Plug Documentation

Release 0.0.1

Plug Developers

Mar 12, 2018

CONTENTS:

Here you will find all the information necessary to begin building and deploying your own distributed ledger network with Plug. Start with the *Getting started* manual.

CHAPTER

ONE

GETTING STARTED

This document will walk you through the process of setting up your environment, installing Plug, running a node, publishing a transaction and verifying the result.

1.1 Prerequisites

1.1.1 Python

You will need Python 3.6 or later.

You can verify which version of Python is installed by running the following command:

```
$ python --version
Python 3.6.4
```

If the version number is less than 3.6 (for example, if you see `Python 2.7.14`), then you will first need to install a newer version of Python.

Tip: To prevent conflicts with existing programs on your computer, it is recommended that you use a package manager such as Anaconda, Homebrew or MacPorts (note: the latter two are macOS only).

For example, to install Python 3.6 using MacPorts, use the following commands after installing MacPorts:

```
$ sudo port install python36 python_select
$ sudo port select --set python3 python36
$ sudo port select --set python python36
$ python --version
```

1.1.2 Virtualenv

Virtualenvs help prevent conflicts when working on multiple Python projects, by isolating the Python libraries and tools specific to each project.

Before you install the Plug SDK, it highly recommended that you create and activate a new virtualenv.

To create a new virtualenv, you can use pipenv:

```
$ sudo easy_install pipenv
```

We will use pipenv in the subsequent documentation, as it will create the virtualenv for you automatically.

If you are using a different utility, such as virtualenv or virtualenvwrapper, be sure to create the new virtualenv and activate it before continuing.

1.2 Installing From Source

Note: Items marked *(TBA)* will be updated once the Plug codebase is open-sourced.

To install Plug from source, first install the Plug SDK from *(TBA)*:

```
$ pipenv install -e (TBA)
```

Next, install `plug_demo` in the same environment where you installed `plug`; this package provides some example code that will be used to create a demo Node.

```
$ pipenv install -e (TBA)
```

After installing Plug SDK and Plug Demo you will have access to the `plug` command-line tool. This tool will assist you in creating a new node and running it.

1.3 Running a Node

Next, we will get a Plug node up and running.

To start, copy this config into a new file called `node.yaml`:

```
plug:
    runner:
        refresh_window: PT5S
        voting_model_fqdn: plug.model.VotingPowerModel
        max_block_size: 100
        quorum: 51
        network_id: 1ee06262-a70e-4760-87b0-f4df093cc440

    middlewares: []

    models:
      - plug_demo.balance.BalanceModel

    transforms:
      - plug_demo.balance.BalanceTransfer

    initial_state:
      plug_demo.balance.BalanceModel:
        19jpsGgLeUtQTV4FUided956EmcvDGEB12: {"balance": 100}
        1JE1wQmtJiQtt4eXYWB3xNrbb3ji37GVCH: {"balance": 100}
```

Then run the following commands to initialize and run your node:

```
$ plug new -n1 node.yaml
$ cd nodes/node_0/
$ plug run
```

Tip: The `-n1` argument tells `plug new` to create a 1-node network. Try specifying different values to configure multiple nodes (for example, use `-n3` to create a network with 3 nodes).

1.4 Interacting with Your Node

Once your node is up and running, you can interact with its HTTP API by going to http://localhost:8181/api/v1.

As an example, try posting a new `BalanceTransfer` transaction:

1. Click on **transaction** to expand the section.
2. Click on **POST /api/v1/transaction** to expand the section.
3. Enter the following JSON next to the **transaction** parameter:

```
{
  "fqdn": "plug.consensus.Transaction",
  "transform": {
    "fqdn": "plug_demo.balance.BalanceTransfer",
    "sender": "19jpsGgLeUtQTV4FUided956EmcvDGEB12",
    "receiver": "1JE1wQmtJiQtt4eXYWB3xNrbb3ji37GVCH",
    "amount": 10
  },
  "proofs": {
    "19jpsGgLeUtQTV4FUided956EmcvDGEB12": {
      "address": "19jpsGgLeUtQTV4FUided956EmcvDGEB12",
      "fqdn": "plug.proof.SingleKeyProof",
      "nonce": 0,
      "challenge":
↪"c8551de4804422957653dcc9cb84740cee1a35532b8bd8c6e32ffa977bf9a167",
      "verifying_key": {
        "fqdn": "plug.key.ED25519VerifyingKey",
        "verifying_key":
↪"a9d3b9f401367556cd816a3c307a0c2fd9887ea48e24b0cb4201f0c1ea013497"
      },
      "signature":
↪"10e0b80e230542b54b4beebb83d921eaa2009aa5f3373c3e8a4b2b196835375dada6db3bf1c456a4f07174cad99a6
↪"
    }
  }
}
```

4. Click the **Try it out!** button. Note the `transaction_hash` value in the **Response Body**.

Tip: Check the output from your Plug node after you post your transaction, to see the consensus process in action.

Once the node has received the transaction, we can check to see if it has been confirmed (added to a block):

1. Click on **GET /api/v1/transaction/{transaction_hash}** to expand the section.
2. Enter the transaction hash (from after you posted the transaction above) next to the **transaction_hash** parameter.
3. Click the **Try it out!** button. If the **Response Body** says `"status": "confirmed"`, then your transaction was processed successfully!

Once the transaction is confirmed, we can check the blockchain state to see the results:

1. Click on **state** to expand the section.
2. Click on **GET /api/v1/state/{height}** to expand the section.
3. Enter `-1` next to the **height** parameter.
4. Click the **Try it out!** button!

