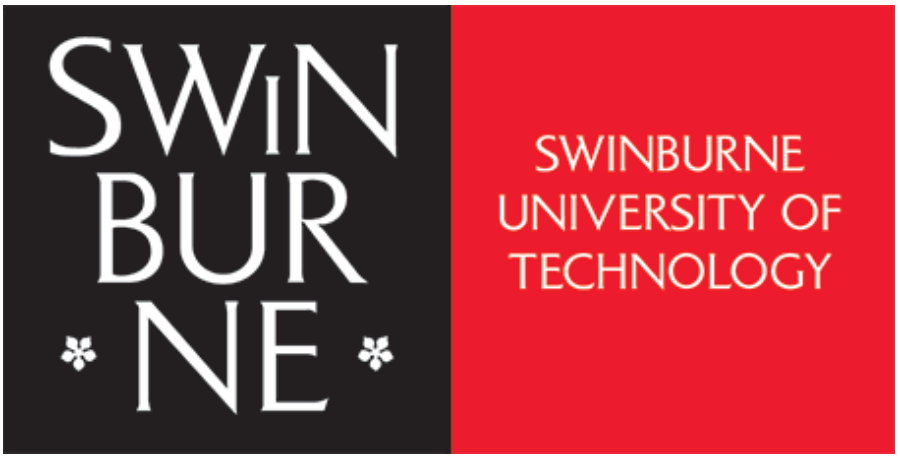


2023-COS30049-Computing Technology Innovation Project

Workshop Guide

Workshop 08 - SE

Deploy and Interact with Smart Contracts

Objective:

This workshop is designed for students who already have a basic understanding of Solidity. In the previous lesson, Solidity Smart Contract Development - Fundamentals, we learned the basic syntax of Solidity. This week, we can use Web3.py to interact directly with our local Ganache node to better understand its principles. This lesson uses Web3.py as an example to implement the basics of compiling a contract, deploying it to the local Ganache network, and interacting with the contract.

Workshop Structure:

Part 1: Introduction & Installation of Ganache (20 minutes)

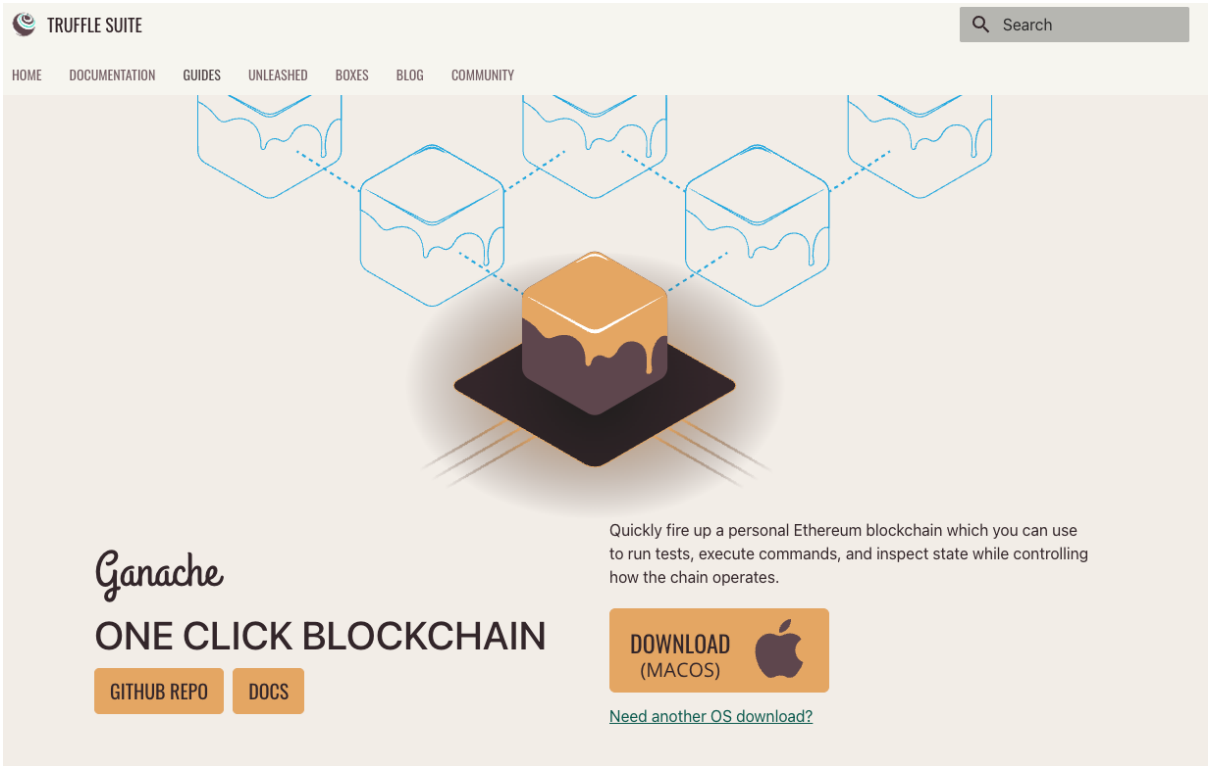
Ganache is a personal blockchain simulator and debugging tool for Ethereum blockchain development. It is an important tool in the Ethereum developer ecosystem, primarily used to simulate Ethereum blockchain networks locally, allowing developers to develop, test, and debug smart contracts and decentralized applications (DApps) without spending real Ether.

