Clase Transaction

Una transacción en el contexto de blockchain es un cambio de estado en la red. Las transacciones de nuestra blockchain permiten que dos usuarios puedan transferirse dinero virtual.

Vamos a crear un archivo en nuestro directorio de trabajo llamado transaction.py. En él, vamos a instanciar una clase de nombre Transaction.

In [3]:
```
class Transaction():
    pass
```

Cada transacción tiene 3 elementos principales.

- ¿Quién manda la transaccion? (sender)
- ¿Quién la recibe? (recipient)
- ¿Cuánto dinero es? (value)

Cada que se instancie una transacción en local, la vamos a inicializar con sus elementos principales.

In [4]:
```
from bin.account import Account

class Transaction():
    def __init__(self, sender: Account, value: int, recipient: Account):
        pass
```

Además, añadiremos atributos "informativos" que sirven para guardar información importante de la transaccion.

- Timestamp: Hora en la que la transaccion fue añadida a la red.
- Block: Bloque al que pertenece la transaccion.
- Signature: Firma digital de la transaccion.
- Status: Estado actual de la transaccion.

Así como añadir a la propia transaccion al historial de transacciones de quien envía la transaccion. Aunque se rechace la transacción, quedara el registro guardado en su cuenta.

In [5]:
```
from bin.account import Account

class Transaction:
    def __init__(self, sender: Account, value: int, recipient: Account):
        self.sender = sender
        self.value = value
        self.recipient = recipient
        # Hora en la que la transaccion se instancia
        self.time = datetime.now().strftime("%d/%m/%Y %H:%M:%S")
        self.block = None
        self.signature = None
        self.status = TxStatus.PENDIENTE
```

```
        # Al instanciarse una tx, esta debe reflejarse en la cuenta que la envia. (
        sender.list_of_all_transactions.append(self)
```

El atributo status lo necesitamos para ir cambiando si nuestra transacción esta en espera, pendiente o aprobada. Para ello, vamos a crear una clase Enum que nos ayude a hacerlo visible.

Enum

Un Enum nos ayuda a definir nuestros propios tipos de datos. En este caso, vamos a diseñar uno que nos dé 3 casos de una transacción:

- Completada
- Rechazada
- En espera

La clase TxStatus la vamos a poner antes de nuestra clase Transaction en nuestro archivo transaction.py

In [6]:
```
from enum import Enum
from bin.account import Account

class TxStatus(Enum):
    PENDIENTE = 0
    CONFIRMADA = 1
    DECLINADA = 2

class Transaction():
    def __init__(self, sender: Account, value: int, recipient: Account):
        ...
        self.status = TxStatus.PENDIENTE
```

Un ejemplo visual de cómo se veria seria el siguiente:

In [7]:
```
tx = Transaction(None, None, None)
tx.status
```

Out[7]:
```
<TxStatus.PENDIENTE: 0>
```

Hablando de sus métodos...

Implementaremos un método llamado **to_dict** que exporte un diccionario con el "encabezado" de la transacción. Es la información que se codifica por un hash a la hora de obtener su firma digital.

In [8]:
```
    def to_dict(self):
        """Exporta la transaccion en formato: dict."""
        return {
            'sender': self.sender.nickname,
            'recipient': self.recipient.nickname,
            'value': self.value,
            'time' : self.time}
```

Implementaremos dos métodos que nos ayuden a:

- Firmar transacciones.
- Verificar la firma digital.

Pero para ello, primero veamos este ejemplo:

Firmado y Verificado de transacciones utilizando una cuenta

Firmado

Digamos que nuestra transacción es lo siguiente:

In [9]:
```
transaction = {
    "sender": "Pedro",
    "receiver": "maria",
    "amount": 10
}
```

Si trataramos de ingresar en un hash este contenido tendriamos un error muy grande: (probar en local)

In [1]:
```
from Crypto.Hash import SHA256
# hash = SHA256.new(transaction) # Sale un errorsaso.
```

El algoritmo SHA256 solo acepta valores tipo byte, por lo tanto necesitamos convertir nuestro diccionario en una cadena de bytes.

Primero convertimos de dict -> str y después de str -> bytes.

In [11]:
```
transaction_str = str(transaction)
# Ya lo convertimos a String, ahora necesitamos convertirlo a bites
transaction_byte = transaction_str.encode()
type(transaction_byte)
```

Out[11]:
```
bytes
```

Una vez lo tenemos en bytes podemos pasarlo por un algoritmo SHA256 sin problema.

In [12]:
```
tx_hashed = SHA256.new(transaction_byte)
# Imprimimos el hash en su version hexadecimal
tx_hashed.hexdigest()
```

Out[12]:
```
'6a9519ccc6c7ecececf8801e699fe72544eccb46a37a7387bc416d4f347141a8'
```

Ya que tenemos el contenido de nuestra transacción y su hash, podemos obtener su firma. Hagamos otro ejemplo.

Pedro es el dueño de la transacción. Por lo tanto, Pedro tiene una cuenta con sus respectivas llaves, y puede firmarlas con su llave privada.