In the **Response Body**, you should see something that looks like this:

```
{
  "fqdn": "plug.consensus.State",
  "height": 1,
  "models": {
    "plug_demo.balance.BalanceModel": {
      "bin": 100,
      "19jpsGgLeUtQTV4FUided956EmcvDGEB12": {
        "balance": 90
      },
      "1JE1wQmtJiQtt4eXYWB3xNrbb3ji37GVCH": {
        "balance": 110
      }
    }
    ...
```

Your transaction was successfully processed; 10 tokens were transferred from one address to another!

1.5 Where to Go from Here

Now that you're able to run and interact with a Plug node, it's time to dive deeper into the technology and understand how the *Plug blockchain* and *consensus process* work.

CHAPTER

TWO

THE PLUG BLOCKCHAIN

2.1 Terminology

Before discussing the details of the Plugin framework architecture, it's worth defining certain basic terms. Some of these terms are so widely used (and occasionally abused) across the industry that it's worth reviewing these explanations, even if you are already familiar with blockchain technology.

Note: Additional terms are defined in the *Glossary*.

2.1.1 Blockchain

In the strict technical sense, the term *blockchain* refers to an authenticated data structure comprising cryptographically-linked blocks of data.

The blockchain structure enforces "append-only" modification and (intentionally) provides no way to edit existing data stored in the blocks.

It implies an ability to reliably verify the contents of the blocks, but **it does not imply how those blocks are distributed between network participants, nor does it proscribe how network participants can arrive at consensus regarding the contents of the blocks.**

2.1.2 Distributed Ledger Technology

Distributed Ledger Technology (DLT) is a system that allows reliable replication of data between multiple nodes, according to an agreed-upon consensus algorithm.

DLT depends on an authenticated data structure such as a blockchain in order to operate. Without the ability to securely and independently verify the data, the DLT degenerates into simple master-master replication and can no longer guarantee the integrity of data being replicated.

2.1.3 Consensus

A consensus algorithm defines the way nodes of the DLT agree on the contents of the next block.

There are many types of consensus algorithms:

- Consensus may require a certain arbitrary cryptographic challenge to be continuously solved ("Proof of Work").
- It may require participants to provide proof of ownership of special tokens or other virtual assets ("Proof of Stake").

- And many others.

Regardless, the algorithm must be deterministic and independently verifiable, so successful consensus algorithms will likely rely on strong cryptography.

Proof of Work Consensus

Proof of Work is the most often-mentioned mechanism for achieving consensus. Proof of Work requires that a contributor does a deterministically-difficult amount of work that is then easy to check.

Bitcoin, for example, does this by making miners generate hashes until they find one with that starts with a certain number of zeroes ("difficulty"). This artificially slows down block creation and makes it computationally (and thus financially) expensive to participate in the Bitcoin consensus process.

Anyone can mine blocks, but given the current normalized difficulty, it takes a ridiculously long time for non-specialized hardware to mine a valid block.

Note: This process is currently the only known way to reliably implement a public, permissionless consensus where anybody can participate.

Proof of Stake Consensus

Proof of stake operates more like a traditional weighted voting model.

Each participant locks up some value as a promise of their good intentions inside the system ("stake"). At fixed intervals, a voting round occurs, wherein each participant who has locked up some stake gets to vote, with their vote having weight proportional to the amount that they staked.

For example, suppose Alice and Bob each stake 50 units of value into the system. They now have equal voting power; because neither has majority voting power, both Alice and Bob must vote for the same block in order for that block to be selected by consensus.

Note: Anyone can *propose* a block, but only those with stake can *vote*.

This model works very well for permissioned ledgers, where participants have to be given an explicit prior permission to join. In this case, their stake contribution can be verified as part of the permission-granting process.

2.1.4 Forking

Forking occurs when, starting from a certain block, the chain of blocks splits into two or more chains.

Both chains descend from the same parent, but the subsequent blocks are added according to different rules.

Forks may be intentional (for example, to roll out changes to the consensus algorithm) or unintentional (for example, if a partition causes some of the network participants to select a different block from the rest of the network).

Depending on the consensus algorithm, forks may be permanent, temporary, or disallowed entirely.

2.2 Plug Blockchain Details

Similar to most blockchains, the Plug blockchain consists of *Blocks*. Each Block contains *Transactions* that describe changes made since the previous Block was created.

However, the Plug blockchain also contains *States*, which are snapshots of the contents of the blockchain. A new State is created after each Block is appended to the blockchain.

2.2.1 States

Each State object is a key-value mapping comprised of many different *models*. Each model has a fully-qualified domain name (abbreviated "FQDN"), which is a string value that uniquely identifies it.

For example:

```
{
   "plug.balances.models.BalanceModel": {
        "Alice": 100,
        "Carol": 100
    },

    "plug.loans.models.LoanModel": {
        "Bob": [
            {"owner": "Edgar", "balance": 200},
            {"owner": "Glenda", "balance": 300}
        ],

        "Dave": [
            {"owner": "Frank", "balance": 200}
        ]
    }
}
```

In the above example, `plug.balances.models.BalanceModel` and `plug.loans.models.LoanModel` are the FQDNs for two different models.

Inside each model there may be an arbitrary number of key-value pairs. The exact keys and values depend on the business logic of the corresponding models.

Tip: See *ÐApp Basics* for more information models and how they work.

In addition to models, each State contains:

- Height: indicates how many other States occur before it in the blockchain.
- Previous block hash: indicates the hash of the block that created this State (see *Blocks* below).
- Hash: uniquely identifies the State object, created by generating a digest of the State's contents (models, height and previous block index).

  Note: No two States will have the same hash, even if they contain the exact same models. At minimum, each State has a unique height and previous block hash, all of which are used to compute the hash.

The initial State may be referenced as Genesis State, but normally it's called "State 0" (the "0" refers to its height).

Tip: The most recent State in the blockchain is sometimes called the "State of the World", as it contains the most up-to-date information for dapps to use.

