LESSON MC1 – Intro To `Web3.py`

University of West Attica

Department of Electrical and Electronics Engineering

Ioannis Christidis

Christoforos Kachris

Support by Ethereum Foundation ESP

What will we accomplish!

This is the first lesson that we will be talking about interaction with the blockchain from a microcontroller. In this example we will use a Raspberry Pi. (technically microprocessor)

We will code in python, and we will use the most common library for blockchain interaction called `web3.py`.

`Web3.py` is a Python library for interacting with the Ethereum-based blockchains. It provides tools to connect to nodes, read data from SC, send transactions, and deploy SC.

We will start with a smaller example to show off the basics of `web3.py` and later we will build our solution for the `ICR`.

What will we accomplish! (2)

In the following lessons we will see how to:

1. Connect with blockchain using HTTPS and `WebSocket`
2. Read data from a SC
3. Make transactions on a SC
4. Subscribe to events emitted from a SC
5. Sign messages using an account's private key

In this lessons we will assume that you are already familiar with python so this section will be a bit more advanced.

You will be needing `python3` and `pip` in order to download and use the necessary libraries to follow these lessons.

The external libraries that we are going to use are:

`web3py version 7.6.0`

`python-dotenv version 1.0.1`

Requirements for Blockchain connection and interaction.

In order to, connect to a blockchain we need an RPC `endpoint` URL. For public blockchains like the one that we are going to work with, we can get one for free from a blockchain infrastructure provider like Alchemy. You can choose the network and the type of endpoint like:

HTTPS: https://https://eth-sepolia.g.alchemy.com/v2/YOUR_API_KEY

WebSocket: wss://https://eth-sepolia.g.alchemy.com/v2/YOUR_API_KEY

You will also need a blockchain account that you can get from Metamask.

Finally, you will need some ETH. We are going to be deploying our DApp to Sepolia testnet and to acquire some SepoliaETH, you can visit a faucet like the following:

Chainlink

Alchemy

Metamask

Connecting to blockchain

The first thing that we do to connect to the blockchain is import Web3 to our project.

```
from web3 import Web3
```

Next we can use our HTTPS RPC URL to create a connection to the blockchain like this.

```
HTTPS_URL = "https://eth-sepolia.g.alchemy.com/v2/<YOUR-API-KEY>"
# Create a Web3 connection
web3 = Web3(Web3.HTTPProvider(HTTPS_URL))
```

To check if the connection was successful, you can do the following:

```
# Check if connected
if web3.is_connected():
  print("Connected to the Sepolia testnet successfully!")
  print(f"Chain ID: {web3.eth.chain_id}")
  return web3
else:
  print("Failed to connect to the Sepolia testnet.")
  return None
```

If you run the script above you will receive an output like this:

```
Connected to the Sepolia testnet successfully!
Chain ID: 11155111
```

The **Chain ID** is a unique identifier assigned to a specific blockchain network to distinguish it from other networks.

Connect to Blockchain Function

We will turn the connection into a function that we will use later to the rest of our scripts. So, we will create a file called `connect_to_bc.py`.

```python
from web3 import Web3
def https_connection_to_blockchain():
    # Find a URL
    HTTPS_URL = "https://eth-sepolia.g.alchemy.com/v2/<YOUR-API-KEY>"
    try:
        # Create a Web3 connection
        web3 = Web3(Web3.HTTPProvider(HTTPS_URL))

        # Check if connected
        if web3.is_connected():
            print("Connected to the Sepolia testnet successfully!")
            print(f"Chain ID: {web3.eth.chain_id}")
            return web3
        else:
            print("Failed to connect to the Sepolia testnet.")
            return None
    except Exception as e:
        print(f"An error occurred: {e}")
        return None
```

How to interact with a smart contract.

To interact with a SC, we will need its `address` on the blockchain and its `ABI`.

We will use a SC already deployed on Sepolia testnet which is similar to the `SimpleStorage.sol` we created in the introduction portion of this course, but you can deploy your own if you want to. To get the abi you can paste it on a file in remix, compile it and copy the `ABI` at the bottom of the SOLIDITY COMPILER sidebar.

```
//SPDX-License-Identifier: MIT
pragma solidity 0.8.26;

contract SimpleStorage {

    event NumberSet(uint indexed _storedNumber);

    uint256 storedNumber;

    function setNumber(uint256 _storedNumber) public {
        storedNumber = _storedNumber;
        emit NumberSet(storedNumber);
    }

    function getNumber() public view returns (uint256) {
        return  storedNumber;
    }

}
```

How to interact with a smart contract. (2)

The ABI is in json format so from now on we will store the abis in json files on our project.

We will name the the .json file of SimpleStorage ABI simple_storage_abi.json and we will paste the ABI inside.

