py-evm Documentation

Release 0.1.0-alpha.17

Ethereum Foundation

Dec 08, 2018

General

CHAPTER 1

Trinity & Py-EVM

Py-EVM is a new implementation of the Ethereum Virtual Machine (EVM) written in Python. Trinity is the client software that connects to the Ethereum network and runs on top of Py-EVM.

Trinity and Py-EVM aim to replace existing Python Ethereum implementations to eventually become the defacto standard for the Python ecosystem.

If none of this makes sense to you yet we recommend to checkout the Ethereum website as well as a higher level description of the Ethereum project.

1.1 Py-EVM goals

The main focus is to enrich the Ethereum ecosystem with a Python implementation that:

- Supports Ethereum 1.0 as well as 2.0 / Serenity
- Is well documented
- Is easy to understand
- Has clear APIs
- Runs fast and resource friendly
- Is highly flexible to support:
  - Public chains
  - Private chains
  - Consortium chains
  - Advanced research

1.2 Trinity goals

While Py-EVM provides the low level APIs of the Ethereum protocol, it does not aim to implement a full or light node directly.

Trinity is a refernece implementation on top of Py-EVM that aims to:

- Provide a reference implementation for an Ethereum 1.0 node (alpha)
- Support "full" and "light" modes
- Fully support mainnet as well as several testnets
- Provide a reference implementation of an Ethereum 2.0 / Serenity beacon node (pre-alpha)
- Provide a reference implementation of an Ethereum 2.0 / Sereneity validator node (pre-alpha)

Note: Trinity is currently in **public alpha** and can connect and sync to the main ethereum network. While it isn't meant for production use yet, we encourage the adventurous to try it out. Follow along the *Trinity Quickstart* to get things going.

1.3 Further reading

Here are a couple more useful links to check out.

- *Trinity Quickstart*
- Source Code on GitHub
- Public Gitter Chat
- *Get involved*

CHAPTER 2

Table of contents

2.1 Introduction

2.1.1 Trinity & Py-EVM

Py-EVM is a new implementation of the Ethereum Virtual Machine (EVM) written in Python. Trinity is the client software that connects to the Ethereum network and runs on top of Py-EVM.

Trinity and Py-EVM aim to replace existing Python Ethereum implementations to eventually become the defacto standard for the Python ecosystem.

If none of this makes sense to you yet we recommend to checkout the Ethereum website as well as a higher level description of the Ethereum project.

Py-EVM goals

The main focus is to enrich the Ethereum ecosystem with a Python implementation that:

- Supports Ethereum 1.0 as well as 2.0 / Serenity
- Is well documented
- Is easy to understand
- Has clear APIs
- Runs fast and resource friendly
- Is highly flexible to support:
  - Public chains
  - Private chains
  - Consortium chains
  - Advanced research

Trinity goals

While Py-EVM provides the low level APIs of the Ethereum protocol, it does not aim to implement a full or light node directly.

Trinity is a refernece implementation on top of Py-EVM that aims to:

- Provide a reference implementation for an Ethereum 1.0 node (alpha)
- Support "full" and "light" modes
- Fully support mainnet as well as several testnets
- Provide a reference implementation of an Ethereum 2.0 / Serenity beacon node (pre-alpha)
- Provide a reference implementation of an Ethereum 2.0 / Sereneity validator node (pre-alpha)

Note: Trinity is currently in **public alpha** and can connect and sync to the main ethereum network. While it isn't meant for production use yet, we encourage the adventurous to try it out. Follow along the *Trinity Quickstart* to get things going.

Further reading

Here are a couple more useful links to check out.

- *Trinity Quickstart*
- Source Code on GitHub
- Public Gitter Chat
- *Get involved*

2.2 Quickstart

2.2.1 Installation

This is the quickstart guide for Trinity. If you only care about running a Trinity node, this guide will help you to get things set up. If you plan to develop on top of Py-EVM or contribute to the project you may rather want to checkout the *Contributing Guide* which explains how to set everything up for development.

Installing on Ubuntu

Trinity requires Python 3.6 as well as some tools to compile its dependencies. On Ubuntu, the `python3.6-dev` package contains everything we need. Run the following command to install it.

```
apt-get install python3.6-dev
```

Trinity is installed through the pip package manager, if pip isn't available on the system already, we need to install the `python3-pip` package through the following command.

```
apt-get install python3-pip
```

Note: **Optional:** Often, the best way to guarantee a clean Python 3 environment is with virtualenv. If we don't have `virtualenv` installed already, we first need to install it via pip.

```
pip install virtualenv
```

Then, we can initialize a new virtual environment `venv`, like:

```
virtualenv -p python3 venv
```

This creates a new directory `venv` where packages are installed isolated from any other global packages.

To activate the virtual directory we have to *source* it

```
. venv/bin/activate
```

Finally, we can install the `trinity` package via pip.

```
pip3 install -U trinity
```

Installing on macOS

First, install LevelDB and the latest Python 3 with brew:

```
brew install python3 leveldb
```

Note: **Optional:** Often, the best way to guarantee a clean Python 3 environment is with virtualenv. If we don't have `virtualenv` installed already, we first need to install it via pip.

```
pip install virtualenv
```

Then, we can initialize a new virtual environment `venv`, like:

```
virtualenv -p python3 venv
```

This creates a new directory `venv` where packages are installed isolated from any other global packages.

To activate the virtual directory we have to *source* it

```
. venv/bin/activate
```

Then, install the `trinity` package via pip:

```
pip3 install -U trinity
```

Installing through Docker

Trinity can also be installed using `Docker` which can be a lightweight alternative where no changes need to be made to the host system apart from having `Docker` itself installed.

Note: While we don't officially support Windows just yet, running Trinity through `Docker` is a great way to bypass this current limitation as Trinity can run on any system that runs `Docker` with support for linux containers.

Using `Docker` we have two different options to choose from.

1. Run an existing official image

This is the default way of running Trinity through `Docker`. If all we care about is running a Trinity node, using one of the latest released versions, this method is perfect.

Run:

```
docker run -it ethereum/trinity
```

Alternatively, we can run a specific image version, following the usual docker version schema.

```
docker run -it ethereum/trinity:0.1.0-alpha.13
```

2. Build your own image

Alternatively, we may want to try out a specific (unreleased) version. In that case, we can create our very own image directly from the source code.

```
make create-docker-image version=my-own-version
```

After the image has been successfully created, we can run it by invoking:

```
docker run -it ethereum/trinity:my-own-version
```

2.2.2 Running Trinity

After Trinity is installed we should have the `trinity` command available to start it.

```
trinity
```

While it may take a couple of minutes before Trinity can start syncing against the Ethereum mainnet, it should print out some valuable information right away which should look something like this. If it doesn't please file an issue to help us getting that bug fixed.

```
    INFO  05-29 01:57:02        main
  ______     _       _ __
/_  __/____(_)___  (_) /___  __
  / / / ___/ / __ \/ / __/ / / /
/ / / /  / / / / / / /_/ /_/ /
/_/ /_/  /_/_/ /_/_/\__/\__, /
                      /____/
    INFO  05-29 01:57:02        main  Trinity/0.1.0a4/linux/cpython3.6.5
    INFO  05-29 01:57:02        main  network: 1
    INFO  05-29 01:57:02         ipc  IPC started at: /root/.local/share/trinity/
↪mainnet/jsonrpc.ipc
    INFO  05-29 01:57:02      server  Running server...
    INFO  05-29 01:57:07      server  enode://
↪09d34ecb0de1806ab0e68cb2d822b967292dc021df06aab9a55aa4d2e1b2e04ae73560137407a48073286026e12dd60d26
↪0.0.0:30303
    INFO  05-29 01:57:07      server  network: 1
```

(continues on next page)

(continued from previous page)

```
    INFO  05-29 01:57:07        peer  Running PeerPool...
    INFO  05-29 01:57:07        sync  Starting fast-sync; current head: #0
```

Once Trinity successfully connected to other peers we should see it starting to sync the chain.

```
INFO  05-29 02:23:13       chain  Starting sync with ETHPeer <Node(0xaff0@90.114.124.
↪196)>
INFO  05-29 02:23:14       chain  Imported chain segment in 0 seconds, new head: #191␣
↪(739b)
INFO  05-29 02:23:15       chain  Imported chain segment in 0 seconds, new head: #383␣
↪(789c)
INFO  05-29 02:23:16       chain  Imported chain segment in 0 seconds, new head: #575␣
↪(a1d0)
INFO  05-29 02:23:17       chain  Imported chain segment in 0 seconds, new head: #767␣
↪(aeb6)
```

Running as a light client

Warning: It may take a **very** long time for Trinity to find an LES node with open slots. This is not a bug with trinity, but rather a shortage of nodes serving LES. Please consider running your own LES server to help improve the health of the network.

Use the `--light` flag to instruct Trinity to run as a light node.

Ropsten vs Mainnet

Trinity currently only supports running against either the Ethereum Mainnet or Ropsten testnet. Use `--ropsten` to run against Ropsten.

```
trinity --ropsten
```

2.2.3 Connecting to preferred nodes

If you would like to have Trinity prioritize connecting to specific nodes, you can use the `--preferred-node` command line flag. This flag takes an enode URI as a single argument and will instruct Trinity to prioritize connecting to this node.

```
trinity --preferred-node enode://
↪a41defa74e8d9d4152699cb9a0d195377da95833769ad6b386092ac3b16c184eb4ef4b4f02889e0b5097ff50fb5847ba99
↪0.0.1:30304
```

Using `--preferred-node` is a good way to ensure Trinity running in `--light` mode connects to known peers who serve LES.

2.2.4 Retrieving Chain information via web3

While just running `trinity` already causes the node to start syncing, it doesn't let us interact with the chain directly (apart from the JSON-RPC API).

However, we can attach an interactive shell to a running Trinity instance with the `attach` subcommand. The interactive `ipython` shell binds a web3 instance to the `w3` variable.

```
trinity attach
```

Now that Trinity runs in an interactive shell mode, let's try to get some information about the latest block by calling `w3.eth.getBlock('latest')`.

```
In [9]: w3.eth.getBlock('latest')
Out[9]:
AttributeDict({'difficulty': 743444339302,
'extraData': HexBytes('0x476574682f4c5649562f76312e302e302f6c696e75782f676f312e342e32
↪'),
'gasLimit': 5000,
'gasUsed': 0,
'hash': HexBytes('0x1a8487dfb8de7ee27b9cca30b6f3f6c9676eae29c10eef39b86890ed15eeed01
↪'),
'logsBloom': HexBytes(
↪'0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪'),
'mixHash': HexBytes(
↪'0xf693b8e4bc30728600da40a0578c14ddb7ad08a64e329a19d9355d5665588aef'),
'nonce': HexBytes('0x7382884a72533c59'),
'number': 12479,
'parentHash': HexBytes(
↪'0x889c36c51463f100cf50ec2e2a92886aa7ebb3f99fa8c817343214a92f967a29'),
'receiptsRoot': HexBytes(
↪'0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421'),
'sha3Uncles': HexBytes(
↪'0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347'),
'stateRoot': HexBytes(
↪'0x6ad1ecb7d516c679e7c476956159051fa32848f3ba631a47c3fb72937ed86987'),
'timestamp': 1438368997,
'transactionsRoot': HexBytes(
↪'0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421'),
'miner': '0xbb7B8287f3F0a933474a79eAe42CBCa977791171',
'totalDifficulty': 3961372514945562,
'uncles': [],
'size': 544,
'transactions': []})
```

You can attach to an existing Trinity process using the `attach` comand.

```
trinity attach
```

For a list of JSON-RPC endpoints which are expected to work, see this issue: https://github.com/ethereum/py-evm/issues/178

Warning: Trinity is currently in public alpha. **Keep in mind**:

- It is expected to have bugs and is not meant to be used in production
- Things may be ridiculously slow or not work at all
- Only a subset of JSON-RPC API calls are currently supported

2.3 Release notes

Trinity and Py-EVM are moving fast. Learn about the latest improvements in the release notes.

2.3.1 Trinity

0.1.0-alpha.17

Released November 20, 2018

- #1488: Bugfix: Bugfix for state sync to limit the number of open files.
- #1478: Maintenance: Improve logging messages during fast sync to include performance metrics
- #1476: Bugfix: Ensure that network connections are properly close when a peer doesn't successfully complete the handshake.
- #1474: Bugfix: EthStats fix for displaying correct uptime metrics
- #1471: Maintenance: Upgrade `mypy` to `0.641`
- #1469: Maintenance: Add logging to show when fast sync has completed.
- #1467: Bugfix: Don't add peers which disconnect during the boot process to the peer pool.
- #1465: Bugfix: Proper handling for when `SIGTERM` is sent to the main Trinity process.
- #1463: Bugfix: Better handling for bad server responses by EthStats client.
- #1443: Maintenance: Merge the `--nodekey` and `--nodekey-path` flags.
- #1438: Bugfix: Remove warnings when printing the ASCII Trinity header
- #1437: Maintenance: Update to use f-strings for string formatting
- #1435: Maintenance: Enable Constantinople fork on Ropsten chain
- #1434: Bugfix: Fix incorrect mainnet genesis parameters.
- #1421: Maintenance: Implement `eth_syncing` JSON-RPC endpoint
- #1410: Maintenance: Implement EIP1283 for updated logic for `SSTORE` opcode gas costs.
- #1395: Bugfix: Fix gas cost calculations for `CREATE2` opcode
- #1386: Maintenance: Trinity now prints a message to make it more clear why Trinity was shutdown.
- #1387: Maintenance: Use colorized output for `WARNING` and `ERROR` level logging messages.
- #1378: Bugfix: Fix address generation for `CREATE2` opcode.
- #1374: Maintenance: New `ChainTipMonitor` service to keep track of the highest TD chain tip.
- #1371: Maintenance: Upgrade `mypy` to `0.630`
- #1367: Maintenance: Improve logging output to include more contextual information
- #1361: Maintenance: Remove `HeaderRequestingPeer` in favor of `BaseChainPeer`
- #1353: Maintenance: Decouple peer message handling from syncing.
- #1351: Bugfix: Unhandled `DecryptionError`
- #1348: Maintenance: Add default server URIs for mainnet and ropsten.
- #1347: Maintenance: Improve code organization within `trinity` module

- #1343: Bugfix: Rename `Chain.network_id` to be `Chain.chain_id`
- #1342: Maintenance: Internal rename of `ChainConfig` to `TrinityConfig`
- #1336: Maintenance: Implement plugin for EthStats reporting.
- #1335: Maintenance: Relax some constraints on the ordered task management constructs.
- #1332: Maintenance: Upgrade `pyrlp` to `1.0.3`
- #1317: Maintenance: Extract peer selection from the header sync.
- #1312: Maintenance: Turn on warnings by default if in a prerelease

0.1.0-alpha.16

Released September 27, 2018

- #1332: Bugfix: Comparing rlp objects across processes used to fail sporadically, because of a changing object hash (fixed by upgrading pyrlp to 1.0.3)
- #1326: Maintenance: Squash a stack trace in the logs when a peer sends us an invalid public key during handshake
- #1325: Bugfix: When switching to a new peer to sync headers, it might have started from too far behind the tip, and get stuck
- #1327: Maintenance: Squash some log warnings from trying to make a request to a peer (or receive a response) while it is shutting down
- #1321: Bugfix: Address a couple race condition exceptions when syncing headers from a new peer, and other downstream processing is in progress
- #1316: Maintenance: Reduce size of images in documentation
- #1313: Maintenance: Remove miscellaneous things that are generating python warnings (eg~ using deprecated methods)
- #1279: Reliability: Atomically persist when storing: a block, a chain of headers, or a cluster of trie nodes
- #1304: Maintenance: Refactor AtomicDB to return an explict database instance to write into
- #1296: Maintenance: Require new AtomicDB in chain and header DB layers
- #1295: Maintenance: New AtomicDB interface to enable a batch of atomic writes (all succeed or all fail)
- #1290: Bugfix: more graceful recovery when re-launching sync on a fork
- #1277: Maintenance: add a cancellable `call_later` to all services
- #1226: Performance: enable multiple peer requests to a single fast peer when other peers are slow
- #1254: Bugfix: peer selection when two peers have exactly the same throughput
- #1253: Maintenance: prefer f-string formatting in p2p, trinity code

0.1.0-alpha.15

- #1249: Misc bugfixes for fast sync reliability.
- #1245: Improved exception messaging for `BaseService`
- #1244: Use `time.perf_counter` or `time.monotonic` over `time.time`
- #1242: Bugfix: Unhandled `MalformedMessage`.

- #1235: Typo cleanup.
- #1236: Documentation cleanup
- #1237: Code cleanup
- #1232: Bugfix: Correctly enforce timeouts on peer requests and add lock mechanism to support concurrency.
- #1229: CI cleanup
- #1228: Merge `KademliaProtocol` and `DiscoveryProtocol`
- #1225: Expand peer stats tracking
- #1221: Implement Discovery V5 Protocol
- #1219: Re-organize and document fixture filler tools
- #1214: Implement `BaseService.is_operational`.
- #1210: Convert sync to use streaming queue instead of batches.
- #1209: Chain Builder tool
- #1205: Bugfix: ExchangeHandler stats crash
- #1204: Consensus bugfix for uncle validation
- #1151: Change to `import_block` to return chain re-organization data.
- #1197: Increase wait time for database IPC socket.
- #1194: Unify `ValidationError` to use `eth-utils` exception class.
- #1190: Improved testing for peer authentication
- #1189: Detect crashed sub-services and exit
- #1179: `LightNode` now uses `Server` for incoming peer connections.
- #1182: Convert `fix-unclean-shutdown` CLI command to be a plugin

0.1.0-alpha.14

- #1081 #1115 #1116: Reduce logging output during state sync.
- #1063 #1035 #1089 #1131 #1132 #1138 #1149 #1159: Implement round trip request/response API.
- #1094 #1124: Make the node processing during state sync more async friendly.
- #1097: Keep track of which peers are missing trie nodes during state sync.
- #1109 #1135: Python 3.7 testing and experimental support.
- #1136 #1120: Module re-organization in preparation of extracting `p2p` and `trinity` modules.
- #1137: Peer subscriber API now supports specifying specific msg types to reduce msg queue traffic.
- #1142 #1165: Implement JSON-RPC endpoints for: `eth_estimateGas`, `eth_accounts`, `eth_call`
- #1150 #1176: Better handling of malformed messages from peers.
- #1157: Use shared pool of workers across all services.
- #1158: Support specifying granular logging levels via CLI.
- #1161: Use a tmpfile based LevelDB database for cache during state sync to reduce memory footprint.
- #1166: Latency and performance tracking for peer requests.