This State is normally referred to as "State -1" because it is the last State in the blockchain (in some programming languages, index "-1" references the last item in a collection).

2.2.2 Transactions

Although ĐApps rely on States to track changes to the blockchain's contents, the State objects themselves are immutable.

Instead, changes to the blockchain contents are carried out by creating and broadcasting Transactions.

Each Transaction is made up of two parts:

- A *Transform* which identifies a routine to execute and the runtime parameters (if applicable).
- A collection of Proofs which authorize the Transform on behalf of one or more Addresses.

The logic in the Transform code determine which Addresses (if any) must authorize the Transaction. For more information, see *ĐApp Basics*.

2.2.3 Blocks

At fixed intervals, each node in the network collects unconfirmed Transactions into Blocks to attach to the blockchain.

Important: The order that Transactions appear in the Block is significant, for purposes of resolving conflicts.

Transactions that appear earlier in the Block are understood as having occurred before Transactions that appear later in the Block.

In addition to the list of Transactions, each Block also contains a "previous state hash", which is the hash of the State object that it is modifying (see *States* above).

Because the Block contains a reference to a State, any node can take that State, apply the Transactions from the Block and arrive at a new State. Assuming that all of the nodes in the network use the same logic, all nodes will compute the same State after each Block.

Note: If one or more nodes uses different logic to process the Transactions, a *partition* will occur.

2.2.4 Blockchain Structure

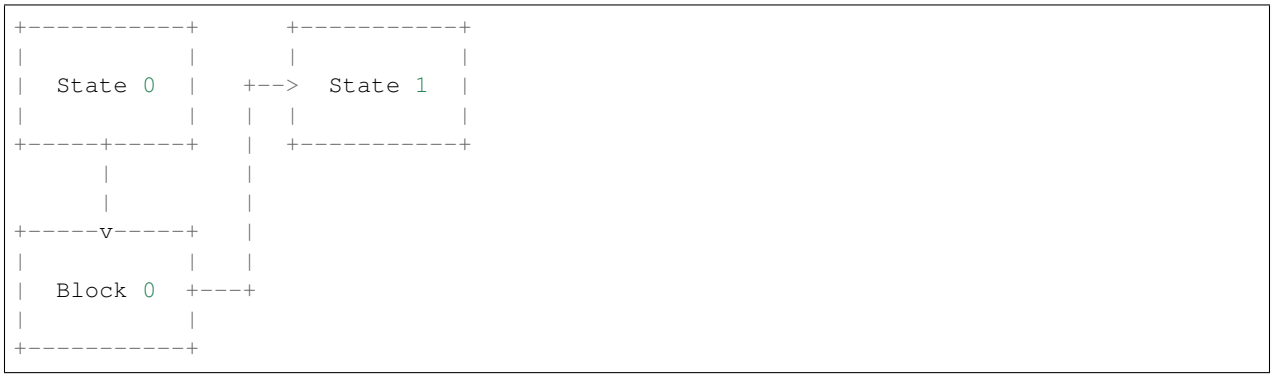
Building from States and Blocks, a chain-like structure begins to emerge.

Because Plug's blockchain has intermediate States, the chain doesn't link from Block to Block. Rather, each Block links to the State that it modifies, and correspondingly, each State links to the Block that created it.

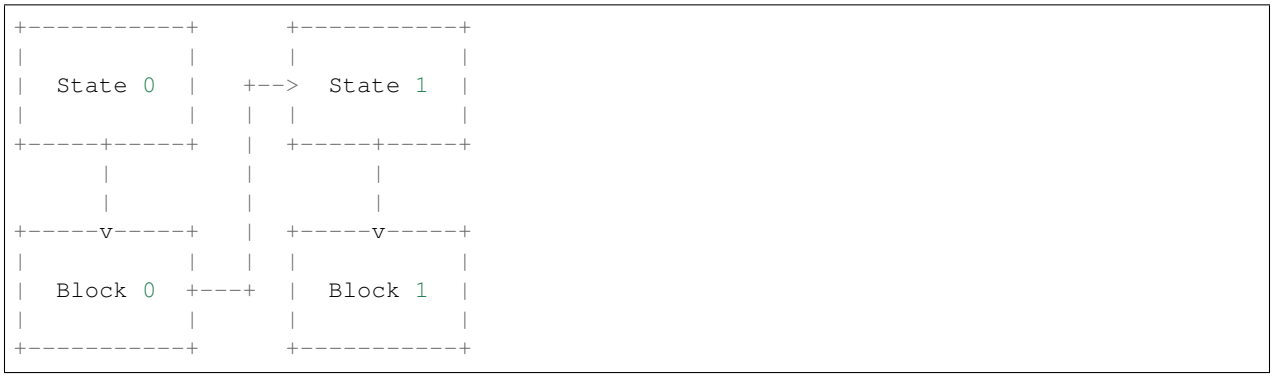
As an example, imagine starting at State 0:

After an interval, several Transactions are collected into Block 0, which builds upon State 0:

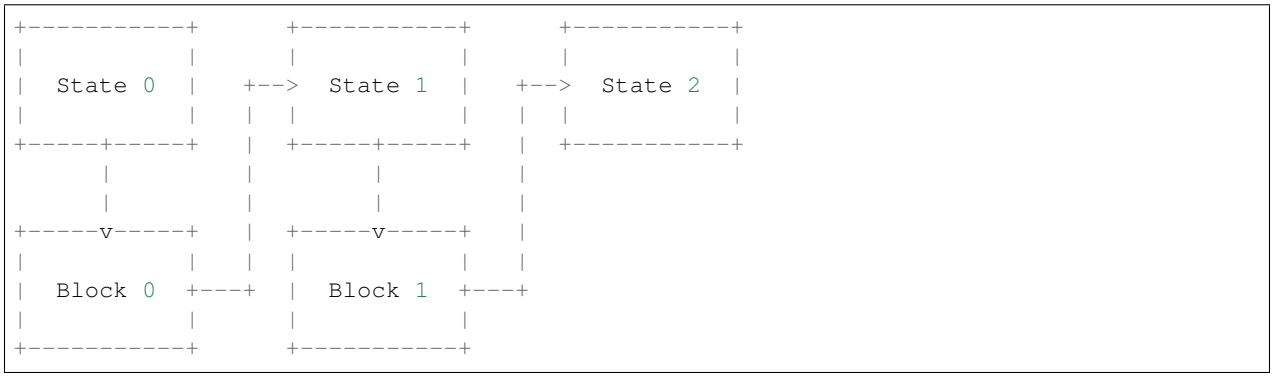
Once the network agrees to make Block 0 the next one in the chain, each node applies the Transactions in this Block to State 0 to produce the next state:



After the next interval, more Transactions are collected into Block 1:



The process continues, and Block 1 will be applied to State 1, forming the next full state:



And so on.

The process will run indefinitely as long as there is a quorum of nodes in the network to agree on Blocks, and new valid Transactions are coming in.

2.2.5 Plug's Consensus Algorithm

Plug uses a leaderless two-phase commit algorithm with variable *Voting Power*.

This means that each and every node in the network has equal power to gather Transactions, form a Block, and vote on the Blocks that make the most sense, according to current Block validation rules.

However, votes may have different weights, according to Voting Power distribution for a given network.

Tip: In practice, this operates similarly to *Proof of Stake*.

Example

Assume a network of three nodes: A, B and C. No blocks exist yet, and the current State of the network is State 0.

Node A receives a valid Transaction from a client through its HTTP gateway. The Transaction is valid, so the node adds it to its unconfirmed Transactions pool.

After an interval (5 seconds by default), the consensus process begins:

1. Node A recognizes a new Transaction in the pool and tries to form a Block.
2. Node A forms a *Block proposal* and sends it out to other nodes in the network.
3. Nodes B and C receive the Block proposal and check that the proposed Block is valid.
4. Nodes B and C confirm that the Block is valid and start voting on the Block. This is the process of *voting* — phase one of the consensus algorithm.
5. Votes are exchanged. If any Block proposal receives a quorum of *Voting Power*, it is considered a *voted Block*. This concludes phase 1.

   Note: In this example, there is only one Block proposal, but in some cases there may be several Block proposals being voted on simultaneously.

   Regardless, each node may only vote for a single Block during the consensus process.

   ---

6. As each node recognizes the voted Block, it votes again, this time to *commit* the Block.
7. Once a voted Block has received quorum Voting Power backing, it is considered committed to be the next Block in the chain. Phase two is now complete.
8. Each node adds this Block to the chain and persists it to permanent storage ("upgrading").

At the start of the next interval, the process repeats.

Important: The consensus algorithm depends on several conditions in order to function correctly:

- There has to be Voting Power available in the system, and each node must have access to the corresponding private key(s). This is discussed in more detail in *Voting and Voting Power* below.

- There has to be a required quorum defined. The current default is 51% (of all Voting Power present in the network).
- There has to be an active network connection between nodes that will allow them to propagate the Blocks and votes.

If any of the above conditions are not met, the network will not be able to reach consensus on new blocks.

Voting and Voting Power

Voting in Plug consensus is strongly tied to Voting Power (often abbreviated as VP).

Voting Power is represented by one or more arbitrary integer values assigned to addresses, stored in the blockchain's *State*.

For example:

```
{
    "plug.model.VotingPowerModel": {
        "19joM8wBG7bAmBypMj23DBmmjGmqfxL4Bj": 100,
        "14RixTSeLtit5GJoDKdEJ23ob74vcvcFvv": 100
    }
}
```

The above example allocates an identical amount of Voting Power each to two different addresses. Note that the actual amount is not so important; what's critical is the *relative weight* of each address, compared to the *total weight* across all addresses.

Tip: `plug.model.VotingPowerModel` is the FQDN for Plug's Voting Power *model*.

To use this Voting Power (i.e. cast a vote) a node must possess the private key corresponding to the address that the Voting Power is assigned to.

Tip: When running the `plug new` command, each node is allocated a private key for one of the Voting Power addresses, and the Voting Power is divided equally among all nodes.

Plug does not currently support reallocating Voting Power, but there is no technical limitation that prevents this. For example, it is theoretically possible to create new voting addresses on-the-fly and assign them Voting Power (i.e., to add more nodes to the network).

It would even be possible to add nodes to a network and give them zero Voting Power. That is, the nodes will be able to participate in the network by propagating unconfirmed Transactions, but they will not be able to vote for Blocks.

2.2.6 Partitions

Partitioning is a situation where some or all of the network's Voting Power is unavailable. This can occur because:

- Network connectivity is impaired, and nodes cannot propagate the votes on new Blocks; or
- The nodes cannot agree on the rules of the network (such as whether a proposed Block is valid).

When a partition occurs, the affected nodes will become out-of-sync with the network, or (in extreme cases) prevent the network from reaching consensus on new Blocks.

Note: A *partition* does not necessarily lead to a *fork*.

In a *fork*, the network splits into two or more separate chains. Each of these chains has a common ancestor, but the subsequent Blocks are different.

Whether a fork occurs on a given side of a partition depends on two factors:

- The amount of Voting Power owned by the nodes on that side of the partition.
- The minimum Voting Power threshold required to commit to a Block (see *Plug's Consensus Algorithm* above).

If there is not enough Voting Power to meet the minimum threshold, then the nodes on that side of the partition will not be able to commit to new Blocks — instead of forking, that part of the network will stall.

