DEVELOPING DAPPS IN ETHEREUM: THEORY AND PRACTICE

4TH SCIENTIFIC SCHOOL ON BLOCKCHAIN & DISTRIBUTED LEDGER TECHNOLOGIES

ANDREA PINNA - ROBERTO TONELLI

CAGLIARI 12-15 SEPTEMBER 2023

DEVELOPMENT OF DECENTRALIZED APPLICATIONS

CONTENTS

DAPP

A decentralized application (dApp) is a software system implemented through the use of blockchain (or more generally distributed ledger) technologies.

This means that a dApp is characterized by having its logic based on smart contracts (or programs) executed within the blockchain. This makes the application decentralized and capable of inheriting the characteristics of the blockchain.

DAPP

There are many decentralized application projects today.

There are portals for investors that resemble those of ICOs,

https://dappradar.com
https://www.dapp.review/explore
https://www.dappt.io/

DAPP

In practice, a dApp is a system composed of **multiple components,** some of which may be non-decentralized, for example, storage and calculation. However, today there are solutions for decentralized computing (nuNet) and decentralized storage (ipfs, filecoin).

In general, a **dApp is composed of**:

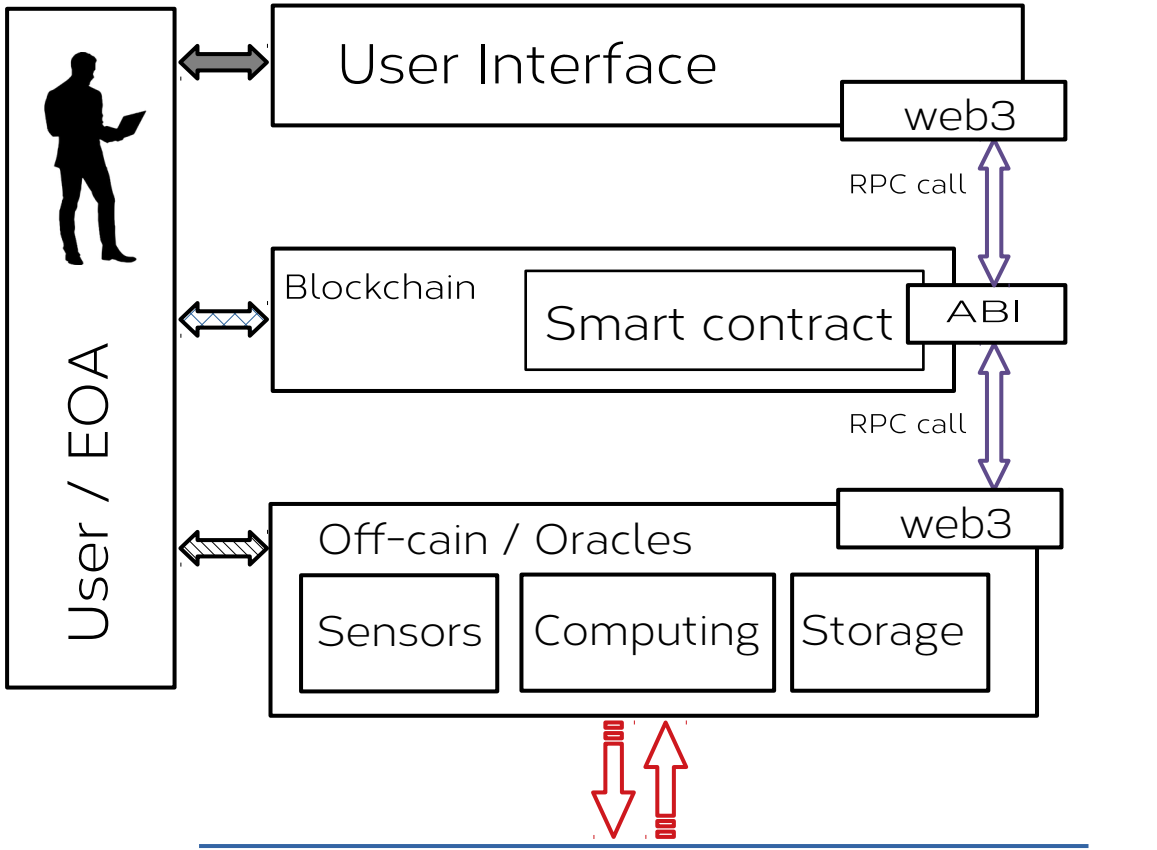
One or more **on-chain** components

One or more **off-chain** components.