- #1173: Better APIs for background task running for `Service` classes.
- #1182: Convert `fix-unclean-shutdown` command to be a plugin.

0.1.0-alpha.13

- Remove specified `eth-account` dependency in favor of allowing `web3.py` specify the correct version.

0.1.0-alpha.12

- #1058 #1044: Add `fix-unclean-shutdown` CLI command for cleaning up after a dirty shutdown of the `trinity` CLI process.
- #1041: Bugfix for ensuring CPU count for process pool is always greater than `0`
- #1010: Performance tuning during fast sync. Only check POW on a subset of the received headers.
- #996 Experimental new Plugin API: Both the transaction pool and the `console` and `attach` commands are now written as plugins.
- #898: New experimental transaction pool. Disabled by default. Enable with `--tx-pool`. (**warning**: has known issues that effect sync performance)
- #935: Protection against eclipse attacks.
- #869: Ensure connected peers are on the same side of the DAO fork.

Minor Changes

- #1081: Reduce `DEBUG` log output during state sync.
- #1071: Minor fix for how version string is generated for trinity
- #1070: Easier profiling of `ChainSyncer`
- #1068: Optimize `evm.db.chain.ChainDB.persist_block` for common case.
- #1057: Additional `DEBUG` logging of peer uptime and msg stats.
- #1049: New integration test suite for trinity CLI
- #1045 #1051: Bugfix for generation of block numbers for `GetBlockHeaders` requests.
- #1011: Workaround for parity bug parity #8038
- #987: Now serving requests from peers during fast sync.
- #971 #909 #650: Benchmarking test suite.
- #968: When launching `console` and `attach` commands, check for presence of IPC socket and log informative message if not found.
- #934: Decouple the `Discovery` and `PeerPool` services.
- #913: Add validation of retrieved contract code when operating in `--light` mode.
- #908: Bugfix for transitioning from syncing chain data to state data during fast sync.
- #905: Support for multiple UPNP devices.

0.1.0-alpha.11

- Bugfix for `PreferredNodePeerPool` to respect `max_peers`

0.1.0-alpha.10

- More bugfixes to enforce `--max-peers` in `PeerPool._connect_to_nodes`

0.1.0-alpha.9

- Bugfix to enforce `--max-peers` for incoming connections.

0.1.0-alpha.7

- Remove `min_peers` concept from `PeerPool`
- Add `--max-peers` and enforcement of maximum peer connections maintained by the `PeerPool`.

0.1.0-alpha.6

- Respond to `GetBlockHeaders` message during fast sync to prevent being disconnected as a *useless peer*.
- Add `--profile` CLI flag to Trinity to enable profiling via `cProfile`
- Better error messaging with Trinity cannot determine the appropriate location for the data directory.
- Handle `ListDeserializationError` during handshake.
- Add `net_version` JSON-RPC endpoint.
- Add `web3_clientVersion` JSON-RPC endpoint.
- Handle `rlp.DecodingError` during handshake.

2.4 Cookbooks

The Cookbooks are collections of simple recipes that demonstrate good practices to accomplish common tasks. The examples are usually short answers to simple "How do I..." questions that go beyond simple API descriptions but also don't need a full guide to become clear.

2.4.1 EVM Cookbook

Using the Chain object

A "single" blockchain is made by a series of different virtual machines for different spans of blocks. For example, the Ethereum mainnet had one virtual machine for blocks 0 till 1150000 (known as Frontier), and another VM for blocks 1150000 till 1920000 (known as Homestead).

The `Chain` object manages the series of fork rules, after you define the VM ranges. For example, to set up a chain that would track the mainnet Ethereum network until block 1920000, you could create this chain class:

```
>>> from eth import constants, Chain
>>> from eth.vm.forks.frontier import FrontierVM
>>> from eth.vm.forks.homestead import HomesteadVM
>>> from eth.chains.mainnet import HOMESTEAD_MAINNET_BLOCK

>>> chain_class = Chain.configure(
```

(continues on next page)

(continued from previous page)

```
...     __name__='Test Chain',
...     vm_configuration=(
...         (constants.GENESIS_BLOCK_NUMBER, FrontierVM),
...         (HOMESTEAD_MAINNET_BLOCK, HomesteadVM),
...     ),
... )
```

Then to initialize, you can start it up with an in-memory database:

```
>>> from eth.db.atomic import AtomicDB
>>> from eth.chains.mainnet import MAINNET_GENESIS_HEADER

>>> # start a fresh in-memory db

>>> # initialize a fresh chain
>>> chain = chain_class.from_genesis_header(AtomicDB(), MAINNET_GENESIS_HEADER)
```

Creating a chain with custom state

While the previous recipe demos how to create a chain from an existing genesis header, we can also create chains simply by specifing various genesis parameter as well as an optional genesis state.

```
>>> from eth_keys import keys
>>> from eth import constants
>>> from eth.chains.mainnet import MainnetChain
>>> from eth.db.atomic import AtomicDB
>>> from eth_utils import to_wei, encode_hex



>>> # Giving funds to some address
>>> SOME_ADDRESS = b'\x85\x82\xa2\x89V\xb9%\x93M\x03\xdd\xb4Xu\xe1\x8e\x85\x93\x12\xc1
↪'
>>> GENESIS_STATE = {
...     SOME_ADDRESS: {
...         "balance": to_wei(10000, 'ether'),
...         "nonce": 0,
...         "code": b'',
...         "storage": {}
...     }
... }

>>> GENESIS_PARAMS = {
...     'parent_hash': constants.GENESIS_PARENT_HASH,
...     'uncles_hash': constants.EMPTY_UNCLE_HASH,
...     'coinbase': constants.ZERO_ADDRESS,
...     'transaction_root': constants.BLANK_ROOT_HASH,
...     'receipt_root': constants.BLANK_ROOT_HASH,
...     'difficulty': constants.GENESIS_DIFFICULTY,
...     'block_number': constants.GENESIS_BLOCK_NUMBER,
...     'gas_limit': constants.GENESIS_GAS_LIMIT,
...     'extra_data': constants.GENESIS_EXTRA_DATA,
...     'nonce': constants.GENESIS_NONCE
... }
```

(continues on next page)

(continued from previous page)

```
>>> chain = MainnetChain.from_genesis(AtomicDB(), GENESIS_PARAMS, GENESIS_STATE)
```

Getting the balance from an account

Considering our previous example, we can get the balance of our pre-funded account as follows.

```
>>> current_vm = chain.get_vm()
>>> account_db = current_vm.state.account_db
>>> account_db.get_balance(SOME_ADDRESS)
10000000000000000000000
```

Building blocks incrementally

The default `Chain` is stateless and thus does not keep a tip block open that would allow us to incrementally build a block. However, we can import the `MiningChain` which does allow exactly that.

```
>>> from eth.chains.base import MiningChain
```

Please check out the *Understanding the mining process* guide for a full example that demonstrates how to use the `MiningChain`.

2.5 Guides

This section aims to provide hands-on guides to demonstrate how to use Trinity and the Py-EVM. If you are looking for detailed API descriptions check out the *API section*.

2.5.1 Trinity

This section aims to provide hands-on guides to demonstrate how to use Trinity. If you are looking for detailed API descriptions check out the *API section*.

Quickstart

Installation

This is the quickstart guide for Trinity. If you only care about running a Trinity node, this guide will help you to get things set up. If you plan to develop on top of Py-EVM or contribute to the project you may rather want to checkout the *Contributing Guide* which explains how to set everything up for development.

Installing on Ubuntu

Trinity requires Python 3.6 as well as some tools to compile its dependencies. On Ubuntu, the `python3.6-dev` package contains everything we need. Run the following command to install it.

```
apt-get install python3.6-dev
```

Trinity is installed through the pip package manager, if pip isn't available on the system already, we need to install the `python3-pip` package through the following command.

```
apt-get install python3-pip
```

Note: **Optional:** Often, the best way to guarantee a clean Python 3 environment is with virtualenv. If we don't have `virtualenv` installed already, we first need to install it via pip.

```
pip install virtualenv
```

Then, we can initialize a new virtual environment `venv`, like:

```
virtualenv -p python3 venv
```

This creates a new directory `venv` where packages are installed isolated from any other global packages.

To activate the virtual directory we have to *source* it

```
. venv/bin/activate
```

Finally, we can install the `trinity` package via pip.

```
pip3 install -U trinity
```

Installing on macOS

First, install LevelDB and the latest Python 3 with brew:

```
brew install python3 leveldb
```

Note: **Optional:** Often, the best way to guarantee a clean Python 3 environment is with virtualenv. If we don't have `virtualenv` installed already, we first need to install it via pip.

```
pip install virtualenv
```

Then, we can initialize a new virtual environment `venv`, like:

```
virtualenv -p python3 venv
```

This creates a new directory `venv` where packages are installed isolated from any other global packages.

To activate the virtual directory we have to *source* it

```
. venv/bin/activate
```

Then, install the `trinity` package via pip:

```
pip3 install -U trinity
```

Installing through Docker

Trinity can also be installed using `Docker` which can be a lightweight alternative where no changes need to be made to the host system apart from having `Docker` itself installed.

Note: While we don't officially support Windows just yet, running Trinity through `Docker` is a great way to bypass this current limitation as Trinity can run on any system that runs `Docker` with support for linux containers.

Using `Docker` we have two different options to choose from.

1. Run an existing official image

This is the default way of running Trinity through `Docker`. If all we care about is running a Trinity node, using one of the latest released versions, this method is perfect.

Run:

```
docker run -it ethereum/trinity
```

Alternatively, we can run a specific image version, following the usual docker version schema.

```
docker run -it ethereum/trinity:0.1.0-alpha.13
```

2. Build your own image

Alternatively, we may want to try out a specific (unreleased) version. In that case, we can create our very own image directly from the source code.

```
make create-docker-image version=my-own-version
```

After the image has been successfully created, we can run it by invoking:

```
docker run -it ethereum/trinity:my-own-version
```

Running Trinity

After Trinity is installed we should have the `trinity` command available to start it.

```
trinity
```

While it may take a couple of minutes before Trinity can start syncing against the Ethereum mainnet, it should print out some valuable information right away which should look something like this. If it doesn't please file an issue to help us getting that bug fixed.

```
    INFO  05-29 01:57:02         main
  ______     _       _ __
/_  __/____(_)___  (_) /___  __
  / / / ___/ / __ \/ / __/ / / /
/ / / /  / / / / / / /_/ /_/ /
/_/ /_/  /_/_/ /_/_/\__/\__, /
                        /____/
    INFO  05-29 01:57:02         main  Trinity/0.1.0a4/linux/cpython3.6.5
    INFO  05-29 01:57:02         main  network: 1
    INFO  05-29 01:57:02          ipc  IPC started at: /root/.local/share/trinity/
↪mainnet/jsonrpc.ipc
```

(continues on next page)

(continued from previous page)

```
    INFO  05-29 01:57:02      server  Running server...
    INFO  05-29 01:57:07      server  enode://
↪09d34ecb0de1806ab0e68cb2d822b967292dc021df06aab9a55aa4d2e1b2e04ae73560137407a48073286026e12dd60d26
↪0.0.0:30303
    INFO  05-29 01:57:07      server  network: 1
    INFO  05-29 01:57:07        peer  Running PeerPool...
    INFO  05-29 01:57:07        sync  Starting fast-sync; current head: #0
```

Once Trinity successfully connected to other peers we should see it starting to sync the chain.

```
INFO  05-29 02:23:13       chain  Starting sync with ETHPeer <Node(0xaff0@90.114.124.
↪196)>
INFO  05-29 02:23:14       chain  Imported chain segment in 0 seconds, new head: #191␣
↪(739b)
INFO  05-29 02:23:15       chain  Imported chain segment in 0 seconds, new head: #383␣
↪(789c)
INFO  05-29 02:23:16       chain  Imported chain segment in 0 seconds, new head: #575␣
↪(a1d0)
INFO  05-29 02:23:17       chain  Imported chain segment in 0 seconds, new head: #767␣
↪(aeb6)
```

Running as a light client

Warning: It may take a **very** long time for Trinity to find an LES node with open slots. This is not a bug with trinity, but rather a shortage of nodes serving LES. Please consider running your own LES server to help improve the health of the network.

Use the `--light` flag to instruct Trinity to run as a light node.

Ropsten vs Mainnet

Trinity currently only supports running against either the Ethereum Mainnet or Ropsten testnet. Use `--ropsten` to run against Ropsten.

```
trinity --ropsten
```

Connecting to preferred nodes

If you would like to have Trinity prioritize connecting to specific nodes, you can use the `--preferred-node` command line flag. This flag takes an enode URI as a single argument and will instruct Trinity to prioritize connecting to this node.

```
trinity --preferred-node enode://
↪a41defa74e8d9d4152699cb9a0d195377da95833769ad6b386092ac3b16c184eb4ef4b4f02889e0b5097ff50fb5847ba99
↪0.0.1:30304
```

Using `--preferred-node` is a good way to ensure Trinity running in `--light` mode connects to known peers who serve LES.

Retrieving Chain information via web3

While just running `trinity` already causes the node to start syncing, it doesn't let us interact with the chain directly (apart from the JSON-RPC API).

However, we can attach an interactive shell to a running Trinity instance with the `attach` subcommand. The interactive `ipython` shell binds a web3 instance to the `w3` variable.

```
trinity attach
```

Now that Trinity runs in an interactive shell mode, let's try to get some information about the latest block by calling `w3.eth.getBlock('latest')`.

```
In [9]: w3.eth.getBlock('latest')
Out[9]:
AttributeDict({'difficulty': 743444339302,
'extraData': HexBytes('0x476574682f4c5649562f76312e302e302f6c696e75782f676f312e342e32
↪'),
'gasLimit': 5000,
'gasUsed': 0,
'hash': HexBytes('0x1a8487dfb8de7ee27b9cca30b6f3f6c9676eae29c10eef39b86890ed15eeed01
↪'),
'logsBloom': HexBytes(
↪'0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪'),
'mixHash': HexBytes(
↪'0xf693b8e4bc30728600da40a0578c14ddb7ad08a64e329a19d9355d5665588aef'),
'nonce': HexBytes('0x7382884a72533c59'),
'number': 12479,
'parentHash': HexBytes(
↪'0x889c36c51463f100cf50ec2e2a92886aa7ebb3f99fa8c817343214a92f967a29'),
'receiptsRoot': HexBytes(
↪'0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421'),
'sha3Uncles': HexBytes(
↪'0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347'),
'stateRoot': HexBytes(
↪'0x6ad1ecb7d516c679e7c476956159051fa32848f3ba631a47c3fb72937ed86987'),
'timestamp': 1438368997,
'transactionsRoot': HexBytes(
↪'0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421'),
'miner': '0xbb7B8287f3F0a933474a79eAe42CBCa977791171',
'totalDifficulty': 3961372514945562,
'uncles': [],
'size': 544,
'transactions': []})
```

You can attach to an existing Trinity process using the `attach` comand.

```
trinity attach
```

For a list of JSON-RPC endpoints which are expected to work, see this issue: https://github.com/ethereum/py-evm/issues/178

> **Warning:** Trinity is currently in public alpha. **Keep in mind**:
>
> - It is expected to have bugs and is not meant to be used in production

- Things may be ridiculously slow or not work at all
- Only a subset of JSON-RPC API calls are currently supported

Architecture

This guide is intended to provide an overview of the general application architecture that Trinity follows.

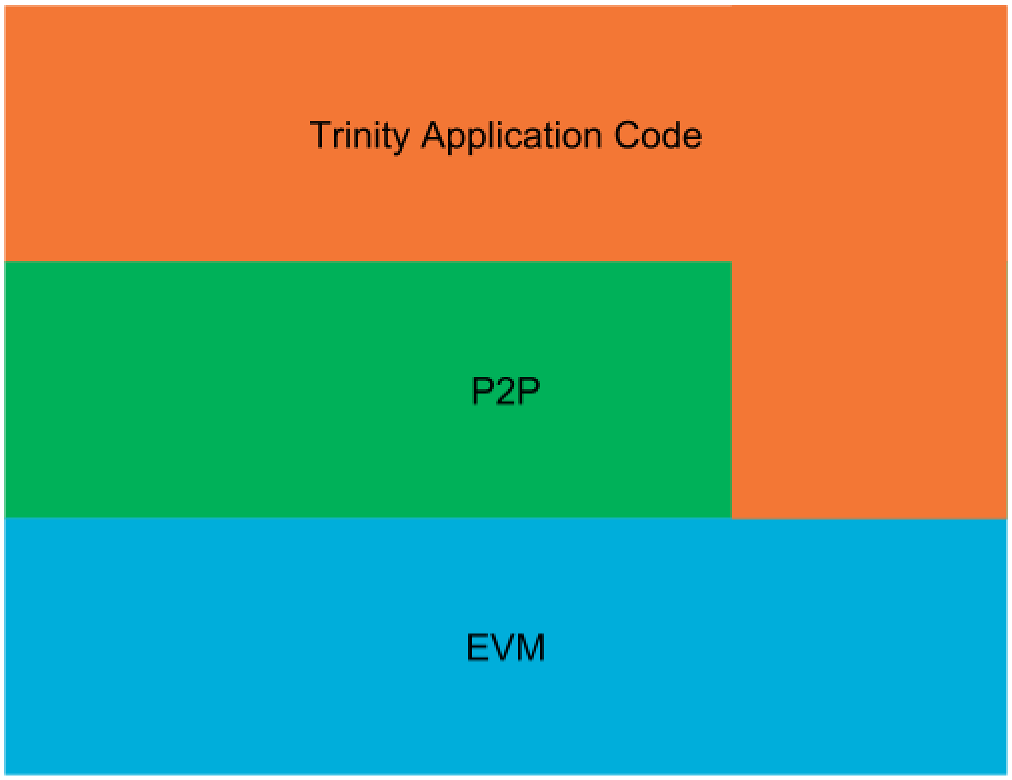
Layering

Trinity is layered to be highly flexible and is based on other independent projects that provide the foundation for lower levels.

