Title	LOTTERY PROJECT USING BROWNIE A FULL SCALE DAPP
Description	Intro
Date	June 20th, 2022
Author	Arashtad
Author URI	https://Arashtad.com

In this article, we are going to get started with the lottery project using Brownie. The main purpose of a lottery project in every network is to check the reliability of the randomness and use it for different purposes such as the lottery itself. In the lottery project, we are going to create a decentralized application using Brownie. In the end, we will be able to run the smart contract via Etherscan.

USING BROWNIE FOR LOTTERY PROJECT

In this tutorial, we are going to first write a smart contract related to a lottery and write scripts related to testing and deploying the smart contract. We also want to make it a full-scale decentralized application, meaning that it is going to be an end-to-end Dapp with easy to use user experience. Every lottery needs a random variable. As randomness is a very complicated concept when it is going to be applied on the internet and here we are going to run it via the blockchain, it must be protected from hacks and cheating. What makes randomness really complex, is that we are dealing with deterministic variables rather than probabilistic ones. As a result, we are going to use some Chainlink tools to cover this complexity.

In this first part, we only focus on the lottery.sol related to the smart contracts, its dependency files, and the Brownie-config.yaml in addition to seeing how the Solidity scripts are written and how they work.

GETTING STARTED WITH THE LOTTERY PROJECT

It is also important to notice that this tutorial is written to explain the GitHub repository published by Patrick Alpha C more explicitly and solve the probable issues that the programmers might face when running and facing bugs related to the Solidity version or the configuration of the project. With that said let's get started:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.6.6;

import
"@chainlink/contracts/src/v0.6/interfaces/AggregatorV3Interface.sol"
;
import "@openzeppelin/contracts/access/Ownable.sol";
import "@chainlink/contracts/src/v0.6/VRFConsumerBase.sol";
```

We have first declared the version of Solidity and the license comment before that. Also, we have imported 3 necessary Solidity libraries including AggregatorV3Interface which is used for fetching the Ethereum price from Chainlink oracle, openzeppelin Ownable.sol which is used for determining the owner of the contract and finally the VRFConsumer.sol which is used to get a random number.

```
contract Lottery is VRFConsumerBase, Ownable {
        address payable[] public players;
    address payable public recentWinner;
    uint256 public randomness;
    uint256 public usdEntryFee;
    AggregatorV3Interface internal ethUsdPriceFeed;
    enum LOTTERY_STATE {
        OPEN,
        CLOSED,
        CALCULATING_WINNER
    }
    LOTTERY_STATE public lottery_state;
    uint256 public fee;
    bytes32 public keyhash;
    event RequestedRandomness(bytes32 requestId);

    constructor(address _priceFeedAddress,
            address _vrfCoordinator,
            address _link,
            uint256 _fee,
            bytes32 _keyhash
            ) public VRFConsumerBase(_vrfCoordinator, _link) {
        usdEntryFee = 50 * (10**18);
        ethUsdPriceFeed = AggregatorV3Interface(_priceFeedAddress);
        lottery_state = LOTTERY_STATE.CLOSED;
        fee = _fee;
        keyhash = _keyhash;
    }
}
```

We start the contract named Lottery and determine their declarations ownable, VRFConsumerBase. Inside the contract we define the variables that we are going to work with including:

- The array of the addresses of the participants in the lottery defined here as players (declared as payable and public),
- The recent winner who has won the prize in the last lottery,
- The randomness which is the random number we receive from VRFC,
- UsdEntryFee which is the minimum amount that a player needs to participate in the lottery, ethUsdPriceFeed which is the conversion rate of ETH to USD.

The lottery state is declared as Enum and has 3 states Open: Close, and Calculating winner.

1. Open is when everyone can participate in the lottery,
2. Closed is when nobody can participate
3. and calculating is used when the random number related to the winner is being calculated.

We also have a constructor for some variables which are going to be explained later.

```
function enter() public payable {
    // $50 minimum
    require(lottery_state == LOTTERY_STATE.OPEN);
    require(msg.value >= getEntranceFee(), "Not enough ETH!");
    players.push(msg.sender);
}
```

The above function is related to the entrance to the lottery. At first, the state of the lottery is open and we check if the players pay the minimum entrance fee to participate in it.

```
function getEntranceFee() public view returns (uint256) {
    (, int256 price, , , ) = ethUsdPriceFeed.latestRoundData();
    uint256 adjustedPrice = uint256(price) * 10**10;
    // 18 decimals
    // $50, $2,000 / ETH
    // 50/2,000
    // 50 * 100000 / 2000
    uint256 costToEnter = (usdEntryFee * 10**18) / adjustedPrice;
    return costToEnter;
}
```

The above function is responsible for checking the price of the Ethereum in USD and determining the price of participation in ETH.

