PAB

Manuel Pepe

Jun 26, 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 your strategy development.

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).

PAB also comes with a pytest plugin that allows for easy strategy testing (see *Testing*).

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, token: str, controller: str, filepath: str = "compound.csv
↪", 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:
```

(continues on next page)

(continued from previous page)

```
        """ Calls compound function of the `controller` contract to compound pending profits
        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* and *Strategy Development* 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

Barebones

This is an example of the most basic structure required by a PAB project.

```
MyPABProject
├── abis
│   └── SomeSmartContract.abi
├── .env
├── config.json
├── contracts.json
├── tasks.json
└── strategies.py
```

This structure does not have tests.

Recommended

A better project structure that takes into account testing with pytest.

This is the recommended structure when testing a PAB project:

```
MyPABProject
├── abis
│   └── SomeSmartContract.abi
├── .env
├── config.json
├── contracts.json
├── tasks.json
├── strategies.py
├── requirements.txt   # If you have extra dependencies
├── pytest.ini
└── tests
    ├── __init__.py
    ├── conftest.py
    ├── truffle        # Default directory for testing contracts source.
    │   ├── contracts
    │   │   ├── MainContract.sol
    │   │   └── AnotherContract.sol
    │   ├── migrations
    │   │   └── 1_initial_migration.js
    │   └── truffle-config.js
    └── test_basic.py
```

For more info on testing strategies see *Testing*.

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):
```

(continues on next page)

(continued from previous page)

```
        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, CONTROLLER, 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",
            "controller": "CONTROLLER",
            "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

PAB is a framework for developing and running custom tasks in crypto blockchains.

```
usage: pab [-h] {init,run,list-strategies,create-keyfile} ...
```

2.4.1 Sub-commands:

init

Initialize PAB project in current directory.

```
pab init [-h] [-d DIRECTORY]
```

options

-d, --directory Initialize project in diferent directory.

run

Run PAB

```
pab run [-h] [-k KEYFILES] [-e ENVS] {tasks,strat} ...
```

options

-k, --keyfiles List of keyfiles separated by commas.

Default: “”

-e, --envs List of environments separated by commas.

Default: “”

Sub-commands:

tasks

Run and schedule all tasks from ‘tasks.json’.

```
pab run tasks [-h]
```

strat

Run a single strategy by name.

```
pab run strat [-h] --strategy STRATEGY
```

options

--strategy Name of the strategy to run

list-strategies

List strategies and parameters

```
pab list-strategies [-h] [-v] [-j]
```

options

-v, --verbose Print strategy parameters

Default: False

-j, --json Print strategies as JSON

Default: False

create-keyfile

Create keyfile. You'll need your private key and a new password for the keyfile.

```
pab create-keyfile [-h] [-o OUTPUT]
```

options

-o, --output Output location for keyfile.

Default: "key.file"

2.5 Testing

PAB comes with a pytest plugin `pab.test`. It allows you to test your strategies against local deployments of Smart Contracts by providing some usefull functionallity like starting a Ganache server, building and deploying your contracts with Truffle and automatically fill in some necessary/usefull info like Ganache test accounts private keys and connection endpoint, and the temporary contract addresses on the local network.

2.5.1 Example Use Case

Say you have a strategy that increments a counter on a Smart Contract every time it runs by calling a `incrementCounter()` method on the contract.

Maybe a good test would be to deploy a contract into a test network, run the strategy a couple of times and check that the counter is correctly updated. The PAB.test plugin eases this process, and allows you to easily write an automated test that you can run through `pytest`.

You would only need a truffle project with one contract that has a compatible ABI, this can be a Mock or even the live contract code if you have access to it.

For our example:

```
contract Counter {
    unit8 counter;

    constructor() {
        counter = 0;
    }

    function getCounter() public view returns (uint8) {
        return counter;
    }

    function incrementCounter () public {
        counter += 1;
    }
}
```

Then, you can write a test like this:

```
from strategies import MyStrategy

def test_asd(setup_project, get_strat):
    with setup_project() as pab:
        params = {"contract": "Counter"}
        strat: MyStrategy = get_strat(pab, 'MyStrategy', params)
        assert strat.vault_token.functions.getCounter().call() == 0
        strat.run()
        assert strat.vault_token.functions.getCounter().call() == 1
        strat.run()
        assert strat.vault_token.functions.getCounter().call() == 2
```

That's it, when the plugin loads it will deploy the `Counter` contract into a test server and automatically register the contract with the correct test address and ABI. `Counter` **doesn't** need to be registered as a contract name.

2.5.2 PAB Testing Plugin

The `pab.test` plugin will start a local Ganache server, test and deploy your contracts with Truffle and install some usefull fixtures for your tests.