The three main layers from top to bottom are:

- **Trinity Application Code**
- **Peer to Peer (P2P)**
- **Ethereum Virtual Machine (Py-EVM)**

They can be visualized as seen in the following graphic:



The graphic should be understood in such a way that only the higher levels know and use the lower levels. Consequently, the Trinity application code knows and uses both the P2P and EVM layers. However, the P2P layer uses only the EVM layer and the EVM layer neither knows nor uses any higher layers.

Let's go briefly over each layer to understand its main purpose. We won't go into very much detail for each layer in this guide but rather link to more specific guides that explain the nitty gritty details.

Let's start bottom up.

EVM

EVM stands for Ethereum Virtual Machine and is the lowest level layer that Trinity utilizes to build and validate blocks, run transactions and execute their code (EVM byte code) to eventually apply transitions to the state of the Ethereum blockchain.

Notice that the EVM is a seperate project and has no dependency against Trinity and that other projects are free to use it as the *Py-EVM* project independently from Trinity.

P2P

The peer to peer layer implements the communication protocol that each Ethereum node follows to talk with each other. Ethereum uses an Kademlia-like distributed hash table to store information about Ethereum nodes.

Trinity Application Code

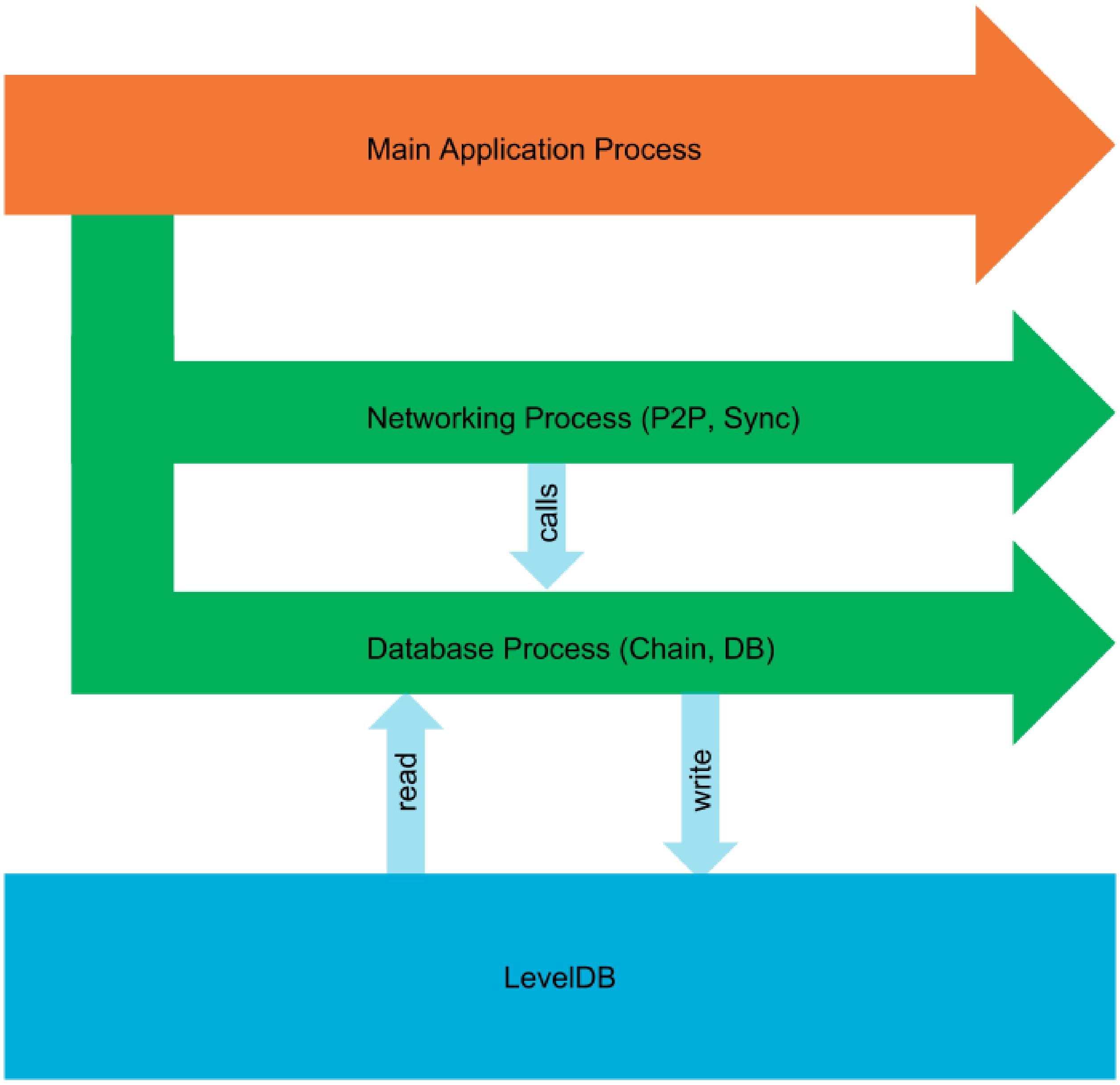
Everything that makes an Ethereum node an Ethereum node and is not part of the EVM or P2P layer is handled by the Trinity application layer itself. That includes a lot of networking, orchestrating the existing building blocks for different modes of operations (e.g. light vs full mode), handling of the different CLI arguments, providing interactive access to the node and the network as well as lot of other things.

Processes

An Ethereum node is quite a busy kind of application. There's a constant flow of actions such as responding to peers, running transactions and validating blocks that will keep the machine busy.

Since Python doesn't play very well with multi threading (mainly because of Pythons GIL), often the best way to achieve an architecture that can handle concurrency efficiently is through the usage of multiple processes as well as asynchronous IO. Notice that the usage of asynchronous IO alone doesn't cut it since a lot concurrent jobs are effectively CPU bound rather than IO bound.

On startup, Trinity spawns three main processes that we'll briefly explain here.



Main Application Process

This is the main process of Trinity that spawns up implicitly when we run the `trinity` command. It is responsible for parsing the command line arguments, orchestrating the building blocks to run the kind of node the user wants to run and eventually kicks off the networking and the database process.

Database Process

The database process exposes several chain-related operations, all of which are bundled in this single process. These aren't necessarily low-level get/set operations, but also include higher-level APIs, such as the *`import_block()`* API.

The way this works is by facilitazing Pythons `BaseManager` API and exposing several `BaseProxy` proxies to coordinate inter-process access to these APIs.

Since Trinity uses *LevelDB* as its default database, it is a given requirement (of *LevelDB*) that all database reads and writes are done by a single process.

Networking Process

The networking process is what kicks of the peer to peer communication and starts the syncing process. It does so by running an instance of `Node()` in an event loop.

Notice that the instance of `Node()` has access to the APIs that the database processes exposes. In pracice that means, that the network process controls the connections to other peers, starts of the syncing process but will call APIs that run inside the database processes when it comes to actual importing of blocks or reading and writing of other things from the database.

The networking process also host an instance of the `PluginManager` to run plugins that need to deeply integrate with the networking process (Further reading: *Writing Plugins*).

Plugin Processes

Apart from running these three core processes, there may be additional processes for plugins that run in isolated processes. Isolated plugins are explained in depth in the *Writing Plugins* guide.

Writing Plugins

Trinity aims to be a highly flexible Ethereum node to support lots of different use cases beyond just participating in the regular networking traffic.

To support this goal, Trinity allows developers to create plugins that hook into the system to extend its functionality. In fact, Trinity dogfoods its Plugin API in the sense that several built-in features are written as plugins that just happen to be shipped among the rest of the core modules. For instance, the JSON-RPC API, the Transaction Pool as well as the `trinity attach` command that provides an interactive REPL with *Web3* integration are all built as plugins.

Trinity tries to follow the practice: If something can be written as a plugin, it should be written as a plugin.

What can plugins do?

Plugin support in Trinity is still very new and the API hasn't stabilized yet. That said, plugins are already pretty powerful and are only becoming more so as the APIs of the underlying services improve over time.

Here's a list of functionality that is currently provided by plugins:

- JSON-RPC API
- Transaction Pool
- EthStats Reporting
- Interactive REPL with Web3 integration
- Crash Recovery Command

Understanding the different plugin categories

There are currently three different types of plugins that we'll all cover in this guide.

- Plugins that overtake and redefine the entire `trinity` command
- Plugins that spawn their own new isolated process
- Plugins that run in the shared *networking* process

Plugins that redefine the Trinity process

This is the simplest category of plugins as it doesn't really *hook* into the Trinity process but hijacks it entirely instead. We may be left wonderering: Why would one want to do that?

The only reason to write such a plugin is to execute some code that we want to group under the `trinity` command. A great example for such a plugin is the `trinity attach` command that gives us a REPL attached to a running Trinity instance. This plugin could have easily be written as a standalone program and associated with a command such as `trinity-attach`. However, using a subcommand `attach` is the more idiomatic approach and this type of plugin gives us simple way to develop exactly that.

We build this kind of plugin by subclassing from *`BaseMainProcessPlugin`*. A detailed example will follow soon.

Plugins that spawn their own new isolated process

Of course, if all what plugins could do is to hijack the *trinity* command, there wouldn't be much room to actually extend the *runtime functionality* of Trinity. If we want to create plugins that boot with and run alongside the main node activity, we need to write a different kind of plugin. These type of plugins can respond to events such as a peers connecting/disconnecting and can access information that is only available within the running application.

The JSON-RPC API is a great example as it exposes information such as the current count of connected peers which is live information that can only be accessed by talking to other parts of the application at runtime.

This is the default type of plugin we want to build if:

- we want to execute logic **together** with the command that boots Trinity (as opposed to executing it in a separate command)
- we want to execute logic that integrates with parts of Trinity that can only be accessed at runtime (as opposed to e.g. just reading things from the database)

We build this kind of plugin subclassing from *`BaseIsolatedPlugin`*. A detailed example will follow soon.

Plugins that run inside the networking process

If the previous category sounded as if it could handle every possible use case, it's because it's actually meant to. In reality though, not all internal APIs yet work well across process boundaries. In practice, this means that sometimes we want to make sure that a plugin runs in the same process as the rest of the networking code.

Warning: The need to run plugins in the networking process is declining as the internals of Trinity become more and more multi-process friendly over time. While it isn't entirely clear yet, there's a fair chance this type of plugin will become obsolete at some point and may eventually be removed.

We should only choose this type of plugin category if what we are trying to build cannot be built with a *BaseIsolatedPlugin*.

We build this kind of plugin subclassing from *BaseAsyncStopPlugin*. A detailed example will follow soon.

The plugin lifecycle

Plugins can be in one of the following status at a time:

- `NOT_READY`
- `READY`
- `STARTED`
- `STOPPED`

The current status of a plugin is also reflected in the *status()* property.

Note: Strictly speaking, there's also a special state that only applies to the *BaseMainProcessPlugin* which comes into effect when such a plugin hijacks the Trinity process entirely. That being said, in that case, the resulting process is in fact something entirely different than Trinity and the whole plugin infrastruture doesn't even continue to exist from the moment on where that plugin takes over the Trinity process. This is why we do not list it as an actual state of the regular plugin lifecycle.

Plugin state: `NOT_READY`

Every plugin starts out being in the `NOT_READY` state. This state begins with the instantiation of the plugin and lasts until the *on_ready()* hook was called which happens as soon the core infrastructure of Trinity is ready.

Plugin state: `READY`

After Trinity has finished setting up the core infrastructure, every plugin has its *PluginContext* set and *on_ready()* is called. At this point the plugin has access to important information such as the parsed arguments or the `TrinityConfig`. It also has access to the central event bus via its *event_bus()* property which enables the plugin to communicate with other parts of the application including other plugins.

Plugin state: `STARTED`

A plugin is in the `STARTED` state after the *start()* method was called. Plugins call this method themselves whenever they want to start which may be based on some condition like Trinity being started with certain parameters or some event being propagated on the central event bus.

Note: Calling *start()* while the plugin is in the `NOT_READY` state or when it is already in `STARTED` will cause an exception to be raised.

Plugin state: STOPPED

A plugin is in the STOPPED state after the stop() method was called and finished any tear down work.

Defining plugins

We define a plugin by deriving from either *BaseMainProcessPlugin*, *BaseIsolatedPlugin* or *BaseAsyncStopPlugin* depending on the kind of plugin that we intend to write. For now, we'll stick to *BaseIsolatedPlugin* which is the most commonly used plugin category.

Every plugin needs to overwrite name so voilà, here's our first plugin!

```
class PeerCountReporterPlugin(BaseIsolatedPlugin):

    @property
    def name(self) -> str:
        return "Peer Count Reporter"
```

Of course that doesn't do anything useful yet, bear with us.

Configuring Command Line Arguments

More often than not we want to have control over if or when a plugin should start. Adding command-line arguments that are specific to such a plugin, which we then check, validate, and act on, is a good way to deal with that. Implementing *configure_parser()* enables us to do exactly that.

This method is called when Trinity starts and bootstraps the plugin system, in other words, **before** the start of any plugin. It is passed a ArgumentParser as well as a _SubParsersAction which allows it to amend the configuration of Trinity's command line arguments in many different ways.

For example, here we are adding a boolean flag --report-peer-count to Trinity.

```
    def configure_parser(self,
                         arg_parser: ArgumentParser,
                         subparser: _SubParsersAction) -> None:
        arg_parser.add_argument(
            "--report-peer-count",
            action="store_true",
            help="Report peer count to console",
        )
```

To be clear, this does not yet cause our plugin to automatically start if --report-peer-count is passed, it simply changes the parser to be aware of such flag and hence allows us to check for its existence later.

Note: For a more advanced example, that also configures a subcommand, checkout the trinity attach plugin.

Defining a plugins starting point

Every plugin needs to have a well defined starting point. The exact mechanics slightly differ in case of a *BaseMainProcessPlugin* but remain fairly similar for the other types of plugins which we'll be focussing on for now.

Plugins need to implement the `do_start()` method to define their own bootstrapping logic. This logic may involve setting up event listeners, running code in a loop or any other kind of action.

Warning: Technically, there's nothing preventing a plugin from performing starting logic in the `on_ready()` hook. However, doing that is an anti pattern as the plugin infrastructure won't know about the running plugin, can't propagate the `PluginStartedEvent` and the plugin won't be properly shut down with Trinity if the node closes.

Let's assume we want to create a plugin that simply periodically prints out the number of connected peers.

While it is absolutely possible to put this logic right into the plugin, the preferred way is to subclass `BaseService` and implement the core logic in such a standalone service.

```
class PeerCountReporter(BaseService):

    def __init__(self, event_bus: Endpoint) -> None:
        super().__init__()
        self.event_bus = event_bus

    async def _run(self) -> None:
        self.run_daemon_task(self._periodically_report_stats())
        await self.cancel_token.wait()

    async def _periodically_report_stats(self) -> None:
        while self.is_operational:
            try:
                response = await asyncio.wait_for(
                    self.event_bus.request(PeerCountRequest()),
                    timeout=1.0
                )
                self.logger.info("CONNECTED PEERS: %s", response.peer_count)
            except asyncio.TimeoutError:
                self.logger.warning("TIMEOUT: Waiting on PeerPool to boot")
            await asyncio.sleep(5)
```

Then, the implementation of `do_start()` is only concerned about running the service on a fresh event loop.

```
    def do_start(self) -> None:
        loop = asyncio.get_event_loop()
        service = PeerCountReporter(self.event_bus)
        asyncio.ensure_future(exit_with_service_and_endpoint(service, self.event_bus))
        asyncio.ensure_future(service.run())
        loop.run_forever()
        loop.close()
```

If the example may seem unnecessarily complex, it should be noted that plugins can be implemented in many different ways, but this example follows a pattern that is considered best practice within the Trinity Code Base.

Starting a plugin

As we've read in the previous section not all plugins should run at any point in time. In fact, the circumstances under which we want a plugin to begin its work may vary from plugin to plugin.

We may want a plugin to only start running if:

- a certain (combination) of command line arguments was given

- another plugin or group of plugins started
- a certain number of connected peers was exceeded / undershot
- a certain block number was reached
- ...

Hence, to actually start a plugin, the plugin needs to invoke the *start()* method at any moment when it is in its `READY` state. Let's assume a simple case in which we simply want to start the plugin if Trinity is started with the `--report-peer-count` flag.

```
    def on_ready(self) -> None:
        if self.context.args.report_peer_count:
            self.start()
```

In case of a `BaseIsolatedProcessPlugin`, this will cause the `do_start()` method to run on an entirely separated, new process. In other cases `do_start()` will simply run in the same process as the plugin manager that the plugin is controlled by.

Communication pattern

For most plugins to be useful they need to be able to communicate with the rest of the application as well as other plugins. In addition to that, this kind of communication needs to work across process boundaries as plugins will often operate in independent processes.

To achieve this, Trinity uses the Lahja project, which enables us to operate a lightweight event bus that works across processes. An event bus is a software dedicated to the transmission of events from a broadcaster to interested parties.

This kind of architecture allows for efficient and decoupled communication between different parts of Trinity including plugins.

For instance, a plugin may be interested to perform some action every time that a new peer connects to our node. These kind of events get exposed on the EventBus and hence allow a wide range of plugins to make use of them.