```
function startLottery() public onlyOwner {
    require(lottery_state == LOTTERY_STATE.CLOSED,"Can't start a
new lottery yet!");
    lottery_state = LOTTERY_STATE.OPEN;
}
```

The above function starts the lottery and before that checks if the lottery has not yet got closed.

```
function endLottery() public onlyOwner {
    lottery_state = LOTTERY_STATE.CALCULATING_WINNER;
    bytes32 requestId = requestRandomness(keyhash, fee);
    emit RequestedRandomness(requestId);
}
```

The above function gives the authority to only the owner of the contract. This function also determines the winner to end the lottery. This process is done by requesting a random number.

```
function fulfillRandomness(bytes32 _requestId, uint256
_randomness)internal override{
    require(lottery_state == LOTTERY_STATE.CALCULATING_WINNER,"You
aren't there yet!");
    require(_randomness > 0, "random-not-found");
    uint256 indexOfWinner = _randomness % players.length;
    recentWinner = players[indexOfWinner];
    recentWinner.transfer(address(this).balance);
    // Reset
    players = new address payable[](0);
    lottery_state = LOTTERY_STATE.CLOSED;
    randomness = _randomness;
}
```

The above function requires the lottery state to be in the calculating winner. It also checks whether the given random number is positive. Then it uses the remainder operator to calculate the id of the winner from the random number it gets. After that, it transfers the balance of the contract to the player that has won the lottery And finally changes the state of the lottery to closed.

CONFIGURATION

Notice that up to here you need to enter the following into the Brownie-config.yaml:

```
dependencies:
     smartcontractkit/chainlink-brownie-contracts@1.1.1
     OpenZeppelin/openzeppelin-contracts@3.4.0
compiler:
     solc:
          remappings:
               '@chainlink=smartcontractkit/chainlink-brownie-
contracts@1.1.1'
               '@openzeppelin=OpenZeppelin/openzeppelin-
contracts@3.4.0'
dotenv: .env
networks:
     default: development
     development:

keyhash:'0x2ed0feb3e7fd2022120aa84fab1945545a9f2ffc9076fd6156fa96eaf
f4c1311'
          fee: 100000000000000000
     rinkeby:

vrf_coordinator:'0xb3dCcb4Cf7a26f6cf6B120Cf5A73875B7BBc655B'

eth_usd_price_feed:'0x8A753747A1Fa494EC906cE90E9f37563A8AF630e'
          link_token: '0x01BE23585060835E02B77ef475b0Cc51aA1e0709'

keyhash:'0x2ed0feb3e7fd2022120aa84fab1945545a9f2ffc9076fd6156fa96eaf
f4c1311'
          fee: 100000000000000000
          verify: True
     mainnet-fork:

eth_usd_price_feed:'0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419'
          verify: False
wallets:
     from_key: ${PRIVATE_KEY}
```

Up to here, we only need the following to be able to compile the Brownie project:

```
dependencies:
     smartcontractkit/chainlink-brownie-contracts@1.1.1
     OpenZeppelin/openzeppelin-contracts@3.4.0
compiler:
     solc:
          remappings:
               '@chainlink=smartcontractkit/chainlink-brownie-
contracts@1.1.1'
               '@openzeppelin=OpenZeppelin/openzeppelin-
contracts@3.4.0'
```

In the next parts, we are going to see how we can deploy and test this smart contract on different networks.

DEPLOYMENT OF THE LOTTERY PROJECT USING BROWNIE

In this section, we are going to continue the lottery project and focus on its deployment of it. The main python file as always is the deploy.py where we are going to deploy the four stages of a lottery: deploy the lottery, start the lottery, enter the lottery and end the lottery. The prize here is paid with a Chainlink token. As a result, we will add the ChainlinkTokenInterface contract to the contracts directory.

MANAGING THE FOLDERS OF THE LOTTERY PROJECT DIRECTORY

In order to deploy the smart contract that we have just written, we need to complete the deployment and test folders and also add some other files like the dependencies of our smart contract. As always, the first step is to tyoe in the terminal:

```
brownie init
```

Then, you will see the folders and files that are created afterward. Then it is time to write our deploy.py file. As you know every lottery contract has 4 stages, deploying the lottery, starting it, entering the lottery, and ending it. So accordingly, we have 4 functions:

```
from scripts.helpful_scripts import get_account, get_contract,
fund_with_link
from brownie import Lottery, network, config
import time

def deploy_lottery():
    account = get_account()
    lottery =
Lottery.deploy(get_contract("eth_usd_price_feed").address,

        get_contract("vrf_coordinator").address,
        get_contract("link_token").address,
        config["networks"][network.show_active()]["fee"],
        config["networks"][network.show_active()]["keyhash"],
        {"from": account},
        publish_source=config["networks"]
[network.show_active()].get("verify", False),)

    print("Deployed lottery!")
    return lottery
```

In every contract deployment, we should determine some of the specifications related to a contract such as contract addresses (address of the dependency contracts) and some config specifications about every network.

```
def start_lottery():
    account = get_account()
    lottery = Lottery[-1]
    starting_tx = lottery.startLottery({"from": account})
    starting_tx.wait(1)
    print("The lottery is started!")
```

The above function starts the lottery. At first, it gets the account address which uses the get account function from helpful_scripts.py, then it gets the latest lottery smart contract, After that, it starts the lottery transaction using the startLottery function inside the contract.

```
def enter_lottery():
    account = get_account()
    lottery = Lottery[-1]
    value = lottery.getEntranceFee() + 100000000
    tx = lottery.enter({"from": account, "value": value})
    tx.wait(1)
    print("You entered the lottery!")
```

The above function also gets the account and the latest lottery contract records, then determine the value of the entrance fee (notice that 10000000 that has been added up with the entrance fee, is not a big number as it is in the Wei unit). Then, we enter the participant by using the enter function from the lottery.sol contract.

```
def end_lottery():
    account = get_account()
    lottery = Lottery[-1]
    # fund the contract
    # then end the lottery
    tx = fund_with_link(lottery.address)
    tx.wait(1)
    ending_transaction = lottery.endLottery({"from": account})
    ending_transaction.wait(1)
    time.sleep(180)
    print(f"{lottery.recentWinner()} is the new winner!")
```

In the above function, again get the account and the lottery contract, fund the winner with some link token (using the function written in helpful_scripts.py), and ends the transaction using the function in the lottery contract.sol.

```
def main():
    deploy_lottery()
    start_lottery()
    enter_lottery()
    end_lottery()
```

In the above main function, we apply other functions in the sequence of a lottery.

HELPFUL_SCRIPTS.PY

Now, it is time to go after helpful_scripts.py file and explain the codes:

```
from brownie import (accounts, network, config, MockV3Aggregator,
VRFCoordinatorMock, LinkToken,
    Contract, interface,)
FORKED_LOCAL_ENVIRONMENTS = ["mainnet-fork", "mainnet-fork-dev"]
LOCAL_BLOCKCHAIN_ENVIRONMENTS = ["development", "ganache-local"]

def get_account(index=None, id=None):
    if index:
        return accounts[index]
    if id:
        return accounts.load(id)
    if ( network.show_active() in LOCAL_BLOCKCHAIN_ENVIRONMENTS or
network.show_active() in
            FORKED_LOCAL_ENVIRONMENTS):
        return accounts[0]
    return accounts.add(config["wallets"]["from_key"])
```

The above function determines the address of the available account according to the active network.

```
contract_to_mock = {"eth_usd_price_feed":
MockV3Aggregator,"vrf_coordinator": VRFCoordinatorMock,
    "link_token": LinkToken,}
def get_contract(contract_name):
    contract_type = contract_to_mock[contract_name]
    if network.show_active() in LOCAL_BLOCKCHAIN_ENVIRONMENTS:
        if len(contract_type) <= 0:
            # MockV3Aggregator.length
            deploy_mocks()
            contract = contract_type[-1]
            # MockV3Aggregator[-1]
    else:
        contract_address = config["networks"]
[network.show_active()][contract_name]
        contract = Contract.from_abi(contract_type._name,
contract_address, contract_type.abi)
    return contract
```

The above function will grab the contract addresses from the Brownie config when defined. Otherwise, it will deploy a mock version of that contract, and return that mock contract.

```
DECIMALS = 8
INITIAL_VALUE = 200000000000
def deploy_mocks(decimals=DECIMALS, initial_value=INITIAL_VALUE):
    account = get_account()
    MockV3Aggregator.deploy(decimals, initial_value, {"from":
account})
    link_token = LinkToken.deploy({"from": account})
    VRFCoordinatorMock.deploy(link_token.address, {"from":
account})
    print("Deployed!")
```

The above function deploys the mock version of the 2 contracts: 1. VRFCoordinator and 2. V3Aggregator contracts.

