PAB

Manuel Pepe

Jan 03, 2022

CONTENTS:

PAB is a framework that helps with development and automation of periodic tasks on blockchains.

PAB allows you to quickly implement Strategies without worring about some implementation details, like connecting to a blockchain, retrieving contracts and sending transactions. It also comes with a simple scheduler so you can setup jobs and forget about them (don't forget about them).

CHAPTER

ONE

INTRODUCTION

With PAB, you quickstart your blockchain development and prototyping. After running *pab init* to create a new project, you can jump right into developing your own strategies.

With little more configuration, you can connect to any Web3 compatible network using an RPC, load contracts from the network, and use any account you have the Private Key of to authenticate against the network (if you need to make transactions).

1.1 Sample Strategy

Here's a basic sample strategy to give you an idea of the *Strategy API*:

```
import csv
from datetime import datetime
from pab.strategy import BaseStrategy

class CompoundAndLog(BaseStrategy):
    """ Finds pool in `controller` for `token`, compounds the given pool for
    `self.accounts[account_index]` and logs relevant data into a csv file. """

    def __init__(self, *args, filepath: str = "compound.csv", token: str = '',␣
↪controller: str = '', account_index: int = 0):
        super().__init__(*args)
        self.filepath = filepath
        self.account = self.accounts[account_index]
        self.token = self.contracts.get(token)
        self.controller = self.contracts.get(controller)
        self.pool_id = self.controller.functions.getPoolId(self.token.address).call()
        if not self.pool_id:
            raise Exception(f"Pool not found for {self.token} in {self.controller}")

    def run(self):
        """ Strategy entrypoint. """
        balance = self.get_balance()
        new_balance = self.compound()
        self.write_to_file(balance, new_balance)
        self.logger.info(f"Current balance is {balance}")

    def compound(self) -> int:
        """ Calls compound function of the `controller` contract to compound pending␣
↪profits
```

(continues on next page)

(continued from previous page)

```
        on the `token` pool. """
        self.transact(self.account, self.controller.functions.compound, (self.pool_id,␣
↪))
        return self.get_balance()

    def get_balance(self) -> int:
        """ Returns accounts balance on `controller` for `token` pool. """
        return self.controller.functions.balanceOf(
            self.account.address,
            self.pool_id
        ).call()

    def write_to_file(self, old_balance: int, new_balance: int):
        """ Write some number to a file. """
        now = datetime.now().strftime('%Y-%m-%d %I:%M:%S')
        diff = new_balance - old_balance
        with open(self.filepath, 'a') as fp:
            writer = csv.writer(fp)
            writer.writerow([now, new_balance, diff])
```

BaseStrategy childs can use *self.accounts*, *self.contracts* and *self.transact*. They also need to implement the *run()* method.

For more information on the Strategy API see *Strategy API* docs.

CHAPTER

TWO

GUIDE

2.1 Setup Project

2.1.1 Create project directory and venv

First you need to create a directory to contain your project files and create a virtualenv.

```
$ mkdir -p ~/projects/MyNewProject
$ python3.10 -m venv venv
$ source venv/bin/activate
```

2.1.2 Install PAB and initialize project

Then install the *PyAutoBlockchain* dependency in your virtualenv and run *pab init* to create the basic project structure.

```
(venv) $ pip install PyAutoBlockchain
(venv) $ pab init  # Initialize project in current directory
```

2.1.3 Project Structure

...

2.2 Strategy Development

Custom Strategies are automatically loaded when runing *pab run*. PAB loads your strategies from a *strategies* module at your current working directory.

Any subclass of *pab.strategy.BaseStrategy* in the strategies module can be used to run a task. For more info on how to configure tasks, see *Configuring Tasks*.

2.2.1 Basic Strategy

Open *strategies/__init__.py* and write the following:

```
from pab.strategy import BaseStrategy
from pab.utils import amountToDecimal

class LogBalanceToFileStrategy(BaseStrategy):
    def __init__(self, *args, accix: int = 0):
        super().__init__(*args)
        self.user = self.accounts[0]

    def run(self):
        balance = self.blockchain.w3.get_balance(self.user)
        self.logger.info(f"Balance: {amountToDecimal(balance)})
```

This simple strategy will only log the balance of some account. It uses the *BaseStrategy.accounts* to retrieve the account at the *accix* index, and the current blockchain connection from *BaseStrategy.blockchain.w3* to get the account balance.