For an event to be usable across processes it needs to be pickable and in general should be a shallow Data Transfer Object (DTO)

Every plugin has access to the event bus via its *event_bus()* property and in fact we have already used it in the above example to get the current number of connected peers.

Note: This guide will soon cover communication through the event bus in more detail. For now, the Lahja documentation gives us some more information about the available APIs and how to use them.

Distributing plugins

Of course, plugins are more fun if we can share them and anyone can simply install them through `pip`. The good news is, it's not hard at all!

In this guide, we won't go into details about how to create Python packages as this is already covered in the official Python docs .

Once we have a `setup.py` file, all we have to do is to expose our plugin under `trinity.plugins` via the `entry_points` section.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
from setuptools import setup

setup(
    name='trinity-peer-count-reporter-plugin',
    py_modules=['peer_count_reporter_plugin'],
    entry_points={
        'trinity.plugins': 'peer_count_reporter_plugin=peer_count_reporter_
↪plugin:PeerCountReporterPlugin',
    },
)
```

Check out the official documentation on entry points for a deeper explanation.

A plugin where the `setup.py` file is configured as described can be installed by `pip install <package-name>`` and immediately becomes available as a plugin in Trinity.

Note: Plugins installed from a local directory (instead of the pypi registry), such as the sample plugin described in this article, must be installed with the `-e` parameter (Example: `pip install -e ./trinity-external-plugins/examples/peer_count_reporter`)

2.5.2 Py-EVM

This section aims to provide hands-on guides to demonstrate how to use Py-EVM. If you are looking for detailed API descriptions check out the *API section*.

Quickstart

Note: This quickstart is aspirational. The code examples may not work yet.

Installation

```
pip install py-evm
```

Sync and interact with the Ropsten chain

Currently we only provide a light client that will sync only block headers, although it can fetch block bodies on demand. The easiest way to try it is by running the lightchain_shell, which will run the LightPeerChain in the background and let you use the python interpreter to interact with it:

```
$ python -i -m eth.lightchain_shell -db /tmp/testnet.db
```

That will immediately give you a python shell, with a chain variable that you can use even before it has finished syncing:

```
>>> chain.get_canonical_head()
<BlockHeader #2200794 e3f9c6bb>
```

Some `LightPeerChain` methods (e.g. those that need data from block bodies) are coroutines that need to be executed by asyncio's event loop, so for those we provide a helper that will schedule their execution and wait for the result:

```
>>> wait_for_result(chain.get_canonical_block_by_number(42))
<FrontierBlock(#Block #42)>
```

Accessing an existing chain database

The `Chain` object manages the series of fork rules contained in every blockchain. It requires that you define the VM ranges. Some pre-built chains are available for your convenience. To access the Mainnet chain you can use:

```
from eth import MainnetChain
from eth.chains.mainnet import MAINNET_GENESIS_HEADER
from eth.db.backends.level import LevelDB
from eth.db.chain import ChainDB

# Read the previously saved chain database
chaindb = ChainDB(LevelDB('/tmp/mainnet.db'))

# Load the saved database into a mainnet chain object
chain = MainnetChain(chaindb)
```

Then you can read data about the chain that you already downloaded. For example:

```
highest_block_num = chain.get_canonical_head().block_number

block1 = chain.get_canonical_block_by_number(1)
assert block1.number == 1

blockhash = block1.hash()
vm = chain.get_vm()
blockgas = vm.get_cumulative_gas_used(block1)
```

The methods available on the block are variable. They depend on what fork you're on. The mainnet follows "Frontier" rules at the beginning, then Homestead, and so on. To see block features for Frontier, see the API for `FrontierBlock`.

Building an app that uses Py-EVM

One of the primary use cases of the `Py-EVM` library is to enable developers to build applications that want to interact with the ethereum ecosystem.

In this guide we want to build a very simple script that uses the `Py-EVM` library to create a fresh blockchain with a pre-funded address to simply read the balance of that address through the regular `Py-EVM` APIs. Frankly, not the most exciting application in the world, but the principle of how we use the `Py-EVM` library stays the same for more exciting use cases.

Setting up the application

Let's get started by setting up a new application. Often, that process involves lots of repetitive boilerplate code, so instead of doing it all by hand, let's just clone the Ethereum Python Project Template which contains all the typical things that we want.

To clone this into a new directory `demo-app` run:

```
git clone https://github.com/carver/ethereum-python-project-template.git demo-app
```

Then, change into the directory

```
cd demo-app
```

Add the Py-EVM library as a dependency

To add `Py-EVM` as a dependency, open the `setup.py` file in the root directory of the application and change the `install_requires` section as follows.

```
install_requires=[
    "eth-utils>=1,<2",
    "py-evm==0.2.0a26",
],
```

Warning: Make sure to also change the `name` inside the `setup.py` file to something valid (e.g. `demo-app`) or otherwise, fetching dependencies will fail.

Next, we need to use the `pip` package manager to fetch and install the dependencies of our app.

Note: Optional: Often, the best way to guarantee a clean Python 3 environment is with virtualenv. If we don't have `virtualenv` installed already, we first need to install it via pip.

```
pip install virtualenv
```

Then, we can initialize a new virtual environment `venv`, like:

```
virtualenv -p python3 venv
```

This creates a new directory `venv` where packages are installed isolated from any other global packages.

To activate the virtual directory we have to *source* it

```
. venv/bin/activate
```

To install the dependencies, run:

```
pip install -e .[dev]
```

Congrats! We're now ready to build our application!

Writing the application code

Next, we'll create a new directory `app` and create a file `main.py` inside. Paste in the following content.

```python
from eth import constants
from eth.chains.mainnet import MainnetChain
from eth.db.backends.memory import MemoryDB

from eth_utils import to_wei, encode_hex


MOCK_ADDRESS = constants.ZERO_ADDRESS
DEFAULT_INITIAL_BALANCE = to_wei(10000, 'ether')

GENESIS_PARAMS = {
    'parent_hash': constants.GENESIS_PARENT_HASH,
    'uncles_hash': constants.EMPTY_UNCLE_HASH,
    'coinbase': constants.ZERO_ADDRESS,
    'transaction_root': constants.BLANK_ROOT_HASH,
    'receipt_root': constants.BLANK_ROOT_HASH,
    'difficulty': constants.GENESIS_DIFFICULTY,
    'block_number': constants.GENESIS_BLOCK_NUMBER,
    'gas_limit': constants.GENESIS_GAS_LIMIT,
    'extra_data': constants.GENESIS_EXTRA_DATA,
    'nonce': constants.GENESIS_NONCE
}

GENESIS_STATE = {
    MOCK_ADDRESS: {
        "balance": DEFAULT_INITIAL_BALANCE,
        "nonce": 0,
        "code": b'',
        "storage": {}
    }
}

chain = MainnetChain.from_genesis(MemoryDB(), GENESIS_PARAMS, GENESIS_STATE)

mock_address_balance = chain.get_vm().state.account_db.get_balance(MOCK_ADDRESS)

print("The balance of address {} is {} wei".format(
    encode_hex(MOCK_ADDRESS),
    mock_address_balance)
)
```

Runing the script

Let's run the script by invoking the following command.

```
python app/main.py
```

We should see the following output.

```
The balance of address 0x0000000000000000000000000000000000000000 is␣
↪10000000000000000000000 wei
```

Architecture

The primary use case for Py-EVM is supporting the public Ethereum blockchain.

However, it is architected with a strong focus on configurability and extensibility. Use of Py-EVM for alternate use cases such as private chains, consortium chains, or even chains with fundamentally different VM semantics should be possible without any changes to the core library.

The following abstractions are used to represent the full consensus rules for a Py-EVM based blockchain.

- Chain: High level API for interacting with the blockchain.
- VM: High level API for a single fork within a Chain
- VMState: The current state of the VM, transaction execution logic and the state transition function.
- Message: Representation of the portion of the transaction which is relevant to VM execution.
- Computation: The computational state and result of VM execution.
- Opcode: The logic for a single opcode.

The Chain

The term **Chain** is used to encapsulate:

- The state transition function (e.g. VM opcodes and execution logic)
- Protocol rules (e.g. block rewards, header rewards, difficulty calculations, transaction execution)
- The chain data (e.g. **Headers**, **Blocks**, **Transactions** and **Receipts**)
- The state data (e.g. **balance**, **nonce**, **code** and **storage**)
- The chain state (e.g. tracking the chain head, canonical blocks)

Note: While a chain is used to *wrap* these concepts, many of them are actually defined at lower layers such as the underlying **Virtual Machines**.

The `Chain` object itself is largely an interface and orchestration layer. Most of the `Chain` APIs merely serving as a passthrough to the appropriate `VM`.

A chain has one or more underlying **Virtual Machines** or VMs. The chain contains a mapping which defines which VM should be active for which blocks.

The chain for the public mainnet Ethereum blockchain would have a separate VM defined for each fork ruleset (e.g. **Frontier**, **Homestead**, **Tangerine Whistle**, **Spurious Dragon**, **Byzantium**).

The VM

The term **VM** is used to encapsulate:

- The state transition function for a single fork ruleset.
- Orchestration logic for transaction execution.
- Block construction and validation.
- Chain data storage and retrieval APIs

The `VM` object loosely mirrors many of the Chain APIs for retrieval of chain state such as blocks, headers, transactions and receipts. It is also responsible for block level protocol logic such as block creation and validation.

The VMState

The term **VMState** is used to encapsulate:

- Execution context for the VM (e.g. `coinbase` or `gas_limit`)
- The state root defining the current VM state.
- Some block validation

The Message

The term **Message** comes from the yellow paper. It encapsulates the information from the transaction needed to initiate the outermost layer of VM execution.

- Parameters like `sender`, `value`, `to`

The message can be thought of as the VM's internal representation of a transaction.

The Computation

The term **Computation** is used to encapsulate:

- The computational state during VM execution (e.g. memory, stack, gas metering)
- The computational results of VM execution (e.g. return data, gas consumption and refunds, execution errors)

This abstraction is the interface through which opcode logic is implemented.

The Opcode

The term **Opcode** is used to encapsulate:

- A single instruction within the VM such as the `ADD` or `MUL` opcodes.

Opcodes are implemented as TODO

Understanding the mining process

From the *EVM Cookbook* we can already learn how to use the `Chain` class to create a single blockchain as a combination of different virtual machines for different spans of blocks.

In this guide we want to build up on that knowledge and look into the actual mining process.

Note: Mining is an overloaded term and in fact the names of the mentioned APIs are subject to change.

Mining

The term *mining* can refer to different things depending on our point of view. Most of the time when we read about *mining*, we talk about the process where several parties are *competing* to be the first to create a new valid block and pass it on to the network.

In this guide, when we talk about the `mine_block()` API, we are only referring to the part that creates, validates and sets a block as the new canonical head of the chain but not necessarily as part of the mentioned competition to be the first. In fact, the `mine_block()` API is internally also called when we import existing blocks that others created.

Mining an empty block

Usually when we think about creating blocks we naturally think about adding transactions to the block first because, after all, one primary use case for the Ethereum blockchain is to process *transactions* which are wrapped in blocks.

For the sake of simplicity though, we'll mine an empty block as a first example (meaning the block will not contain any transactions)

As a refresher, he's how we create a chain as demonstrated in the *Using the chain object recipe* from the cookbook.

```
from eth.db.atomic import AtomicDB
from eth.chains.mainnet import MAINNET_GENESIS_HEADER

# increase the gas limit
genesis_header = MAINNET_GENESIS_HEADER.copy(gas_limit=3141592)

# initialize a fresh chain
chain = chain_class.from_genesis_header(AtomicDB(), genesis_header)
```

Since we decided to not add any transactions to our block let's just call `mine_block()` and see what happens.

```
# initialize a fresh chain
chain = chain_class.from_genesis_header(AtomicDB(), genesis_header)

chain.mine_block()
```

Aw, snap! We're running into an exception at `check_pow()`. Apparently we are trying to add a block to the chain that doesn't qualify the Proof-of-Work (PoW) rules. The error tells us precisely that the `mix_hash` of our block does not match the expected value.

```
Traceback (most recent call last):
  File "scripts/benchmark/run.py", line 111, in <module>
    run()
  File "scripts/benchmark/run.py", line 52, in run
    block = chain.mine_block()  #**pow_args
  File "/py-evm/eth/chains/base.py", line 545, in mine_block
    self.validate_block(mined_block)
  File "/py-evm/eth/chains/base.py", line 585, in validate_block
    self.validate_seal(block.header)
  File "/py-evm/eth/chains/base.py", line 622, in validate_seal
    header.mix_hash, header.nonce, header.difficulty)
  File "/py-evm/eth/consensus/pow.py", line 70, in check_pow
    encode_hex(mining_output[b'mix digest']), encode_hex(mix_hash)))

eth.exceptions.ValidationError: mix hash mismatch;
0x7a76bbf0c8d0e683fafa2d7cab27f601e19f35e7ecad7e1abb064b6f8f08fe21 !=
0x0000000000000000000000000000000000000000000000000000000000000000
```

Let's lookup how `check_pow()` is implemented.

```
def check_pow(block_number: int,
              mining_hash: Hash32,
              mix_hash: Hash32,
```

(continues on next page)

(continued from previous page)

```
                 nonce: bytes,
                 difficulty: int) -> None:
    validate_length(mix_hash, 32, title="Mix Hash")
    validate_length(mining_hash, 32, title="Mining Hash")
    validate_length(nonce, 8, title="POW Nonce")
    cache = get_cache(block_number)
    mining_output = hashimoto_light(
        block_number, cache, mining_hash, big_endian_to_int(nonce))
    if mining_output[b'mix digest'] != mix_hash:
        raise ValidationError("mix hash mismatch; {0} != {1}".format(
            encode_hex(mining_output[b'mix digest']), encode_hex(mix_hash)))
    result = big_endian_to_int(mining_output[b'result'])
    validate_lte(result, 2**256 // difficulty, title="POW Difficulty")
```

Just by looking at the signature of that function we can see that validating the PoW is based on the following parameters:

- `block_number` - the number of the given block
- `difficulty` - the difficulty of the PoW algorithm
- `mining_hash` - hash of the mining header
- `mix_hash` - together with the `nonce` forms the actual proof
- `nonce` - together with the `mix_hash` forms the actual proof

The PoW algorithm checks that all these parameters match correctly, ensuring that only valid blocks can be added to the chain.

In order to produce a valid block, we have to set the correct `mix_hash` and `nonce` in the header. We can pass these as key-value pairs when we call `mine_block()` as seen below.

```
chain.mine_block(nonce=valid_nonce, mix_hash=valid_mix_hash)
```

This call will work just fine assuming we are passing the correct `nonce` and `mix_hash` that corresponds to the block getting mined.

Retrieving a valid nonce and mix hash

Now that we know we can call `mine_block()` with the correct parameters to successfully add a block to our chain, let's briefly go over an example that demonstrates how we can retrieve a matching `nonce` and `mix_hash`.

Note: Py-EVM currently doesn't offer a stable API for actual PoW mining. The following code is for demonstration purpose only.

Mining on the main ethereum chain is a competition done simultanously by many miners, hence the *mining difficulty* is pretty high which means it will take a very long time to find the right `nonce` and `mix_hash` on commodity hardware. In order for us to have something that we can tinker with on a regular laptop, we'll construct a test chain with the `difficulty` set to `1`.

Let's start off by defining the `GENESIS_PARAMS`.

```
from eth import constants

GENESIS_PARAMS = {
```

(continues on next page)

(continued from previous page)

```
        'parent_hash': constants.GENESIS_PARENT_HASH,
        'uncles_hash': constants.EMPTY_UNCLE_HASH,
        'coinbase': constants.ZERO_ADDRESS,
        'transaction_root': constants.BLANK_ROOT_HASH,
        'receipt_root': constants.BLANK_ROOT_HASH,
        'difficulty': 1,
        'block_number': constants.GENESIS_BLOCK_NUMBER,
        'gas_limit': 3141592,
        'timestamp': 1514764800,
        'extra_data': constants.GENESIS_EXTRA_DATA,
        'nonce': constants.GENESIS_NONCE
  }
```

Next, we'll create the chain itself using the defined GENESIS_PARAMS and the latest ByzantiumVM.

```
from eth import MiningChain
from eth.vm.forks.byzantium import ByzantiumVM
from eth.db.backends.memory import AtomicDB


klass = MiningChain.configure(
    __name__='TestChain',
    vm_configuration=(
        (constants.GENESIS_BLOCK_NUMBER, ByzantiumVM),
    ))
chain = klass.from_genesis(AtomicDB(), GENESIS_PARAMS)
```

Now that we have the building blocks available, let's put it all together and mine a proper block!

```
from eth.consensus.pow import mine_pow_nonce


# We have to finalize the block first in order to be able read the
# attributes that are important for the PoW algorithm
block = chain.get_vm().finalize_block(chain.get_block())

# based on mining_hash, block number and difficulty we can perform
# the actual Proof of Work (PoW) mechanism to mine the correct
# nonce and mix_hash for this block
nonce, mix_hash = mine_pow_nonce(
    block.number,
    block.header.mining_hash,
    block.header.difficulty)

block = chain.mine_block(mix_hash=mix_hash, nonce=nonce)
```

```
>>> print(block)
Block #1
```

Let's take a moment to fully understand what this code does.

1. We call *finalize_block()* on the underlying VM in order to retrieve the information that we need to calculate the nonce and the mix_hash.

2. We then call mine_pow_nonce() to retrieve the proper nonce and mix_hash that we need to mine the block and satisfy the validation.

3. Finally we call mine_block() and pass along the nonce and the mix_hash

Note: The code above will essentially perform `finalize_block` twice. Keep in mind this code is for demonstration purpose only and that Py-EVM will provide a pluggable system in the future to allow PoW mining among other things.

Mining a block with transactions

Now that we've learned the basics of how the mining process works, let's revisited our example and add a transaction before we mine another block. There are a couple of concepts we need to dive into in order to accomplish that goal.

Every transaction goes from a sender `Address` to a receiver `Address`. Each transaction takes some computational power to get processed that is measured in a unit called `gas`.

In practice, we have to pay the miners to put our transaction in a block. However, there is no *technical* reason why we have to pay for the computing power, but only an economical, i.e. in reality we'll usually have trouble finding a miner who's willing to include a transaction that doesn't pay for its computational costs.

In this example, however, **we are the miner** which means we are free to include any transactions we like. In the spirit of this guide, let's start simple and create a transaction that sends zero ether from one address to another address. Keep in mind that even if the value being transferred is zero, there's still a computational cost for the processing but since we are the miner, we'll mine it anyway even if no one is willing to pay for it!

Let's first setup the sender and receiver.

