Blockchain y Criptomonedas II

UCEMA - QUANt

Clase I: 10/11/2024

Matias Selser

Sobre Mi

Educación:

- 08/2022 - 05/2024 : Maestría en Matemática - *Columbia University*
- 03/2020 - 05/2022 : Maestría en Finanzas - *Universidad Torcuato di Tella*
- 03/2013 - 07/2020 : Ingeniería Electrónica - *Instituto Tecnológico de Buenos Aires*

Experiencia relevante:

- 08/2023 - Actual : Execution Lead - *Terrace.fi*
- 08/2022 - 08/2023 : Quantitative Developer - *MRM Analytics*
- 02/2022 - 06/2022 : Quantitative Developer - *Privi Protocol*
- 08/2020 - 01/2022 : Quantitative Developer - *Alma Global Strategies*

Publicación:

- 2021: ¨*Optimal Market Making by Reinforcement Learning*¨

Contacto:

- Email: ms6517@columbia.edu
- LinkedIn: https://www.linkedin.com/in/matias-selser

Metodología

- Formato: dos partes de 75 minutos + break de 15 minutos
- Módulos: pequeñas unidades de contenido
- Actividades: después de los módulos
- Tareas: challenge opcional

Agenda

- Módulo I: Introducción a la Blockchain y Ethereum
- Módulo II: Transacciones en Ethereum
- Módulo III: Infraestructura en Ethereum
- Módulo IV: Web3.js and Web3.py
- Módulo V: Uniswap V2
- Conclusión y resumen
- Tarea propuesta

Módulo I
Introducción a la Blockchain y Ethereum

1.1 Introducción a la *blockchain*:

- Repaso:
  - Blockchain: cadena de bloques
  - Asociados entre sí (cada bloque contiene el hash del bloque anterior)
  - Con información sobre cambios en el estado de la naturaleza
    - *Wallet* A sent *Wallet* B 1 eth
    - *Wallet* X interacted with *Wallet* Y (contract)
    - …

- Descentralización:
  - *“Ledger”* [libro mayor] descentralizado
    - No tiene dueño
    - Múltiples copias distribuidas a lo largo de la red en los nodos
  - Nodos:
    - Mantienen una copia actualizada del libro
    - Públicos y privados
    - *Read Only* y R/W
    - Cada uno tiene su propia mempool

- Seguridad:
  - Criptografía:
    - Transacciones seguras
    - Clave pública y privada
  - Consenso:
    - PoW y PoS (entre otros) aseguran la validez de las transacciones

- Rol de los mineros (PoW):
  - Resuelven algoritmos matemáticos complejos computacionalmente
    - Data + *random nonce* = hash del bloque tiene que empezar con n ceros
- Rol de los validadores (PoS):
  - *Stake* [apuesta] de parte del nodo para ser el creador
  - Mientras más apuesta, mejores chances
  - El nodo elegido propone el bloque
  - Se valida, si tiene transacciones fraudulentas, pierde lo apostado

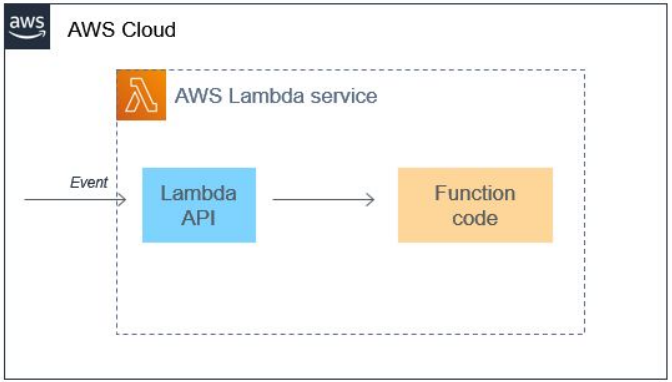
¡Ambos son críticos!

1.2 Introducción a Ethereum:

- ¿Qué diferencia a Ethereum de Bitcoin?
  - Smart contracts
  - dApps montadas sobre los smart contracts
  - EVM (*Ethereum Virtual Machine*)
    - Máquina virtual que ejecuta las transacciones
    - “Regula” que la lógica sea válida

- *Smart Contracts*
  - Programación orientada a eventos
    - Los *Smart Contracts* se almacenan en una dirección en la blockchain
    - Cuando se quiere ejecutar alguna función, se interactúa con esa dirección
    - No son procesos que quedan ejecutándose en el fondo
      - No tiene timers, por ejemplo!