2.2.2 Strategies In-Depth

Accounts

Accounts, also called *wallets*, are used in blockchains as user controlled addresses that can sign transactions and data. For info on how to configure a accounts in PAB read *Loading Accounts*.

To use configured accounts in a BaseStrategy subclass, you can access the `pab.strategy.BaseStrategy.accounts` attribute (e.g. *self.accounts[0]*, *self.accounts[1]*).

```
class MyStrategy(BaseStrategy):
    def run(self):
        user = self.accounts[0]
```

Accounts Attributes

Accounts in *self.accounts* are instances of LocalAccount.

Loading Order

As you see in *Loading Accounts*, there are two ways of loading accounts but only one list. The way the list is filled is by first loading accounts from environment variables into their fixed indexes, and then filling the gaps from 0 to N with keyfiles.

This means that if you have the following environment variables:

```
# .env.prod
PAB_PK1=ACC-A
PAB_PK2=0xACC-B
PAB_PK5=0xACC-C
```

And you run with two keyfiles like this:

```
$ pab run -e prod -k ACC-D.keyfile,ACC-E.keyfile
```

The accounts dictionary for a strategy will look like this:

```
>>> print(self.accounts)
{
    0: LocalAccount("0xACC-D"),
    1: LocalAccount("0xACC-A"),
    2: LocalAccount("0xACC-B"),
    3: LocalAccount("0xACC-E"),
    5: LocalAccount("0xACC-C")
}
```

To avoid the confusion that using both methods might cause, we recomend you stick to one method of loading accounts.

Contracts

PAB automatically loads the contracts defined in *Registering Contracts*. Strategies can fetch them by name using the *pab.strategy.BaseStrategy.contracts* attribute.

For example:

```
class MyStrategy(BaseStrategy):
    def run(self):
        contract = self.contacts.get("MY_CONTRACT")
```

Transactions

Subclasses of BaseStrategy will have a *pab.strategy.BaseStrategy.transact()* method that you can use to sign and send transactions.

For example:

```
class MyStrategy(BaseStrategy):
    def run(self):
        user = self.accounts[0]
        contract = self.contacts.get("MY_CONTRACT")
        params = ("param1", 2)
        rcpt = self.transact(user, contract.functions.someFunction, params)
```

Read-Only Queries

You can make readonly queries directly from the contract, without using *self.transact*.

```
class MyStrategy(BaseStrategy):
    def run(self):
        contract = self.contacts.get("MY_CONTRACT")
        params = ("param1", 2)
        some_data = contract.functions.getSomeData(*params).call()
```

Read-Only queries do not consume gas.

Blockchain and Web3

To access the underlying Web3 connection you can use the *pab.blockchain.Blockchain.w3* attribute. You can get the current *Blockchain* object from your strategie' s *pab.strategy.BaseStrategy.blockchain*.

2.3 Configuration

2.3.1 Blockchain Connection Setup

An RPC is needed for PAB to communicate with the blockchain networks. Some known RPCs with free tiers are Infura and MaticVigil.

RPC endpoint can be loaded from the *PAB_CONF_ENDPOINT* environment variable or from the *endpoint* config.

2.3.2 Loading Accounts

Multiple accounts can be dynamically loaded from the environment or keyfiles. All accounts can be used by any BaseStrategy child, for more info on how to use them see *Accounts* and *Transactions*.

From Environment

You can set the environment variables *PAB_PK1*, *PAB_PK2*, etc as the private keys for the accounts.

For example:

```
$ export PAB_PK0="0xSomePrivateKey"
$ pab run
```

From Keyfiles

A keyfile is a file that contains your private key, encrypted with a password. You can create a keyfile with `pab create-keyfile`.

You can then load them with *pab run –keyfiles key1.file,key2.file*. Accounts loaded through keyfiles require a one-time interactive authentication at the start of the execution.

For example, to create a keyfile and use it:

```
$ pab create-keyfile -o me.kf
Enter private key: 0xSomePrivateKey
Enter keyfile password:
Repeat keyfile password:
Keyfile written to 'me.kf'
$ pab run -k me.kf
Enter me.kf password:
```

2.3.3 Registering Contracts

Contracts are loaded from the *contracts.json* file at the project root. An example would be:

```
{
    "MYTOKEN": {
        "address": "0x12345",
        "abifile": "mytoken.abi"
    }
}
```

In this example, you also need to create the abifile at *abis/mytoken.abi* with the ABI data. You need to do this for all contracts.

