TRY dApp: a dApp for the NFT Lottery

Federico Bernacca

1 Code

The structure, classes and most significant functions of the application are shown below.

1.1 Project structure

All files are located inside the **nft_dapp** repo, which has the following structure.

- **nft_dapp**
  - **contracts** (contains the smart contract)
  - **helpers**
    - **nft_collectible.py** (helper class to represent a NFT collectible)
    - **ticket.py** (helper class to represent a NFT collectible)
  - **processors**
    - **Lottery.py** (Class that manages smart contract function calls)
    - **Nft.py** (Class that manages smart contract function calls)
    - **Contract.py** (Class containig helpful method to deploy a contract and to create a transaction)
  - **static** (contains static files (images, css styles))
  - **templates** (contains html templates)
  - **views**
    - **auth.py** (routes that manages a user (login, logout, profile page))
    - **home.py** (routes that renders homepage, accounts, collectibles templates)
    - **lottery.py** (routes that manage lottery operations at high level)
    - **notification.py** (route that asks for the lottery events and get the notifications)
  - **app.py** (entry point of the application)
  - **auth.py** (manage the access to the routes and define the user class)
  - **keys.json** (blockchain addresses and private keys generated by ganache)

1.2 ContractProcessor

The following class contains the definition of two functions: one to deploy contracts and one to create a transaction in json format.

```
import json


class ContractProcessor:
    """
    This class is used to deploy a contract and to build a transaction.
    """

    ADDRESS_ZERO = "0x0000000000000000000000000000000000000000"

    @staticmethod
    def deploy_contract(contract_name, *constructor_args):
        """
        This method is used to deploy a contract.
        :param contract_name: Name of the contract to deploy.
        :param constructor_args: Arguments to pass to the contract constructor.
        :return: The address of the contract and the contract instance.
        """
        from app import w3, manager

        truffle_file = json.load(open("./build/contracts/" + contract_name + ".json"))
        abi = truffle_file["abi"]
        bytecode = truffle_file["bytecode"]

        # Initialize a contract object with the smart contract compiled artifacts
        contract = w3.eth.contract(bytecode=bytecode, abi=abi)

        # build a transaction by invoking the buildTransaction() method
        from the smart contract constructor function
        construct_txn = contract.constructor(*constructor_args).buildTransaction(
            {
                "from": manager.address,
                "nonce": w3.eth.getTransactionCount(manager.address),
                "gas": 30000000,
                "gasPrice": w3.toWei("21", "gwei"),
            }
        )

        # sign the deployment transaction with the private key
        signed = w3.eth.account.sign_transaction(construct_txn, manager.key)

        # broadcast the signed transaction to your local network using
        sendRawTransaction() method and get the transaction hash
        tx_hash = w3.eth.sendRawTransaction(signed.rawTransaction)

        # collect the Transaction Receipt with contract address when the transaction
        is mined on the network
        tx_receipt = w3.eth.waitForTransactionReceipt(tx_hash)
        contract_address = tx_receipt["contractAddress"]

        # Initialize a contract instance object using the contract address
        which can be used to invoke contract functions
        contract_instance = w3.eth.contract(abi=abi, address=contract_address)
        return contract_address, contract_instance


    @staticmethod
    def create_transaction(_from: str, _to: str, _value: int):
        """
        Create a transaction object in the format required by the web3.py library
        :param _from: Address from which the transaction is sent.
        :param _to: Address to which the transaction is sent.
        :param _value: Amount of ether to send.
        :return: A transaction object.
        """
        from app import w3
```

```
wei = w3.toWei(_value, "ether")
tx = {
    "from": _from,
    "to": _to,
    "value": wei,
    "gas": 2618850,
    "gasPrice": w3.toWei("40", "gwei"),
}
return tx
```