- *Smart Contracts (continuación)*
  - Programación orientada a eventos
    - Similar al funcionamiento de las *Lambda Functions* de AWS



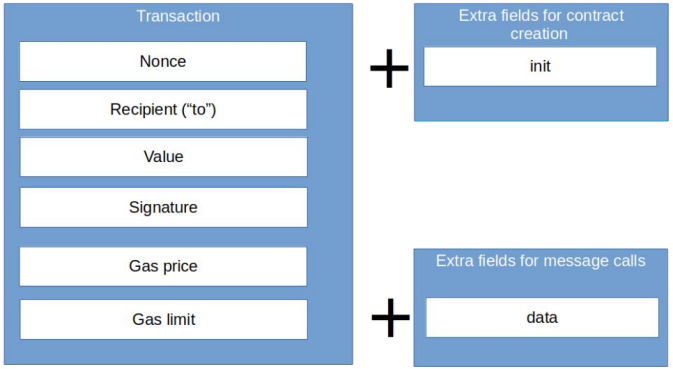
1.3 *Test chains*:

- ¿Para qué sirven?
  - Similaridad con el concepto de “staging” del desarrollo convencional
  - Validación de la lógica de los *smart contracts*
  - Espacio de aprendizaje sin riesgo
- Ejemplos
  - Ropsten, Rinkerby, Kovan (ETH)
  - Optimistic Goerli (Optimism)
  - Amoy (Polygon)
  - …

- Faucets [canilla]
  - Proveen gas o tokens de forma gratuita para desarrollo y testeo para usuarios
  - Ejemplo:
    - MATIC: link

Módulo II
Transacciones en Ethereum

2.1 ¿Qué es una transacción?



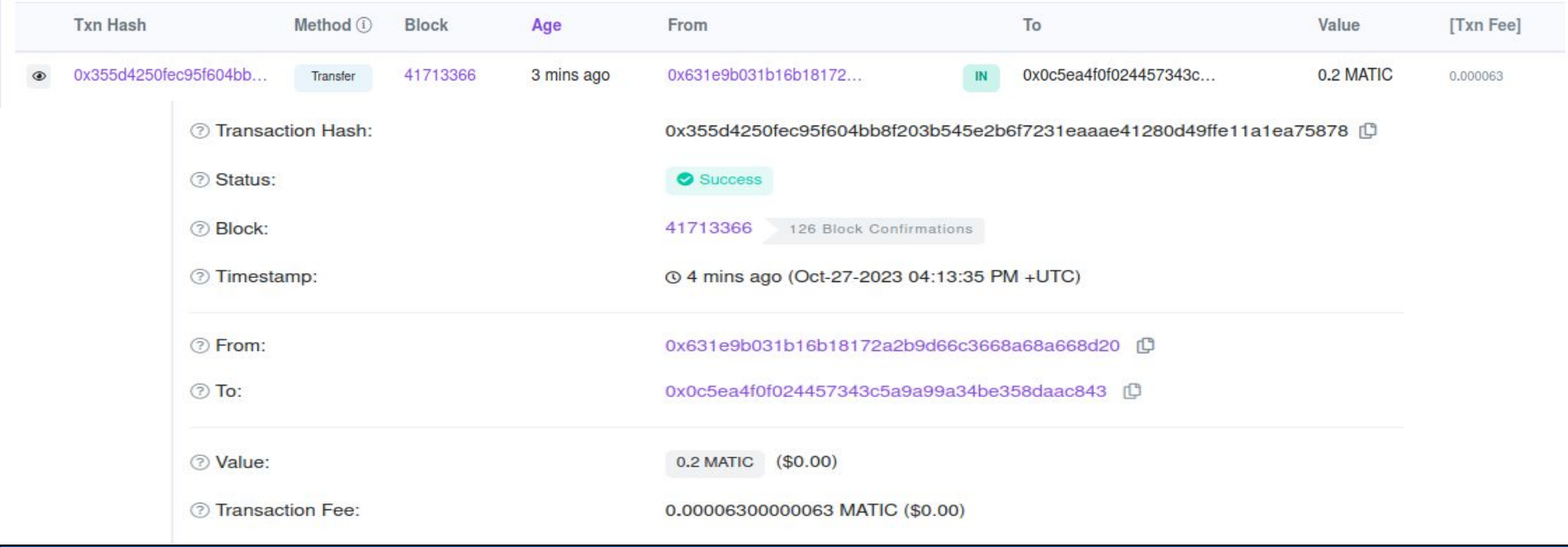
- Esencialmente, cuatro componentes básicos:
  - Origen: *wallet* que generó la transacción
  - Destino: *wallet* que va a recibir la transacción (puede ser un *smart contract*)
  - Value: cuánto gas va a recibir el destinatario (no es lo mismo que el gas para la transacción)
  - Data: si es un smart contract, qué método se busca invocar y con qué parámetros

2.1 ¿Qué es una transacción?