Strategies can then get and use this contract with *self.contracts.get("MYTOKEN")*.

2.3.4 Configuring Tasks

Tasks are loaded from the *tasks.json* file at the project root. The following example defines a single task to execute, using the strategy *BasicCompound* that repeats every 24hs.

Multiple contract names (BNB, WBTC, PAIR, MASTERCHEF, ROUTER) are passed to the strategy as params. The strategy later uses these names to query the contracts from *BaseStrategy.contracts*.

```
[
    {
        "strategy": "BasicCompound",
        "name": "Compound BNB-WBTC",
        "repeat_every": {
            "days": 1
        },
        "params": {
            "swap_path": ["BNB", "WBTC"],
            "pair": "PAIR",
            "masterchef": "MASTERCHEF",
            "router": "ROUTER",
            "pool_id": 11
        }
    }
]
```

Tasks are defined as dictionaries with:

- *strategy*: Class name of strategy (must be subclass of *pab.strategy.BaseStrategy*, see *pab list-strategies*)
- *name*: Name, just for logging.
- *params*: Dictionary with strategy parameters. (see *pab list-strategies -v*)
- *repeat_every*: _Optional_. Dictionary with periodicity of the process, same arguments as *datetime.timedelta*.

Run *pab list-strategies -v* to see available strategies and parameters.

2.4 Running PAB

To initialize the PAB process that periodically runs all tasks:

```
$ pab run
```

2.4.1 Enviornments

To load a *.env.<env_name>* with PAB you can use:

```
$ pab run -e env_name
```

You can load multiple envfiles separating them with commas. They must be located at the project root.

CHAPTER

THREE

API

3.1 Strategy API

For more info see *Strategies In-Depth*.

pab.strategy.**import_strategies**(*root: pathlib.Path*)
Imports *strategies* module from *root* directory.

class pab.strategy.**BaseStrategy**(*blockchain:* pab.blockchain.Blockchain, *name: str*)
Abstract Base Class for custom strategies.

blockchain: *pab.blockchain.Blockchain*
Current blockchain connection.

property accounts: Dict[int, 'LocalAccount']
Returns available accounts in current blockchain. You can access specific accounts with numeric indexes

property contracts: ContractManager
Returns a pab.contract.ContractManager. You can use *self.contracts.get(name)* to retrieve a contract by name.

abstract run()
Strategy entrypoint. Must be defined by all childs.

transact(*account: LocalAccount*, *func: callable*, *args: tuple*) → TxReceipt
Makes a transaction on the current blockchain.

exception pab.strategy.**PABError**
Base class for errors while running tasks. Unhandleded PABErrors will prevent further excecutions of a strategy.

exception pab.strategy.**RescheduleError**
Strategies can raise this exception to tell PAB to reschedule them in known scenarios.

exception pab.strategy.**SpecificTimeRescheduleError**(*message: str*, *next_at: Optional[int] = None*)
Same as *RescheduleError* but at a specific time passed as a timestamp.

3.2 Blockchain API

class pab.blockchain.**Blockchain**(*root: pathlib.Path*, *config: pab.config.Config*, *accounts: Dict[int, LocalAccount]*)

Web3 connection manager.

rpc: str
Network RPC URL

id: int
Network Chain ID

name: str
Network name

w3: Web3
Internal Web3 connection

accounts: Dict[int, 'LocalAccount']
List of loaded accounts

contracts: ContractManager
Initialized contract manager

transact(*account: LocalAccount*, *func: callable*, *args: tuple*) → TxReceipt
Uses internal transaction handler to submit a transaction.

3.3 Contract API

class pab.contract.**ContractManager**(*w3: Web3*, *root: pathlib.Path*)

Stores contract definitions (address and location of the abi file). Reads and returns contracts from the network.

_load_contracts() → list['Contract']
Reads *contracts.json* from *self.root* and checks format.

_check_valid_contract_data(*data: dict*) → None

_check_abifile_exists(*abifile*) → None

get(*name: str*) → Contract
Returns contract by name. Contract must be defined in *CONTRACTS_FILE* and *ABIS_DIR*.

exception pab.contract.**ContractDefinitionError**

3.4 Transaction API

class pab.transaction.**TransactionHandler**(*w3: web3.Web3*, *chain_id: int*, *config: Config*)

w3
Internal Web3 connection.

chain_id
ChainID of current blockchain for transactions.

config
Config data.

transact(*account: LocalAccount*, *func: callable*, *args: tuple*, *timeout: Optional[int] = None*) → TxReceipt
Submits transaction and returns receitp.