1.3 LotteryProcessor

The following class contains the definition of functions that call the methods of the smart contract **Lottery.sol**, and the registration to the events of the same.

```
from flask_login import current_user
from processors.contract import ContractProcessor


class LotteryProcessor:

    # Filter for Lottery create event
    lottery_created_event = None

    # Filter for Lottery open round event
    round_opened_event = None

    # Filter for Lottery close event
    lottery_closed_event = None

    # Filter for Lottery draw winning numbers event
    winning_numbers_drawn_event = None

    # Filter for Lottery assign prize event
    prize_assigned = None

    # Filter for Lottery mint token event
    token_minted = None

    @staticmethod
    def init_filters():
        from app import lottery_instance

        """
        Initialize all the filters for the Lottery contract.
        """
        LotteryProcessor.lottery_created_event = (
            lottery_instance.events.LotteryCreated.createFilter(
                fromBlock=1, toBlock="latest"
            )
        )
        LotteryProcessor.round_opened_event = (
            lottery_instance.events.RoundOpened.createFilter(
                fromBlock=1, toBlock="latest"
            )
        )
        LotteryProcessor.lottery_closed_event = (
            lottery_instance.events.LotteryClosed.createFilter(
                fromBlock=1, toBlock="latest"
            )
        )
        LotteryProcessor.winning_numbers_drawn_event = (
            lottery_instance.events.WinningNumbersDrawn.createFilter(
                fromBlock=1, toBlock="latest"
            )
        )
        LotteryProcessor.prize_assigned = (
            lottery_instance.events.PrizeAssigned.createFilter(
                fromBlock=1, toBlock="latest"
            )
        )
        LotteryProcessor.token_minted = (
            lottery_instance.events.TokenMinted.createFilter(
                fromBlock=1, toBlock="latest"
            )
        )
```

```python
@staticmethod
def is_open():
    """
    :return: True if the lottery is open, False otherwise
    """
    from app import lottery_instance

    return lottery_instance.functions.isLotteryActive().call()

@staticmethod
def is_already_minted(id: int):
    """
    :param id: id of the collectible
    :return: True if the collectible is already minted, False otherwise
    """
    from app import nft_instance

    return (
        nft_instance.functions.ownerOf(id).call() != ContractProcessor.ADDRESS_ZERO
    )

@staticmethod
def is_round_active():
    """
    :return: True if the round is active, False otherwise
    """
    from app import lottery_instance

    return lottery_instance.functions.isRoundActive().call()

@staticmethod
def is_round_finished():
    """
    :return: True if the round is finished, False otherwise
    """
    from app import lottery_instance

    return lottery_instance.functions.isRoundFinished().call()

@staticmethod
def is_winning_ticket_extracted():
    """
    :return: True if the winning ticket is already extracted, False otherwise
    """
    from app import lottery_instance

    return lottery_instance.functions.areNumbersDrawn().call()

@staticmethod
def mint(id: int, collectible: str, rank: int):
    """
    Mint a collectible
    :param id: id of the collectible
    :param collectible: collectible name
    :param rank: rank of the collectible
    :return: Transaction result
    """
    from app import w3, lottery_instance, lottery_address, manager

    try:
        tx = ContractProcessor.create_transaction(
            manager.address, lottery_address, 0
        )
        tx_hash = lottery_instance.functions.mint(id, rank, collectible).transact(
            tx
        )
        tx_receipt = w3.eth.waitForTransactionReceipt(tx_hash)
        print(tx_receipt)
        return tx_receipt["status"]
    except Exception as e:
        return 0
```

```
@staticmethod
def buy_ticket(
    one: int, two: int, three: int, four: int, five: int, powerball: int
):
    """
    Buy a ticket for the current user.
    :param one: first number
    :param two: second number
    :param three: third number
    :param four: fourth number
    :param five: fifth number
    :param powerball: powerball number
    :return: Transaction result
    """
    from app import w3, lottery_instance, lottery_address

    try:
        tx = ContractProcessor.create_transaction(
            current_user.id, lottery_address, 1
        )
        tx_hash = lottery_instance.functions.buy(
            one, two, three, four, five, powerball
        ).transact(tx)
        tx_receipt = w3.eth.waitForTransactionReceipt(tx_hash)
        print(tx_receipt)
        return tx_receipt["status"]
    except Exception as e:
        print(e)
        return 0

@staticmethod
def create_lottery():
    """
    Create the lottery
    :return: Transaction result
    """
    from app import w3, lottery_instance, lottery_address, manager

    try:
        tx = ContractProcessor.create_transaction(
            manager.address, lottery_address, 0
        )
        tx_hash = lottery_instance.functions.createLottery().transact(tx)
        tx_receipt = w3.eth.waitForTransactionReceipt(tx_hash)
        print(tx_receipt)
        return tx_receipt["status"]
    except Exception as e:
        print(e)
        return 0

@staticmethod
def open_round():
    """
    Open the round
    :return: Transaction result
    """
    from app import w3, lottery_instance, lottery_address, manager

    try:
        tx = ContractProcessor.create_transaction(
            manager.address, lottery_address, 0
        )
        tx_hash = lottery_instance.functions.openRound().transact(tx)
        tx_receipt = w3.eth.waitForTransactionReceipt(tx_hash)
        print(tx_receipt)
        return tx_receipt["status"]
    except Exception as e:
        print(e)
        return 0
```

```
@staticmethod
def close_lottery():
    """
    Close the lottery
    :return: Transaction result
    """
    from app import w3, lottery_instance, lottery_address, manager

    try:
        tx = ContractProcessor.create_transaction(
            manager.address, lottery_address, 0
        )
        tx_hash = lottery_instance.functions.closeLottery().transact(tx)
        tx_receipt = w3.eth.waitForTransactionReceipt(tx_hash)
        print(tx_receipt)
        return tx_receipt["status"]
    except Exception as e:
        print(e)
        return 0

@staticmethod
def extract_winning_ticket():
    """
    Extract the winning ticket
    :return: Transaction result
    """
    from app import w3, lottery_instance, lottery_address, manager

    try:
        tx = ContractProcessor.create_transaction(
            manager.address, lottery_address, 0
        )
        tx_hash = lottery_instance.functions.drawNumbers().transact(tx)
        tx_receipt = w3.eth.waitForTransactionReceipt(tx_hash)
        print(tx_receipt)
        return tx_receipt["status"]
    except Exception as e:
        print(e)
        return 0

@staticmethod
def give_prizes():
    """
    Give the prizes
    :return: Transaction result
    """
    from app import w3, lottery_instance, lottery_address, manager

    try:
        tx = ContractProcessor.create_transaction(
            manager.address, lottery_address, 0
        )
        tx_hash = lottery_instance.functions.givePrizes().transact(tx)
        tx_receipt = w3.eth.waitForTransactionReceipt(tx_hash)
        print(tx_receipt)
        return tx_receipt["status"]
    except Exception as e:
        print(e)
        return 0
```