Partitions generally require manual intervention to identify and address the root of the problem, whether it is a connectivity issue, consensus disagreement or anything else.

Tip: Plug currently requires a minimum Voting Power threshold of 51%, making it impossible for the network to fork.

In the event of a partition, at most one side of the partition will still have enough Voting Power available to reach consensus on new blocks; the rest of the network will stall until the partition disappears.

2.3 Bootstrapping a Blockchain

In order to bootstrap a network into existence, you need to provide an initial state which defines network participants and their role in the network.

This state includes an initial set of key-value pairs for each model your network implements (in many cases, the initial state for a model is simply an empty object).

For example, a simple genesis state for deploying to a Plug network might look like the following:

```
{
   "plug.balances.models.BalanceModel": {
        "Alice": 100,
        "Carol": 100
    },

    "plug.loans.models.LoanModel": {},

    "plug.model.VotingPowerModel": {
        "1PK2Tx4JjU182gJTzbbnLGT5Hnvuzif9eY": {value: 100},
        "1375GHTkghEey5oVMPSg43jZrqf54dViDG": {value: 100},
        "12gU6PEebSqsdWqyD96NkMVPQAALK379wj": {value: 100},
        "1FLtKabezvkSXDo3R5as5D51G7rZDFd3VM": {value: 100},
        "18CpfWbCB6MYSMQcc6bHC1kpTxMUYcohAT": {value: 100}
    }
}
```

Note: The initial state for the `plug.model.VotingPowerModel` model is particularly important, as it defines which nodes (or more accurately, which signing keys) are permitted to participate in the *consensus process*.

CHAPTER

THREE

RUNNING PLUG NODES

3.1 Configuration

To run a Plug node, you will first need a `config.yaml` configuration file.

Here is a sample `config.yaml`:

```
# Required
plug:
    runner:
        refresh_window: PT5S
        voting_model_fqdn: plug.model.VotingPowerModel
        max_block_size: 100
        quorum: 51
        network_id: 1ee06262-a70e-4760-87b0-f4df093cc440

    middlewares: []

    models:
      - plug_demo.balance.BalanceModel

    transforms:
      - plug_demo.balance.BalanceTransfer

    # Optional
    initial_state:
      plug_demo.balance.BalanceModel:
        root: 100
```

`plug.runner.refresh_window` (required) Specifies the number of seconds between new blocks.

`plug.runner.voting_model_fqdn` (required) Specifies the FQDN of the consensus algorithm to use.

`plug.runner.max_block_size` (required) Specifies the maximum number of transactions allowed in a single block.

`plug.runner.quorum` (required) Specifies the minimum % of voting power needed to confirm a block.

`plug.runner.network_id` (required) A unique alphanumeric identifier for the blockchain.

> **Tip:** v4 UUIDs work well for this.

`plug.middlewares` (required) List of middleware classes that the node will use.

`plug.models` (required) Whitelist of models that the node will use.

> **Tip:** Core models (e.g., `plug.model.VotingPowerModel`) do not need to be included here.

`plug.transforms` (required) Whitelist of transforms that the node will use.

`plug.initial_state` (optional) Used to set the contents of each model in the genesis state.

> **Tip:** You may omit any models that start out empty (without any key-value pairs).

3.2 Creating the Node Metadata

Once you have a `config.yaml` file prepared, you can then generate the metadata files for each node.

Execute the following command:

```
plug new -n3 config.yaml
```

The above command will generate the configuration files to run a network with 3 nodes; if you want to change the number of nodes, specify a different value for the `-n` argument.

Tip: If you get an error "command not found: plug", make sure that the Plug SDK is installed, and the corresponding virtualenv is active.

The `plug new` command will create a directory named `nodes`. Inside, a separate directory is created for each node in the network:

```
> plug new -n3 config.yaml
> ls -F nodes
node_0/  node_1/  node_2/
```

Tip: By default, `plug new` assumes that every node will run on localhost. If you want to configure nodes to run on different hosts, specify the hostnames and/or IP addresses using the `--host` argument. For example:

```
plug new -n3 --host 10.0.1.10 --host 10.0.1.11 --host plug.example.com config.yaml
```

There should be one `--host` per node (i.e., the number of `--host` arguments should match the value of the `-n` argument).

3.3 Starting Each Node

For each node, start a new terminal session. Switch to one of the node directories, and then execute the `plug run` command.

Tip: If you specified hostnames to `plug new` (via the `--host` argument), each directory will be specific to the corresponding host.

Using the example from the previous section, the node metadata directories would be allocated like this:

- `node_0` => `--host 10.0.1.10`
- `node_1` => `--host 10.0.1.11`
- `node_2` => `--host plug.example.com`

Copy each directory to the corresponding host and execute `plug run` on that host.

3.4 Stopping the Node

To stop a Plug node, press Control-C or send a SIGINT to the process.

The node will enter a "warm shutdown" phase, where it attempts to finish the current round of consensus, persist all values to storage, etc.

To shut down the node immediately ("cold shutdown"), press Control-C again or send a second SIGINT to the process.

> **Danger:** Cold shutdown may cause data corruption!

CHAPTER

FOUR

WRITING PLUG ĐAPPS

4.1 ĐApp Basics

Plug provides a powerful framework for building decentralized applications (or "ĐApps" for short). A ĐApp can be thought of as a unit of business logic that runs on the Plug blockchain, intended to satisfy a specific real-life use case.

ĐApps written for Plug are not standalone applications, nor is their code stored on the blockchain like traditional smart contracts.

Rather, Plug ĐApps are Python modules that are installed onto each of the nodes that participate in a Plug network.

Danger: **NEVER** install unknown or untrusted ĐApps onto Plug nodes!

ĐApps can be mixed together, extended and reused just like any other software library. Plug even provides a number of generic ĐApps that you can leverage as building blocks for your custom applications.

