# py-evm Documentation

*Release 0.1.0-alpha.20*

Ethereum Foundation

Jan 14, 2019

# General

Trinity is a program that connects to the Ethereum network to operate as a full or light node. It is built on top of Py-EVM which is a new implementation of the Ethereum Virtual Machine (EVM) written in Python.

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.

# CHAPTER 1

# 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
- Support "full" and "light" modes
- Is well documented
- Is easy to understand
- Has clear APIs
- Runs fast and resource friendly
- Is highly flexible to support:
  - Public chains (including Mainnet, Ropsten and other networks)
  - Private chains
  - Consortium chains
  - Advanced research

**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 *Quickstart* to get things going.

# CHAPTER 2

# Further reading

Here are a couple more useful links to check out.

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

# CHAPTER 3

# Table of contents

## 3.1 Introduction

Trinity is a program that connects to the Ethereum network to operate as a full or light node. It is built on top of Py-EVM which is a new implementation of the Ethereum Virtual Machine (EVM) written in Python.

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.

### 3.1.1 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
- Support "full" and "light" modes
- Is well documented
- Is easy to understand
- Has clear APIs
- Runs fast and resource friendly
- Is highly flexible to support:
    - Public chains (including Mainnet, Ropsten and other networks)
    - Private chains
    - Consortium chains
    - Advanced research

**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 *Quickstart* to get things going.

### 3.1.2 Further reading

Here are a couple more useful links to check out.

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

## 3.2 Quickstart

### 3.2.1 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
```

### 3.2.2 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://
↪09d34ecb0de1806ab0e68cb2d822b967292dc021df06aab9a55aa4d2e1b2e04ae73560137407a48073286026e12dd60d265
↪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
```

## 3.2.3 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.

## 3.2.4 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(
↪'0x00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪'),
```

(continues on next page)

(continued from previous page)

```
'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

## 3.3 Release notes

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

### 3.3.1 Trinity

#### 0.1.0-alpha.20

Released December 13, 2018

- #1579: Feature: Full Constantinople support, with all* tests passing
- #1590: Performance: CodeStream speedup
- #1576: Bugfix: require recent enough py-ecc to avoid busted py-ecc release (see #1572)
- #1577: Maintenance: Show state diffs on all state failures (see #1573)

- #1570: Maintenance: Cleanup sporadic unclean shutdown of peer request
- #1580: Maintenance: The logged delta in expected vs actual account balance was backwards
- #1573: Maintenance: Display state diffs on failing tests, for much easier EVM debugging
- #1567: Performance: Reduce event bus traffic by enabling point-to-point communication
- #1569: Bugfix: Increase Kademlia timeouts to work on high-latency networks
- #1530: Maintenance: Rename logging level from `trace` (reserved for EVM tracing) to `debug2`
- #1553: Maintenance: Dynamically tune peer timeouts with historical latency (also #1583)
- #1560: Bugfix: Constantinople CREATE2 gas usage
- #1559: Feature: Mainnet configuration now defaults to Constantinople rules at 7080000
- #1557: Docs: Clarify that local plugins must be installed with `-e`
- #1538: Maintenance: Variety of dependency resolution warning cleanups
- #1549: Maintenance: Separate Plugin space for `trinity` and `trinity-beacon`
- #1554: Maintenance: Enable asynchronous iterators that can be cancelled by a service
- #1523: Maintenance: Much faster testing of valid PoW chains
- #1536: Maintenance: Add `trinity-beacon` command as a placeholder for future Beacon Chain
- #1500: Performance: Be smarter about validating the bloom filter, to avoid duplicate hashing
- #1537: Maintenance: Use new event bus feature to avoid the old hack for clean shutdown
- #1544: Docs: Quickstart fix – use `trinity attach` instead of console
- #1541: Docs: Simplify and de-duplicate readme
- #1533: Bugfix: Light chain data lookups regressed during genesis file feature. Fixed
- #1524: Bugfix: Validate header chain continuity during light sync
- #1528: Maintenance: Computation code reorg and gas logging bugfix
- #1522: Bugfix: Increase the system recursion limit for EVM requirements, but never decrease it
- #1519: Docs: Document why we must spawn instead of fork on linux (spoiler: asyncio)
- #1516: Maintenance: Add test for `trinity attach`
- #1299: Feature: Launch via custom genesis file (See EIP proposal)
- #1496: Bugfix: Regular chain sync crash
- The research team has started adding Beacon Chain code to the underlying py-evm repo. It's all a work in progress, but for those who like to follow along:
  - #1508: Rework Eth2.0 Types
  - #1543: Beacon Chain network commands and protocol scaffolding
  - #1521: Rework helper functions - part 1
  - #1552: Beacon Chain protocol class and handshake
  - #1555: Rename data structures and constants
  - #1563: Rework helper functions - part 2
  - #1574: Beacon block request handler

### 0.1.0-alpha.18,19

That sound you make when you burp in the middle of a hiccup. Hiccurp?