The components communicate with each other **via interfaces** and communications based on the internet.

DAPP

Example of Architecture



DAPP

One of the most relevant aspects of dApps concerns the communication method between the different components.

In general, access to the blockchain component by off-chain components depends on:

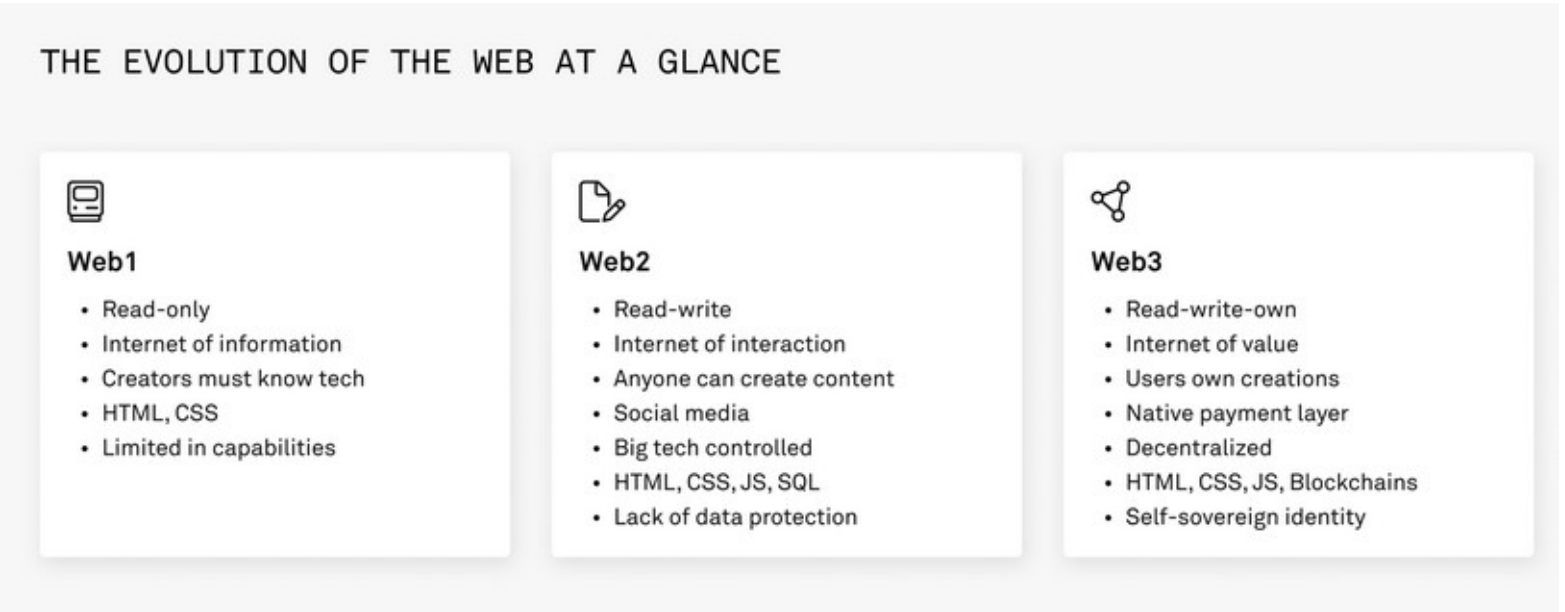
- knowledge of the ABI of contracts (or the equivalent in other CBs)

- creation of a connection (for example RPC or websoket type) to a node or provider.

- use of libraries, which allow communication with the node using the procedures exposed by it (in general we talk about web3 libraries)

WEB 3

All programmable blockchains have a set of APIs available with which the dApps can interact. For different programming languages there are web3 libraries that allow interaction with remote blockchain nodes. We talk about web3 to indicate the third evolution of the internet.



WEB 3 LIBRARIES

The list of ethereum RPCs is available at this link,

https://ethereum.org/it/developers/docs/apis/json-rpc/

Web3 libraries simplify communication with nodes. There are different implementations of the web3 libraries for different languages.