Why Ganache:

- **Local Blockchain Simulator**: Ganache enables developers to create a local Ethereum blockchain network that is not connected to the real Ethereum mainnet or test networks but runs entirely locally. This allows developers to iterate and test their smart contracts and DApps more quickly.
- **Rapid Development and Testing**: Ganache provides a quick way to create Ethereum blocks, process transactions, and simulate various Ethereum network scenarios, helping developers iterate and test their smart contracts and DApps rapidly.
- **Visual Interface:** Ganache also has a user-friendly visual interface that displays detailed information about the local blockchain network, including accounts, transaction history, and block data, making it easy for developers to monitor and debug their applications.
- **Built-in Accounts and Test Ether:** Ganache provides virtual Ethereum accounts and test Ether when it starts, which can be used to simulate transactions and smart contract execution.

- **Integrated Development Environment (IDE) Compatibility:** Ganache seamlessly integrates with various Ethereum development tools and integrated development environments (IDEs) like Truffle, Remix, and others, making the development process smoother.

Install and config Ganache:



Visit the Ganache website and choose your platform

For Mac users (GUI)

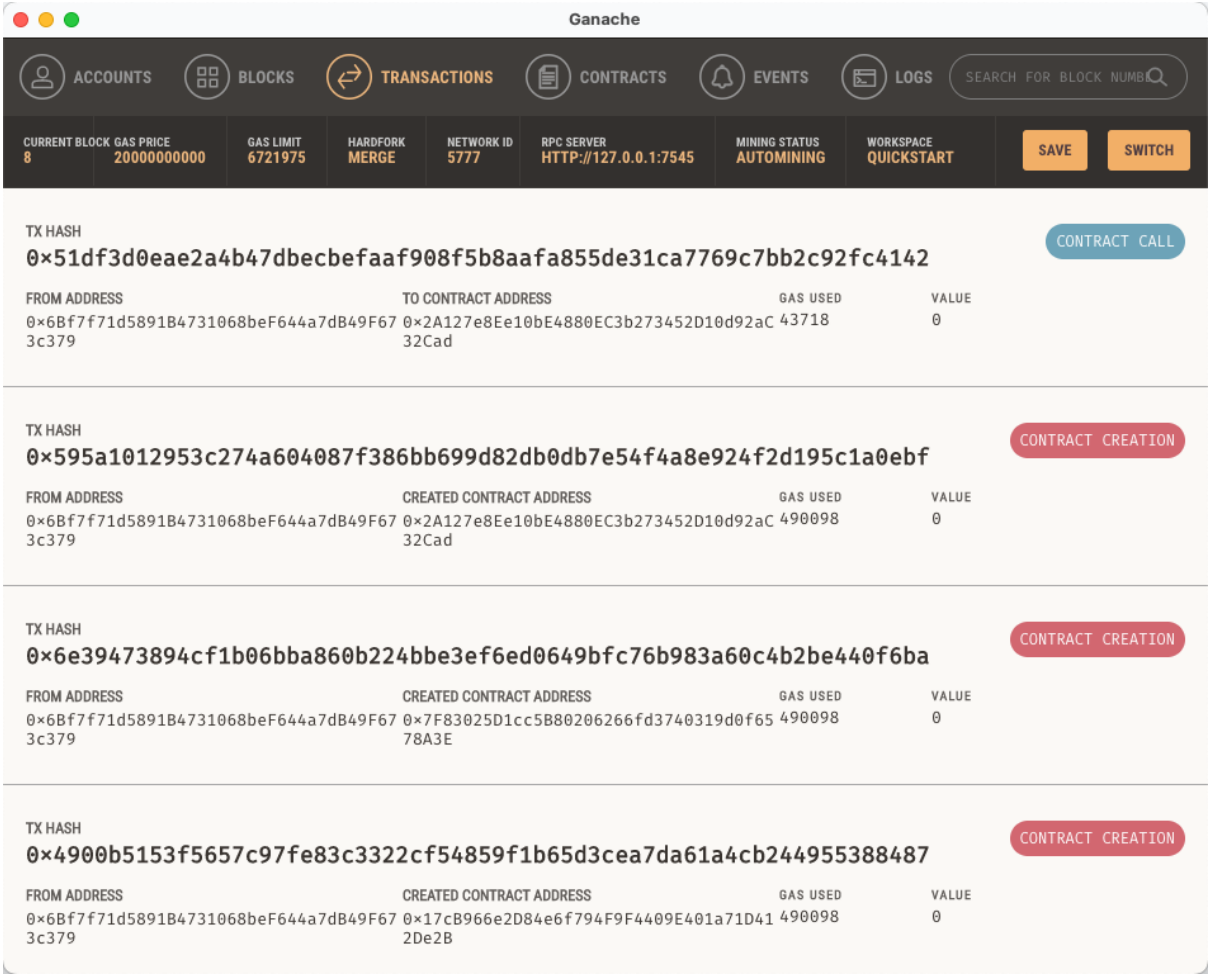
Visit https://trufflesuite.com/ganache/ and choose MAC OS

For Windows users (GUI)