```
from eth_keys import keys
from eth_utils import decode_hex
from eth_typing import Address

SENDER_PRIVATE_KEY = keys.PrivateKey(
  decode_hex('0x45a915e4d060149eb4365960e6a7a45f334393093061116b197e3240065ff2d8')
)

SENDER = Address(SENDER_PRIVATE_KEY.public_key.to_canonical_address())

RECEIVER = Address(b'\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\x02')
```

One thing that strikes out here is that we only need the plain address for the receiver whereas for the sender we are obtaining an address derived from the `SENDER_PRIVATE_KEY`. That's because we obviously can not send transactions from an address that we don't have the private key to sign it for.

With sender and receiver prepared, let's create the actual transaction.

```
vm = chain.get_vm()
nonce = vm.state.account_db.get_nonce(SENDER)

tx = vm.create_unsigned_transaction(
    nonce=nonce,
    gas_price=0,
    gas=100000,
    to=RECEIVER,
    value=0,
    data=b'',
)
```

Every transaction needs a `nonce` not to be confused with the `nonce` that we previously mined as part of the PoW algorithm. The *transaction nonce* serves as a counter to ensure all transactions from one address are processed in order. We retrieve the current `nonce` by calling `get_nonce(sender)()`.

Once we have the nonce we can call *create_unsigned_transaction()* and pass the nonce among the rest of the transaction attributes as key-value pairs.

- nonce - Number of transactions sent by the sender
- gas_price - Number of Wei to pay per unit of gas
- gas - Maximum amount of gas the transaction is allowed to consume before it gets rejected
- to - Address of transaction recipient
- value - Number of Wei to be transferred to the recipient

The last step we need to do before we can add the transaction to a block is to sign it with the private key which is as simple as calling *as_signed_transaction()* with the SENDER_PRIVATE_KEY.

```
signed_tx = tx.as_signed_transaction(SENDER_PRIVATE_KEY)
```

Finally, we can call apply_transaction() and pass along the signed_tx.

```
chain.apply_transaction(signed_tx)
```

What follows is the complete script that demonstrates how to mine a single block with one simple zero value transfer transaction.

```
>>> from eth_keys import keys
>>> from eth_utils import decode_hex
>>> from eth_typing import Address
>>> from eth import constants
>>> from eth.chains.base import MiningChain
>>> from eth.consensus.pow import mine_pow_nonce
>>> from eth.vm.forks.byzantium import ByzantiumVM
>>> from eth.db.atomic import AtomicDB


>>> GENESIS_PARAMS = {
...     'parent_hash': constants.GENESIS_PARENT_HASH,
...     'uncles_hash': constants.EMPTY_UNCLE_HASH,
...     'coinbase': constants.ZERO_ADDRESS,
...     'transaction_root': constants.BLANK_ROOT_HASH,
...     'receipt_root': constants.BLANK_ROOT_HASH,
...     'difficulty': 1,
...     'block_number': constants.GENESIS_BLOCK_NUMBER,
...     'gas_limit': 3141592,
...     'timestamp': 1514764800,
...     'extra_data': constants.GENESIS_EXTRA_DATA,
...     'nonce': constants.GENESIS_NONCE
... }

>>> SENDER_PRIVATE_KEY = keys.PrivateKey(
...     decode_hex('0x45a915e4d060149eb4365960e6a7a45f334393093061116b197e3240065ff2d8
↪')
... )

>>> SENDER = Address(SENDER_PRIVATE_KEY.public_key.to_canonical_address())

>>> RECEIVER = Address(b'\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\x02')

>>> klass = MiningChain.configure(
...     __name__='TestChain',
```

(continues on next page)

(continued from previous page)

```
...     vm_configuration=(
...         (constants.GENESIS_BLOCK_NUMBER, ByzantiumVM),
...     ))

>>> chain = klass.from_genesis(AtomicDB(), GENESIS_PARAMS)
>>> vm = chain.get_vm()

>>> nonce = vm.state.account_db.get_nonce(SENDER)

>>> tx = vm.create_unsigned_transaction(
...     nonce=nonce,
...     gas_price=0,
...     gas=100000,
...     to=RECEIVER,
...     value=0,
...     data=b'',
... )

>>> signed_tx = tx.as_signed_transaction(SENDER_PRIVATE_KEY)

>>> chain.apply_transaction(signed_tx)
(<ByzantiumBlock(#Block #1...)
>>> # We have to finalize the block first in order to be able read the
>>> # attributes that are important for the PoW algorithm
>>> block = chain.get_vm().finalize_block(chain.get_block())

>>> # based on mining_hash, block number and difficulty we can perform
>>> # the actual Proof of Work (PoW) mechanism to mine the correct
>>> # nonce and mix_hash for this block
>>> nonce, mix_hash = mine_pow_nonce(
...     block.number,
...     block.header.mining_hash,
...     block.header.difficulty
... )

>>> chain.mine_block(mix_hash=mix_hash, nonce=nonce)
<ByzantiumBlock(#Block #1)>
```

Creating Opcodes

An opcode is just a function which takes a *BaseComputation* instance as it's sole argument. If an opcode function has a return value, this value will be discarded during normal VM execution.

Here are some simple examples.

```
def noop(computation):
    """
    An opcode which does nothing (not even consume gas)
    """
    pass

def burn_5_gas(computation):
    """
    An opcode which simply burns 5 gas
    """
    computation.consume_gas(5, reason='why not?')
```

The `as_opcode()` helper

While these examples are demonstrative of *simple* logic, opcodes will traditionally have an intrinsic gas cost associated with them. Py-EVM offers an abstraction which allows for decoupling of gas consumption from opcode logic which can be convenient for cases where an opcode's gas cost changes between different VM rules but its logic remains constant.

`eth.vm.opcode.`**`as_opcode`**(*logic_fn*, *mnemonic*, *gas_cost*)

- The `logic_fn` argument should be a callable conforming to the opcode API, taking a *~eth.vm.computation.Computation* instance as its sole argument.
- The `mnemonic` is a string such as `'ADD'` or `'MUL'`.
- The `gas_cost` is the gas cost to execute this opcode.

The return value is a function which will consume the `gas_cost` prior to execution of the `logic_fn`.

Usage of the *`as_opcode()`* helper:

```
def custom_op(computation):
    ... # opcode logic here

class ExampleComputation(BaseComputation):
    opcodes = {
        b'\x01': as_opcode(custom_op, 'CUSTOM_OP', 10),
    }
```

Opcodes as classes

Sometimes it may be helpful to share common logic between similar opcodes, or the same opcode across multiple fork rules. In these cases, implementing opcodes as classes *may* be the right choice. This is as simple as implementing a `__call__` method on your class which conforms to the opcode API, taking a single `Computation` instance as the sole argument.

```
class MyOpcode:
    def initial_logic(self, computation):
        ...

    def main_logic(self, computation):
        ...

    def cleanup_logic(self, computation):
        ...

    def __call__(self, computation):
        self.initial_logic(computation)
        self.main_logic(computation)
        self.cleanup_logic(computation)
```

With this pattern, the overall structure, as well as much of the logic can be re-used while still allowing a mechanism for overriding individual sections of the opcode logic.

2.6 API

This section aims to provide a detailed description of all APIs. If you are looking for something more hands-on or higher-level check out the existing *guides*.

Warning: We expect each alpha release to have breaking changes to the API.

2.6.1 Trinity

This section aims to provide a detailed description of all APIs. If you are looking for something more hands-on or higher-level check out the existing *Trinity guides*.

Command Line Interface (CLI)

```
usage: trinity [-h] [--version] [--trinity-root-dir TRINITY_ROOT_DIR]
            [-l {debug,info}] [--network-id NETWORK_ID | --ropsten]
            [--sync-mode {full,light} | --light] [--data-dir DATA_DIR]
            [--nodekey NODEKEY] [--nodekey-path NODEKEY_PATH]
            {console,attach} ...

positional arguments:
{console,attach}
    console             run the chain and start the trinity REPL
    attach              open an REPL attached to a currently running chain

optional arguments:
-h, --help            show this help message and exit

sync mode:
--version             show program's version number and exit
--trinity-root-dir TRINITY_ROOT_DIR
                        The filesystem path to the base directory that trinity
                        will store it's information. Default:
                        $XDG_DATA_HOME/.local/share/trinity

logging:
-l {debug,info}, --log-level {debug,info}
                        Sets the logging level

network:
--network-id NETWORK_ID
                        Network identifier (1=Mainnet, 3=Ropsten)
--ropsten             Ropsten network: pre configured proof-of-work test
                        network. Shortcut for `--networkid=3`

sync mode:
--sync-mode {full,light}
--light               Shortcut for `--sync-mode=light`

chain:
--data-dir DATA_DIR   The directory where chain data is stored
--nodekey NODEKEY     Hexadecimal encoded private key to use for the nodekey
```

(continues on next page)

(continued from previous page)