The Ethereum community has collected in two pages all **the libraries** and services useful for the **development of dApps.**

https://ethereum.org/en/developers/docs/apis/javascript/

https://ethereum.org/en/developers/docs/apis/backend/

API WEB 3

Web3 includes:

Gossip methods (to know data regarding the head of the blockchain and for future transactions)

State methods (for querying the state of the blockchain and smart contracts)

History methods (to query data in the blockchain, to learn about blocks, transactions, transaction receipts, etc.)

It also includes **utility** calls like:

Sha3 (returns the keccak-256 of the given data)

LOCAL DEV. TOOLS

DApp development can be done completely locally, before **migrating** the on-chain component to a public or private blockchain. This operation is facilitated by collections of tools and dedicated environments.

The most used suites are **Truffle+Ganache and Hardhat**.

Both allow for the creation of a local blockchain network. The development and compilation of Solidity smart contracts and **testing**.

Both are command line based (**CLI**)

There are also **dashboards** for monitoring (ganache has its own graphical interface).

WEB 3 WITH PYTHON

The web3.py library is available on **Python** which allows you to communicate with a compatible node.

The documentation is available at this link;

https://web3py.readthedocs.io/en/latest/

The **installation** is performed with:

```
$ pip install web3
```

WEB 3 WITH PYTHON

The web3.py library is available on **Python** which allows you to communicate with a compatible node.

The documentation is available at this link;

https://web3py.readthedocs.io/en/latest/

The **installation** is performed with:

```
$ pip install web3
```

WEB 3 WITH PYTHON

In a python file you can import the web3 library

To use it you need to create a connection. The connection via http provider requires the server to be of type JSON-RPC.

```
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.
```

Besides HTTPProvider, you can use IPCProvider, WebsocketProvider and AsyncHTTPProvider

WEB 3 WITH PYTHON

The Web3 class provides the interfaces provided by the Ethereum API.
https://ethereum.org/it/developers/docs/apis/json-rpc/

Example: Let's create a python accountCreation file and use the eth.account.create() function to create a new account.

```
from web3 import Web3, EthereumTesterProvider
w3 = Web3(EthereumTesterProvider)
account = w3.eth.account.create()
print("address:",account.address)
print("private key:", account.key.hex())
```

We can check the connection.

```
if not w3.is_connected():
    raise ConnectionError("Connection failed")
```

EXERCISE

`w3.eth.get_transaction(hash)` takes as input a string containing a hash and returns the details of a validated transaction.

- Create a python program that connects via web3 to the Sepolia network.
- The program prints on the screen the result of the call to the get_transaction API to which the hash of a Sepolia transaction is passed.

https://etherscan.io/

Example of valid transaction hash

0x3ba794db80584bb2d5cb7484a85ec968c4a6144e729bead05b80bcf194ba460f

EXERCISE - SOL

```
from web3 import Web3

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

print(w3.eth.get_transaction('0x3ba794db80584bb2d5cb7484a85ec968c4a6144e729bead
05b80bcf194ba460f'))
```

EXERCISE

Based on the web3py documentation, find web3 methods to know the latest block mined and know the details of that block.

Create a python file that connects to the sepolia node via web3.

Store the height of the last block in a block variable

Record the details of the block in a variable blockData.

Extract and print the hash and timestamp from blockData.

Note: the data can be converted into dicts.

EXERCISE – SOL.

```
from web3 import Web3

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

block = w3.eth.get_block_number()
blockData = w3.eth.get_block(block)

print(dict(blockData).keys())
print('hash = ', blockData['hash'].hex(),'\ntimestamp = ',blockData['timestamp'])
```

Note: .hex() allows you to view the HexByte type data recorded in the hash field.

WEB 3 - ACCOUNT

Sending transactions requires **complete control** of at least one account and therefore of its private key. Account management is provided by the web3.eth.account class

Account management on web3 can be done in three ways.

- **Account generation**: it is possible to create a random account based on an entropy value of your choice.

- **Via node** or provider: the node (or provider as infura, or via bridge as metamask) provides you with your accounts and allows you to sign transactions via the data recorded within it.

- **Via private key**: the account is generated locally using the private key (suitably stored on your device).