- Dependiendo la blockchain, además hay parámetros como
  - gasLimit: cantidad máxima de gas que puede usar la transacción
    - Importante para prevenir for loops bloqueantes
  - gasPrice: prima que se paga para que la transacción se incluya en el bloque
  - nonce: número de transacción, siempre incremental, para mantener las transacciones ordenadas
- Mínimo gas para la transacción = gasPrice x gasLimit

2.2 Tipos de transacción

- ETH transactions: link



Txn Hash	Method	Block	Age	From		To	Value	[Txn Fee]
0x355d4250fec95f604bb...	Transfer	41713366	3 mins ago	0x631e9b031b16b18172...	IN	0x0c5ea4f0f024457343c...	0.2 MATIC	0.000063

Transaction Hash: 0x355d4250fec95f604bb8f203b545e2b6f7231eaaae41280d49ffe11a1ea75878

Status: Success

Block: 41713366 126 Block Confirmations

Timestamp: 4 mins ago (Oct-27-2023 04:13:35 PM +UTC)

From: 0x631e9b031b16b18172a2b9d66c3668a68a668d20

To: 0x0c5ea4f0f024457343c5a9a99a34be358daac843

Value: 0.2 MATIC ($0.00)

Transaction Fee: 0.00006300000063 MATIC ($0.00)

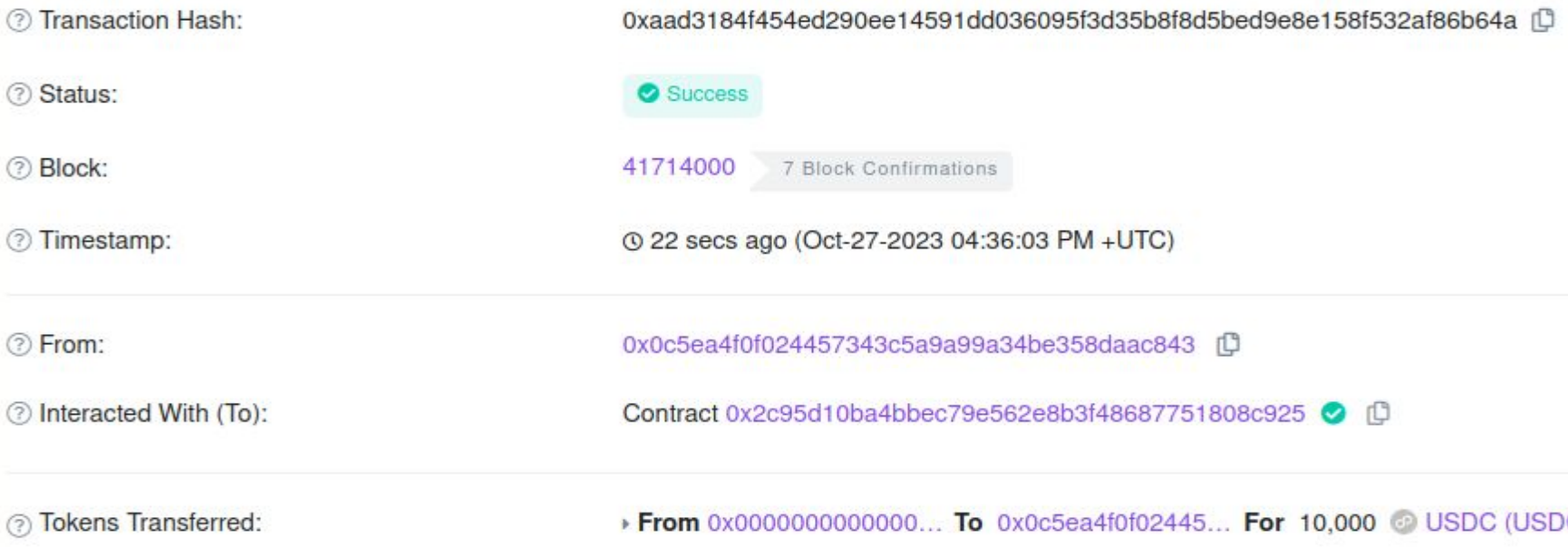
- ETH transactions (continuación):
  - En la transacción anterior se ve:
    - From
    - To
    - Value
  - Además, cuestiones asociadas al procesamiento:
    - Número de bloque
    - Hash de la transacción

- Interacciones con un *smart contract*:
  - Interfaz: expone la forma de interactuar con el contrato
    - Funciona como una API
  - Métodos: describen la función y los parámetros que se necesitan para interactuar con el contrato
  - Eventos: sirven para describir qué está haciendo la lógica.
    - Por ejemplo: el contrato ERC20.sol (un token) en su lógica transfiere el balance deseado del usuario A al usuario B
    - Al hacerlo, emite un evento *Transfer* que contiene información asociada (origen, destino, cantidad)
  - ABI: funciona como una interfaz que se expone dentro de un json. Sirve para interactuar con los contratos fuera del entorno de trabajo de *Solidity*. Se puede exportar de Etherscan!