It works in the following way:

When a pytest session is created, the plugin starts a local Ganache server by spawning a `ganache-cli` subprocess. It will parse the ganache startup output and collect the connection data (host, port) and the auto-generated private keys.

After the Ganache server is running, it will run `truffle deploy` on all directories specified in the `pab-contracts-sources` config (see *Plugin Configuration*). The output of the deployments is parsed to collect the relevant contract data. The default network for deployment is `` `development `` but it can be changed with the `pab-truffle-network` config. All contract sources must be valid Truffle Project.

The last thing done in initialization is to register two pytest fixtures, *`pab.test.setup_project()`* and *`pab.test.get_strat()`*. You can read more about them in the *Test API*.

When the pytest session finishes, the ganache process is stopped.

To use the plugin, you must manually install Ganache and Truffle, probably through npm. The plugin will check if both dependencies are installed as `ganache-cli` and `truffle`.

Test Case Example

The following is a sample test case written with the help of the *PAB Testing Plugin*.

```
# MyProject/tests/test_basic.py
def test_basic_run(setup_project, get_strat):
    with setup_project("MyProject") as pab:
        params = {
            "contract_a": "ABC",
            "contract_b": "DEF"
        }
        strat = get_strat(pab, "MyStrategyABC", params)
        strat.run()
        assert strat.contract_a.functions.getSomeValue.call() == 'Some Value'
```

The example uses the `setup_project` fixture to initialize the PAB test project and return a *pab.core.PAB* instance. Inside the context of `setup_project`, the `get_strat` fixture is used to retrieve a single strategy from the PAB app, initialized with certain parameters. Finally executes the strategy and asserts that a side-effect (in this case, a value change on some contract attribute) happened.

Plugin Configuration

To enable the plugin you need to add `pab.test` to `pytest_plugins` in your `conftest.py`:

```
# MyProject/tests/conftest.py
import pytest
pytest_plugins = ["pab.test"]
```

You can also change some configurations in your `pytest.ini`:

```
# MyProject/pytest.ini
[pytest]
pab-ignore-patterns-on-copy =
    venv/
pab-contracts-sources =
    tests/contracts
pab-truffle-network = development
```

Ganache Configuration

There's not much configuration that's necessary for ganache. PAB starts the process by running `ganache-cli` without extra parameters. By default, this starts a server in at `127.0.0.1:8545`.

Truffle Configuration

The only necessary configuration for truffle is to correctly setup your network parameters. Ganache defaults are `127.0.0.1:8545`, so make sure that there is a network with those parameters.

For example, in your `truffle-config.js`:

```
{
    # More configs above
    networks: {
        development: {
            host: "127.0.0.1",
            port: 8545,
            network_id: "*",
        }
    }
    # More configs below
}
```

Recommended Structure

A sample project structure can be found at *Project Structure*.

2.6 All Configs

2.6.1 Available configurations

Table 1: All Configs

Path	Format	Default	Help
blockchain	string	''	Network name
chainId	int	''	Chain ID
endpoint	string	''	RPC Endpoint
transactions.timeout	int	200	A blocking timeout after making a transaction. Defined in seconds.
transactions.gasPrice.number	float	1.1	Gas multiplier
transactions.gasPrice.unit	string	gwei	Gas unit
transactions.gas.useEstimate	bool	''	If true, gas is estimated using *web3.eth.estimateGas()*
transactions.gas.exact	int	200000	If useEstimate is disabled, this exact amount will be available as gas for all txs.
emails.enabled	bool	''	If true, emails will be sent on failures.
emails.host	string	localhost	SMTP host
emails.port	int	587	SMTP port
emails.user	string	''	SMTP user. Used to login and as FROM value.
emails.password	string	''	SMTP password
emails.recipient	string	''	Recipient for email alerts

All Defaults

```
{
    "blockchain": "",
    "chainid": 0,
    "endpoint": "",
    "transactions": {
        "timeout": 200,
        "gasprice": {
            "number": 1.1,
            "unit": "gwei"
        },
        "gas": {
            "useestimate": false,
            "exact": 200000
        }
    },
    "emails": {
        "enabled": false,
        "host": "localhost",
        "port": 587,
        "user": "",
        "password": "",
```

(continues on next page)

(continued from previous page)

```
        "recipient": ""
    }
}
```

CHAPTER

THREE

API

3.1 Strategy API

For more info see *Strategies In-Depth*.

pab.strategy.**StrategiesDict**

TypeAlias for a mapping of *strategy_name: strategy_class*

pab.strategy.**import_strategies**(*root: Path*)

Imports *strategies* module from *root* directory.

class pab.strategy.**BaseStrategy**(*blockchain:* Blockchain, *name: str*)

Abstract Base Class for custom strategies.