WEB 3 - ACCOUNT

To **generate an account** we use the create account method.

```
new_account = w3.eth.account.create('valore di entropia')
print(new_account)
```

You obtain a signer object, useful for signing transactions.

```
<eth_account.signers.local.LocalAccount object at 0x7f8a9087a920>
```

The object contains the attributes **address** and **key**, and methods **sign_message()**, **sign_transaction()**, **encrypt()**.

WEB 3 - ACCOUNT

You can **encrypt** a generated account and record the data in a file.

Example: we record the private key of an account inside the key.json file using the encrypt method.

```
accountData = w3.eth.account.encrypt(privateKey, mypassword)
with open("key.json", "w") as file:
    file.write(json.dumps(accountData))
```

The decrypt method allows you to reconstruct the account using the password used in encrypt.

WEB 3 - ACCOUNT

If we are **connected to a node** in our machine we can recover our accounts and use them directly from the application.

Once connected, the list of account addresses is available from eth.accounts.

Example: the first account in the list will be:

w3.eth.accounts[0]

All accounts in the list are already present in memory and can be used to sign transactions.

WEB 3 - ACCOUNT

Having a **private key** you can take control of an already created account.

We use the "**from_key**" method of account to import it.

The method to use is

```
eth.account.from_key(private_key)
```

Example:

```
privateKey_account1 = "bc0acc845ed3dd70256b132.....5e328bc"
account1 = w3.eth.account.from_key(pk_account1)
```

WEB 3 - ACCOUNT

It is highly **discouraged** to include the private key in a script that we intend to share. We can record it to an env file. or in an encrypted file not to be included in the project, .

Example: If the private key is inside an **env** file:

```
import os
pk = os.environ.get('PRIVATE_KEY')
```

If the key is in an **encrypted file** as seen above, you must decrypt it first.

Example: if the key is encrypted in the file `key.json`

```
with open("key.json", "r") as file:
    data = json.loads(file)
pk =  w3.eth.account.decrypt(data,password)
```

EXERCISE

From metamask: Export the private key of one of the accounts that has non-zero balance on Seopolia.

Create a python script that allows you to encrypt your private key with a password of your choice.

Create another script to retrieve the private key and use it to generate the account object.

Print the balance of this account on sepolia.

WEB 3-SIMPLE TRANSACTION

Web3 provides a full control on transaction parameters. A transaction is a dictionary having some specific keys, including:

to, value, gas, maxFeePerGas, 'maxPriorityFeePerGas', nonce, chainId.

```
transaction = {
    'to': '0xF0109f...5aA624EaC1F55',
    'value': 1000000000,
    'gas': 2000000,                    # = w3.eth.estimate_gas(transaction)
    'maxFeePerGas': 200000,
    'maxPriorityFeePerGas': 100000,# = web3.eth.max_priority_fee
    'nonce': 0,              # = w3.eth.get_transaction_count(myaccount.address)
    'chainId': 1,
    'data': b''                        # bytecode della transazione,
}
```

WEB 3-SIMPLE TRANSACTION

Some of the transaction parameters may vary based on the state of the blockchain or the transaction itself.
We can modify or add elements to the transaction dictionary via the **update** method.

Nonce: The number of transactions created by the account.

Gas: maximum gas for this transaction. It can be calculated after creating the transaction dictionary and then updating it.

MaxFeePerGas. It corresponds to the maximum base Fee that we want to pay and is dictated by the state of the network, depending on whether or not the target has been achieved. We can determine it starting from that of the last mined block.

```
block = w3.eth.get_block('latest')
print(block.baseFeePerGas)
transaction.update({'maxFeePerGas': block.baseFeePerGas * 1.10})
```

Gasprice: If supported, use instead of `maxFeePerGas`.

```
gasprice = w3.eth.gas_price
```

WEB 3-SIMPLE TRANSACTION

Example.

- Defined part: **chainId, from, to, value**

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

- Variable part: **gas, gasprice, 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
```

WEB 3-SIMPLE TRANSACTION

A transaction **must be signed** using a private key.

```
signed_tx = w3.eth.account.sign_transaction(transaction, fromAccount.key)
```

What you get is a **transaction object** that contains the "rawTransaction" with the transaction data, hash and signature elements.

To execute the transaction you need to use the send_raw_transaction method which takes the rawTransaction as its argument.