```
def fund_with_link(contract_address, account=None,
link_token=None,amount=100000000000000000): # 0.1 LINK
    account = account if account else get_account()
    link_token = link_token if link_token else
get_contract("link_token")
    tx = link_token.transfer(contract_address, amount, {"from":
account})
    tx.wait(1)
    print("Fund contract!")
    return tx
```

The above function funds the winner with a link token. Notice that we should create a file in the interfaces folder for the link token contract and name it LinkTokenInterface.sol. You can copy and paste the contract below:

LINK TOKEN INTERFACE CONTRACT

The below contract which is called LinkTokenInterface.sol, should be added to the contacts folder. We use the functions of this contract in the helpful_scripts.py and deploy.py to fund the winner of the lottery with some ChainLink tokens. If you look at the contract in detail, you will see that the functions and methods are the same as of an ERC-20 token contract.

```
pragma solidity ^ 0.6.6;

interface LinkTokenInterface {
    function allowance(address owner, address spender) external
view returns (uint256 remaining);
    function approve(address spender, uint256 value) external
returns (bool success);
    function balanceOf(address owner) external view returns
(uint256 balance);
    function decimals() external view returns (uint8
decimalPlaces);
    function decreaseApproval(address spender, uint256 addedValue)
external returns (bool success);
    function increaseApproval(address spender, uint256
subtractedValue)external;
    function name() external view returns (string memory
tokenName);
    function symbol() external view returns (string memory
tokenSymbol);
    function totalSupply() external view returns (uint256
totalTokensIssued);
    function transfer(address to, uint256 value) external returns
(bool success);
    function transferAndCall(
        address to,
        uint256 value,
        bytes calldata data
        ) external returns (bool success);
    function transferFrom(
        address from,
        address to,
        uint256 value
        ) external returns (bool success);
}
```

COMPILING OUR COMPLETE LOTTERY PROJECT USING BROWNIE FOR FINAL DEPLOYMENT

In this section, following the lottery project, we are going to complete the scripts that should be written so that we can finally compile our complete contract and be able to deploy it fully. The focus of this section is on the testing of the contract, where we are going to test the different functionalities of different stages of the lottery project. In the following of the lottery project, we are going to complete the scripts that should be written so that we can finally compile our complete contract and be able to deploy it fully.

BROWNIE_CONFIG.SOL

There are some .sol dependencies that need to be copied and pasted into the contracts/test folder so that we can use them to deploy our main lottery smart contract. So make sure you copy them from this link and paste them into your directory.

.ENV FILE

Also, in the .env file enter the private key of your test Metamask account (Do not use a Metamask wallet with real crypto in it), your Etherscan Token, and Infura ID.

```
export WEB3_INFURA_PROJECT_ID=''
export PRIVATE_KEY=''
export ETHERSCAN_TOKEN=''
```

TESTING THE CONTRACT FUNCTIONALITIES

Now, every standard smart contract deployment needs testing and this one is not an exception, we start our test with test_lottery_unit.py:

```
from scripts.helpful_scripts import
( LOCAL_BLOCKCHAIN_ENVIRONMENTS,get_account,fund_with_link,
    get_contract,)
from brownie import Lottery, accounts, config, network, exceptions
from scripts.deploy_lottery import deploy_lottery
from web3 import Web3
import pytest

def test_get_entrance_fee():
    if network.show_active() not in LOCAL_BLOCKCHAIN_ENVIRONMENTS:
        pytest.skip()
    # Arrange
    lottery = deploy_lottery()
    # Act
    # 2,000 eth / usd
    # usdEntryFee is 50
    # 2000/1 == 50/x == 0.025
    expected_entrance_fee = Web3.toWei(0.025, "ether")
    entrance_fee = lottery.getEntranceFee()
    # Assert
    assert expected_entrance_fee == entrance_fee
```

The above test considers the price of ETH is 2000 dollars and the entrance fee is 50 dollars so the 0.025 ether is required to participate in the lottery. as a result, we expect the entrance fee to be 0.025 ether, so we call the getEn-tranceFee() function to check the validity of the result.

```
def test_cant_enter_unless_started():
    # Arrange
    if network.show_active() not in LOCAL_BLOCKCHAIN_ENVIRONMENTS:
        pytest.skip()
    lottery = deploy_lottery()
    # Act / Assert
    with pytest.raises(exceptions.VirtualMachineError):
        lottery.enter({"from": get_account(), "value": lot-
tery.getEntranceFee()})
```

The above test checks whether the participant can enter or not at the times the lottery has been closed or has not opened yet.