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: int | float*)

Same as *RescheduleError* but at a specific time passed as a timestamp.

3.2 Blockchain API

class pab.blockchain.**Blockchain**(*root: Path*, *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.**ContractData**(*name: str*, *address: str*, *abi: str*)

Stores smart contract data

name: str

address: str

abi: str

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

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

_load_contracts() → Dict[str, *ContractData*]

Reads and parses *contracts.json*.

_parse_contract_data(*contracts_data: dict[str, dict]*) → Dict[str, *ContractData*]

Replaces the raw contract data from *contracts.json* with the `ContractData` dataclass.

_check_valid_contract_data(*data: dict*) → None

Validates that the data loaded from *contracts.json* is valid.

_check_abifile_exists(*abifile*) → None

Validates that all abifiles defined in *contracts.json* exist.

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() → Wei

exception pab.transaction.**TransactionError**

3.5 Accounts API

pab.accounts.**create_keyfile**(*path: Path*, *private_key: str*, *password: str*) → None

Creates a keyfile using Web3.eth.account.encrypt.

pab.accounts.**_load_keyfile**(*keyfile: Path*) → Optional[LocalAccount]

pab.accounts.**_get_ix_from_name**(*name*) → Optional[int]

Returns the index from *PAB_PK<INDEX>*.

pab.accounts.**_load_from_env**() → Dict[int, 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, 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:* 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: *BaseStrategy*

Strategy object

next_at: int

Next execution time as timestamp

repeat_every: dict[str, int] | 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: Path*, *blockchain:* Blockchain, *strategies: dict[str, type['BaseStrategy']]*)

Parses a tasks file and loads a TaskList.

REQUIRED_TASK_FIELDS: list[str] = ['name', 'strategy']

Fields that must be declared in all tasks.

root: Path

Root of the project.

blockchain: *Blockchain*

`Blockchain` used by tasks.

strategies: dict[str, type['BaseStrategy']]

Strategies dictionary.

load() → TaskList

Loads TaskList from tasks file.

_load_tasks_json_or_exception(*fhandle: TextIO*) → RawTasksData

Parses TextIO input as JSON, validates and returns raw data. May raise *TasksFileParseError*.

_validate_raw_data(*data: Any*) → None

Validates raw tasks data format. May raise *TasksFileParseError*.

_create_tasklist(*tasks: RawTasksData*) → TaskList

Creates a list of *Task* objects from raw data. May raise *TaskLoadError*.

_create_strat_from_data(*data: dict*) → *BaseStrategy*

Creates a single *Task* object from raw data. May raise *TaskLoadError*.

_find_strat_by_name(*name: str*) → type[*pab.strategy.BaseStrategy*]

Finds a strategy by name. May raise *UnkownStrategyError*.

exception pab.task.**TasksFileParseError**

Error while parsing tasks.json

exception pab.task.**TaskLoadError**

Error while loading a task

exception pab.task.**UnkownStrategyError**

A strategy required by a task could not be found.

3.7 Core API

class pab.core.**PAB**(*root: Path*, *keyfiles: Optional[list[pathlib.Path]] = None*, *envs: Optional[list[str]] = None*)

Loads PAB project from a given path, including configs from all sources, strategies from the *strategies* module, and accounts.

class pab.core.**Runner**(*pab:* PAB)

Base class for PAB Runners.

Runners execute PAB strategies in different manners. All runners must implement `run`.

abstract run()

Main run method.

_abc_impl = <_abc._abc_data object>

class pab.core.**TasksRunner**(**args*)

Loads and runs tasks from *tasks.json*.

ITERATION_SLEEP = 60

run()

Main run method.

process_tasks()

process_item(*item:* Task)

_sleep()

_abc_impl = <_abc._abc_data object>

class pab.core.**StratParam**(*name: 'str'*, *type: 'Callable | str'*, *default: 'Any'*)

name: str

type: Union[Callable, str]

default: Any

class pab.core.**SingleStrategyRunner**(**args*, *strategy: str*, *params: list[str]*)

Runs a single strategy, one time, with custom parameters.

TYPES = {'float': <class 'float'>, 'int': <class 'int'>, 'str': <class 'str'>}

run()

Main run method.

_parse_params(*strat_class: type[*pab.strategy.BaseStrategy*]*) → Namespace

_get_strat_parser(*strat_class: type[*pab.strategy.BaseStrategy*]*) → ArgumentParser

Builds an ArgumentParser from the signature of the current strategy __init__ method.