```
--nodekey-path NODEKEY_PATH
                        The filesystem path to the file which contains the
                        nodekey
```

Extensibility

Warning: The extensibility API isn't stable yet. Expect breaking changes.

Events

class `trinity.extensibility.events.`**`PluginStartedEvent`**(*plugin_type: Type[BasePlugin]*)

Broadcasted when a plugin was started

class `trinity.extensibility.events.`**`ResourceAvailableEvent`**(*resource: Any, resource_type: Type[Any]*)

Broadcasted when a resource becomes available

Exceptions

class `trinity.extensibility.exceptions.`**`EventBusNotReady`**

Raised when a plugin tried to access an `EventBus` before the plugin had received its *`on_ready()`* call.

class `trinity.extensibility.exceptions.`**`UnsuitableShutdownError`**

Raised when *`shutdown()`* was called on a *`PluginManager`* instance that operates in the *`MainAndIsolatedProcessScope`* or when `shutdown_blocking()` was called on a *`PluginManager`* instance that operates in the *`SharedProcessScope`*.

Plugin

PluginContext

class `trinity.extensibility.plugin.`**`PluginContext`**(*endpoint: lahja.endpoint.Endpoint, boot_info: trinity.extensibility.plugin.TrinityBootInfo*)

The *`PluginContext`* holds valuable contextual information and APIs to be used by a plugin. This includes the parsed arguments that were used to launch `Trinity` as well as an `Endpoint` that the plugin can use to connect to the central `EventBus`.

The *`PluginContext`* is set during startup and is guaranteed to exist by the time that a plugin receives its *`on_ready()`* call.

`args`

Return the parsed arguments that were used to launch the application

`event_bus`

Return the `Endpoint` that the plugin uses to connect to the central `EventBus`

shutdown_host (*reason: str*) → None
: Shutdown `Trinity` by broadcasting a `ShutdownRequest` on the `EventBus`. The actual shutdown routine is executed and coordinated by the main application process who listens for this event.

trinity_config
: Return the `TrinityConfig`

BasePlugin

class trinity.extensibility.plugin.**BasePlugin**

configure_parser (*arg_parser: argparse.ArgumentParser*, *subparser: argparse._SubParsersAction*) → None
: Give the plugin a chance to amend the Trinity CLI argument parser. This hook is called before *`on_ready()`*

do_start () → None
: Perform the actual plugin start routine. In the case of a *BaseIsolatedPlugin* this method will be called in a separate process.

This method should usually be overwritten by subclasses with the exception of plugins that set `func` on the `ArgumentParser` to redefine the entire host program.

event_bus
: Get the `Endpoint` that this plugin uses to connect to the `EventBus`

logger
: Get the `Logger` for this plugin.

name
: Describe the name of the plugin.

on_ready () → None
: Notify the plugin that it is ready to bootstrap itself. Plugins can rely on the *`PluginContext`* to be set after this method has been called.

ready () → None
: Set the `status` to `PluginStatus.READY` and delegate to *`on_ready()`*

running
: Return `True` if the `status` is `PluginStatus.STARTED`, otherwise return `False`.

set_context (*context: trinity.extensibility.plugin.PluginContext*) → None
: Set the *`PluginContext`* for this plugin.

start () → None
: Delegate to *`do_start()`* and set `running` to `True`. Broadcast a *`PluginStartedEvent`* on the `EventBus` and hence allow other plugins to act accordingly.

status
: Return the current `PluginStatus` of the plugin.

BaseSyncStopPlugin

class trinity.extensibility.plugin.**BaseSyncStopPlugin**
: A *`BaseSyncStopPlugin`* unwinds synchronoulsy, hence blocks until the shutdown is done.

do_stop() → None
Stop the plugin. Should be overwritten by subclasses.

stop() → None
Delegate to *do_stop()* causing the plugin to stop and setting `running` to `False`.

BaseAsyncStopPlugin

class trinity.extensibility.plugin.**BaseAsyncStopPlugin**
A *BaseAsyncStopPlugin* unwinds asynchronoulsy, hence needs to be awaited.

coroutine do_stop() → None
Asynchronously stop the plugin. Should be overwritten by subclasses.

coroutine stop() → None
Delegate to *do_stop()* causing the plugin to stop asynchronously and setting `running` to `False`.

BaseMainProcessPlugin

class trinity.extensibility.plugin.**BaseMainProcessPlugin**
A *BaseMainProcessPlugin* overtakes the whole main process early before any of the subsystems started. In that sense it redefines the whole meaning of the `trinity` command.

BaseIsolatedPlugin

class trinity.extensibility.plugin.**BaseIsolatedPlugin**
A *BaseIsolatedPlugin* runs in an isolated process and hence provides security and flexibility by not making assumptions about its internal operations.

Such plugins are free to use non-blocking asyncio as well as synchronous calls. When an isolated plugin is stopped it does first receive a SIGINT followed by a SIGTERM soon after. It is up to the plugin to handle these signals accordingly.

do_stop() → None
Stop the plugin. Should be overwritten by subclasses.

start() → None
Prepare the plugin to get started and eventually call `do_start` in a separate process.

PluginManager

BaseManagerProcessScope

class trinity.extensibility.plugin_manager.**BaseManagerProcessScope**
Define the operational model under which a *PluginManager* works. Subclasses define whether a *PluginManager* is responsible to manage a specific plugin and how its *PluginContext* is created.

create_plugin_context(*plugin: trinity.extensibility.plugin.BasePlugin, boot_info: trinity.extensibility.plugin.TrinityBootInfo*) → None
Create the *PluginContext* for the given `plugin`.

is_responsible_for_plugin(*plugin: trinity.extensibility.plugin.BasePlugin*) → bool
Define whether a *PluginManager* operating under this scope is responsible to manage the given `plugin`.

MainAndIsolatedProcessScope

class trinity.extensibility.plugin_manager.**MainAndIsolatedProcessScope**(*event_bus: lahja.eventbus.EventBus*, *main_proc_endpoint: lahja.endpoint.Endpoint*)

create_plugin_context(*plugin: trinity.extensibility.plugin.BasePlugin*, *boot_info: trinity.extensibility.plugin.TrinityBootInfo*) → None
: Create a `PluginContext` that holds a reference to a dedicated new `Endpoint` to enable plugins which run in their own isolated processes to connect to the central `EventBus` that Trinity uses to enable application wide event-driven communication even across process boundaries.

is_responsible_for_plugin(*plugin: trinity.extensibility.plugin.BasePlugin*) → bool
: Return `True` if if the plugin instance is a subclass of `BaseIsolatedPlugin` or `BaseMainProcessPlugin`

SharedProcessScope

class trinity.extensibility.plugin_manager.**SharedProcessScope**(*shared_proc_endpoint: lahja.endpoint.Endpoint*)

create_plugin_context(*plugin: trinity.extensibility.plugin.BasePlugin*, *boot_info: trinity.extensibility.plugin.TrinityBootInfo*) → None
: Create a `PluginContext` that uses the `Endpoint` of the `PluginManager` to communicate with the central `EventBus` that Trinity uses to enable application wide, event-driven communication even across process boundaries.

is_responsible_for_plugin(*plugin: trinity.extensibility.plugin.BasePlugin*) → bool
: Return `True` if if the plugin instance is a subclass of `BaseAsyncStopPlugin`.

PluginManager

class trinity.extensibility.plugin_manager.**PluginManager**(*scope: trinity.extensibility.plugin_manager.BaseManagerPro*

The plugin manager is responsible to register, keep and manage the life cycle of any available plugins.

A `PluginManager` is tight to a specific `BaseManagerProcessScope` which defines which plugins are controlled by this specific manager instance.

This is due to the fact that Trinity currently allows plugins to either run in a shared process, also known as the "networking" process, as well as in their own isolated processes.

Trinity uses two different `PluginManager` instances to govern these different categories of plugins.

Note: This API is very much in flux and is expected to change heavily.

amend_argparser_config(*arg_parser: argparse.ArgumentParser*, *subparser: argparse._SubParsersAction*) → None
: Call `configure_parser()` for every registered plugin, giving them the option to amend the global parser setup.

event_bus_endpoint
: Return the `Endpoint` that the `PluginManager` instance uses to connect to the central `EventBus`.

prepare(*args: argparse.Namespace*, *trinity_config: trinity.config.TrinityConfig*, *boot_kwargs: Dict[str, Any] = None*) → None
Create and set the `PluginContext` and call `ready()` on every plugin that this plugin manager instance is responsible for.

register(*plugins: Union[trinity.extensibility.plugin.BasePlugin, Iterable[trinity.extensibility.plugin.BasePlugin]]*) → None
Register one or multiple instances of `BasePlugin` with the plugin manager.

coroutine shutdown() → None
Asynchronously shut down all running plugins. Raises an `UnsuitableShutdownError` if called on a `PluginManager` that operates in the `MainAndIsolatedProcessScope`.

shutdown_blocking() → None
Synchronously shut down all running plugins. Raises an `UnsuitableShutdownError` if called on a `PluginManager` that operates in the `SharedProcessScope`.

2.6.2 Py-EVM

This section aims to provide a detailed description of all APIs. If you are looking for something more hands-on or higher-level check out the existing *EVM guides*.

Chain

BaseChain

class eth.chains.base.**BaseChain**
The base class for all Chain objects

classmethod get_vm_class(*header: eth.rlp.headers.BlockHeader*) → Type[BaseVM]
Returns the VM instance for the given block number.

classmethod get_vm_class_for_block_number(*block_number: NewType.<locals>.new_type*) → Type[BaseVM]
Returns the VM class for the given block number.

classmethod validate_chain(*root: eth.rlp.headers.BlockHeader*, *descendants: Tuple[eth.rlp.headers.BlockHeader, ...]*, *seal_check_random_sample_rate: int = 1*) → None
Validate that all of the descendents are valid, given that the root header is valid.

By default, check the seal validity (Proof-of-Work on Ethereum 1.x mainnet) of all headers. This can be expensive. Instead, check a random sample of seals using seal_check_random_sample_rate.

Chain

class eth.chains.base.**Chain**(*base_db: eth.db.backends.base.BaseAtomicDB*)
A Chain is a combination of one or more VM classes. Each VM is associated with a range of blocks. The Chain class acts as a wrapper around these other VM classes, delegating operations to the appropriate VM depending on the current block number.

build_block_with_transactions(*transactions: Tuple[eth.rlp.transactions.BaseTransaction, ...], parent_header: eth.rlp.headers.BlockHeader = None*) → Tuple[eth.rlp.blocks.BaseBlock, Tuple[eth.rlp.receipts.Receipt, ...], Tuple[eth.vm.computation.BaseComputation, ...]]

Generate a block with the provided transactions. This does *not* import that block into your chain. If you want this new block in your chain, run `import_block()` with the result block from this method.

Parameters

- **transactions** – an iterable of transactions to insert to the block
- **parent_header** – parent of the new block – or canonical head if `None`

Returns (new block, receipts, computations)

chaindb_class

alias of `eth.db.chain.ChainDB`

create_header_from_parent(*parent_header: eth.rlp.headers.BlockHeader, **header_params*) → eth.rlp.headers.BlockHeader

Passthrough helper to the VM class of the block descending from the given header.

create_transaction(**args, **kwargs*) → eth.rlp.transactions.BaseTransaction

Passthrough helper to the current VM class.

create_unsigned_transaction(**, nonce: int, gas_price: int, gas: int, to: NewType.<locals>.new_type, value: int, data: bytes*) → eth.rlp.transactions.BaseUnsignedTransaction

Passthrough helper to the current VM class.

ensure_header(*header: eth.rlp.headers.BlockHeader = None*) → eth.rlp.headers.BlockHeader

Return `header` if it is not `None`, otherwise return the header of the canonical head.

estimate_gas(*transaction: Union[BaseTransaction, SpoofTransaction], at_header: eth.rlp.headers.BlockHeader = None*) → int

Returns an estimation of the amount of gas the given transaction will use if executed on top of the block specified by the given header.

classmethod from_genesis(*base_db: eth.db.backends.base.BaseAtomicDB, genesis_params: Dict[str, Union[int, None, bytes, NewType.<locals>.new_type, NewType.<locals>.new_type]], genesis_state: Dict[NewType.<locals>.new_type, eth.typing.AccountDetails] = None*) → eth.chains.base.BaseChain

Initializes the Chain from a genesis state.

classmethod from_genesis_header(*base_db: eth.db.backends.base.BaseAtomicDB, genesis_header: eth.rlp.headers.BlockHeader*) → eth.chains.base.BaseChain

Initializes the chain from the genesis header.

get_ancestors(*limit: int, header: eth.rlp.headers.BlockHeader*) → Tuple[eth.rlp.blocks.BaseBlock, ...]

Return *limit* number of ancestor blocks from the current canonical head.

get_block() → eth.rlp.blocks.BaseBlock

Returns the current TIP block.

get_block_by_hash(*block_hash: NewType.<locals>.new_type*) → eth.rlp.blocks.BaseBlock

Returns the requested block as specified by block hash.

get_block_by_header(*block_header: eth.rlp.headers.BlockHeader*) → eth.rlp.blocks.BaseBlock

Returns the requested block as specified by the block header.

get_block_header_by_hash(*block_hash: NewType.<locals>.new_type*) → eth.rlp.headers.BlockHeader
Returns the requested block header as specified by block hash.

Raises BlockNotFound if there's no block header with the given hash in the db.

get_canonical_block_by_number(*block_number: NewType.<locals>.new_type*) → eth.rlp.blocks.BaseBlock
Returns the block with the given number in the canonical chain.

Raises BlockNotFound if there's no block with the given number in the canonical chain.

get_canonical_block_hash(*block_number: NewType.<locals>.new_type*) → NewType.<locals>.new_type
Returns the block hash with the given number in the canonical chain.

Raises BlockNotFound if there's no block with the given number in the canonical chain.

get_canonical_head() → eth.rlp.headers.BlockHeader
Returns the block header at the canonical chain head.

Raises CanonicalHeadNotFound if there's no head defined for the canonical chain.

get_canonical_transaction(*transaction_hash: NewType.<locals>.new_type*) → eth.rlp.transactions.BaseTransaction
Returns the requested transaction as specified by the transaction hash from the canonical chain.

Raises TransactionNotFound if no transaction with the specified hash is found in the main chain.

get_score(*block_hash: NewType.<locals>.new_type*) → int
Returns the difficulty score of the block with the given hash.

Raises HeaderNotFound if there is no matching black hash.

get_transaction_result(*transaction: Union[BaseTransaction, SpoofTransaction], at_header: eth.rlp.headers.BlockHeader*) → bytes
Return the result of running the given transaction. This is referred to as a *call()* in web3.

get_vm(*at_header: eth.rlp.headers.BlockHeader = None*) → BaseVM
Returns the VM instance for the given block number.

import_block(*block: eth.rlp.blocks.BaseBlock*, *perform_validation: bool = True*) → Tuple[eth.rlp.blocks.BaseBlock, Tuple[eth.rlp.blocks.BaseBlock, ...], Tuple[eth.rlp.blocks.BaseBlock, ...]]
Imports a complete block and returns a 3-tuple

- the imported block
- a tuple of blocks which are now part of the canonical chain.
- a tuple of blocks which are were canonical and now are no longer canonical.

validate_block(*block: eth.rlp.blocks.BaseBlock*) → None
Performs validation on a block that is either being mined or imported.

Since block validation (specifically the uncle validation) must have access to the ancestor blocks, this validation must occur at the Chain level.

Cannot be used to validate genesis block.

validate_gaslimit(*header: eth.rlp.headers.BlockHeader*) → None
Validate the gas limit on the given header.

validate_seal(*header: eth.rlp.headers.BlockHeader*) → None
Validate the seal on the given header.

validate_uncles(*block: eth.rlp.blocks.BaseBlock*) → None
Validate the uncles for the given block.

DataBase

Backends

BaseDB

class eth.db.backends.base.**BaseDB**
This is an abstract key/value lookup with all `bytes` values, with some convenience methods for databases. As much as possible, you can use a DB as if it were a `dict`.

Notable exceptions are that you cannot iterate through all values or get the length. (Unless a subclass explicitly enables it).

All subclasses must implement these methods: __init__, __getitem__, __setitem__, __delitem__

Subclasses may optionally implement an _exists method that is type-checked for key and value.

LevelDB

class eth.db.backends.level.**LevelDB**(*db_path: pathlib.Path = None*, *max_open_files: int = None*)

MemoryDB

class eth.db.backends.memory.**MemoryDB**(*kv_store: Dict[bytes, bytes] = None*)

Account

BaseAccountDB

class eth.db.account.**BaseAccountDB**

make_state_root() → NewType.<locals>.new_type
Generate the state root with all the current changes in AccountDB

Returns the new state root

persist() → None
Send changes to underlying database, including the trie state so that it will forever be possible to read the trie from this checkpoint.

AccountDB

class eth.db.account.**AccountDB**(*db: eth.db.backends.base.BaseDB*, *state_root: NewType.<locals>.new_type = b'Vxe8x1fx17x1bxccUxa6xffx83Exe6x92xc0xf8n[Hxe0x1bx99lxadxc0x01b/xb5xe3cxb4!'*

make_state_root() → NewType.<locals>.new_type
Generate the state root with all the current changes in AccountDB

Returns the new state root

persist() → None
Send changes to underlying database, including the trie state so that it will forever be possible to read the trie from this checkpoint.

Journal

JournalDB

class eth.db.journal.**JournalDB**(*wrapped_db: eth.db.backends.base.BaseDB*)
A wrapper around the basic DB objects that keeps a journal of all changes. Each time a recording is started, the underlying journal creates a new changeset and assigns an id to it. The journal then keeps track of all changes that go into this changeset.

Discarding a changeset simply throws it away inculding all subsequent changesets that may have followed. Commiting a changeset merges the given changeset and all subsequent changesets into the previous changeset giving precidence to later changesets in case of conflicting keys.

Nothing is written to the underlying db until *persist()* is called.

The added memory footprint for a JournalDB is one key/value stored per database key which is changed. Subsequent changes to the same key within the same changeset will not increase the journal size since we only need to track latest value for any given key within any given changeset.

commit(*changeset_id: uuid.UUID*) → None
Commits a given changeset. This merges the given changeset and all subsequent changesets into the previous changeset giving precidence to later changesets in case of any conflicting keys.

If this is the base changeset then all changes will be written to the underlying database and the Journal starts a new recording.

discard(*changeset_id: uuid.UUID*) → None
Throws away all journaled data starting at the given changeset

persist() → None
Persist all changes in underlying db

record() → uuid.UUID
Starts a new recording and returns an id for the associated changeset

reset() → None
Reset the entire journal.

Chain

BaseChainDB

class eth.db.chain.**BaseChainDB**(*db: eth.db.backends.base.BaseAtomicDB*)

ChainDB

class eth.db.chain.**ChainDB**(*db: eth.db.backends.base.BaseAtomicDB*)

add_receipt(*block_header: eth.rlp.headers.BlockHeader*, *index_key: int*, *receipt: eth.rlp.receipts.Receipt*) → NewType.<locals>.new_type
Adds the given receipt to the provide block header.

Returns the updated *receipts_root* for updated block header.

add_transaction(*block_header: eth.rlp.headers.BlockHeader*, *index_key: int*, *transaction: BaseTransaction*) → NewType.<locals>.new_type
Adds the given transaction to the provide block header.

Returns the updated *transactions_root* for updated block header.

exists(*key: bytes*) → bool
Returns True if the given key exists in the database.

get(*key: bytes*) → bytes
Return the value for the given key or a KeyError if it doesn't exist in the database.

get_block_transaction_hashes(*block_header: eth.rlp.headers.BlockHeader*) → Iterable[NewType.<locals>.new_type]
Returns an iterable of the transaction hashes from th block specified by the given block header.

get_block_transactions(*header: eth.rlp.headers.BlockHeader, transaction_class: Type[BaseTransaction]*) → Iterable[BaseTransaction]
Returns an iterable of transactions for the block speficied by the given block header.

get_block_uncles(*uncles_hash: NewType.<locals>.new_type*) → List[eth.rlp.headers.BlockHeader]
Returns an iterable of uncle headers specified by the given uncles_hash

get_receipts(*header: eth.rlp.headers.BlockHeader, receipt_class: Type[eth.rlp.receipts.Receipt]*) → Iterable[eth.rlp.receipts.Receipt]
Returns an iterable of receipts for the block specified by the given block header.

get_transaction_by_index(*block_number: NewType.<locals>.new_type, transaction_index: int, transaction_class: Type[BaseTransaction]*) → BaseTransaction
Returns the transaction at the specified *transaction_index* from the block specified by *block_number* from the canonical chain.

Raises TransactionNotFound if no block

get_transaction_index(*transaction_hash: NewType.<locals>.new_type*) → Tuple[NewType.<locals>.new_type, int]
Returns a 2-tuple of (block_number, transaction_index) indicating which block the given transaction can be found in and at what index in the block transactions.

Raises TransactionNotFound if the transaction_hash is not found in the canonical chain.

persist_block(*block: BaseBlock*) → Tuple[Tuple[NewType.<locals>.new_type, ...], Tuple[NewType.<locals>.new_type, ...]]
Persist the given block's header and uncles.

Assumes all block transactions have been persisted already.

persist_trie_data_dict(*trie_data_dict: Dict[NewType.<locals>.new_type, bytes]*) → None
Store raw trie data to db from a dict

persist_uncles(*uncles: Tuple[eth.rlp.headers.BlockHeader]*) → NewType.<locals>.new_type
Persists the list of uncles to the database.

Returns the uncles hash.

Exceptions

exception `eth.exceptions.`**`BlockNotFound`**
Raised when the block with the given number/hash does not exist.

exception `eth.exceptions.`**`CanonicalHeadNotFound`**
Raised when the chain has no canonical head.

exception `eth.exceptions.`**`ContractCreationCollision`**
Raised when there was an address collision during contract creation.

exception `eth.exceptions.`**`FullStack`**
Raised when the stack is full.

exception `eth.exceptions.`**`Halt`**
Raised when an opcode function halts vm execution.

exception `eth.exceptions.`**`HeaderNotFound`**
Raised when a header with the given number/hash does not exist.

exception `eth.exceptions.`**`IncorrectContractCreationAddress`**
Raised when the address provided by transaction does not match the calculated contract creation address.

exception `eth.exceptions.`**`InsufficientFunds`**
Raised when an account has insufficient funds to transfer the requested value.

exception `eth.exceptions.`**`InsufficientStack`**
Raised when the stack is empty.

exception `eth.exceptions.`**`InvalidInstruction`**
Raised when an opcode is invalid.

exception `eth.exceptions.`**`InvalidJumpDestination`**
Raised when the jump destination for a JUMPDEST operation is invalid.

exception `eth.exceptions.`**`OutOfBoundsRead`**
Raised when an attempt was made to read data beyond the boundaries of the buffer (such as with RETURN-DATACOPY)

exception `eth.exceptions.`**`OutOfGas`**
Raised when a VM execution has run out of gas.

exception `eth.exceptions.`**`ParentNotFound`**
Raised when the parent of a given block does not exist.

exception `eth.exceptions.`**`PyEVMError`**
Base class for all py-evm errors.

exception `eth.exceptions.`**`Revert`**
Raised when the REVERT opcode occured

exception `eth.exceptions.`**`StackDepthLimit`**
Raised when the call stack has exceeded it's maximum allowed depth.

exception `eth.exceptions.`**`StateRootNotFound`**
Raised when the requested state root is not present in our DB.

exception `eth.exceptions.`**`TransactionNotFound`**
Raised when the transaction with the given hash or block index does not exist.

exception eth.exceptions.**VMError**
Base class for errors raised during VM execution.

exception eth.exceptions.**VMNotFound**
Raised when no VM is available for the provided block number.

exception eth.exceptions.**WriteProtection**
Raised when an attempt to modify the state database is made while operating inside of a STATICCALL context.

RLP

Accounts

Account

class eth.rlp.accounts.**Account**(*nonce: int = 0, balance: int = 0, storage_root: bytes = b'Vxe8x1fx17x1bxccUxa6xffx83Exe6x92xc0xf8n[Hxe0x1bx99lxadxc0x01b/xb5xe3cxb4!'*, *code_hash: bytes = b"xc5xd2Fx01x86xf7#<x92~}xb2xdcxc7x03xc0xe5x00xb6Sxcax82';{*
***kwargs*)
RLP object for accounts.

Blocks

BaseBlock

class eth.rlp.blocks.**BaseBlock**(**args, **kwargs*)

classmethod from_header(*header: eth.rlp.headers.BlockHeader, chaindb: eth.db.chain.BaseChainDB*) → eth.rlp.blocks.BaseBlock
Returns the block denoted by the given block header.

Headers

BlockHeader

class eth.rlp.headers.**BlockHeader**(*difficulty: int, block_number: int, gas_limit: int, timestamp: int = None, coinbase: NewType.<locals>.new_type = b'x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00', parent_hash: NewType.<locals>.new_type = b'x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x0 uncles_hash: NewType.<locals>.new_type = b'x1dxccMxe8xdexc7]zxabx85xb5gxb6xccxd4x1axd3x12Ex1bx94x8atx13xf0xa1Bxfd state_root: NewType.<locals>.new_type = b'Vxe8x1fx17x1bxccUxa6xffx83Exe6x92xc0xf8n[Hxe0x1bx99lxadxc0x01b/xb5xe3cx transaction_root: NewType.<locals>.new_type = b'Vxe8x1fx17x1bxccUxa6xffx83Exe6x92xc0xf8n[Hxe0x1bx99lxadxc0x01b/xb5xe3cx receipt_root: NewType.<locals>.new_type = b'Vxe8x1fx17x1bxccUxa6xffx83Exe6x92xc0xf8n[Hxe0x1bx99lxadxc0x01b/xb5xe3cx bloom: int = 0, gas_used: int = 0, extra_data: bytes = b", mix_hash: NewType.<locals>.new_type = b'x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x0 nonce: bytes = b'x00x00x00x00x00x00x00B'*)

classmethod from_parent(*parent: eth.rlp.headers.BlockHeader, gas_limit: int, difficulty: int, timestamp: int, coinbase: NewType.<locals>.new_type = b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00', nonce: bytes = None, extra_data: bytes = None, transaction_root: bytes = None, receipt_root: bytes = None*) → eth.rlp.headers.BlockHeader

Initialize a new block header with the *parent* header as the block's parent hash.

Logs

Log

class eth.rlp.logs.**Log**(*address: bytes, topics: List[int], data: bytes*)

Receipts

Receipt

class eth.rlp.receipts.**Receipt**(*state_root: bytes, gas_used: int, logs: Iterable[eth.rlp.logs.Log], bloom: int = None*)

Transactions

BaseTransactionMethods

class eth.rlp.transactions.**BaseTransactionMethods**

gas_used_by(*computation: eth.vm.computation.BaseComputation*) → int
Return the gas used by the given computation. In Frontier, for example, this is sum of the intrinsic cost and the gas used during computation.

get_intrinsic_gas() → int
Compute the baseline gas cost for this transaction. This is the amount of gas needed to send this transaction (but that is not actually used for computation).

intrinsic_gas
Convenience property for the return value of *get_intrinsic_gas*

validate() → None
Hook called during instantiation to ensure that all transaction parameters pass validation rules.

BaseTransactionFields

class eth.rlp.transactions.**BaseTransactionFields**(**args*, ***kwargs*)

BaseTransaction

class eth.rlp.transactions.**BaseTransaction**(**args*, ***kwargs*)

check_signature_validity() → None
Checks signature validity, raising a ValidationError if the signature is invalid.

classmethod create_unsigned_transaction(*, *nonce: int*, *gas_price: int*, *gas: int*, *to: NewType.<locals>.new_type*, *value: int*, *data: bytes*) → eth.rlp.transactions.BaseUnsignedTransaction
Create an unsigned transaction.

get_message_for_signing() → bytes
Return the bytestring that should be signed in order to create a signed transactions

get_sender() → NewType.<locals>.new_type
Get the 20-byte address which sent this transaction.

sender
Convenience property for the return value of *get_sender*

validate() → None
Hook called during instantiation to ensure that all transaction parameters pass validation rules.

BaseUnsignedTransaction

class eth.rlp.transactions.**BaseUnsignedTransaction**(**args*, ***kwargs*)

as_signed_transaction(*private_key: eth_keys.datatypes.PrivateKey*) → eth.rlp.transactions.BaseTransaction
Return a version of this transaction which has been signed using the provided *private_key*

Tools

Builders

Chain Builder

The chain builder utils are intended to reduce common boilerplace for both construction of chain classes as well as building up some desired chain state.

Note: These tools are best used in conjunction with `cytoolz.pipe`.

Constructing Chain Classes

The following utilities are provided to assist with constructing a chain class.

`eth.tools.builder.chain.`**`fork_at`**`()`

Adds the `vm_class` to the chain's `vm_configuration`.

