistenti e leggere lo stato, sia per deployare nuovi contratti e modificare lo stato della blockchain attraverso transazioni.

Nelle slide viene mostrato come compilare uno smart contract Solidity utilizzando la libreria Python `solcx` e come interagire con un contratto esistente utilizzando Web3. Ecco una descrizione in italiano con le righe di codice inserite:

1. Installazione del compilatore Solidity:

```
import solcx
solcx.install_solc('0.8.18')  # Installa la versione 0.8.18 del compilatore Solidity
```

Questo codice installa la versione desiderata del compilatore Solidity utilizzando la funzione `install_solc` della libreria `solcx`.

2. Compilazione di un contratto Solidity:

```
contract_id, contract_interface = solcx.compile_source(scSource, output_values=['abi', 'bin']).popitem()
 abi = contract_interface['abi']
bytecode = contract_interface['bin']
```

Questo codice compila il codice sorgente Solidity contenuto nella stringa `scSource` utilizzando la funzione `compile_source` di `solcx`. Dopo la compilazione, estrae l'ABI e il bytecode del contratto dal dizionario restituito.

3. Creazione di un'istanza di contratto utilizzando Web3:

```
mycontract = w3.eth.contract(address=contractAddress, abi=contractABI)
```

Questo codice crea un'istanza di contratto locale utilizzando l'indirizzo del contratto sulla blockchain e l'ABI estratta dalla compilazione. Questa istanza di contratto può quindi essere utilizzata per interagire con il contratto esistente sulla blockchain tramite Web3.

Chiamata dei contratti

Una volta generato, possiamo interagire con l'istanza del contratto. L'attributo "functions" di un contratto consente di selezionare una delle funzioni pubbliche del contratto, presenti nell'ABI.
Per le chiamate a funzioni di visualizzazione o funzioni pure, posso utilizzare il metodo "call()" con la seguente sintassi:

```
risultato = contractInstance.functions.nomeFunzione(parametri).call()
```

Esempio:

```
mycontract = w3.eth.contract(address=indirizzoContratto, abi=ABIContratto)
risultato = mycontract.functions.get().call()
print(risultato)
```

Distribuzione dei Contratti

"Deploying a contract requires:

- l'ABI del contratto,
- il bytecode del contratto,
- la lista dei parametri del costruttore.

La transazione non richiede l'indirizzo di destinazione (verrà generato).
Dopo aver creato l'istanza locale tramite ABI e bytecode, puoi creare la transazione di deploy tramite il metodo build_transaction del costruttore del contratto."

```
transaction = mycontract.constructor(*params).build_transaction(
            {"chainId": chainID,
             "from": account.address,
             "gasPrice" : w3.eth.gas_price,
             "nonce": nonce,
             "value": value,
             })
```

Esecuzione delle funzioni dei contratti

Per eseguire transazioni e modificare lo stato dei contratti è necessario creare una transazione contenente il messaggio e i dati. Come per il deploy del contratto, utilizziamo il metodo `build_transaction` che esiste per ogni funzione del contratto intelligente. Questo metodo prende i dati della transazione (il dizionario) come argomento. La transazione così creata dovrà essere firmata.

```
transaction = mycontract.functions.functionName(Parameters).build_transaction({
        "chainId": chainID,
        "from": account.address,
        "value": value,
        "gasPrice" : w3.eth.gas_price,
        "nonce": nonce(account.address)})
```

The obtained object must be signed and forwarded to the blockchain.

```
signed_tx = w3.eth.account.sign_transaction(transaction, account.key)
tx_hash = w3.eth.send_raw_transaction(signed_tx.rawTransaction)
```

L'esempio nella slide precedente mostra la creazione di una chiamata a un contratto intelligente in cui impostiamo il nome della funzione e l'elenco dei parametri.

```
transazione = mycontract.functions.NomeFunzione(Parametri).build_transaction({ ... })
```

Possiamo creare una 'meta-transazione' in cui il nome della funzione e l'elenco dei parametri possono essere variabili, possiamo utilizzare la funzione `getattr` di Python.

```
Esempio: FUNZIONE è una stringa, *parametri è una tupla.
transazione = getattr(contratto.functions, FUNZIONE)(*parametri).build_transaction({…})
```

In questo modo posso utilizzare lo stesso pezzo di codice per creare più transazioni.

EVENTI

Approfondimento sugli eventi

Gli eventi in Web3 sono un meccanismo utilizzato per monitorare e reagire a determinati avvenimenti che si verificano sulla blockchain Ethereum. Gli eventi vengono emessi dai contratti intelligenti durante l'esecuzione delle loro funzioni e forniscono una modalità di comunicazione asincrona per informare gli utenti esterni su determinati stati o azioni all'interno del contratto.

Ecco alcuni *punti chiave* sugli eventi in Web3:

1. Definizione negli Smart Contract: Gli eventi vengono definiti all'interno dei contratti intelligenti utilizzando la parola chiave `event`. Gli sviluppatori specificano i dettagli degli eventi, come il nome, i parametri e gli eventuali dati aggiuntivi da includere.

2. Registrazione degli Eventi: Quando un evento viene emesso dal contratto, viene registrato sulla blockchain come parte della transazione che ha causato l'emissione dell'evento. Questo consente agli utenti esterni di accedere alla cronologia degli eventi e di recuperare informazioni su di essi.