### 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.

## 3.4 Guides

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*.

### 3.4.1 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
    INFO  05-29 01:57:02            server  Running server...
    INFO  05-29 01:57:07            server  enode://
 ↪09d34ecb0de1806ab0e68cb2d822b967292dc021df06aab9a55aa4d2e1b2e04ae73560137407a48073286026e12dd60d265
 ↪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
```

### 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

### 3.4.2 Architecture

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

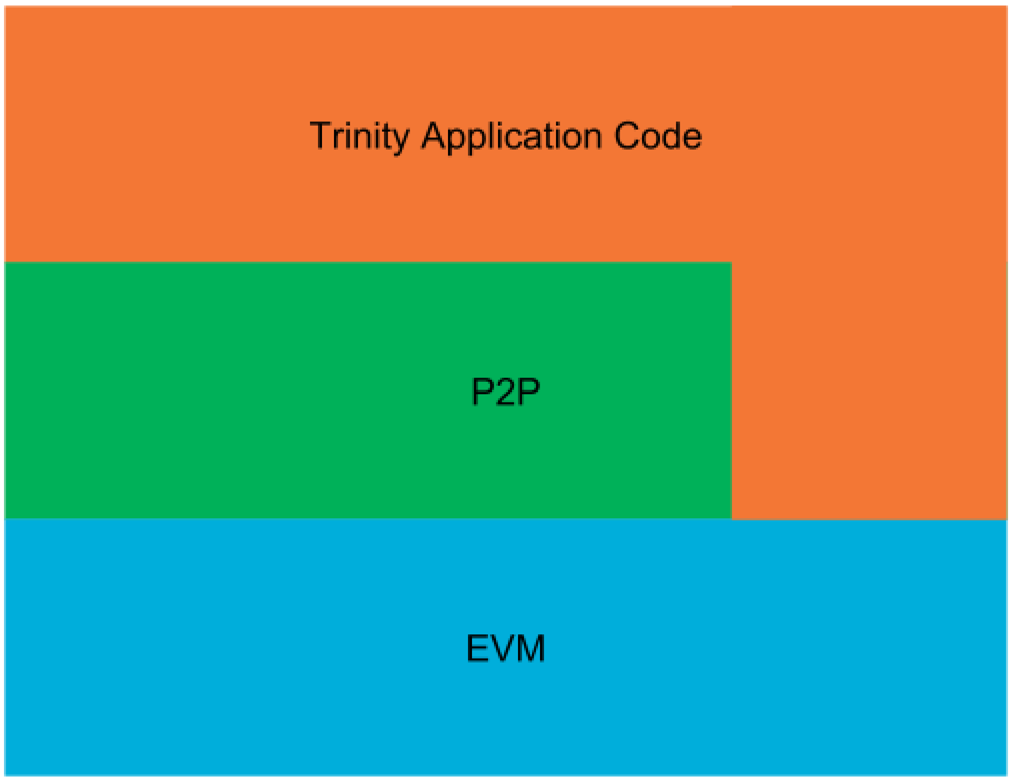
#### Layering

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

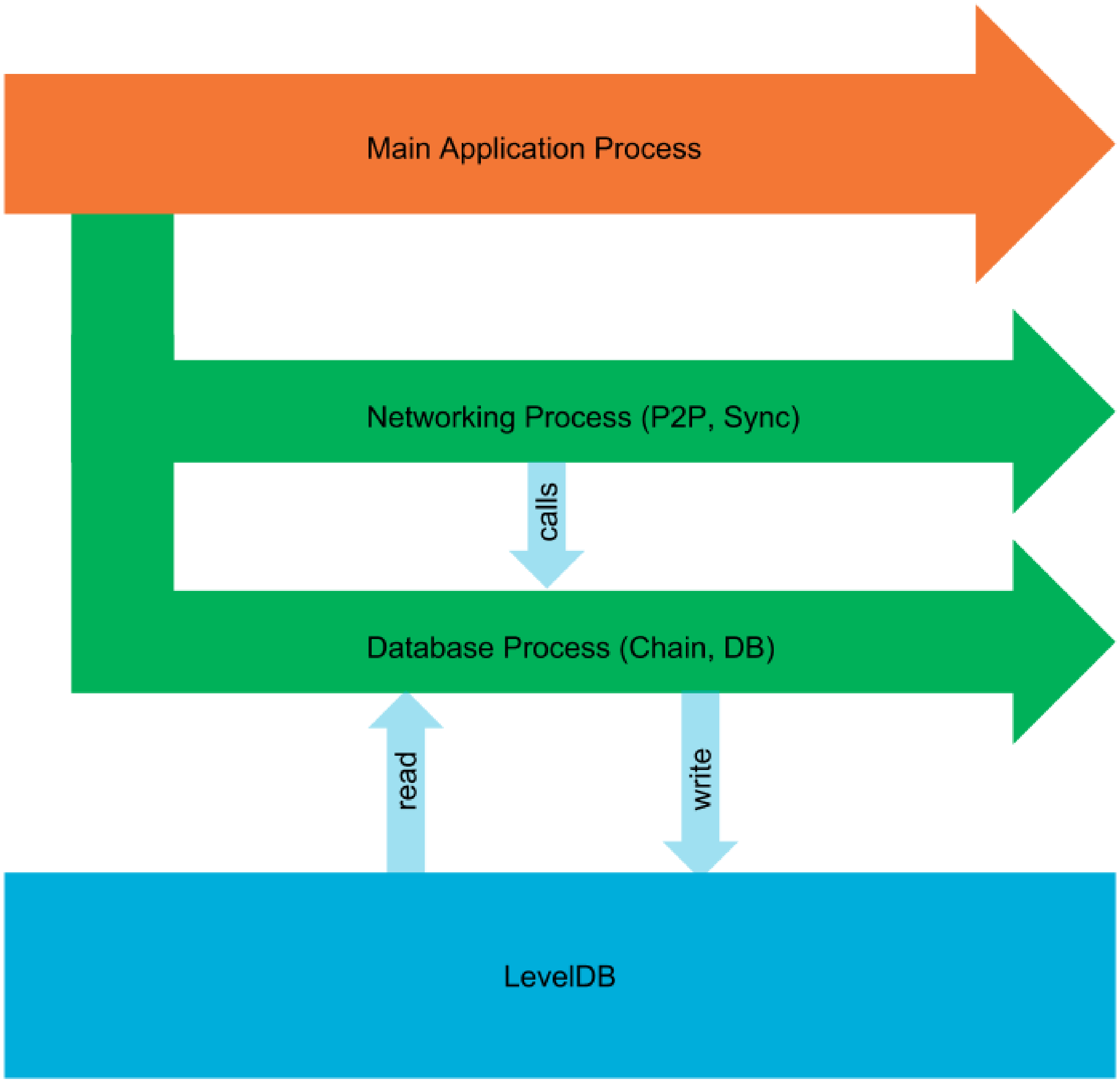
Everything that makes an Ethereum node an Ethereum node and is not part of the EVM or P2P layer is handled by the Trinity application layer itself. That includes a lot of networking, orchestrating the existing building blocks for different modes of operations (e.g. light vs full mode), handling of the 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 utilizing Python's `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 off 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 practice that means that the network process controls the connections to other peers and starts 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 hosts 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.

### 3.4.3 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 as 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 an `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 `BaseIsolatedPlugin`, 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`)

## 3.5 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.

### 3.5.1 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:
```