```
from eth.chains.base import MiningChain
from eth.tools.builder.chain import build, fork_at

FrontierOnlyChain = build(MiningChain, fork_at(FrontierVM, 0))

# these two classes are functionally equivalent.
class FrontierOnlyChain(MiningChain):
    vm_configuration = (
        (0, FrontierVM),
    )
```

Note: This function is curriable.

The following pre-curried versions of this function are available as well, one for each mainnet fork.

- `frontier_at()`
- `homestead_at()`
- `tangerine_whistle_at()`
- `spurious_dragon_at()`
- `byzantium_at()`
- `constantinople_at()`

`eth.tools.builder.chain.`**`dao_fork_at`**`()`

Set the block number on which the DAO fork will happen. Requires that a version of the *`HomesteadVM`* is present in the chain's `vm_configuration`

`eth.tools.builder.chain.`**`disable_dao_fork`**`()`

Set the `support_dao_fork` flag to `False` on the *`HomesteadVM`*. Requires that presence of the *`HomesteadVM`* in the `vm_configuration`

`eth.tools.builder.chain.`**`enable_pow_mining`**`()`
Inject on demand generation of the proof of work mining seal on newly mined blocks into each of the chain's vms.

`eth.tools.builder.chain.`**`disable_pow_check`**`()`
Disable the proof of work validation check for each of the chain's vms. This allows for block mining without generation of the proof of work seal.

Note: blocks mined this way will not be importable on any chain that does not have proof of work disabled.

`eth.tools.builder.chain.`**`name`**`()`
Assign the given name to the chain class.

`eth.tools.builder.chain.`**`chain_id`**`()`
Set the `chain_id` for the chain class.

Initializing Chains

The following utilities are provided to assist with initializing a chain into the genesis state.

`eth.tools.builder.chain.`**`genesis`**`()`
Initialize the given chain class with the given genesis header parameters and chain state.

Building Chains

The following utilities are provided to assist with building out chains of blocks.

`eth.tools.builder.chain.`**`copy`**`()`
Make a copy of the chain at the given state. Actions performed on the resulting chain will not affect the original chain.

`eth.tools.builder.chain.`**`import_block`**`()`
Import the provided `block` into the chain.

`eth.tools.builder.chain.`**`import_blocks`**`(*blocks)` → Callable[eth.chains.base.BaseChain, eth.chains.base.BaseChain]
Variadic argument version of *`import_block()`*

`eth.tools.builder.chain.`**`mine_block`**`()`
Mine a new block on the chain. Header parameters for the new block can be overridden using keyword arguments.

`eth.tools.builder.chain.`**`mine_blocks`**`()`
Variadic argument version of *`mine_block()`*

`eth.tools.builder.chain.`**`chain_split`**`(*splits)` → Callable[eth.chains.base.BaseChain, Iterable[eth.chains.base.BaseChain]]
Construct and execute multiple concurrent forks of the chain.

Any number of forks may be executed. For each fork, provide an iterable of commands.

Returns the resulting chain objects for each fork.

```
chain_a, chain_b = build(
    mining_chain,
    chain_split(
        (mine_block(extra_data=b'chain-a'), mine_block()),
```

(continues on next page)

(continued from previous page)

```
        (mine_block(extra_data=b'chain-b'), mine_block(), mine_block()),
    ),
)
```

eth.tools.builder.chain.**at_block_number**()

Rewind the chain back to the given block number. Calls to things like `get_canonical_head` will still return the canonical head of the chain, however, you can use `mine_block` to mine fork chains.

Builder Tools

The JSON test fillers found in *eth.tools.fixtures* is a set of tools which facilitate creating standard JSON consensus tests as found in the ethereum/tests repository.

Note: Only VM and state tests are supported right now.

State Test Fillers

Tests are generated in two steps.

- First, a *test filler* is written that contains a high level description of the test case.
- Subsequently, the filler is compiled to the actual test in a process called filling, mainly consisting of calculating the resulting state root.

The test builder represents each stage as a nested dictionary. Helper functions are provided to assemble the filler file step by step in the correct format. The `fill_test()` function handles compilation and takes additional parameters that can't be inferred from the filler.

Creating a Filler

Fillers are generated in a functional fashion by piping a dictionary through a sequence of functions.

```
filler = pipe(
    setup_main_filler("test"),
    pre_state(
        (sender, "balance", 1),
        (receiver, "balance", 0),
    ),
    expect(
        networks=["Frontier"],
        transaction={
            "to": receiver,
            "value": 1,
            "secretKey": sender_key,
        },
        post_state=[
            [sender, "balance", 0],
            [receiver, "balance", 1],
        ]
    )
)
```

Note: Note that `setup_filler()` returns a dictionary, whereas all of the following functions such as `pre_state()`, `expect()`, expect to be passed a dictionary as their single argument and return an updated version of the dictionary.

`eth.tools.fixtures.fillers.common.`**`setup_main_filler`**(*name: str*, *environment: Dict[Any, Any] = None*) → Dict[str, Dict[str, Any]]

Kick off the filler generation process by creating the general filler scaffold with a test name and general information about the testing environment.

For tests for the main chain, the *environment* parameter is expected to be a dictionary with some or all of the following keys:

key	description
`"currentCoinbase"`	the coinbase address
`"currentNumber"`	the block number
`"previousHash"`	the hash of the parent block
`"currentDifficulty"`	the block's difficulty
`"currentGasLimit"`	the block's gas limit
`"currentTimestamp"`	the timestamp of the block

`eth.tools.fixtures.fillers.`**`pre_state`**(**raw_state, filler: Dict[str, Any]*) → None

Specify the state prior to the test execution. Multiple invocations don't override the state but extend it instead.

In general, the elements of *state_definitions* are nested dictionaries of the following form:

```
{
    address: {
        "nonce": <account nonce>,
        "balance": <account balance>,
        "code": <account code>,
        "storage": {
            <storage slot>: <storage value>
        }
    }
}
```

To avoid unnecessary nesting especially if only few fields per account are specified, the following and similar formats are possible as well:

```
(address, "balance", <account balance>)
(address, "storage", <storage slot>, <storage value>)
(address, "storage", {<storage slot>: <storage value>})
(address, {"balance", <account balance>})
```

`eth.tools.fixtures.fillers.`**`execution`**()

For VM tests, specify the code that is being run as well as the current state of the EVM. State tests don't support this object. The parameter is a dictionary specifying some or all of the following keys:

key	description
`"address"`	the address of the account executing the code
`"caller"`	the caller address
`"origin"`	the origin address (defaulting to the caller address)
`"value"`	the value of the call
`"data"`	the data passed with the call
`"gasPrice"`	the gas price of the call
`"gas"`	the amount of gas allocated for the call
`"code"`	the bytecode to execute
`"vyperLLLCode"`	the code in Vyper LLL (compiled to bytecode automatically)

`eth.tools.fixtures.fillers.`**`expect`**(*post_state: Dict[str, Any] = None*, *networks: Any = None*, *transaction: eth.typing.TransactionDict = None*) → Callable[..., Dict[str, Any]]

Specify the expected result for the test.

For state tests, multiple expectations can be given, differing in the transaction data, gas limit, and value, in the applicable networks, and as a result also in the post state. VM tests support only a single expectation with no specified network and no transaction (here, its role is played by *`execution()`*).

- `post_state` is a list of state definition in the same form as expected by *`pre_state()`*. State items that are not set explicitly default to their pre state.
- **`networks` defines the forks under which the expectation is applicable. It should be a** sublist of the following identifiers (also available in *ALL_FORKS*):
  - `"Frontier"`
  - `"Homestead"`
  - `"EIP150"`
  - `"EIP158"`
  - `"Byzantium"`
- `transaction` is a dictionary coming in two variants. For the main shard:

key	description
`"data"`	the transaction data,
`"gasLimit"`	the transaction gas limit,
`"gasPrice"`	the gas price,
`"nonce"`	the transaction nonce,
`"value"`	the transaction value

In addition, one should specify either the signature itself (via keys `"v"`, `"r"`, and `"s"`) or a private key used for signing (via `"secretKey"`).

Virtual Machine

Computation

BaseComputation

class eth.vm.computation.**BaseComputation**(*state: eth.vm.state.BaseState*, *message: eth.vm.message.Message*, *transaction_context: eth.vm.transaction_context.BaseTransactionContext*)

The base class for all execution computations.

> **Note:** Each *BaseComputation* class must be configured with:
>
> `opcodes`: A mapping from the opcode integer value to the logic function for the opcode.
>
> `_precompiles`: A mapping of contract address to the precompile function for execution of precompiled contracts.

apply_child_computation(*child_msg: eth.vm.message.Message*) → eth.vm.computation.BaseComputation

Apply the vm message `child_msg` as a child computation.

classmethod apply_computation(*state: eth.vm.state.BaseState*, *message: eth.vm.message.Message*, *transaction_context: eth.vm.transaction_context.BaseTransactionContext*) → eth.vm.computation.BaseComputation

Perform the computation that would be triggered by the VM message.

apply_create_message() → eth.vm.computation.BaseComputation

Execution of an VM message to create a new contract.

apply_message() → eth.vm.computation.BaseComputation

Execution of an VM message.

consume_gas(*amount: int*, *reason: str*) → None

Consume `amount` of gas from the remaining gas. Raise *eth.exceptions.OutOfGas* if there is not enough gas remaining.

extend_memory(*start_position: int*, *size: int*) → None

Extend the size of the memory to be at minimum `start_position + size` bytes in length. Raise *eth.exceptions.OutOfGas* if there is not enough gas to pay for extending the memory.

is_error

Return `True` if the computation resulted in an error.

is_origin_computation

Return `True` if this computation is the outermost computation at `depth == 0`.

is_success

Return `True` if the computation did not result in an error.

memory_read(*start_position: int*, *size: int*) → bytes

Read and return `size` bytes from memory starting at `start_position`.

memory_write(*start_position: int*, *size: int*, *value: bytes*) → None

Write `value` to memory at `start_position`. Require that `len(value) == size`.

output

Get the return value of the computation.

prepare_child_message(*gas: int*, *to: NewType.<locals>.new_type*, *value: int*, *data: bytes*, *code: bytes*, ***kwargs*) → eth.vm.message.Message
Helper method for creating a child computation.

raise_if_error() → None
If there was an error during computation, raise it as an exception immediately.

Raises *VMError* –

refund_gas(*amount: int*) → None
Add `amount` of gas to the pool of gas marked to be refunded.

return_gas(*amount: int*) → None
Return `amount` of gas to the available gas pool.

should_burn_gas
Return `True` if the remaining gas should be burned.

should_erase_return_data
Return `True` if the return data should be zerod out due to an error.

should_return_gas
Return `True` if the remaining gas should be returned.

stack_dup(*position: int*) → None
Duplicate the stack item at `position` and pushes it onto the stack.

stack_pop(*num_items: int = 1*, *type_hint: str = None*) → Any
Pop and return a number of items equal to `num_items` from the stack. `type_hint` can be either `'uint256'` or `'bytes'`. The return value will be an `int` or `bytes` type depending on the value provided for the `type_hint`.

Raise *eth.exceptions.InsufficientStack* if there are not enough items on the stack.

stack_push(*value: Union[int, bytes]*) → None
Push `value` onto the stack.

Raise *eth.exceptions.StackDepthLimit* if the stack is full.

stack_swap(*position: int*) → None
Swap the item on the top of the stack with the item at `position`.

CodeStream

class eth.vm.code_stream.**CodeStream**(*code_bytes: bytes*)

ExecutionContext

class eth.vm.execution_context.**ExecutionContext**(*coinbase: NewType.<locals>.new_type*, *timestamp: int*, *block_number: int*, *difficulty: int*, *gas_limit: int*, *prev_hashes: Tuple[NewType.<locals>.new_type, ...]*)

GasMeter

class eth.vm.gas_meter.**GasMeter**(*start_gas: int, refund_strategy: Callable[[int, int], int] = <function default_refund_strategy>*)

Memory

class eth.vm.memory.**Memory**

VM Memory

read(*start_position: int, size: int*) → bytes

Read a value from memory.

write(*start_position: int, size: int, value: bytes*) → None

Write *value* into memory.

Message

class eth.vm.message.**Message**(*gas: int, to: NewType.<locals>.new_type, sender: NewType.<locals>.new_type, value: int, data: bytes, code: bytes, depth: int = 0, create_address: NewType.<locals>.new_type = None, code_address: NewType.<locals>.new_type = None, should_transfer_value: bool = True, is_static: bool = False*)

A message for VM computation.

Opcode

class eth.vm.opcode.**Opcode**

classmethod as_opcode(*logic_fn: Callable[..., Any], mnemonic: str, gas_cost: int*) → Type[T]

Class factory method for turning vanilla functions into Opcode classes.

VM

BaseVM

class eth.vm.base.**BaseVM**(*header: eth.rlp.headers.BlockHeader, chaindb: eth.db.chain.BaseChainDB*)

classmethod compute_difficulty(*parent_header: eth.rlp.headers.BlockHeader, timestamp: int*) → int

Compute the difficulty for a block header.

Parameters

- **parent_header** – the parent header
- **timestamp** – the timestamp of the child header

configure_header(***header_params*) → eth.rlp.headers.BlockHeader

Setup the current header with the provided parameters. This can be used to set fields like the gas limit or timestamp to value different than their computed defaults.

classmethod create_header_from_parent(*parent_header: eth.rlp.headers.BlockHeader*, ***header_params*) → eth.rlp.headers.BlockHeader

Creates and initializes a new block header from the provided *parent_header*.

static get_block_reward() → int

Return the amount in **wei** that should be given to a miner as a reward for this block.

Note: This is an abstract method that must be implemented in subclasses

classmethod get_nephew_reward() → int

Return the reward which should be given to the miner of the given *nephew*.

Note: This is an abstract method that must be implemented in subclasses

static get_uncle_reward(*block_number: int*, *uncle: eth.rlp.blocks.BaseBlock*) → int

Return the reward which should be given to the miner of the given *uncle*.

Note: This is an abstract method that must be implemented in subclasses

make_receipt(*base_header: eth.rlp.headers.BlockHeader*, *transaction: eth.rlp.transactions.BaseTransaction*, *computation: eth.vm.computation.BaseComputation*, *state: eth.vm.state.BaseState*) → eth.rlp.receipts.Receipt

Generate the receipt resulting from applying the transaction.

Parameters

- **base_header** – the header of the block before the transaction was applied.
- **transaction** – the transaction used to generate the receipt
- **computation** – the result of running the transaction computation
- **state** – the resulting state, after executing the computation

Returns receipt

validate_transaction_against_header(*base_header: eth.rlp.headers.BlockHeader*, *transaction: eth.rlp.transactions.BaseTransaction*) → None

Validate that the given transaction is valid to apply to the given header.

Parameters

- **base_header** – header before applying the transaction
- **transaction** – the transaction to validate

Raises ValidationError if the transaction is not valid to apply

VM

class `eth.vm.base.`**`VM`**(*header: eth.rlp.headers.BlockHeader*, *chaindb: eth.db.chain.BaseChainDB*)

The `BaseVM` class represents the Chain rules for a specific protocol definition such as the Frontier or Homestead network.

> **Note:** Each `BaseVM` class must be configured with:
>
> - `block_class`: The `Block` class for blocks in this VM ruleset.
> - `_state_class`: The `State` class used by this VM for execution.

`apply_all_transactions`(*transactions: Tuple[eth.rlp.transactions.BaseTransaction, ...]*, *base_header: eth.rlp.headers.BlockHeader*) → Tuple[eth.rlp.headers.BlockHeader, Tuple[eth.rlp.receipts.Receipt, ...], Tuple[eth.vm.computation.BaseComputation, ...]]

Determine the results of applying all transactions to the base header. This does *not* update the current block or header of the VM.

Parameters

- **`transactions`** – an iterable of all transactions to apply
- **`base_header`** – the starting header to apply transactions to

Returns the final header, the receipts of each transaction, and the computations

`apply_transaction`(*header: eth.rlp.headers.BlockHeader*, *transaction: eth.rlp.transactions.BaseTransaction*) → Tuple[eth.rlp.headers.BlockHeader, eth.rlp.receipts.Receipt, eth.vm.computation.BaseComputation]

Apply the transaction to the current block. This is a wrapper around `apply_transaction()` with some extra orchestration logic.

Parameters

- **`header`** – header of the block before application
- **`transaction`** – to apply

`create_transaction`(**args*, ***kwargs*) → eth.rlp.transactions.BaseTransaction

Proxy for instantiating a signed transaction for this VM.

`classmethod create_unsigned_transaction`(***, *nonce: int*, *gas_price: int*, *gas: int*, *to: NewType.<locals>.new_type*, *value: int*, *data: bytes*) → eth.rlp.transactions.BaseUnsignedTransaction

Proxy for instantiating an unsigned transaction for this VM.

`execute_bytecode`(*origin: NewType.<locals>.new_type*, *gas_price: int*, *gas: int*, *to: NewType.<locals>.new_type*, *sender: NewType.<locals>.new_type*, *value: int*, *data: bytes*, *code: bytes*, *code_address: NewType.<locals>.new_type = None*) → eth.vm.computation.BaseComputation

Execute raw bytecode in the context of the current state of the virtual machine.

`finalize_block`(*block: eth.rlp.blocks.BaseBlock*) → eth.rlp.blocks.BaseBlock

Perform any finalization steps like awarding the block mining reward.

`classmethod generate_block_from_parent_header_and_coinbase`(*parent_header: eth.rlp.headers.BlockHeader*, *coinbase: NewType.<locals>.new_type*) → eth.rlp.blocks.BaseBlock

Generate block from parent header and coinbase.

classmethod get_block_class() → Type[eth.rlp.blocks.BaseBlock]
Return the `Block` class that this VM uses for blocks.

classmethod get_state_class() → Type[eth.vm.state.BaseState]
Return the class that this VM uses for states.

classmethod get_transaction_class() → Type[eth.rlp.transactions.BaseTransaction]
Return the class that this VM uses for transactions.

import_block(*block: eth.rlp.blocks.BaseBlock*) → eth.rlp.blocks.BaseBlock
Import the given block to the chain.

mine_block(**args*, ***kwargs*) → eth.rlp.blocks.BaseBlock
Mine the current block. Proxies to self.pack_block method.

pack_block(*block: eth.rlp.blocks.BaseBlock*, **args*, ***kwargs*) → eth.rlp.blocks.BaseBlock
Pack block for mining.

Parameters

- **coinbase** (*bytes*) – 20-byte public address to receive block reward
- **uncles_hash** (*bytes*) – 32 bytes
- **state_root** (*bytes*) – 32 bytes
- **transaction_root** (*bytes*) – 32 bytes
- **receipt_root** (*bytes*) – 32 bytes
- **bloom** (*int*) –
- **gas_used** (*int*) –
- **extra_data** (*bytes*) – 32 bytes
- **mix_hash** (*bytes*) – 32 bytes
- **nonce** (*bytes*) – 8 bytes

previous_hashes
Convenience API for accessing the previous 255 block hashes.

validate_block(*block: eth.rlp.blocks.BaseBlock*) → None
Validate the the given block.

classmethod validate_header(*header: eth.rlp.headers.BlockHeader*, *parent_header: eth.rlp.headers.BlockHeader*, *check_seal: bool = True*) → None

Raises eth.exceptions.ValidationError – if the header is not valid

classmethod validate_seal(*header: eth.rlp.headers.BlockHeader*) → None
Validate the seal on the given header.

classmethod validate_uncle(*block: eth.rlp.blocks.BaseBlock*, *uncle: eth.rlp.blocks.BaseBlock*, *uncle_parent: eth.rlp.blocks.BaseBlock*) → None
Validate the given uncle in the context of the given block.

Stack

class eth.vm.stack.**Stack**
VM Stack

dup (*position: int*) → None
Perform a DUP operation on the stack.

pop (*num_items: int*, *type_hint: str*) → Union[int, bytes, Tuple[Union[int, bytes], ...]]
Pop an item off the stack.

Note: This function is optimized for speed over readability.

push (*value: Union[int, bytes]*) → None
Push an item onto the stack.

swap (*position: int*) → None
Perform a SWAP operation on the stack.

State

BaseState

class eth.vm.state.**BaseState** (*db: eth.db.backends.base.BaseDB*, *execution_context: eth.vm.execution_context.ExecutionContext*, *state_root: bytes*)
The base class that encapsulates all of the various moving parts related to the state of the VM during execution. Each `BaseVM` must be configured with a subclass of the `BaseState`.

Note: Each `BaseState` class must be configured with:

- `computation_class`: The `BaseComputation` class for vm execution.
- `transaction_context_class`: The `TransactionContext` class for vm execution.

apply_transaction (*transaction: BaseTransaction*) → Tuple[bytes, BaseComputation]
Apply transaction to the vm state

Parameters **transaction** – the transaction to apply

Returns the new state root, and the computation

block_number
Return the current `block_number` from the current `execution_context`

coinbase
Return the current `coinbase` from the current `execution_context`

commit (*snapshot: Tuple[bytes, Tuple[uuid.UUID, uuid.UUID]]*) → None
Commit the journal to the point where the snapshot was taken. This will merge in any changesets that were recorded *after* the snapshot changeset.

difficulty
Return the current `difficulty` from the current `execution_context`

gas_limit
Return the current `gas_limit` from the current `transaction_context`

classmethod get_account_db_class () → Type[eth.db.account.BaseAccountDB]
Return the `BaseAccountDB` class that the state class uses.

get_ancestor_hash (*block_number: int*) → NewType.<locals>.new_type
Return the hash for the ancestor block with number `block_number`. Return the empty bytestring `b''` if the block number is outside of the range of available block numbers (typically the last 255 blocks).

get_computation(*message: eth.vm.message.Message*, *transaction_context: BaseTransactionContext*) → BaseComputation

Return a computation instance for the given *message* and *transaction_context*

classmethod get_transaction_context_class() → Type[BaseTransactionContext]

Return the `BaseTransactionContext` class that the state class uses.

revert(*snapshot: Tuple[bytes, Tuple[uuid.UUID, uuid.UUID]]*) → None

Revert the VM to the state at the snapshot

snapshot() → Tuple[bytes, Tuple[uuid.UUID, uuid.UUID]]

Perform a full snapshot of the current state.

Snapshots are a combination of the `state_root` at the time of the snapshot and the id of the changeset from the journaled DB.

state_root

Return the current `state_root` from the underlying database

timestamp

Return the current `timestamp` from the current `execution_context`

BaseTransactionExecutor

class `eth.vm.state.`**BaseTransactionExecutor**(*vm_state: eth.vm.state.BaseState*)

BaseTransactionContext

class `eth.vm.transaction_context.`**BaseTransactionContext**(*gas_price: int*, *origin: NewType.<locals>.new_type*)

This immutable object houses information that remains constant for the entire context of the VM execution.

Forks

Frontier

FrontierVM

class `eth.vm.forks.frontier.`**FrontierVM**(*header: eth.rlp.headers.BlockHeader*, *chaindb: eth.db.chain.BaseChainDB*)

block_class

alias of `eth.vm.forks.frontier.blocks.FrontierBlock`

static compute_difficulty(*parent_header: eth.rlp.headers.BlockHeader*, *timestamp: int*) → int

Computes the difficulty for a frontier block based on the parent block.

static get_block_reward() → int

Return the amount in **wei** that should be given to a miner as a reward for this block.

Note: This is an abstract method that must be implemented in subclasses

classmethod get_nephew_reward() → int

Return the reward which should be given to the miner of the given *nephew*.

Note: This is an abstract method that must be implemented in subclasses

static get_uncle_reward(*block_number: int*, *uncle: eth.rlp.blocks.BaseBlock*) → int

Return the reward which should be given to the miner of the given *uncle*.

Note: This is an abstract method that must be implemented in subclasses

FrontierState

class eth.vm.forks.frontier.state.**FrontierState**(*db: eth.db.backends.base.BaseDB*, *execution_context: eth.vm.execution_context.ExecutionContext*, *state_root: bytes*)

account_db_class

alias of `eth.db.account.AccountDB`

computation_class

alias of `eth.vm.forks.frontier.computation.FrontierComputation`

transaction_context_class

alias of `eth.vm.forks.frontier.transaction_context.FrontierTransactionContext`

transaction_executor

alias of `FrontierTransactionExecutor`

FrontierComputation

class eth.vm.forks.frontier.computation.**FrontierComputation**(*state: eth.vm.state.BaseState*, *message: eth.vm.message.Message*, *transaction_context: eth.vm.transaction_context.BaseTransactionC*

A class for all execution computations in the `Frontier` fork. Inherits from `BaseComputation`

apply_create_message() → eth.vm.computation.BaseComputation

Execution of an VM message to create a new contract.

apply_message() → eth.vm.computation.BaseComputation

Execution of an VM message.

Homestead

HomesteadVM

class eth.vm.forks.homestead.**HomesteadVM**(*header: eth.rlp.headers.BlockHeader*, *chaindb: eth.db.chain.BaseChainDB*)

block_class
alias of eth.vm.forks.homestead.blocks.HomesteadBlock

static compute_difficulty(*parent_header: eth.rlp.headers.BlockHeader*, *timestamp: int*) → int
Computes the difficulty for a homestead block based on the parent block.

HomesteadState

class eth.vm.forks.homestead.state.**HomesteadState**(*db: eth.db.backends.base.BaseDB*, *execution_context: eth.vm.execution_context.ExecutionContext*, *state_root: bytes*)

computation_class
alias of *eth.vm.forks.homestead.computation.HomesteadComputation*

HomesteadComputation

class eth.vm.forks.homestead.computation.**HomesteadComputation**(*state: eth.vm.state.BaseState*, *message: eth.vm.message.Message*, *transaction_context: eth.vm.transaction_context.BaseTransactio*

A class for all execution computations in the `Frontier` fork. Inherits from *FrontierComputation*

apply_create_message() → eth.vm.computation.BaseComputation
Execution of an VM message to create a new contract.

TangerineWhistle

TangerineWhistleVM

class eth.vm.forks.tangerine_whistle.**TangerineWhistleVM**(*header: eth.rlp.headers.BlockHeader*, *chaindb: eth.db.chain.BaseChainDB*)

TangerineWhistleState

class eth.vm.forks.tangerine_whistle.state.**TangerineWhistleState**(*db: eth.db.backends.base.BaseDB, execution_context: eth.vm.execution_context.ExecutionCo state_root: bytes*)

computation_class
alias of *eth.vm.forks.tangerine_whistle.computation.TangerineWhistleComputation*

TangerineWhistleComputation

class eth.vm.forks.tangerine_whistle.computation.**TangerineWhistleComputation**(*state: eth.vm.state.BaseSta message: eth.vm.message.Mes transaction_context: eth.vm.transaction_*

A class for all execution computations in the TangerineWhistle fork. Inherits from *HomesteadComputation*

SpuriousDragon

SpuriousDragonVM

class eth.vm.forks.spurious_dragon.**SpuriousDragonVM**(*header: eth.rlp.headers.BlockHeader, chaindb: eth.db.chain.BaseChainDB*)

block_class
alias of eth.vm.forks.spurious_dragon.blocks.SpuriousDragonBlock

SpuriousDragonState

class eth.vm.forks.spurious_dragon.state.**SpuriousDragonState**(*db: eth.db.backends.base.BaseDB, execution_context: eth.vm.execution_context.ExecutionContext, state_root: bytes*)

computation_class
alias of *eth.vm.forks.spurious_dragon.computation.SpuriousDragonComputation*

transaction_executor
alias of SpuriousDragonTransactionExecutor

SpuriousDragonComputation

class eth.vm.forks.spurious_dragon.computation.**SpuriousDragonComputation**(*state: eth.vm.state.BaseState, message: eth.vm.message.Message, transaction_context: eth.vm.transaction_contex*

A class for all execution computations in the SpuriousDragon fork. Inherits from *HomesteadComputation*

apply_create_message() → eth.vm.computation.BaseComputation
Execution of an VM message to create a new contract.

Byzantium

ByzantiumVM

class eth.vm.forks.byzantium.**ByzantiumVM**(*header: eth.rlp.headers.BlockHeader, chaindb: eth.db.chain.BaseChainDB*)

block_class
alias of eth.vm.forks.byzantium.blocks.ByzantiumBlock

compute_difficulty
https://github.com/ethereum/EIPs/issues/100

static get_block_reward() → int
Return the amount in **wei** that should be given to a miner as a reward for this block.

Note: This is an abstract method that must be implemented in subclasses

ByzantiumState

class eth.vm.forks.byzantium.state.**ByzantiumState**(*db: eth.db.backends.base.BaseDB, execution_context: eth.vm.execution_context.ExecutionContext, state_root: bytes*)

computation_class
alias of *eth.vm.forks.byzantium.computation.ByzantiumComputation*

ByzantiumComputation

class eth.vm.forks.byzantium.computation.**ByzantiumComputation**(*state: eth.vm.state.BaseState*, *message: eth.vm.message.Message*, *transaction_context: eth.vm.transaction_context.BaseTransactio*

A class for all execution computations in the `Byzantium` fork. Inherits from *SpuriousDragonComputation*

2.7 Contributing

Thank you for your interest in contributing! We welcome all contributions no matter their size. Please read along to learn how to get started. If you get stuck, feel free to reach for help in our Gitter channel.

2.7.1 Setting the stage

First we need to clone the Py-EVM repository. Py-EVM depends on a submodule of the common tests across all clients, so we need to clone the repo with the `--recursive` flag. Example:

```
$ git clone --recursive https://github.com/ethereum/py-evm.git
```

Optional: Often, the best way to guarantee a clean Python 3 environment is with virtualenv. If we don't have `virtualenv` installed already, we first need to install it via pip.

```
pip install virtualenv
```

Then, we can initialize a new virtual environment `venv`, like:

```
virtualenv -p python3 venv
```

This creates a new directory `venv` where packages are installed isolated from any other global packages.

To activate the virtual directory we have to *source* it

```
. venv/bin/activate
```

After we have activated our virtual environment, installing all dependencies that are needed to run, develop and test all code in this repository is as easy as:

```
pip install -e .[dev]
```

2.7.2 Running the tests

A great way to explore the code base is to run the tests.

We can run all tests with:

```
pytest
```

However, running the entire test suite does take a very long time so often we just want to run a subset instead, like:

```
pytest tests/core/padding-utils/test_padding.py
```

We can also install `tox` to run the full test suite which also covers things like testing the code against different Python versions, linting etc.

It is important to understand that each Pull Request must pass the full test suite as part of the CI check, hence it is often convenient to have `tox` installed locally as well.

2.7.3 Code Style

When multiple people are working on the same body of code, it is important that they write code that conforms to a similar style. It often doesn't matter as much which style, but rather that they conform to one style.

To ensure your contribution conforms to the style being used in this project, we encourage you to read our style guide.

2.7.4 Type Hints

The code bases is transitioning to use type hints. Type hints make it easy to prevent certain types of bugs, enable richer tooling and enhance the documentation, making the code easier to follow.

All new code is required to land with type hints with the exception of test code that is not expected to use type hints.

All parameters as well as the return type of defs are expected to be typed with the exception of `self` and `cls` as seen in the following example.

```
def __init__(self, wrapped_db: BaseDB) -> None:
    self.wrapped_db = wrapped_db
    self.reset()
```

2.7.5 Documentation

Good documentation will lead to quicker adoption and happier users. Please check out our guide on how to create documentation for the Python Ethereum ecosystem.

2.7.6 Pull Requests

It's a good idea to make pull requests early on. A pull request represents the start of a discussion, and doesn't necessarily need to be the final, finished submission.

GitHub's documentation for working on pull requests is available here.

Once you've made a pull request take a look at the Circle CI build status in the GitHub interface and make sure all tests are passing. In general pull requests that do not pass the CI build yet won't get reviewed unless explicitly requested.

2.7.7 Releasing

Pandoc is required for transforming the markdown README to the proper format to render correctly on pypi.

For Debian-like systems:

```
apt install pandoc
```

Or on OSX:

```
brew install pandoc
```

To release a new version:

```
bumpversion $$VERSION_PART_TO_BUMP$$
git push && git push --tags
make release
```

How to bumpversion

The version format for this repo is `{major}.{minor}.{patch}` for stable, and `{major}.{minor}.{patch}-{stage}.{devnum}` for unstable (`stage` can be alpha or beta).

To issue the next version in line, use bumpversion and specify which part to bump, like `bumpversion minor` or `bumpversion devnum`.

If you are in a beta version, `bumpversion stage` will switch to a stable.

To issue an unstable version when the current version is stable, specify the new version explicitly, like `bumpversion --new-version 4.0.0-alpha.1 devnum`

How to release docker images

To create a docker image:

```
make create-docker-image version=<version>
```

By default, this will create a new image with two tags pointing to it:

- `ethereum/trinity:<version>` (explicit version)
- `ethereum/trinity:latest` (latest until overwritten with a future "latest")

Then, push to docker hub:

```
docker push ethereum/trinity:<version>
# the following may be left out if we were pushing a patch for an older version
docker push ethereum/trinity:latest
```

2.8 Code of Conduct

2.8.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

2.8.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language

- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

2.8.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

2.8.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

2.8.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at piper@pipermerriam.com. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

2.8.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at https://www.contributor-covenant.org/version/1/4/code-of-conduct.html

Python Module Index

Index