We will also create a script called load_json.py that will allow us to load the ABIs to the rest of our files. Just remember to add all the .json files to the root folder.

```
import json

def load_contract_abi(abi_file_path):
    with open(abi_file_path, "r") as file:
        return json.load(file)
```

And with that we can call the ABI of a SC to any file like this.

```
import load_json
abi = load_json.load_contract_abi("abi.json")
```

Read from Smart Contract

To read data from a SC we first need to create an instance of the SC using the `Web3py` library, its `address` and `ABI`.

```
from connect_to_bc import https_connection_to_blockchain
import load_json


w3 = https_connection_to_blockchain()


# smart contract address
contract_address = "0x84606F263db7B839d484d0991d789cC0e688748C"


# smart contract abi
contract_abi = load_json.load_contract_abi("simple_storage_abi.json")


# instantiate contract with address and abi
contract = w3.eth.contract(address=contract_address, abi=contract_abi)
```

Now you can just call any read function from the SC like this:

```
# call getNumer() function from the samrt contract
stored_number = contract.functions.getNumber().call()

# print the result
print(stored_number)
```

Try running it and see what is the number stored on the SC.

Write to Smart Contract

To write on a SC or make a transaction to it you will need to do a few more steps.

First of all, you will need an account that has some ETH to make the transaction.

Each account has an address and a private key (pk). To do the transaction, you will need both.

You can use the account that collected ETH from the faucets.

From MetaMask, you can copy the address of your account and pk.

DISCLAIMER: ONLY USE METAMASK ACCOUNTS FOR TESTING. During testing, you may expose your pks or recovery phrases, either accidentally or in logs/code. If these are compromised, your real funds on the mainnet could be stolen.

NEVER PASTE YOUR MAINNET PRIVATE KEY AS TEXT.

Even the approach we are gonna use is not secure, but we are doing it for testing and we are going to be using a test account.

Write to Smart Contract (2)

ONLY FOR TESTING PURPOSES we will create a `.env` file and we will place the pk there under the name `MY_PRIVATE_KEY`. The address is public so we can paste it directly to our script.

```
# Libraries needed to use the .env file
from dotenv import load_dotenv
import os

from connect_to_bc import https_connection_to_blockchain
import load_json

load_dotenv()

# Address of the user that will make the transaction
MY_ADDRESS='0xf41Fd20b5C7453b9044122115138e541C813ab55'
# Private key of the user that will make the transaction
MY_PRIVATE_KEY=os.getenv("MY_PRIVATE_KEY") # NEVER REVEAL THIS


w3 = https_connection_to_blockchain()


contract_address = "0x84606F263db7B839d484d0991d789cC0e688748C"
contract_abi = load_json.load_contract_abi("simple_storage_abi.json")
contract = w3.eth.contract(address=contract_address, abi=contract_abi)
```

Write to Smart Contract (3)

First, we have to build the transaction.

```
# Build the transaction
# We call the function setNumber and we add as input the number 2 (you can change that)
transaction = contract.functions.setNumber(2).build_transaction({
    "from": MY_ADDRESS,
    "nonce": w3.eth.get_transaction_count(MY_ADDRESS), # The nonce is a unique number assigned to each transaction sent from an
address.
    "gas": 200000, # Gas Limit: The maximum amount of gas you are willing to allow for this transaction (you can change that)
    "gasPrice": w3.to_wei("20", "gwei"), # Gas Price: The price (in wei) you are willing to pay per unit of gas.
})
```

Next, we will sign the transaction using our pk.

```
# Sign the transaction
signed_tx = w3.eth.account.sign_transaction(transaction, MY_PRIVATE_KEY)
```

Finally, we send the transaction to the blockchain.

```
# Send the transaction
tx_hash = w3.eth.send_raw_transaction(signed_tx.raw_transaction)
```

We can also wait for the receipt of the transaction like this:

```
# Wait for the transaction receipt
receipt = w3.eth.wait_for_transaction_receipt(tx_hash)
```

Subscribe to Events

For event subscription we will constantly listen to the blockchain and wait until a specific event is emitted. This means that we will need real-time communication and a persistent connection which is why we are going to use WebSocket.

Get your WebSocket RPC URL from Alchemy.

Now, instead of plain `Web3` we will use `AsyncWeb3` with `asyncio`.

For this part, we will not even need the `ABI` since we are going to specify the event we will subscribe to. We just need the SC address.

```
import asyncio
from web3 import AsyncWeb3, WebSocketProvider


WSS_URL = "wss://eth-sepolia.g.alchemy.com/v2/<YOUR-API-KEY>"
contract_address = "0x84606F263db7B839d484d0991d789cC0e688748C"
```

We are going to build an `async function` with a `try/except statement.`

```
async def subscribe_to_number_set_event():
    try:
        pass
    except Exception as e:
        print(f"Error during subscription: {e}")
```

Subscribe to Events (2)

First, we will define the `async web3` instance with our `WebSocket RPC URL` as parameter.

```
async with AsyncWeb3(WebSocketProvider(WSS_URL)) as w3:
    pass
```

Next, we define the event we want to subscribe to and the SC address that it will be emited from.

```
            number_set_event_topic = w3.keccak(text="NumberSet(uint256)")
            filter_params = {
                "address": contract_address,
                "topics": [number_set_event_topic],
            }
```

Using that we will subscribe to the event.

```
            subscription_id = await w3.eth.subscribe("logs", filter_params)
            print(f"Subscribing to NumberSet event for SimpleStorage at {subscription_id}")
```