1.4 Routes exposed

The following are some of the functions that define the routes exposed by the web application, containing calls to the functions of the **LotteryProcessor** class, and the handling of the results, generating informative messages to the user in case of both positive and negative results; going to control for the latter the type of error generated.

```python
@lottery.route("/lottery/mint", methods=["POST"])
@login_required
@manager_required
def mint():
    """
    Mint a new token for the current user, and redirect to the home page.
    If the user is not the owner, flash a message and redirect to the home page.
    If the user is the owner, mint a new token and redirect to the home page.
    """
    collectible = request.form.get("collectible")
    id = request.form.get("collectibleId")
    if not collectible and not id:
        abort(400)

    collectible = COLLECTIBLES.get(int(id))
    tx_result = LotteryProcessor.mint(
        collectible.id, collectible.collectible, collectible.rank
    )
    if tx_result:
        flash("Collectible minted successfully")
        COLLECTIBLES[int(id)].owner = NftProcessor.owner_of(collectible.id)
        return redirect(request.referrer or url_for("home.index"))

    # Check if the lottery is open
    if not LotteryProcessor.is_open():
        flash("The lottery is closed")
        return redirect(request.referrer or url_for("lottery.lottery_home"))

    # Check if the token is already owned
    if LotteryProcessor.is_already_minted(int(id)):
        flash("The token is already owned")
        return redirect(request.referrer or url_for("lottery.lottery_home"))

    flash("Error during minting")
    return redirect(request.referrer or url_for("home.index"))
```

```python
@lottery.route("/lottery/buy-ticket", methods=["POST"])
@login_required
@user_required
def buy_ticket():
    """
    Buy a ticket for the current user and redirect to the home page
    if the transaction is successful.
    Else, flash a message and redirect to the home page.
    """
    one = int(request.form.get("one"))
    two = int(request.form.get("two"))
    three = int(request.form.get("three"))
    four = int(request.form.get("four"))
    five = int(request.form.get("five"))
    powerball = int(request.form.get("powerball"))
    if not one or not two or not three or not four or not five or not powerball:
        abort(400)

    tx_result = LotteryProcessor.buy_ticket(one, two, three, four, five, powerball)
    if tx_result:
        flash("Tickets bought successfully")
        TICKETS.append(
            Ticket(
                buyer=current_user.id,
                one=one,
                two=two,
                three=three,
                four=four,
                five=five,
                powerball=powerball,
            )
        )
        return redirect(url_for(".lottery_home"))

    # Check if the lottery is open
    if not LotteryProcessor.is_open():
        flash("The lottery is closed")
        return redirect(url_for("lottery.lottery_home"))

    # Check if the round is active
    if not LotteryProcessor.is_round_active():
        flash("The round is not active")
        return redirect(url_for("lottery.lottery_home"))

    # Generic error message
    flash("Error during buying tickets")
    return redirect(url_for("lottery.lottery_home"))
```