```
tx_hash = w3.eth.send_raw_transaction(signed_tx.rawTransaction)
```

Now, to confirm, just **wait for the receipt** using the appropriate method.

```
tx_receipt = w3.eth.wait_for_transaction_receipt(tx_hash, timeout=120, poll_latency=0.1)
```

Note: Timeout and poll_latency are optional

WEB 3 - TRANSACTIONS

Once we have the receipt we can, among other things, **analyze the gas used**.

```
print('cumulativeGasUsed =', tx_receipt['cumulativeGasUsed']) #blocco
print('gasUsed =', tx_receipt['gasUsed']) # tx
```

Note:

If the transaction is already validated it is possible **to read the receipt** from the blockchain without waiting for it.

```
tx_receipt = w3.eth.get_transaction_receipt(transaction_hash)
```

WEB 3 – TRANSACTIONS DEMO

DEMO: SimpleSend.py

WEB 3 - CONTRACTS

WEB 3 - CONTRACTS

Web3 allows you to interact with contracts in **different ways**:

- contracts **already present** in the blockchain: given the address and the abi we can create a reference to the contract in the blockchain.

- creation of **new contracts**: given the bytecode and the ABI it is possible to deploy a contract on the blockchain and obtain its address.

Through contract references we can **perform**:

- **static-call** to pure or view functions,
- **transactions** to activate other functions.

WEB 3 - CONTRACTS

To interact with existing contracts and capture events **we need the contract's ABI**.

Next we need a **local contract instance**, to which we pass the contract address and the abi.

It is possible to **get the ABI** on remix (copying it to the clipboard after compilation), or by **using solc** or Truffle after the compilation.

WEB 3 - CONTRACT

Compiling on Python.

In general, if we have the source file of the contract (.sol) we can generate the abi by compiling it. In this way we just need to have the sol file without having to use other tools.

To compile a solidity file you need to install the solcx library

COMPILING SMART CONTRACTS

On python it is possible to compile a solidity smart contract via the **py-solcx library** (to be installed).
First you need to install the desired compiler version.

```
import solcx
solcx.install_solc('0.8.18') # install the 0.8.18
```

Assuming that scSource is a string containing the solidity source:

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

From the contract_interface we get abi and bytecode.

```
abi = contract_interface['abi']
bytecode = contract_interface['bin']
```

WEB 3 - CONTRACTS

Web3 allows you to interact directly with an **already deployed contract** via a contract instance.

Given an address and the abi we can create a **“contract” object** (a local instace of the contract) using the following sintax:

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

WEB 3 – CALL CONTRACTS

Once generated, we can **interact with the contract instance**.

The functions attribute of a contract allows you to select one of the public functions of the contract, present in the abi.

For calls to **view or pure functions** I can use the **call() method** with the following syntax:

```
results = contractInstance.functions.functionName(parameters).call()
```

Esempio:

```
mycontract = w3.eth.contract(address=contractAddress, abi=contractABI)
result = mycontract.functions.get().call()
print(result)
```

EXERCISE

On **remix**, deploy a contract on the Sepolia testnet that exposes a **"getBalanceOf" function** which returns the balance of an address passed per argument.

Create a python script that:

- connects to the Sepolia network
- Create a local instance of the contract (via the **abi** and the **address**)
- call the getBalanceOf function passing an address as argument.

WEB3 – CONTRACTS DEPLOY

Deploying a contract requires:
- the ABI of the contract,
- The bytecode of the contract,
- the list of the contructor parameters.

The transaction does not require the destination address (**it will be generated**).

WEB3 – CONTRACT DEPLOY

After created the local instance via **abi** and **bytecode**, you can create the **deployment transaction** via the **build_transaction** method of a contract constructor.

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

WEB3 – CONTRACT DEPLOY

Example:

From remix, we compile the **1_storage.sol contract (one of default contracts)** and copy the abi and the bytecode onto two appropriate python variables.

```
abi = json.load( '…..' )
bytecode = '6080604 … 033'
```

Then we create the contract object for deployment.

```
contract = w3.eth.contract(abi=abi, bytecode=bytecode)
```

WEB3 – CONTRACT DEPLOY

Example (cont)

Let's create the transaction for the deployment

```
transaction = mycontract.constructor(*params).build_transaction(
        {"chainId": 11155111,
          "value": value,
         })
```

Let's add the nonce info:

```
transaction.update({ 'nonce' : w3.eth.get_transaction_count(account1.address) })
```

And we sign the transaction with our private key.