_build_signed_txn(*account: LocalAccount*, *func: callable*, *args: tuple*) → SignedTransaction
Builds a signed transactoin ready to be sent to the network.

_txn_details(*account: LocalAccount*, *call: callable*) → dict
Returns transaction details such as chainId, gas, gasPrice and nonce.

gas(*call: callable*) → int
Returns gas allocated for transaction. Depending on the PAB configs it returns an estimation or a fixed value.

_estimate_call_gas(*call: callable*) → int
Returns estimated gas for a given call.

gas_price() → web3.types.Wei

exception pab.transaction.**TransactionError**

3.5 Accounts API

pab.accounts.**create_keyfile**(*path: pathlib.Path*, *private_key: str*, *password: str*) → None
Creates a keyfile using Web3.eth.account.encrypt.

pab.accounts.**_load_keyfile**(*keyfile: pathlib.Path*) → Optional[eth_account.signers.local.LocalAccount]
Loads accounts from keyfile. Asks for user input.

pab.accounts.**_get_ix_from_name**(*name*) → Optional[int]
Returns the index from *PAB_PK<INDEX>*.

pab.accounts.**_load_from_env**() → Dict[int, eth_account.signers.local.LocalAccount]
Private keys are loaded from the environment variables that follow the naming convention *PAB_PK<ix>*. *ix* will be the index in the accounts list.

pab.accounts.**load_accounts**(*keyfiles: list[pathlib.Path]*) → Dict[int, eth_account.signers.local.LocalAccount]
Load accounts from environment variables and keyfiles

exception pab.accounts.**AccountsError**

exception pab.accounts.**KeyfileOverrideException**

3.6 Task API

pab.task.**TaskList**
Type for an explicit list of Tasks

alias of list[Task]

pab.task.**RawTasksData**
Type for an explicit list of task data dictionaries.

alias of List[dict]

class pab.task.**Task**(*id_: int*, *strat:* pab.strategy.BaseStrategy, *next_at: int*, *repeat_every: Optional[dict] = None*)
Container for a strategy to be executed in the future

RUN_ASAP: int = -10
Constant. Means job should be rescheduled to run ASAP.

RUN_NEVER: int = -20
Constant. Means job should't be rescheduled.

id
Internal Task ID

strategy: *pab.strategy.BaseStrategy*
Strategy object

next_at: int
Next execution time as timestamp

repeat_every: dict | None
Repetition data. A dict that functions as kwargs for *datetime.timedelta*

last_start: int
Last execution start time as timestamp

reschedule() → None
Calculates next execution if applies and calls *schedule_for()*

repeats() → bool
True if Task has repetition data.

next_repetition_time() → int
Returns next repetition time based on *last_start* and *repeat_every*.

schedule_for(*next_at: int*) → None
Updates `self.next_at` for a specific time. Will disable job if value is *Task.RUN_NEVER*.

is_ready() → bool
Returns True if job is ready to run based on *next_at*.

process() → None
Calls *_process()* and handles *pab.strategy.RescheduleError*.

_process() → None
Runs strategy and updates schedule.

class pab.task.**TaskFileParser**(*root: pathlib.Path*, *blockchain:* pab.blockchain.Blockchain, *strategies: list[*pab.strategy.BaseStrategy*]*)
Parses a tasks file and loads a TaskList.

load() → *pab.task.TaskList*
Loads TaskList from tasks file.

_load_tasks_json_or_exception(*fhandle: TextIO*) → *pab.task.RawTasksData*
Parses TextIO input as JSON, validates and returns raw data. May raise *TasksLoadError*.

_validate_raw_data(*data: Any*) → None
Validates raw tasks data format.

_create_tasklist(*tasks:* pab.task.RawTasksData) → *pab.task.TaskList*
Creates a list of *Task* objects from raw data.

_create_strat_from_data(*data: dict*) → *pab.strategy.BaseStrategy*
Creates a single *Task* object from raw data.

_find_strat_by_name(*name: str*) → Optional[Callable]
Finds a strategy by name. May raise *UnkownStrategyError*.

exception pab.task.**TasksLoadError**

exception pab.task.**UnkownStrategyError**

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

G

I

K

L

M

N

P

R

S

T

U

W