3. Ascolto degli Eventi tramite Web3: Gli sviluppatori possono utilizzare la libreria Web3 per ascoltare gli eventi emessi dai contratti intelligenti. Web3 fornisce metodi per filtrare gli eventi in base a criteri specifici, come il nome dell'evento, i parametri associati o il mittente della transazione.

4. Utilizzo per la Notifica e l'Aggiornamento dello Stato: Gli eventi sono spesso utilizzati per notificare gli utenti esterni di determinati avvenimenti sulla blockchain e per aggiornare lo stato dell'applicazione o del frontend di conseguenza. Ad esempio, un'applicazione di scambio potrebbe utilizzare gli eventi per notificare agli utenti quando vengono eseguite transazioni di acquisto o vendita.

5. Tracciabilità e Auditabilità: Gli eventi forniscono una maggiore tracciabilità e auditabilità delle azioni eseguite sui contratti intelligenti. Poiché vengono registrati sulla blockchain, è possibile accedere alla cronologia completa degli eventi per verificare le azioni eseguite nel corso del tempo.

Quindi, eventi in Web3 forniscono un modo potente per comunicare e reagire agli avvenimenti che si verificano sulla blockchain Ethereum, consentendo agli sviluppatori di costruire applicazioni più interattive, responsabili e trasparenti.

Event Management

La gestione degli eventi è un processo che consiste nel "ascoltare" gli eventi emessi dai contratti intelligenti e quindi eseguire operazioni specifiche quando si verificano eventi specifici. L'operazione si basa sul filtraggio degli eventi.

Per rilevare un evento emesso da un contratto intelligente è necessario creare un'istanza del filtro utilizzando il metodo `create_filter`.

Example: we create a filter that reads the events starting from the last block created.

```
event_filter = mycontract.events.NAME_EVENT.create_filter(fromBlock='latest',
                                                          argument_filters={'NAME_ARG': value })
```

Un oggetto filtro creato in questo modo può essere utilizzato per gestire gli eventi. Ad esempio, possiamo creare un programma che ascolta un evento tramite il filtro.

Esempio: Utilizziamo il metodo `get_new_entries()` per ottenere l'elenco di tutti gli eventi emessi dall'ultima lettura.

```
while True:
    for event in event_filter.get_new_entries():
        print(event) # Cosa fare in caso di rilevamento dell'evento
    time.sleep(2) # tempo di attesa
```

In questo esempio, il ciclo while continua a eseguire un'iterazione infinita. All'interno del ciclo, viene eseguita un'iterazione sulla lista di nuovi eventi ottenuti tramite il filtro. Per ogni evento rilevato, viene eseguita un'azione specifica, come la stampa dell'evento stesso o l'esecuzione di altre operazioni correlate. Dopo ogni iterazione, il programma attende per un certo periodo di tempo (in questo caso 2 secondi) prima di continuare con la successiva iterazione.

- Altri filtri:
  È inoltre possibile creare oggetti filtro per attività di gossip come il monitoraggio transazioni in sospeso e nuovi blocchi.
  Per questi tipi di filtri viene utilizzato il metodo w3.eth.filter() che prende come argomento una stringa per il tipo di attività che si desidera monitorare.

  Pending transactions: `filter = w3.eth.filter("pending")`

  New blocks: `filter = w3.eth.filter("latest")`

ORACLE

Oracle è un componente fondamentale nell'ecosistema delle blockchain, consentendo l'interazione tra la blockchain e il mondo esterno attraverso la fornitura di dati affidabili e sicuri.

La blockchain e i contratti intelligenti non possono leggere direttamente dati dall'esterno. Questa limitazione è conosciuta come il problema dell'oracolo. Un oracolo è un sistema progettato per fornire dati esterni a un contratto intelligente, al fine di risolvere il problema dell'oracolo. Creiamo un oracolo semplice in grado di fornire a un contratto intelligente dati su richiesta. Il sistema sarà composto da tre componenti:

- un'applicazione utente che legge dal contratto intelligente
    - L'applicazione utente è un programma che legge i dati ed esegue la richiesta di aggiornamento dei dati in tempo utile.

```
# Data reading
def get():
    return mycontract.functions.get().call()


# Update request
def request():
    receipt = metaTransaction.metaTransaction(w3, account1, mycontract, 0, 'request')
    return receipt['transactionHash'].hex()
```

- il contratto intelligente, che registra i dati e li fornisce su richiesta.
- un oracolo, composto da un servizio esterno in grado di inviare dati aggiornati al contratto intelligente
    - Logica dell'oracolo. In questo esempio, i dati vengono generati casualmente. Nelle applicazioni pratiche, l'oracolo sarà in grado di leggere i dati da fonti esterne.

```
oracleContract = w3.eth.contract(abi=abi,address=address)
event_filter = oracleContract.events.dataRequest.create_filter(fromBlock='latest')

def handle_event(event):
    print("NEW REQUEST: ", event)
    data = data_retrieving()
    print("LISTEN New Data:", data)
    print("LISTEN nonce ",w3.eth.get_transaction_count(account1.address))
    try:
        receipt = metaTransaction(w3, account1, oracleContract, 0, 'update',data)
        print(receipt['transactionHash'].hex())
    except:
        print("Transaction pending")

def update_data():
    while True:
        data = data_retrieving()
        print("UPDATE New Data:", data)
        print("UPDATE nonce", w3.eth.get_transaction_count(account1.address))
        try:
            receipt = metaTransaction(w3, account1, oracleContract, 0, 'update',data)
            print(receipt['transactionHash'].hex())
        except:
            print("Transaction pending")
        time.sleep(60)
```