```
signed_tx = w3.eth.account.sign_transaction(transaction, PK)
```

WEB3 – CONTRACT DEPLOY

Example (cont)

Finally we send the transaction:

```
tx_hash = w3.eth.send_raw_transaction(signed_tx.rawTransaction)
```

And we wait for the receipt:

```
tx_receipt = w3.eth.wait_for_transaction_receipt(tx_hash)
```

Once we have obtained the receipt, (that contains the address) we can interact with the contract, creating the instance.

```
contract = w3.eth.contract(address=tx_receipt["contractAddress"], abi=abi)
```

WEB3 – FUNCTION EXECUTION

WEB3 – CONTRACT EXECUTION

To execute transactions and change the status of contracts you need to create a transaction containing the message and data.

Like for contract deployment, we use the **build_transaction** method that exists for every function of the smart contract.

This takes the transaction data (the dictionary) as argument.

The transaction so created will need to be signed.

WEB3 – CONTRACT EXECUTION

Example: we create a transaction to execute the "functionName" function to which we pass a list of Parameters.

```
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)
```

WEB3 – CONTRACT EXECUTION

The example in the previous slide shows the creation of a call to a SC where we set the name of the function and the list of parameters.

```
transaction = mycontract.functions.FunctionName(Parameters).build_transaction({ ... })
```

We can create a **'meta-transaction'** where the function name and parameter list can be variables, we can use Python's getattr function.

Example: **FUNCTION** is a **string**, ***parameters** is a **tuple**.

```
transaction = getattr(contract.functions, FUNCTION)(*parameters).build_transaction({...})
```

This way I can use the same piece of code to create multiple transactions.

EXERCISE

You want to update the contract “certification” we discussed previously to allow a professor to sign a certificate offline and allow the contract owner to publish the signed certificate.

In this way, the professor does not have to pay gas for signing the certificate..

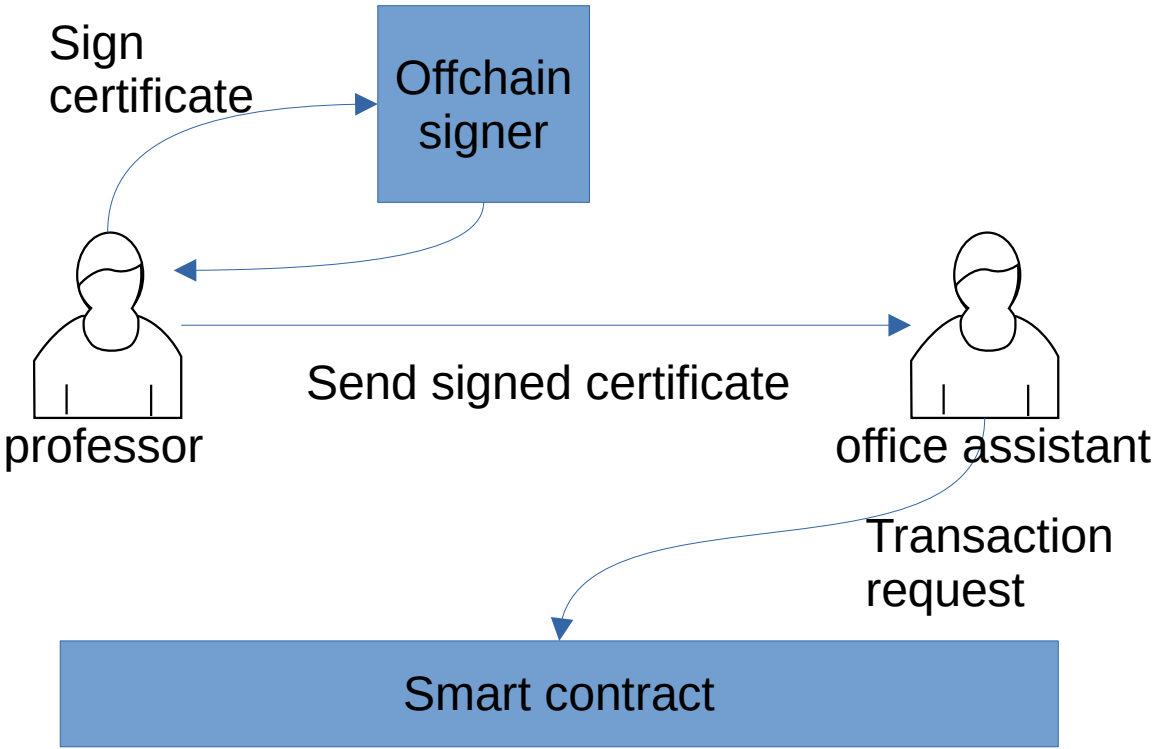
- Add the functions to check the signature (by using openzeppelin)

- In pyton, write the code to sign the hash of a message. The signature will be verified in the contract.

Demo: SimpleSignature

EXERCISE

The schema is similar to the following:



EXERCISE

On solidity, update the solidty file to create two functions:

```
function createCertificate(address professor, bytes memory signature, address attendee, uint8
grade) public onlyOwner{
        require(professors[professor], "invalid professor");
        require(attendance[attendee], "invalid attendee");

        bytes32 hash = keccak256(abi.encodePacked(attendee, grade));
        require(checkSignature(professor, hash, signature), "invalid signature");
        saveCertificate(professor, signature, attendee, grade);
    }

    // this function uses the Openzeppelin library for signature verifiaction, in a easier
way.
function checkSignature(address signer, bytes32 hash, bytes memory signature) private view
returns(bool)
    {
        return SignatureChecker.isValidSignatureNow(signer, hash, signature);
    }
```

EXERCISE

What you want to sign is a piece of data composed of:

- address of the attendee (address)
- grade (uint8)

In particular, via private key, the professor signs the hash generated by the **solidity_keccak** method.

```
dataHash=w3.solidity_keccak(['address','uint8'], [attendee,grade])
hash_str = dataHash.hex()

signed_message = w3.eth.account.signHash(hash_str, account1.key)
sign = signed_message.signature.hex()
```

DEMO: certificates_signature.sol

WEB3 – EVENTS

WEB3 – EVENT MANAGEMENT

Event management is a process that consists of "**listening**" to the events emitted by smart contracts and then performing specific operations when specific events occur. The operation is based on **event filtering**.

To detect an event emitted by a smart contract you need to **create a filter** instance using the **create_filter** method

WEB3 – EVENT MANAGEMENT

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 })
```

For Details
https://web3py.readthedocs.io/en/stable/web3.contract.html#web3.contract.Contract.events.your_event_name.create_filter

WEB3 – EVENT MANAGEMENT

A filter object created in this way can be used to **handle events**.

For example, we can create a program that listens for an event via the filter.

Example: We use the **get_new_entries()** method to get the list of all events emitted since the last reading.

```
while True:
    for event in event_filter.get_new_entries():
        print(event) # What to do  in case of event detection
    time.sleep(2)  # sleeping time
```

WEB3 – EVENT MANAGEMENT

An event captured by the application is a dictionary.

Example: we captured an event named signal with argument "val" of value 10

```
AttributeDict(
  {'args': AttributeDict({'val': 10}),
   'event': 'signal',
   'logIndex': 61,
   'transactionIndex': 44,
   'transactionHash':
HexBytes('0x1652fa98d2233a1c55ec35f1f3ba7b40db095baced5a792b4b67264b8a63bf34'),
   'address': '0xa33DC7C302DDAd420a12b1005b9c3905Be81Cf70',
   'blockHash':
HexBytes('0x8043bf6c72fe5bd5e1811d77e4d2396d3b26c4a948b6d7a90c52e82122ee6074'),
   'blockNumber': 3622637}
)
```

WEB3 – OTHER FILTERS

Filter objects can also be created for **gossip** tasks such as monitoring pending transactions and new blocks.