(continues on next page)

(continued from previous page)

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

### 3.5.2 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.

**normalized_name**
The normalized (computer readable) 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.

### 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.

**process**
Return the `Process` created by the isolated plugin.

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

**stop**() → None
Set the `status` to *STOPPED*' but rely on the *PluginManager* to tear down the process. This allows isolated plugins to be taken down concurrently without depending on a running event loop.

### 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`.

## 3.6 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.

### 3.6.1 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]
```

### 3.6.2 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.

### 3.6.3 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.

### 3.6.4 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()
```

### 3.6.5 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.

### 3.6.6 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.

### 3.6.7 Releasing

#### One time setup

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
```

#### Final test before each release

Before releasing a new version, build and test the package that will be released:

```
git checkout master && git pull

make package

# in another shell, navigate to the virtualenv mentioned in output of ^

# load the virtualenv with the packaged trinity release
source package-smoke-test/bin/activate

# smoke test the release
trinity --ropsten
```

#### Push the release to github & pypi

After confirming that the release package looks okay, release a new version:

```
make release bump=$$VERSION_PART_TO_BUMP$$

# While trinity and py-evm are colocated,
```

(continues on next page)

(continued from previous page)

```
# manually change trinity & py-evm version in setup_trinity.py
make release-trinity
```

### Which version part to bump

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).

During a release, specify which part to bump, like `make release bump=minor` or `make release bump=devnum`.

If you are in a beta version, `make release bump=stage` will switch to a stable.

To issue an unstable version when the current version is stable, specify the new version explicitly, like `make release bump="--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
```

## 3.7 Code of Conduct

### 3.7.1 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.

### 3.7.2 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

### 3.7.3 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.

### 3.7.4 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.

### 3.7.5 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.

### 3.7.6 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

# Index