```
def test_can_start_and_enter_lottery():
    # Arrange
    if network.show_active() not in LOCAL_BLOCKCHAIN_ENVIRONMENTS:
        pytest.skip()
    lottery = deploy_lottery()
    account = get_account()
    lottery.startLottery({"from": account})
    # Act
    lottery.enter({"from": account, "value":
lottery.getEntranceFee()})
    # Assert
    assert lottery.players(0) == account
```

The above test checks whether the player who has participated in the lottery is recorded in the list of lottery players in other words it checks if the player has entered or not.

```
def test_can_end_lottery():
    # Arrange
    if network.show_active() not in LOCAL_BLOCKCHAIN_ENVIRONMENTS:
        pytest.skip()
    lottery = deploy_lottery()
    account = get_account()
    lottery.startLottery({"from": account})
    lottery.enter({"from": account, "value":
lottery.getEntranceFee()})
    fund_with_link(lottery)
    lottery.endLottery({"from": account})
    assert lottery.lottery_state() == 2
```

The above test checks whether the end lottery function works. To do so, it first passes all the other tests and applies the other stages of the lottery and at the end tests the lottery state.

```
def test_can_pick_winner_correctly():
    # Arrange
    if network.show_active() not in LOCAL_BLOCKCHAIN_ENVIRONMENTS:
        pytest.skip()
    lottery = deploy_lottery()
    account = get_account()
    lottery.startLottery({"from": account})
    lottery.enter({"from": account, "value":
lottery.getEntranceFee()})
    lottery.enter({"from": get_account(index=1), "value":  lot-
tery.getEntranceFee()})
    lottery.enter({"from": get_account(index=2), "value":  lot-
tery.getEntranceFee()})
    fund_with_link(lottery)
    starting_balance_of_account = account.balance()
    balance_of_lottery = lottery.balance()
    transaction = lottery.endLottery({"from": account})
    request_id = transaction.events["RequestedRandomness"]
["requestId"]
    STATIC_RNG = 777

get_contract("vrf_coordinator").callBackWithRandomness(request_id,
STATIC_RNG, lottery.address,
        {"from": account})
    # 777 % 3 = 0
    assert lottery.recentWinner() == account
    assert lottery.balance() == 0
    assert account.balance() == starting_balance_of_account + bal-
ance_of_lottery
```

The above test uses all of the operations of the lottery that have been tested to test the correctness of the winner picking. To do so, it first enters an account into the lottery and at the end, tests whether the winner is in the same account, whether the lottery contract balance has turned 0 as a result of sending money to the winner, and whether the winner's account has been added up with the balance of the lottery contract.

```
from brownie import network
import pytest
from scripts.helpful_scripts import ( LOCAL_BLOCKCHAIN_ENVIRONMENTS,
    get_account, fund_with_link,)
from scripts.deploy_lottery import deploy_lottery
import time

def test_can_pick_winner():
    if network.show_active() in LOCAL_BLOCKCHAIN_ENVIRONMENTS:
        pytest.skip()
    lottery = deploy_lottery()
    account = get_account()
    lottery.startLottery({"from": account})
    lottery.enter({"from": account, "value":
lottery.getEntranceFee()})
    lottery.enter({"from": account, "value":
lottery.getEntranceFee()})
    fund_with_link(lottery)
    lottery.endLottery({"from": account})
    time.sleep(180)
    assert lottery.recentWinner() == account
    assert lottery.balance() == 0
```

The above test also does the same process in a little bit different manner.

It is important to know that we should do all of the tests one by one to be able to debug the functions. As you can see, we started our first test with the first stage of the contract.

FINAL WORD ON LOTTERY PROJECT DEPLOYMENT USING BROWNIE

Firstly, we have managed to get started with the lottery project and have written the smart contract for it. In addition to that, we have added some dependency contracts like VRFConsumerBase, Openzeppelin, and V3AggregatorInterface.

Secondly, we have managed to write the deploy.py and helpful_scripts.py to interact with the lottery smart contract and deploy the different stages of a lottery such as deploying the lottery, starting it, entering it using different accounts, and ending it. In the end, the winner is going to be awarded some Chainlink tokens.

Thirdly, we have managed to complete the whole lottery project script file so that we can finally compile and deploy the smart contract. In most of the parts of this tutorial, we focused on testing the contract scripts for testing the different functionalities related to the different stages of the contract.