- Código: IERC20.sol:
  - Métodos:
    - totalSupply
    - balanceOf
    - transfer
    - allowance
    - approve
    - transferFrom
  - Eventos:
    - Transfer
    - Approval

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

// https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v3.0.0/contracts/token/ERC20/IERC20.sol
interface IERC20 {
    function totalSupply() external view returns (uint);

    function balanceOf(address account) external view returns (uint);

    function transfer(address recipient, uint amount) external returns (bool);

    function allowance(address owner, address spender) external view returns (uint);

    function approve(address spender, uint amount) external returns (bool);

    function transferFrom(
        address sender,
        address recipient,
        uint amount
    ) external returns (bool);

    event Transfer(address indexed from, address indexed to, uint value);
    event Approval(address indexed owner, address indexed spender, uint value);
}
```

- Contrato USDC en polygon mumbai:



Hands on: creación de una criptomoneda!

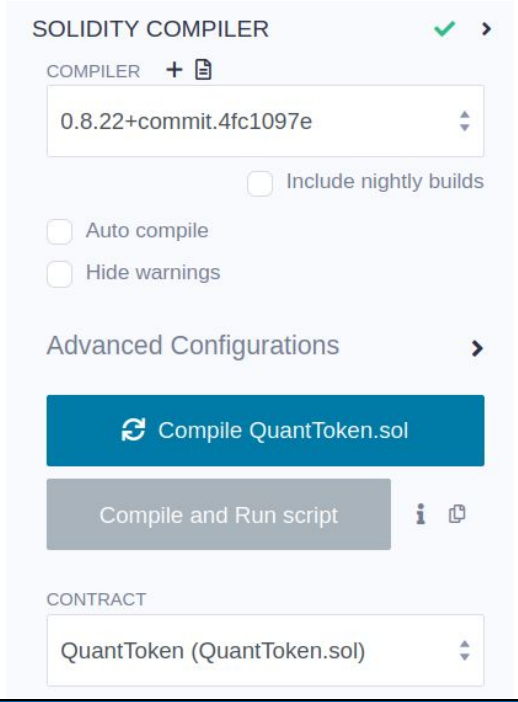
- Abrir Remix. Remix es un IDE para desarrollar Solidity
- Crear un nuevo proyecto
- Crear un archivo QuantToken.sol [código en GitHub]

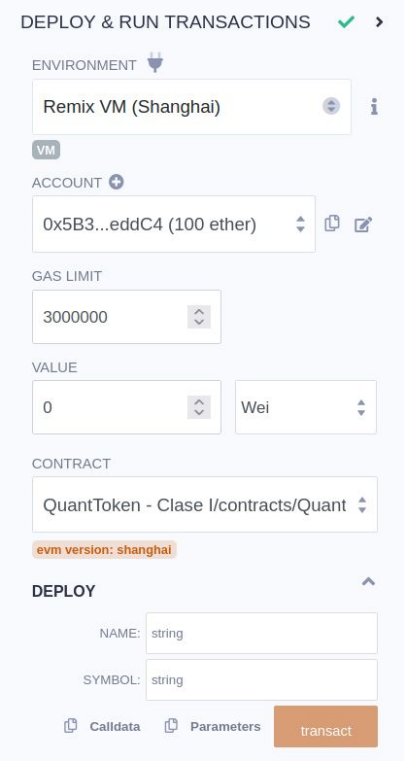
QuantToken.sol | ERC20/ERC20.sol

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.20;
3
4  import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v4.0.0/contracts/token/ERC20/ER
5
6  contract QuantToken is ERC20 {
7      constructor(string memory name, string memory symbol) ERC20(name, symbol) {    infinite gas 100980
8          _mint(msg.sender, 100 * 10 ** uint(decimals()));
9          }
10
11 }
```

Hands on: creación de una criptomoneda! (continuación)

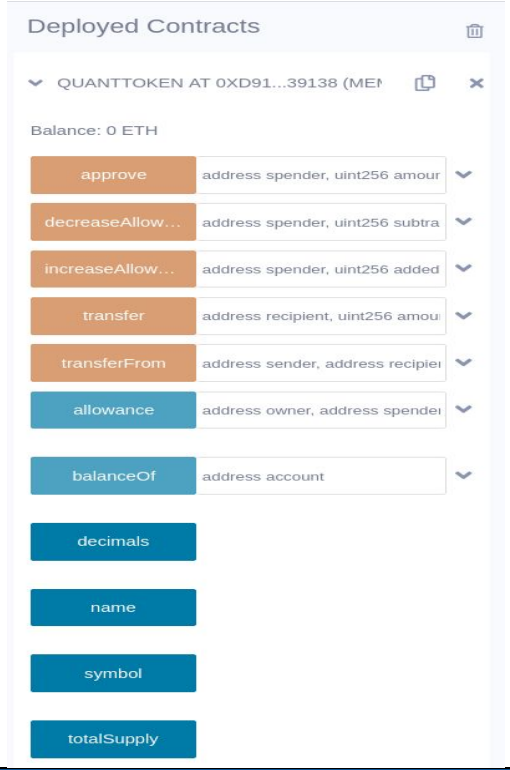
- Compilar el código
- Deploy!