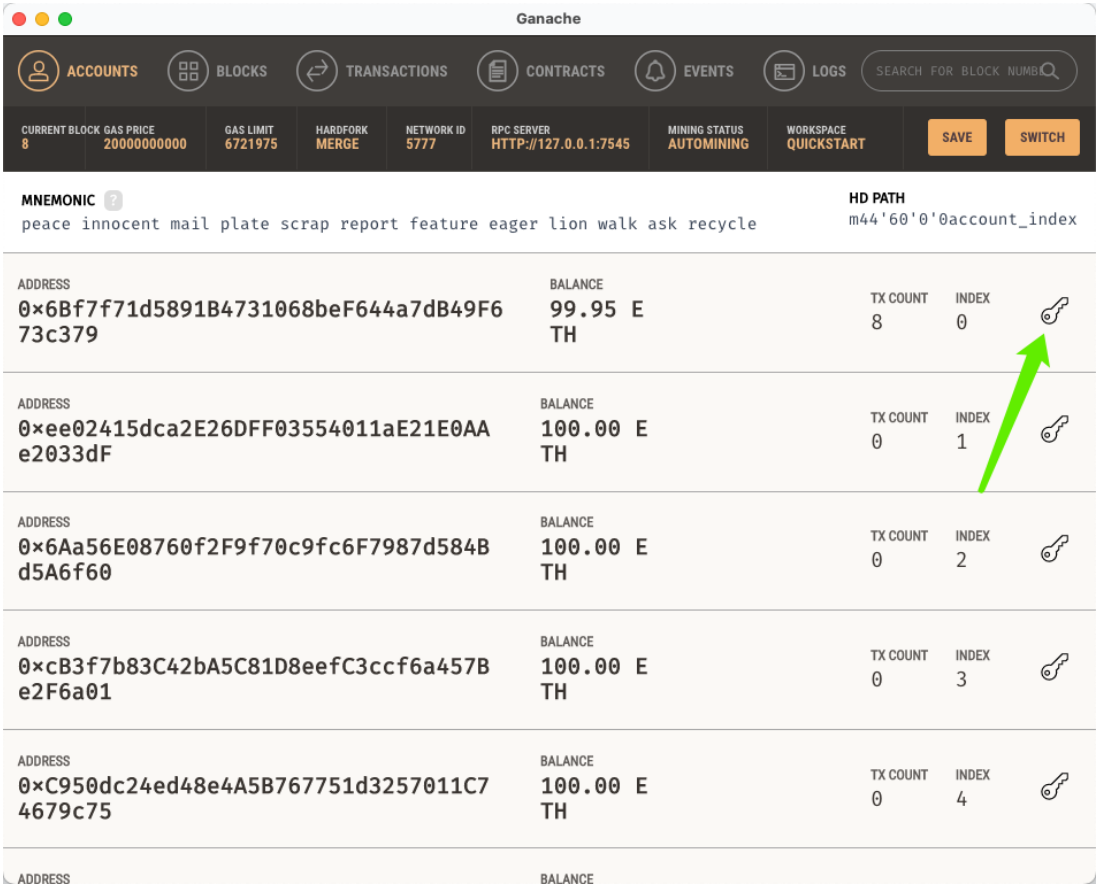
Visit https://trufflesuite.com/ganache/ and choose Windows

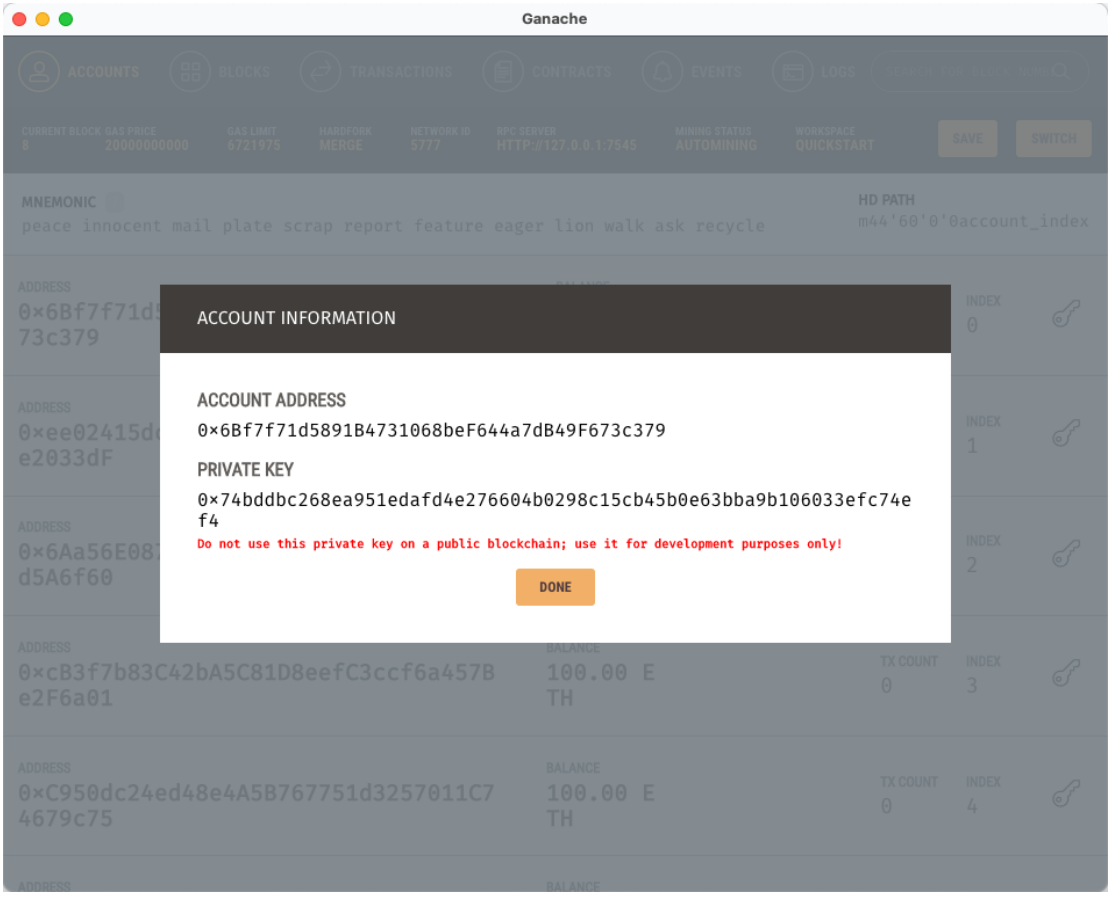
Usage of Ganache:

View the existed Transactions



View the existed Accounts (private key & account address)


Ganache
ACCOUNTS
BLOCKS
TRANSACTIONS
CONTRACTS
EVENTS
LOGS
SEARCH FOR BLOCK NUMB
CURRENT BLOCK 8
GAS PRICE 20000000000
GAS LIMIT 6721975
HARDFORK MERGE
NETWORK ID 5777
RPC SERVER HTTP://127.0.0.1:7545
MINING STATUS AUTOMINING
WORKSPACE QUICKSTART
SAVE
SWITCH
MNEMONIC
peace innocent mail plate scrap report feature eager lion walk ask recycle
HD PATH
m44'60'0'0account_index
ADDRESS 0x6Bf7f71d5891B4731068beF644a7dB49F673c379
BALANCE 99.95 ETH
TX COUNT 8
INDEX 0
ADDRESS 0xee02415dca2E26DFF03554011aE21E0AAe2033dF
BALANCE 100.00 ETH
TX COUNT 0
INDEX 1
ADDRESS 0x6Aa56E08760f2F9f70c9fc6F7987d584Bd5A6f60
BALANCE 100.00 ETH
TX COUNT 0
INDEX 2
ADDRESS 0xcB3f7b83C42bA5C81D8eefC3ccf6a457Be2F6a01
BALANCE 100.00 ETH
TX COUNT 0
INDEX 3
ADDRESS 0xC950dc24ed48e4A5B767751d3257011C74679c75
BALANCE 100.00 ETH
TX COUNT 0
INDEX 4
ADDRESS
BALANCE


Ganache
ACCOUNT INFORMATION
ACCOUNT ADDRESS
0x6Bf7f71d5891B4731068beF644a7dB49F673c379
PRIVATE KEY
0x74bddbc268ea951edafd4e276604b0298c15cb45b0e63bba9b106033efc74ef4
Do not use this private key on a public blockchain; use it for development purposes only!
DONE

Part 2: Introduction & Installation of Web3.py (20 minutes)

Web3.py is a Python client library for interacting with the Ethereum blockchain. It provides a set of Python APIs that allow developers to interact with the Ethereum blockchain, build decentralized applications (DApps), and perform various operations such as sending transactions, querying account balances, and invoking smart contracts.