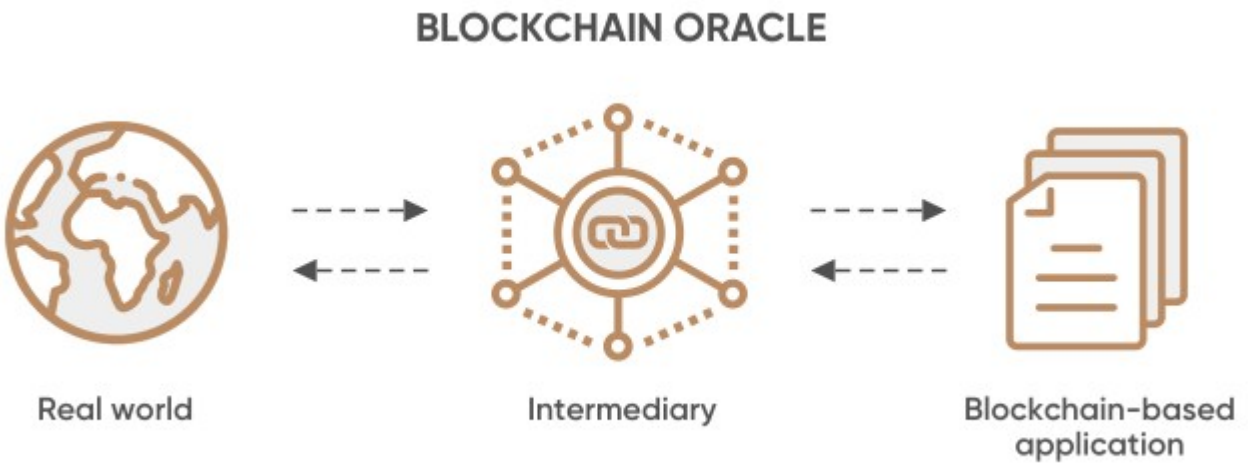
For these types of filters, the w3.eth.filter() method is used which takes as an argument a string for the type of activity you want to monitor.

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

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

ORACLE CREATION

Blockchain and smart contracts **cannot read data directly from outside**.
This limitation is known as the **oracle problem**.
An oracle is a system designed to provide external data to a smart contract, in order to solve the oracle problem.



ORACOLE CREATION

We **create a simple oracle** capable of providing a smart contract with data on request.

The system will consist of three components:

- a user application that reads from the smart contract,

- the smart contract, which records data and provides it on request.

- an oracle, consisting of an external service capable of sending updated data to the smart contract

DEMO: simpleOracle

ORACOLE CREATION

Let's write a smart contract responsible for providing data:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;
contract MyOracle {
    address owner;
    string data;
    uint256 lastUpdate;
    event dataRequest(address);

    constructor(){
        owner = msg.sender;
    }
    function get() public view returns(string memory, uint256){
        return(data, lastUpdate);
    }
    function request() public {
        emit dataRequest(msg.sender);
    }

    function update(string memory _data) public {
        require(owner == msg.sender, "only the contract creator");
        data = _data;
        lastUpdate = block.timestamp;
    }
}
```

ORACLE CREATION

The user application is a program that reads the data and executes the request to update the data in time.

```
...
# 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()
```

ORACLE CREATION

In this example, the oracle service is a running program that

- **Automatically send**s updated data to the smart contract, every 60 seconds.
- **Captures the dataRequest** event and immediately sends the updated data to the contract.

We use **thread management** to make the same program perform both operations.

In this example we use random data as updated data. This of course could be any data taken from
external sources and resources such as dbms or from API.

ORACLE CREATION

Oracle logic. In this example, the data is generated randomly. In practical applications the oracle will be able to read data from external sources.

```
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)
```

ORACLE CREATION

Auto update function.

```
def update_data():
    while True:
        data = str(random.random())
        print("UPDATE New Data:", data)
        try:
            receipt = function_call(w3, oracleContract, 'update', account1, 0, data)
            print(receipt['transactionHash'].hex())
        except:
            print("Transaction pending")
        time.sleep(60)
```

ORACLE CREATION

Running the treads

```
update_thread = threading.Thread(target=update_data, args=())
listening_thread = threading.Thread(target=listening, args=())

update_thread.start()
listening_thread.start()
```

SIMPLE INTERFACE

Let's build a simple user interface using PySimpleGUI
The buttons on the interface will activate one of the functions of the User
dApp.

The interface for the user application will have:

- A button to read data from the blockchain
- A text to indicate the data received.
- A button to request data updating from the oracle
- A text to indicate the hash of the transaction

SIMPLE INTERFACE

Final result