Hands on: creación de una criptomoneda! (continuación)

- Finalmente, interacción



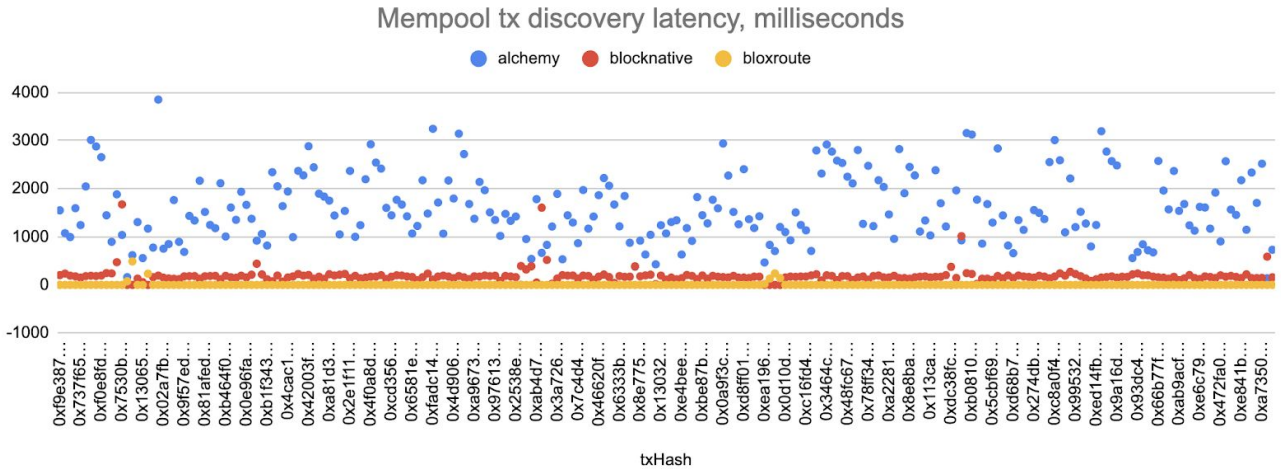
Módulo III
Infraestructura en Ethereum

3.1 Nodos RPC

- ¿Qué son?
  - *RPC* significa *Remote Procedure Call*
  - Sirven como modo de interacción entre un nodo de la *blockchain* y los sistemas externos
  - Acceso a los datos
    - Los nodos permiten que aplicaciones externas accedan a la información almacenada en la *blockchain*
  - Transmisión de las transacciones
    - Cuando los usuarios generan transacciones, los nodos *RPC* son los responsables de transmitir esa transacción a la red

- ¿Qué son? (continuación)
  - Monitoreo de eventos
    - Los nodos permiten que los desarrolladores se suscriban a los mencionados eventos!
  - Proveedores
    - Existen diversos proveedores como *Infura* y *Alchemy* que ofrecen nodos RPC como servicio

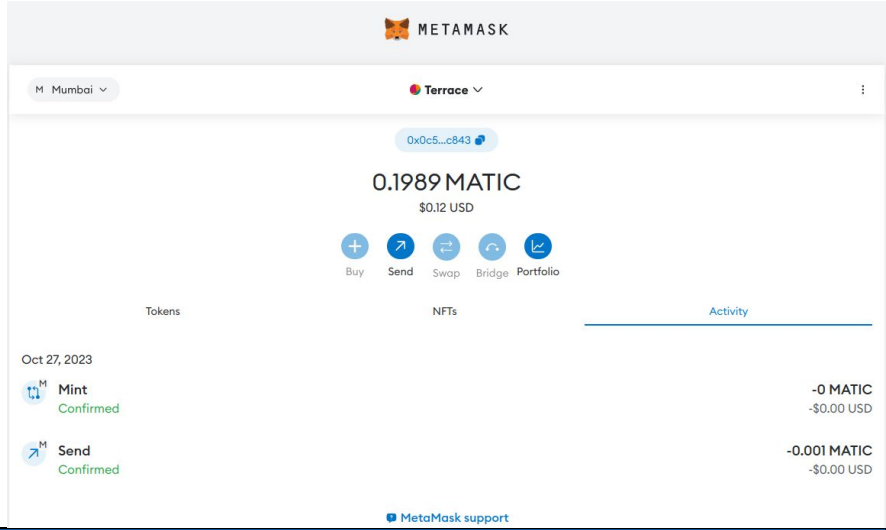
- Distintos proveedores
  - Gratuitos
    - Infura
    - Alchemy
  - Pagos
    - BloxRoute
    - BlockNative



3.2 *Providers and Signers*

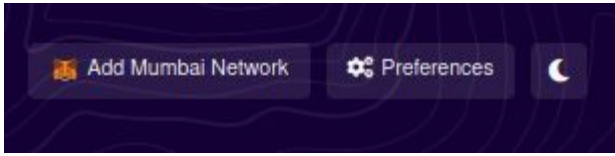
- *Providers*
  - "*Bridge to Ethereum*"
    - Funcionan como conectores entre la blockchain y el exterior
    - Facilitan la conexión entre las *dApps* y los *smart contracts*
- *Signers*
  - Permiten la creación de transacciones seguras
  - Vitales para asegurar la validez de las interacciones en la blockchain

- *Metamask*
  - Combina el funcionamiento de un *provider* y un *signer*
  - Permite el manejo de múltiples cuentas
  - Permite el manejo de múltiples *chains*