4.1.1 Anatomy of a ĐApp

Each ĐApp is comprised of two types of components:

- *Models* define domain-specific data types/structures.
- *Transforms* are the controllers that execute the business logic of the ĐApp.

A single ĐApp may contain any number and combination of the above.

Models

Similarly to models in an ORM-powered application, ĐApp models provide an object-oriented way to model data that your ĐApps store in the *blockchain state*.

To create a new model, write a class that extends *`plug.abstract.Model`* and define a unique `fqdn`.

Your model's FQDN must be universally unique, as this value will identify instances of your model on any Plug blockchain, regardless of what other ĐApps are also installed.

Here is an example of a (minimal) model:

```
from plug.abstract import Model

class MyModel(Model):
    fqdn = "com.my-company.MyModel"
```

An empty model, however, isn't very interesting (let alone useful), so you will likely want to add a `default_factory` which is used to initialize a new instance of the model.

Here is an example of a model that describes a voucher redeemable at a partner establishment:

```
class VoucherModel(Model):
    fqdn = "com.my-company.VoucherModel"

    @staticmethod
    def default_factory(issuer=None, recipient=None, value=0):
        return {
            "issuer": issuer,
            "recipient": recipient,
            "value": value,
        }
```

Tip: For an alternative way to define model structure, see *The Plug ORM*.

Schemas

A model may optionally define a schema, which specifies validations and/or transformations that will be applied to newly-created instances of that model.

For example, a schema can be used to ensure that a specified attribute always has a string value, or it could be used to enforce required attributes, etc.

To define a schema for a model, set its `schema` attribute:

```
class VoucherModel(Model):
    fqdn = "com.my-company.VoucherModel"

    schema = {
        "issuer": {
            "type": "string",
            "required": True,
        },

        "recipient": {
            "type": "string",
            "required": True,
        },

        "value": {
            "type": "int",
            "required": True,
        },
    }

    @staticmethod
    def default_factory(issuer=None, recipient=None, value=0):
        return {
            "issuer":       issuer,
            "recipient":    recipient,
            "value":        value,
        }
```

Tip: Plug uses Cerberus to enforce schemas.

Refer to Cerberus Validation Rules for a list of valid schema directives.

Transforms

Transforms execute the business logic of your ĐApp. They can be thought of similarly to controllers in an MVC framework.

To write a Transform, extend the *plug.abstract.Transform* class and define a unique FQDN.

Just like for models, each Transform's FQDN must be universally unique.

The Transform's functionality is divided (primarily) into two methods:

- `verify()` checks the inputs for a transaction.
- `apply()` carries out the business logic, making changes to the State.

Important: If `verify()` raises an exception, it will prevent `apply()` from being invoked. This is the mechanism by which your transforms can reject invalid transactions.

Along with the input values from the transaction, `verify()` also receives a copy of the blockchain state (known as a state slice), which it can use to verify whether this Transform will be applicable either now or sometime in the future.

`apply()` also receives a state slice. Unlike `verify()` however, any changes that `apply()` makes to the state slice will be persisted to the blockchain state once the corresponding Block is committed.

Tip: Depending on the type of validation, you may opt to put some of the validation logic in `apply()` instead of `validate()`:

When `validate()` raises an exception, the corresponding Transaction will be dropped from the unconfirmed Transaction pool.

If `apply()` raises an exception, however, the Transaction will be kept in the unconfirmed Transaction poll so that the node can try to include it in future blocks.

Here is an example transform used for balance transfers:

```
import typing

class BalanceModel(Model):
    fqdn = "com.my-company.model.Balance"

    @staticmethod
    def default_factory():
        return {
            "balance": 0,
        }


class BalanceTransferTransform(Transform):
    fqdn = "com.my-company.transform.BalanceTransfer"
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, sender: str, receiver: str, amount: int) -> None:
        """
        Initialize the transform using data from the transaction.
        """
        self.sender = sender
        self.receiver = receiver
        self.amount = amount

    def required_authorizations(self) -> typing.Set[str]:
        """
        Indicate which addresses must have valid proofs attached to
        the transaction.
        """
        return {
            self.sender,
        }

    def required_keys(self) -> typing.Set[str]:
        """
        Define the model fields that must be included in the
        transaction.
        """
        return {
            self.sender,
            self.receiver,
        }

    @staticmethod
    def required_models() -> typing.Set[str]:
        """
        Define the models that must be included in the transaction.
        """
        return {
            BalanceModel.fqdn,
        }

    def verify(self, state: typing.Mapping) -> None:
        """
        Verifies the transaction against the current blockchain
        state.
        """
        balances = state[BalanceModel.fqdn]

        if self.amount <= 0:
            raise Exception(_("Cannot send 0 or less"))

        if balances[self.sender]["balance"] < self.amount:
            raise Exception(_("Not enough money"))

    def apply(self, state: typing.MutableMapping) -> None:
        """
        Updates the blockchain state from a valid transaction.
        """
        balances = state[BalanceModel.fqdn]
        balances[self.sender]["balance"] -= self.amount
        balances[self.receiver]["balance"] += self.amount
```

4.1.2 Activating ĐApps

In order to activate your ĐApp, you must register your models and transforms in your node's YAML configuration.

See *Running Plug Nodes* for more information.

4.2 The Plug ORM

For Python developers that are used to SQLAlchemy, Django, and other frameworks that provide ORM functionality, Plug also ships with an ORM library that you can leverage in your ĐApp, providing an object-oriented interface for your ĐApp's models.

In the Plug ORM, every model actually has 2 separate classes:

- A `ModelInstance` class which defines the attributes and logic for each instance of the model (similar in concept to a Django model).
- A `Model` class with defines the FQDN and provides methods for creating and loading model instances from the blockchain state (similar in concept to a Django manager).