_validate_and_get_param_type(*param:* StratParam) → Callable

_get_strat_params(*strat*) → set[*pab.core.StratParam*]

Recursively retrieve a list of all keyword parameters for a strategy constructor.

_get_base_params() → list[str]

Returns the names of the base parameters of `BaseStrategy`.

_abc_impl = <_abc._abc_data object>

3.8 Test API

pab.test.**log**(*msg: str*)

Print message to screen during pytest execution.

pab.test.**pytest_addoption**(*parser*)

pab.test.**pytest_configure**(*config*)

pab.test.**_check_dependencies_installed**(*commands: list[str]*) → None

Checks if a list of commands are available on the system usin shutil.which.

class pab.test.**ParsedGanacheOutput**(*accounts: list[str]*, *endpoint: str*)

Dataclass for ganache data.

accounts: list[str]

endpoint: str

pab.test.**_parse_ganache_output**(*proc: Popen*) → *ParsedGanacheOutput*

Parses ganache process stdout for accounts and endpoint.

class pab.test.**PABPlugin**(*config*)

This plugin will:

- Verify that *ganache-cli* and *truffle* are installed.
- Start a *ganache-cli* subprocess.
- Collect the private keys of all test accounts in the local ganache network.
- Build and deploy your contracts with *truffle* into the local ganache network.
- Collect the addresses of the deployed contracts.
- Install fixtures

GANACHE_CMD = 'ganache-cli'

TRUFFLE_CMD = 'truffle'

pytest_sessionstart(*session*)

Pytest start hook

pytest_sessionfinish(*session*, *exitstatus*)

Pytest cleanup hook

_start_ganache_process()

Starts ganache-cli process.

_collect_ganache_data()

Parses ganache process output and collects accounts and network data.

_deploy_contracts_with_truffle()

Test, build and deploy contracts with ganache.

class pab.test.**Contract**(*name: 'str'*, *source: 'str'*, *address: 'str'*, *abi: 'str'*)

name: str

source: str

address: str

abi: str

class pab.test.**TruffleDeployer**(*sources: list[str]*, *network: str*)

CMD = 'truffle'

deploy() → dict[str, *pab.test.Contract*]

Spawns a truffle deploy process for each source, parses process output and returns a mapping of *contract_name: Contract*.

_run_truffle(*cwd: str*)

_parse_data_and_validate(*sources_data: dict[str, dict]*) → dict[str, *pab.test.Contract*]

_load_abis(*contracts: dict[str,* pab.test.Contract*]*) → dict[str, *pab.test.Contract*]

Load the contract ABIs from the truffle sources 'build' directory.

class pab.test.**TruffleOutputParser**

Parser for truffle output. Currently only supports the deploy process and returns a mapping of *contract_name: contract_data*.

RE_START_CONTRACT = re.compile("^(Deploying|Replacing) '(.*)'$")

RE_CONTRACT_DATA = re.compile('^> ([a-zA-Z]+):\\s+(.*)$')

class States(*value*)

An enumeration.

START_CONTRACT_DATA = 1

IN_CONTRACT_DATA = 2

END_CONTRACT_DATA = 3

OUTSIDE_CONTRACT_DATA = 4

in_state(*states:* States | *list[*States*]*)

parse_contracts_data(*proc: CompletedProcess*) → dict[str, dict]

Parses subprocess output. Returns a mapping of *contract_name: contract_data*.

pab.test.**_copy_project**(*dest: Path*, *ignored_patterns: Optional[list[str]] = None*) → Path

Copies CWD to *dest*.

pab.test.**_set_test_envfile**(*path: Path*, *extra_vars: dict[str, str]*, *plugin:* PABPlugin)

Creates '.env.test' file in *path*. *extra_vars* is a dict of envvars to add to the file. Also adds envvars from *plugin.envs* and accounts from *plugin.accounts*.

pab.test.**_replace_contracts**(*pab:* PAB, *plugin:* PABPlugin)

Replaces contracts in a PAB instance with the ones from *plugin.contracts*.

pab.test.**_temp_environ**()

ContextManager that Saves a copy of the environment and restores the environment variables to their original state after exiting. New variables will be removed and value changes will be undone.

pab.test.**setup_project**(*pytestconfig*)

This fixture will:

- Copy project to a temporary directory.
- Change CWD to temporary directory.
- Replace the contract adresses loaded from contracts.json with the local addresses.
- Create a '.env.test' environment file with local accounts and rpc configs.
- Restore environment variables after execution.

- Return a PAB instance.

pab.test.**get_strat**()

Fixture that returns an initialized strategy from a PAB instance.

exception pab.test.**PABTestException**

exception pab.test.**MissingDependencies**

exception pab.test.**StrategyNotFound**

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

G

I

K

L

M

N

O

P