Hands on: set up metamask + alchemy

- Crear con metamask una wallet
- Agregar amoi a metamask
- Mintear MATIC a la cuenta con un faucet
- Agregar el token a Metamask
- Crear una cuenta en Alchemy
- Crear un proyecto nuevo que genere un nodo RPC para mumbai
- Visualizar el endpoint



Break

Módulo IV
Web3.js and Web3.py

4.1 Introducción a Web3

- ¿Qué es Web3?
  - Librería de Python para interactuar con *EVM-compatible blockchains*
  - Documentación
  - Simplifica la tarea de enviar transacciones, interactuar con *smart contracts*, y obtener data de la blockchain
- Importancia de Web3 para el desarrollo de aplicaciones en blockchain
  - Librería más difundida
  - “fork” de Web3.js (muy popular en el desarrollo de web3 en javascript)

4.2 Trabajando con web3

- Métodos

`Eth.get_block`(*block_identifier=eth.default_block, full_transactions=False*)

- Delegates to `eth_getBlockByNumber` or `eth_getBlockByHash` RPC Methods

Returns the block specified by `block_identifier`. Delegates to `eth_getBlockByNumber` if `block_identifier` is an integer or one of the predefined block parameters `'latest', 'earliest', 'pending', 'safe', 'finalized'` - otherwise delegates to `eth_getBlockByHash`.

Throws `BlockNotFound` error if the block is not found.

`Eth.get_transaction_receipt`(*transaction_hash*)

- Delegates to `eth_getTransactionReceipt` RPC Method

Returns the transaction receipt specified by `transaction_hash`. If the transaction cannot be found throws `web3.exceptions.TransactionNotFound`.

If `status` in response equals 1 the transaction was successful. If it is equals 0 the transaction was reverted by EVM.

`Eth.call`(*transaction, block_identifier=web3.eth.default_block, state_override=None, ccip_read_enabled=True*)

- Delegates to `eth_call` RPC Method

Executes the given transaction locally without creating a new transaction on the blockchain. Returns the return value of the executed contract.

The `transaction` parameter is handled in the same manner as the **`send_transaction()`** method.

`Eth.send_transaction`(*transaction*)

- Delegates to `eth_sendTransaction` RPC Method

Signs and sends the given `transaction`

The `transaction` parameter should be a dictionary with the following fields.

- `from` : `bytes or text`, checksum address or ENS name - (optional, default: `web3.eth.defaultAccount`) The address the transaction is sent from.
- `to` : `bytes or text`, checksum address or ENS name - (optional when creating new contract) The address the transaction is directed to.
- `gas` : `integer` - (optional) Integer of the gas provided for the transaction execution. It will return unused gas.
- `maxFeePerGas` : `integer or hex` - (optional) maximum amount you're willing to pay, inclusive of `baseFeePerGas` and `maxPriorityFeePerGas`. The difference between `maxFeePerGas` and `baseFeePerGas + maxPriorityFeePerGas` is refunded to the user.
- `maxPriorityFeePerGas` : `integer or hex` - (optional) the part of the fee that goes to the miner
- `gasPrice` : `integer` - Integer of the gasPrice used for each paid gas **LEGACY** - unless you have a good reason to use `gasPrice`, use `maxFeePerGas` and `maxPriorityFeePerGas` instead.
- `value` : `integer` - (optional) Integer of the value send with this transaction
- `data` : `bytes or text` - The compiled code of a contract OR the hash of the invoked method signature and encoded parameters. For details see Ethereum Contract ABI.
- `nonce` : `integer` - (optional) Integer of a nonce. This allows to overwrite your own pending transactions that use the same nonce.

If the `transaction` specifies a `data` value but does not specify `gas` then the `gas` value will be populated using the **`estimate_gas()`** function with an additional buffer of `100000` gas up to the `gasLimit` of the latest block. In the event that the value returned by **`estimate_gas()`** method is greater than the `gasLimit` a `ValueError` will be raised.

- Filtros

Filters

The following methods are available on the `web3.eth` object for interacting with the filtering API.

`Eth.filter`(*filter_params*)

- Delegates to `eth_newFilter`, `eth_newBlockFilter`, and `eth_newPendingTransactionFilter` RPC Methods.

This method delegates to one of three RPC methods depending on the value of `filter_params`.

- If `filter_params` is the string `'pending'` then a new filter is registered using the `eth_newPendingTransactionFilter` RPC method. This will create a new filter that will be called for each new unmined transaction that the node receives.
- If `filter_params` is the string `'latest'` then a new filter is registered using the `eth_newBlockFilter` RPC method. This will create a new filter that will be called each time the node receives a new block.
- If `filter_params` is a dictionary then a new filter is registered using the `eth_newFilter` RPC method. This will create a new filter that will be called for all log entries that match the provided `filter_params`.

This method returns a `web3.utils.filters.Filter` object which can then be used to either directly fetch the results of the filter or to register callbacks which will be called with each result of the filter.