4.2.1 ModelInstances

To define a new model using the Plug ORM, start by creating a `ModelInstance` class:

```
from plug.orm import ModelInstance

class Account(ModelInstance):
```

Next, define the fields that each instance will have. This looks very similar to defining the fields for a Django model, but with a few subtle differences.

The following field types are available:

plug.orm.fields.Unspecified The default field type. The value is stored as-is in the blockchain state.

If desired, you may provide a default value for the field.

For example, to add `nickname` and `balance` fields to the `Account` ModelInstance:

```
from plug.orm import fields, ModelInstance

class Account(ModelInstance):
    nickname = fields.Unspecified()  # type: str
    balance = fields.Unspecified(default=0)  # type: int
```

plug.orm.fields.NamedTuple The value is an instance of a `namedtuple`. The field will ensure that the value has the proper type when it is retrieved from the blockchain state.

You must provide the `namedtuple` type for the field. E.g.:

```
from collections import namedtuple
from plug.orm import fields, ModelInstance

Color = namedtuple("Color", ("red", "green", "blue", "alpha"))

class Account(ModelInstance):
    color = fields.NamedTuple(Color)  # type: Color
```

plug.orm.fields.Collection Defines a field that contains a collection (e.g., list) of values with uniform type.

You must provide a field type that describes the value in the collection.

For example, to add a collection of `namedtuple` values to the `Account` ModelInstance:

```
import typing
from collections import namedtuple
from plug.orm import fields, ModelInstance

Currency = namedtuple("Currency", ("name", "symbol"))

class Account(ModelInstance):
    currencies = fields.Collection(
        fields.NamedTuple(Currency)
    )  # type: typing.List[Currency]
```

In the above example, the `Account.currencies` field will contain a list of `Currency` objects.

Tip: You can nest collections (e.g., for a list-of-lists structure).

If desired, you also may specify a custom Python type for the collection.

E.g., to use a bytearray instead of a list:

```
class Account(ModelInstance):
    public_key = fields.Collection(python_type=bytearray)  # type: bytearray
```

Important: Only `MutableSequence` Python types are compatible with this field.

In particular, note that tuples are not mutable, so they cannot be used with Collection fields.

---

plug.orm.fields.Set Effectively the same as *plug.orm.fields.Collection*, except that it holds a set of values:

```
import typing
from collections import namedtuple
from plug.orm import fields, ModelInstance

Currency = namedtuple("Currency", ("name", "symbol"))

class Account(ModelInstance):
    currencies = fields.Set(
        fields.NamedTuple(Currency)
    )  # type: typing.Set[Currency]
```

As with *plug.orm.fields.Collection*, you may specify a custom Python type.

For example to use an ordered set:

```
from ordered_set import OrderedSet
from plug.orm import fields, ModelInstance

Currency = namedtuple("Currency", ("name", "symbol"))
```

(continues on next page)

(continued from previous page)

```
class Account(ModelInstance):
    currencies = fields.Set(
        field_type=fields.NamedTuple(Currency),
        python_type=OrderedSet,
    )  # type: OrderedSet
```

Important: Only `MutableSet` Python types are compatible with this field.

plug.orm.fields.Mapping Effectively the same as *plug.orm.fields.Collection*, except that it holds a mapping (e.g., dict) of values:

```
import typing
from collections import namedtuple
from plug.orm import fields, ModelInstance

Color = namedtuple("Color", ("red", "green", "blue", "alpha"))

class Account(ModelInstance):
    aliases = fields.Mapping(
        fields.NamedTuple(Color)
    )  # type: typing.Dict[str, Color]
```

Tip: You can nest mappings (e.g., for a mapping-of-lists structure).

As with *plug.orm.fields.Collection*, you may specify a custom Python type.

For example, to use an `OrderedDict`:

```
from collections import namedtuple, OrderedDict
from plug.orm import fields, ModelInstance

Color = namedtuple("Color", ("red", "green", "blue", "alpha"))

class Account(ModelInstance):
    aliases = fields.Mapping(
        field_type=fields.NamedTuple(Color),
        python_type=OrderedDict,
    )  # type: OrderedDict
```

Important: Only `MutableMapping` Python types are compatible with this field.

Primary Keys

Every Model also has a `pk` field. This field contains the unique key for the record in the blockchain state (often a Plug address).

Working with Models

Model fields are interactive in exactly the same way as regular attributes:

```
>>> from plug.orm import fields, Model, ModelInstance

>>> class Account(ModelInstance):
...     nickname = fields.Unspecified()  # type: str
...     balance = fields.Unspecified(default=0)  # type: int

>>> alice = Account()
>>> alice.nickname = "Mad Monies"
>>> alice.balance = 10000000000

>>> print(f"Alice has {alice.balance} in her {alice.nickname} account.")
Alice has 10000000000 in her Mad Monies account.
```

When initializing a ModelInstance, you may also provide field values to the initializer:

```
>>> bob = Account(nickname="Nest Egg")
>>> print(f"Bob's {bob.nickname} has a balance of {bob.balance}.")
Bob's Nest Egg has a balance of 0.
```

Business Logic

In addition to fields, ModelInstances may contain any additional attributes and methods, depending on the needs of your application:

```
from plug.orm import fields, ModelInstance

class Account(ModelInstance):
    nickname = fields.Unspecified()  # type: str
    balance = fields.Unspecified(default=0)  # type: int

    def check_balance(self, target: int) -> bool:
        """
        Returns whether the account balance is >= the specified
        amount.
        """
        return target <= self.balance
```

4.2.2 Models

Once you've defined the structure and behavior of a ModelInstance, you will need to create the corresponding Model class.