In [13]:
```
from bin.account import Account
pedro = Account(100, 'pedro')

# Pedro procede a firmar la transacción
signature = pedro.signer.sign(tx_hashed)
```

```
# Nuestra firma se ve de la siguiente manera
signature
```

Out[13]:
```
b':\xf3V\n\x92SN\n\xf5\xd9\xc6Ym\xdc\xd7\xfb\xf2\xcb\xf9\xa2I\xdfI\xe5\xc7\xd5\x98
+E\xc8\xae\xa7<k\xb0SZ\x99/\xbe\xb4f\xa9\x9fS4\x9e\xe4\xda&H\x07Y\x9ez\x9c\xe2\xe3
\x85w\x08j)*.]\xb2\xed\x1ee\xde\x03\xc2x\x90r~\xde\x98D\xee:\xfe\xf8\x05\r\xad\x83
K7\xcd\x02\xca\xd5}\x8d\xfa4\xf9\xd3\xb5\x10\xe9T(\x9c\xc40{M\xc4\x13\xff\xa4\xae
\xfa\x04Q\x1b\xf2\xe8\xd5\xeb!\xc4p\x14\xcc'
```

Verificado

Para verificar una transacción el proceso es muy similar.

Primero hay que hacer el mismo proceso de pasar la transaccion a un algoritmo hash.

In [14]:
```
tx_hashed = SHA256.new(str(transaction).encode()) # Todo el hash en una sola linea
tx_hashed
```

Out[14]:
```
<Crypto.Hash.SHA256.SHA256Hash at 0x1d2a4ba3640>
```

Solo que en vez de usar el firmador, vamos a usar el verificador.

Para verificar necesitamos:

- La firma de la transacción.
- El hash de la transacción previamente obtenido.

El verificador alza un error si la firma es inválida. Por eso utilizamos try/except.

In [15]:
```
try:
    pedro.verifier.verify(tx_hashed, signature)
    print(True)
except:
    print(False)
```

```
True
```

In [16]:
```
# Si lo intentaramos con una firma falsa el resultado seria distinto
firma_falsa = SHA256.new('transaccion falsa'.encode())
try:
    pedro.verifier.verify(firma_falsa, signature)
    print(True)
except:
    print(False)
```

```
False
```

Las condiciones para que salga error son:

- El contenido de la transacción haya sido alterado.
- La firma es incorrecta.
- Quien firma no es el autor de la transacción.

Vamos a añadir estas funciones a los métodos de nuestra clase Transaction. Nuestro código completo quedaría así:

In [20]:
```
from datetime import datetime
```

```
from enum import Enum
from bin.account import Account
from Crypto.Signature.pkcs1_15 import PKCS115_SigScheme
from Crypto.Hash import SHA256

class TxStatus(Enum):
    PENDIENTE = 0
    CONFIRMADA = 1
    DECLINADA = 2

class Transaction:
    def __init__(self, sender: Account, value: int, recipient: Account):
        self.sender = sender
        self.value = value
        self.recipient = recipient
        self.time = datetime.now().strftime("%d/%m/%Y %H:%M:%S")
        self.block = None
        self.signature = None
        self.status = TxStatus.PENDIENTE
        # Al instanciarse una tx, esta debe reflejarse en la cuenta que la envia. (
        sender.list_of_all_transactions.append(self)


    def to_dict(self):
        """Exporta la transaccion en formato: dict."""
        return {
            'sender': self.sender.nickname,
            'recipient': self.recipient.nickname,
            'value': self.value,
            'time' : self.time}

    def sign_transaction(self): # (1)
        """Funcion que recibe un objeto transaccion y devuelve
        la firma de la transaccion en bytes"""
        print("Firmando transaccion...")
        msg = str(self.to_dict()).encode()
        hash = SHA256.new(msg)
        signer = self.sender.signer
        signature = signer.sign(hash)
        # print("Signature:", binascii.hexlify(signature))
        self.signature = signature

    def verify_signature(self) -> bool: # (1)
        """Aqui se verifican las transacciones"""
        print("Verificando la firma de la transaccion...")
        msg = str(self.to_dict()).encode()
        hash = SHA256.new(msg)
        verifier = self.sender.verifier
        try:
            verifier.verify(hash, self.signature)
            print("La firma es valida.")
            return True
        except:
            print("La firma es invalida.")
            return False

    def change_status(self, new_status): # (3)
        if new_status == 'CONFIRMADA':
            self.status = TxStatus.CONFIRMADA
        elif new_status == 'PENDIENTE':
            self.status = TxStatus.PENDIENTE
```

```
        elif new_status == 'DECLINADA':
            self.status = TxStatus.DECLINADA
```

Lo que aplicamos se encuentra en las funciones sign_transaction, verify_signature() y to_dict. Por otro lado...

Nuestro método change_status cambia el estado de una transacción. Hay que recordar que los estados cambian dependiendo del evento, por ejemplo:

- Cuando una transacción se crea, empieza siendo Pendiente.
- Cuando es parte de la cadena de bloques, pasa a estar confirmada.
- Si la red tiene algún error, podría rechazar la transacción.

Si ya vimos las cuentas, sus transacciones y el dónde se almacenan (bloques), solo nos queda ver el cómo se anexa un bloque a la red.