Subscribe to Events (3)

Next, we start an asynchronous loop that listens for incoming messages (payloads) from our WebSocket connection.

```
async for payload in w3.socket.process_subscriptions():
    pass
```

We extract the result field from the incoming payload.

```
result = payload["result"]
```

We decode the second topic (the first is always the event signature hash) in the event's data (to do this we will need to `import decode from eth_abi.abi` that comes with `web3.py`).

```
from eth_abi.abi import decode

stored_number = decode(["uint256"], result["topics"][1])[0]
print(f"Event received: Stored Number: {stored_number}")
```

Finally, we unsubsribe from the event.

```
await w3.eth.unsubscribe(subscription_id)
print("Unsubscribed from NumberSet events.")
return
```

You generally want to avoid long time subscribes unless you have your own infrastructure provider that does not charge per request unlike Alchemy.

Subscribe to Events (4)

You can check the way it works by running the script and use remix to make a transaction on the SC. Make sure you do not leave it running for too long as it will burn your free recources from Alchemy.

```python
import asyncio
from web3 import AsyncWeb3, WebSocketProvider
from eth_abi.abi import decode
WSS_URL = "wss://eth-sepolia.g.alchemy.com/v2/<YOUR-URL>"
contract_address = "0x84606F263db7B839d484d0991d789cC0e688748C"
async def subscribe_to_number_set_event():
    try:
        async with AsyncWeb3(WebSocketProvider(WSS_URL)) as w3:
            number_set_event_topic = w3.keccak(text="NumberSet(uint256)")
            filter_params = {
                "address": contract_address,
                "topics": [number_set_event_topic],
            }
            subscription_id = await w3.eth.subscribe("logs", filter_params)
            print(f"Subscribing to NumberSet event for SimpleStorage at {subscription_id}")
            async for payload in w3.socket.process_subscriptions():
                result = payload["result"]
                stored_number = decode(["uint256"], result["topics"][1])[0]
                print(f"Event received: Stored Number: {stored_number}")
                await w3.eth.unsubscribe(subscription_id)
                print("Unsubscribed from NumberSet events.")
                return
    except Exception as e:
        print(f"Error during subscription: {e}")
if __name__ == "__main__":
    asyncio.run(subscribe_to_number_set_event())
```

Signing Messages

Signing messages is a key cryptographic feature used in Ethereum and other blockchain ecosystems to verify ownership of a specific pk without revealing the pk itself. Message signing involves using a pk to create a unique signature for a given message.

This signature can then be used to:

- Prove Ownership: Show that the signer owns the pk without revealing it.
- Authenticate: Verify the message sender's identity in decentralized systems.

In our case, we will use this, later, in our project to ensure that the ICR car can only be unlocked to the current renter.

For now, let's see a minimal example on the next slide.

Signing Messages (2)


```python
from web3 import Web3
from eth_account.messages import encode_defunct
import os
from dotenv import load_dotenv

load_dotenv(override=True)

HTTPS_URL = "https://eth-sepolia.g.alchemy.com/v2/<YOUR-API-KEY>"
# Connect to a Web3 provider
w3 = Web3(Web3.HTTPProvider(HTTPS_URL))
# Private key of the signer
MY_PRIVATE_KEY = os.getenv("MY_PRIVATE_KEY")  # (keep it secure!)
# Message to sign
message = "Hello world"
# Hash the message
message_hash = encode_defunct(text=message)
# Sign the hashed message
signed_message = w3.eth.account.sign_message(
    message_hash, private_key=MY_PRIVATE_KEY
)
# Recover the address from the message using message hash and the signature of the signed message
address = w3.eth.account.recover_message(message_hash, signature=signed_message.signature)

# Print the signature components
print("Message Hash: ", message_hash)
print("Signed message signature: ", signed_message.signature)
print("address used: ", address)
```

Signing Messages (3)

We will name this script `signing_message.py` and we will use it later.

If you run this script, you will receive an output similar to the one presented bellow.

```
Message Hash:  SignableMessage(version=b'E', header=b'thereum Signed Message:\n10', body=b'Hello world')
Signed messae signature:
b'\x1a\xea\xee;a9\n\xc2\xd3\x86\xc0|\x89\xbeL\x1e\xb8\xc6\xd3\xe3\\N\x87\\\x00\x1a5\x8e*\x04\x87\x86IVBU\xb2,\xbf\xe7z\x13[}\x9a\x95w0\xa5\x1e\x9aMU\x87\xba\xd1Q\x9a\xfa\xa3\x8d-!!\x1c'
address used:  0xf41Fd20b5C7453b9044122115138e541C813ab55
```

In the logs above, you can see that if you have the message hash and signed message signature (with pk) you can find the address that signed the message but not the pk of the account.

We will not explain what the above logs are because signing can get way more complicated so we will keep it simple and learn exactly what we need for our project.

Also, for the project we will use a script to get the signed message signature but on an actual application the user should probably use a wallet for signing, like MetaMask.

Outro

Now that we learned the basics of `web3.py` we will use them in our project in the next lesson.