- **Communication with Ethereum Nodes**: web3.py enables Python applications to communicate with Ethereum nodes for tasks like querying blockchain data and sending transaction requests.
- **Smart Contract Interaction**: web3.py provides functionality for interacting with Ethereum smart contracts. Developers can use it to deploy smart contracts, call functions and methods of contracts, and handle contract events.
- **Blockchain Data Queries**: Developers can use web3.py to query blockchain data such as block information, transaction history, and account balances.
- **Transaction Handling:** web3.py allows developers to create, sign, and send Ethereum transactions for activities like fund transfers and smart contract interactions.

Here is the Official Github link of web3.py

https://github.com/ethereum/web3.py

Here is the web3.py documentation

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

Install web3.py in your laptop:

Step 1: Activate the conda environment

Conda activate <env name>

Step 2: Install the web3.py package by pip

pip install web3

How to use web3.py in the FastAPI project:

```
from web3 import Web3

w3 = Web3(Web3.HTTPProvider("HTTP://127.0.0.1:7545"))
```

Part 3: Write a sample smart contract (30 minutes)

Defines a smart contract named "SimpleStorage" with the following features and structure:

State Variables:

- **uint256 favoriteNumber**: This is an unsigned integer state variable used to store a favorite number.
- bool favoriteBool: This is a boolean state variable used to store a boolean value.

Struct:

- **struct People:** This is a custom struct that includes two member variables: a favorite number and a name.

Public State Variables:

- **People public person**: This is a public state variable representing an instance of the "person" struct. It is initialized as {favoriteNumber: 2, name: "Arthur"}.
- **People[] public people:** This is a public state variable representing an array of People structs, used to store information about multiple people.
- **mapping(string => uint256) public nameToFavoriteNumber**: This is a public state variable representing a mapping from names to favorite numbers, allowing the association of names with their respective numbers.

Functions:

- **store(uint256 _favoriteNumber):** This function accepts a parameter _favoriteNumber and stores it in the favoriteNumber state variable, then returns the stored number.
- **retrieve()**: This function is a view function (does not modify state) that returns the value stored in the favoriteNumber state variable.
- **addPerson(string memory _name, uint256 _favoriteNumber):** This function takes a name _name and a favorite number _favoriteNumber, creates a new People struct instance, adds it to the people array, and associates the name with the number in the nameToFavoriteNumber mapping.

Try to write the smart contract by yourself. After you finish writing, check the smart contract code with the sample code here:

https://codesandbox.io/p/sandbox/fastapi-demo-kkd6yv?file=%2Fweek8%2FsampleContract.sol%3A33%2C2

Part 4: Compile the smart contract in Python (20 minutes)

Once we have written and syntax-checked the Solidity contract with Remix or another editor, we need to read the contract source file and store variables for subsequent compilation.

```
with open("./SimpleStorage.sol", "r") as file:
    simple_storage_file = file.read()
```

The above code reads the contents of the SimpleStorage.sol file into the variable simple_storage_file

How to compile the smart contract ?

Contract compilation requires the solcx tool to be pre-installed.

pip install py-solc-x

Import solcx in the FastAPI:

from solcx import compile_standard, install_solc

```
install_solc("0.6.0")
compiled_sol = compile_standard(
    {
        "language": "Solidity",
        "sources": {"SimpleStorage.sol": {"content": simple_storage_file}},
        "settings": {
            "outputSelection": {
                "*": {"*": ["abi", "metadata", "evm.bytecode", "evm.sourceMap"]}
            }
        },
    },
    solc_version="0.6.0",
)
```

The above code installs version 0.6.0 of Solidity and uses the compile_standard method of the solcx library to compile the contract source file read above, and stores the result of the compilation into the variable compiled_sol.