1.5 Notifications

Following are the route that exposes smart contract event notifications and the script embedded in the web pages that call a GET to this route and updates the user's GUI with relevant notifications in real time.

```
@notification.route("/notifications", methods=["GET"])
def notifications():
    """
    Get all events from the lottery contract.
    return: status 200 and json object of all notifications if any
    else 204 (no content / user not interested in the lottery) status
    """
    if not "starting_block" in session:
        return jsonify(status=204)

    events_entries = (
        LotteryProcessor.lottery_created_event.get_all_entries()
        + LotteryProcessor.lottery_closed_event.get_all_entries()
        + LotteryProcessor.round_opened_event.get_all_entries()
        + LotteryProcessor.winning_numbers_drawn_event.get_all_entries()
        + LotteryProcessor.prize_assigned.get_all_entries()
        + LotteryProcessor.token_minted.get_all_entries()
    )

    # order the events by block number in ascending order
    events_entries.sort(key=lambda x: x.blockNumber)

    events = []
    for e in events_entries:
        block_id = e.blockNumber
        event = e.event
        args = e.args
        # Avoid notifications if they was generated before the user logged in
        if block_id <= session.get("starting_block"):
            continue
        block_id = str(e.blockNumber)
        # Check if the event is already notified
        if not session.get(block_id):
            session[block_id] = []
        if event not in session.get(block_id):
            # check if the event is 'TokenMinted' or 'PrizeAssigned' to update the owner of the collec
            if event == "TokenMinted" or event == "PrizeAssigned":
                # Update the owner of the collectible
                token_id = int(args._tokenId)
                COLLECTIBLES[token_id].owner = NftProcessor.owner_of(token_id)
            # Not display the event if the event is 'TokenMinted'
            if event == "TokenMinted":
                continue
            session[block_id].append(event)
            # get all arguments of the event
            str_args = ""
            for k, v in args.items():
                str_args += f"{k}: {v}; "
            events.append(event + "(" + str_args + ")")

    session["events"] = session.get("events", []) + events

    if len(events) > 0:
        return jsonify(
            status=200, events=events, non_read_events=session.get("events", [])
        )
    else:
        return jsonify(status=204)
```

Script that performs a call to the notification route and updates the GUI in real time.

```
// reload notification every 10 seconds
setInterval(function () {
    $.ajax({
        url: '/notifications',
        success: function (response) {
            console.log(response)
            if (response['status'] == 200) {
                // Get the notifications of contract events
                var events = response['events'];
                var nonReadEvents = response['non_read_events'];
                // Create the notifications
                $('#delete-notifications-btn').remove();
                $('#notifications-count').css('display', 'block');
                $('#notifications-count').text(nonReadEvents.length)

                // Append new notifications
                for (var i = 0; i < events.length; i++) {
                    var event = events[i];
                    var htmlEvent = '<div class="alert alert-success alert-dismissible fade show"
                        role="alert" style="margin: 16px">' +
                        '<strong>' + event + '</strong>' +
                        '<button type="button" class="btn close" data-dismiss="alert"
                        aria-label="Close">' +
                        '<span aria-hidden="true">x</span>' +
                        '</button>' +
                        '</div>';
                    $('#notifications').append(htmlEvent);
                    var smallEvent = '<div style="font-size: 12px; padding: 2px">'
                    + event + '</div>';
                    $('#notifications-body').append(smallEvent);
                    $('#notifications-body').append('<hr>');
                }
                var deleteButton = '<button id="delete-notifications-btn"
                style="background-color: red; font-size: 12px;" type="button"
                class="btn-rounded">Delete</button>';
                $('#notifications-body').append(deleteButton);
            }
        },
    });
}, 10000);
```