When creating a new log filter, the `filter_params` should be a dictionary with the following keys.

- `fromBlock`: `integer/tag` - (optional, default: "latest") Integer block number, or one of predefined block identifiers "latest", "pending", "earliest", "safe", or "finalized".
- `toBlock`: `integer/tag` - (optional, default: "latest") Integer block number, or one of predefined block identifiers "latest", "pending", "earliest", "safe", or "finalized".
- `address`: `string` or list of `strings`, each 20 Bytes - (optional) Contract address or a list of addresses from which logs should originate.
- `topics`: list of 32 byte `strings` or `null` - (optional) Array of topics that should be used for filtering. Topics are order-dependent. This parameter can also be a list of topic lists in which case filtering will match any of the provided topic arrays.

- Contratos

If `address` is *not* provided, the newly created contract class will be returned. That class will then be initialized by supplying the address.

```
from web3 import Web3

w3 = Web3(...)

Contract = w3.eth.contract(abi=...)

# later, initialize contracts with the same metadata at different addresses:
contract1 = Contract(address='0x000000000000000000000000000000000000dEaD')
contract2 = Contract(address='mycontract.eth')
```

`ContractFunction.transact`(*transaction*)

Execute the specified function by sending a new public transaction.

Refer to the following invocation:

```
myContract.functions.myMethod(*args, **kwargs).transact(transaction)
```

The first portion of the function call `myMethod(*args, **kwargs)` selects the appropriate contract function based on the name and provided argument. Arguments can be provided as positional arguments, keyword arguments, or a mix of the two.

The end portion of this function call `transact(transaction)` takes a single parameter which should be a python dictionary conforming to the same format as the `web3.eth.send_transaction(transaction)` method. This dictionary may not contain the keys `data`.

If any of the `args` or `kwargs` specified in the ABI are an `address` type, they will accept ENS names.

If a `gas` value is not provided, then the `gas` value for the method transaction will be created using the `web3.eth.estimate_gas()` method.

Returns the transaction hash.

```
>>> token_contract.functions.transfer(web3.eth.accounts[1], 12345).transact()
"0x4e3a3754410177e6937ef1f84bba68ea139e8d1a2258c5f85db9f1cd715a1bdd"
```

`ContractFunction.call`(*transaction, block_identifier='latest'*)

Call a contract function, executing the transaction locally using the `eth_call` API. This will not create a new public transaction.

Refer to the following invocation:

```
myContract.functions.myMethod(*args, **kwargs).call(transaction)
```

This method behaves the same as the `ContractFunction.transact()` method, with transaction details being passed into the end portion of the function call, and function arguments being passed into the first portion.

Returns the return value of the executed function.

```
>>> my_contract.functions.multiply7(3).call()
21
>>> token_contract.functions.myBalance().call({'from': web3.eth.coinbase})
12345  # the token balance for `web3.eth.coinbase`
>>> token_contract.functions.myBalance().call({'from': web3.eth.accounts[1]})
54321  # the token balance for the account `web3.eth.accounts[1]`
```