Get Compilation Results

After successful compilation, write the compiled contract to a file using the following code

```
with open("compiled_code.json", "w") as file:
    json.dump(compiled_sol, file)
```

Get bytecode and abi

The deployment and interaction of Solidity contracts requires bytecode and abi, which can be written to the corresponding variables for subsequent operations by using the following code

```
# get bytecode
bytecode = compiled_sol["contracts"]["SimpleStorage.sol"]["SimpleStorage"]["evm"][
    "bytecode"
]["object"]

# get abi
abi = compiled_sol["contracts"]["SimpleStorage.sol"]["SimpleStorage"]["abi"]
```

Config your Ganache inside the Python

```
# type your address here
w3 = Web3(Web3.HTTPProvider("HTTP://127.0.0.1:7545"))
# Default is 1337 or with the PORT in your Gaanche
chain_id = 1337
# Find in you account
my_address = "0x6Bf7f71d5891B4731068beF644a7dB49F673c379"
# Find in you account
private_key = "0x74bddbc268ea951edafd4e276604b0298c15cb45b0e63bba9b106033efc74ef4"
```

Part 5: Deploy the smart contract in Python (20 minutes)

Deploying a contract involves three main steps:

Constructing the Transaction:

Before deploying a smart contract, you need to construct a special type of transaction called a "contract creation transaction."

This transaction includes the bytecode of the contract (the code of the smart contract) and any constructor arguments (if the contract has a constructor).

You also need to specify the gas limit and gas price to ensure the contract can be deployed and executed successfully.

Typically, this step involves building a transaction object using Ethereum client libraries like web3.js or web3.py.

```
transaction = SimpleStorage.constructor().build_transaction(
    {
        "chainId": chain_id,
        "gasPrice": w3.eth.gas_price,
        "from": my_address,
        "nonce": nonce,
    }
)
```

Signing the Transaction:

Once you have constructed the contract creation transaction, the next step is to sign the transaction. Signing associates the transaction with the sender's private key, allowing the Ethereum network to verify the transaction's validity.

This step is typically handled by the sender's Ethereum wallet or client library, and the private key should not be exposed directly in code.

The signing process involves encrypting the transaction data with the sender's private key to generate a digital signature.

```
signed_txn = w3.eth.account.sign_transaction(transaction, private_key=private_key)
```

Sending the Transaction:

After the transaction is signed, it can be sent to the Ethereum network by broadcasting it.

Sending a transaction means delivering it to nodes in the Ethereum network to include in the next block.

Once the transaction is included in a block, the smart contract is deployed on the Ethereum network and assigned a unique contract address.

Typically, sending the transaction is also handled by Ethereum client libraries, and this process may take some time as the transaction needs confirmation by the blockchain network.

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

Here is the sample code:

https://codesandbox.io/p/sandbox/fastapi-demo-kkd6yv?file=%2Fweek8%2Fmain.py%3A15%2C1

Interacting with Contracts

Similar to the deployment steps, we can interact with the contract via the web3 library in three steps: constructing the transaction, signing the transaction, and sending the transaction.

```
simple_storage = w3.eth.contract(address=tx_receipt.contractAddress, abi=abi)

store_transaction = simple_storage.functions.store(67).build_transaction(
    {
        "chainId": chain_id,
        "gasPrice": w3.eth.gas_price,
        "from": my_address,
        "nonce": nonce + 1,
    }
)

signed_store_txn = w3.eth.account.sign_transaction(store_transaction, private_key=private_key)
send_store_tx = w3.eth.send_raw_transaction(signed_store_txn.rawTransaction)
tx_receipt = w3.eth.wait_for_transaction_receipt(send_store_tx)
```

Here is the sample code:

https://codesandbox.io/p/sandbox/fastapi-demo-kkd6yv?file=%2Fweek8%2Fmain.py%3A15%2C1

Part 6 : Wrap-up (10 minutes)

Congratulations on completing this workshop on web3.py and Ganache! We hope you found this learning experience valuable and gained a deeper understanding of working with graph databases and the Cypher query language.

Some useful Links:

web3.py official documentation

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

web3.py official Github

https://github.com/ethereum/web3.py

Deploy Smart Contract with Python

https://www.youtube.com/watch?v=zCAhMBedPjc&ab_channel=MammothInteractive