Módulo V
Uniswap V2

5.1 Introducción a Uniswap v2

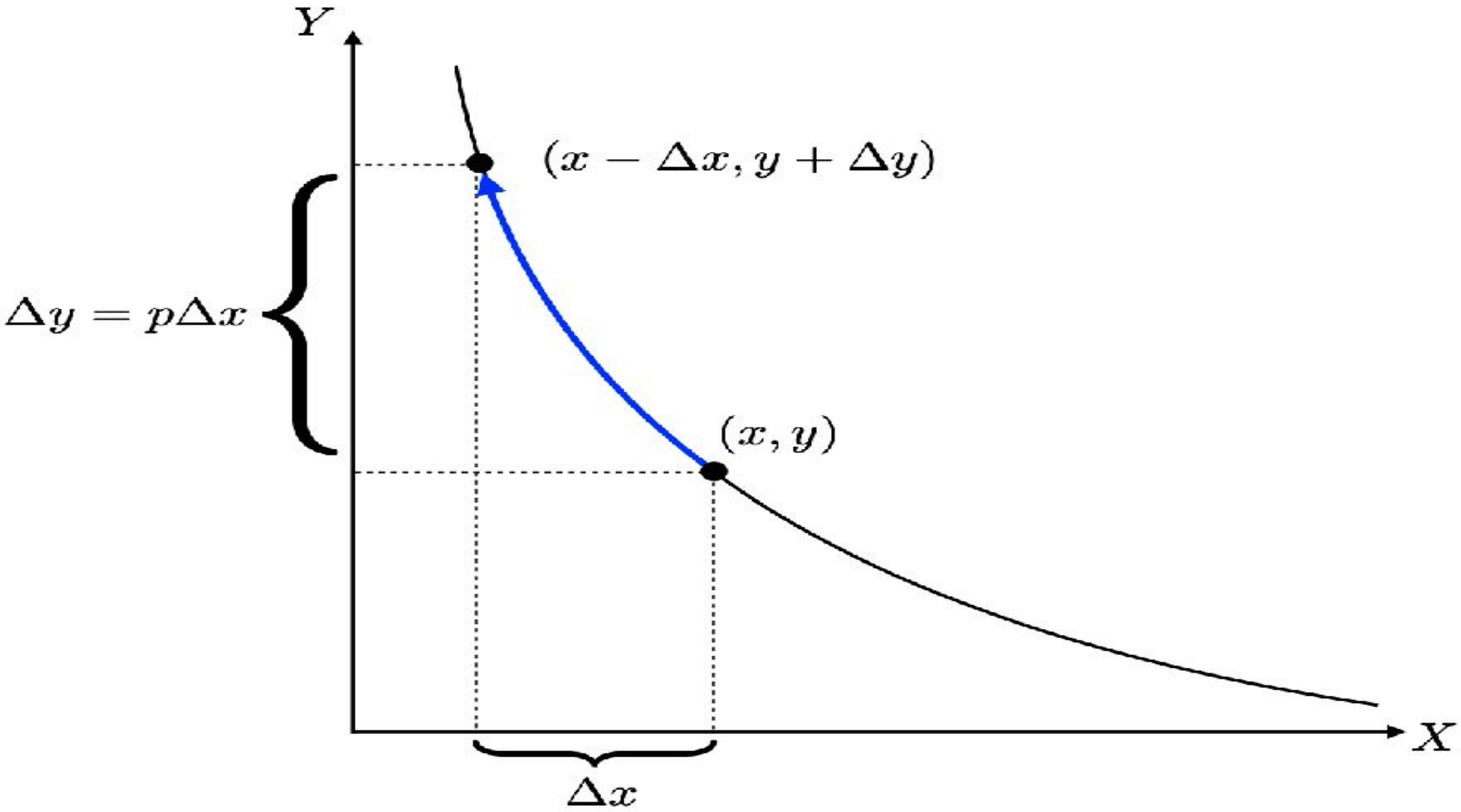
- Whitepaper
- Plataforma descentralizada para intercambio de tokens.
- Utiliza un modelo de "Automated Market Maker" (AMM).
- No requiere libros de órdenes ni intermediarios.
- Los proveedores de liquidez (LPs) depositan pares de tokens y ganan comisiones por transacciones.
- El precio se ajusta automáticamente a través de un algoritmo de oferta y demanda.

5.2 Modelo de Uniswap v2 y su fórmula

- La fórmula básica de Uniswap v2 es:

$$x * y = k$$

- x: cantidad del primer token (ej. ETH).
- y: cantidad del segundo token (ej. DAI).
- k: constante que se mantiene fija.
- La fórmula garantiza que el producto x*y se mantenga constante después de cada transacción.
- El precio del token se ajusta de acuerdo con el balance de tokens en el pool.


Y
$(x - \Delta x, y + \Delta y)$
$\Delta y = p\Delta x$
(x, y)
X
Δx

5.3 Fórmulas clave y transacciones

- Precio de intercambio:

$$k = (x - dx)\ (y + dy)$$

$$\mathbf{dx(y, k, x, dx) = x - k / (y + dy)}$$

- *Slippage*: Cambio en el precio debido a la relación entre el tamaño de la transacción y el tamaño del pool.
- Comisiones para LPs: Los proveedores de liquidez reciben generalmente un 0.30% por transacción.
- Impacto de las grandes transacciones: Transacciones grandes afectan más el precio y el deslizamiento.

Hands on: *real world application*

- Abrir los docs de sushiswap
- Buscar los addresses del router
- Buscar el ABI del router
- Crear el contrato
- Usar el método *quote* del router
- Buscar en el router un swap
- Buscar el ABI del par de UNIV2
- Crear el contrato
- Buscar el contrato de un ERC20 token
- Crear un filtro

Conclusión y resumen

Conclusión y resumen

- Módulo 1:
  - Ideas generales de blockchain: descentralización, seguridad, etc.
  - *Smart contracts!*
  - Test chains, importantes!
- Módulo 2:
  - Transacciones en eth y ERC20
  - Interacciones con smart contracts
- Módulo 3:
  - RPC: providers
  - Signers
- Módulo 4:
  - Web3
  - Interacciones con la blockchain!
- Módulo 5:
  - Intro a Uni V2

Tarea propuesta

Tarea propuesta

- Diseñar una interfaz que le permita al usuario:
  - Obtener un *quote* de un trade: dado un token de entrada, una cantidad de entrada, y un token de salida, calcular la cantidad de tokens que se recibirán si se ejecuta un trade en ese pool
  - Obtener un precio actual: dado un token *base token* y un token *quote token*, obtener el precio
  - Realizar un swap: dado un path (array de addresses), una cantidad de entrada y una cantidad mínima de salida, hacer un intercambio
    - ¿Por qué es importante la cantidad mínima de salida?
    - ¿Funciona para eth/MATIC?
    - ¿Qué hay que hacer con los tokens antes de hacer un swap?