As with regular Plug models, ORM Models hold the FQDN. However, a little bit of extra code is required, to mate the Model with its ModelInstance:

```
from plug.orm import fields, Model, ModelInstance

class Account(ModelInstance):
    nickname = fields.Unspecified()  # type: str
    balance = fields.Unspecified(default=0)  # type: int

class AccountModel(Model[Account]):
    fqdn = "com.my-company.models.Account"
    instance_type = Account
```

Some important things to note from the above example:

- The base class for `AccountModel` is *`plug.orm.Model`*, *not* *`plug.abstract.Model`*!
- *`plug.orm.Model`* is a Generic, so you can type-hint the class with the corresponding ModelInstance. This is optional but highly recommended, as it will make IDE features such as auto-completion and inspections work better.
- In addition to the `fqdn`, the Model must also define an `instance_type` attribute, so that the Plug ORM can link the two classes together.

Query Methods

ORM Models come with a number of features that allow you to manipulate the ModelInstances stored in the blockchain state:

`plug.orm.Model.create(pk, **kwargs)` Create a new ModelInstance with the specified PK and field values.

If a ModelInstance already exists with that PK, an `IntegrityError` will be raised.

```
all_accounts = AccountModel(state_slice)

try:
    new_account = all_accounts.create(
        pk="13AyivXK7hYaBeLVRShrZDwpUQ7nFeWaZg",
        nickname="The Answer",
        balance=42,
    )
except AccountModel.IntegrityError:
    print("Your account already exists!")
else:
    assert new_account.pk in all_accounts
```

`plug.orm.Model.get_if_exists(pk)` Retrieve the ModelInstance with the specified PK, if it exists.

If there is no ModelInstance with that PK, then the method returns `None`.

```
all_accounts = AccountModel(state_slice)

the_account =\
    all_accounts.get_if_exists("13AyivXK7hYaBeLVRShrZDwpUQ7nFeWaZg")

assert (the_account is None) or isinstance(the_account, Account)
```

`plug.orm.Model.get_or_create(pk)` Retrieve the ModelInstance with the specified PK, if it exists.

If there is no ModelInstance with that PK, a new empty ModelInstance will be created and returned.

```
all_accounts = AccountModel(state_slice)

the_account =\
    all_accounts.get_or_create("13AyivXK7hYaBeLVRShrZDwpUQ7nFeWaZg")

assert the_account.pk in all_accounts
```

`plug.orm.Model.require(pk)` Retrieve the ModelInstance with the specified PK.

If there is no ModelInstance with that PK, a `DoesNotExist` exception will be raised.

```
all_accounts = AccountModel(state_slice)

try:
    the_account =\
        all_accounts.require("13AyivXK7hYaBeLVRShrZDwpUQ7nFeWaZg")
except AccountModel.DoesNotExist:
    print("We couldn't find your account!")
```

plug.orm.Model.update(pk, **kwargs) Updates the field values of the ModelInstance with the specified PK.

The modified ModelInstance is returned.

If there is no ModelInstance with that PK, a `DoesNotExist` exception will be raised.

```
all_accounts = AccountModel(state_slice)

try:
    the_account =\
        all_accounts.update(
            pk="13AyivXK7hYaBeLVRShrZDwpUQ7nFeWaZg",
            balance=1000,
        )
except AccountModel.DoesNotExist:
    print("We couldn't find your account!")
else:
    assert the_account.balance == 1000
```

plug.orm.Model.upsert(pk, **kwargs) Updates the field values of the ModelInstance with the specified PK, if it exists.

If there is no ModelInstance with that PK, a new empty one will be created first.

The modified ModelInstance is returned.

```
all_accounts = AccountModel(state_slice)

the_account =\
    all_accounts.upsert(
        pk="13AyivXK7hYaBeLVRShrZDwpUQ7nFeWaZg",
        balance=1000,
    )

assert the_account.pk in all_accounts
assert the_account.balance == 1000
```

Caution: If the ModelInstance already exists, then only the fields that you provide to `upsert()` will be modified.

Be sure to use explicit values, even if you want to set a field to its default value.

```
account_address = "13AyivXK7hYaBeLVRShrZDwpUQ7nFeWaZg"
all_accounts = AccountModel(state_slice)

# The Account record already exists.
all_accounts.create(pk=account_address, balance=1000)

# ``balance`` has a default value of 0...
the_account = all_accounts.upsert(pk=account_address)

# ... but it wasn't provided to ``upsert()``, so the value
# of the record was not modified.
assert the_account.balance == 1000
```

Custom Query Methods

You may also implement any custom query methods for your Model, depending on the needs of your ĐApp:

```
import typing
from plug.orm import Model

class AccountModel(Model[Account]):
    fqdn = "com.my-company.models.Account"
    instance_type = Account

    def with_minimum_balance(self, target: int) -> typing.Generator[Account, None,␣
↪None]:
        """
        Iterate over accounts with balances >= the specified amount.
        """
        for account in self.values():
            if account.check_balance(target):
                yield account
```

4.2.3 Saving Changes

When interacting with ModelInstances outside of query methods, changes are not applied to the state slice automatically:

```
account_address = "13AyivXK7hYaBeLVRShrZDwpUQ7nFeWaZg"
all_accounts = AccountModel(state_slice)

the_account = all_accounts.create(pk=account_address, balance=1000)
the_account.balance = 900

# The account stored in the state slice is unchanged.
assert all_accounts.require(account_address).balance == 1000
```

Instead, you must invoke the ModelInstance's `save()` method:

```
account_address = "13AyivXK7hYaBeLVRShrZDwpUQ7nFeWaZg"
all_accounts = AccountModel(state_slice)

the_account = all_accounts.create(pk=account_address, balance=1000)
the_account.balance = 900

# Save changes to the state slice.
the_account.save()

# The account stored in the state slice is now updated.
assert all_accounts.require(account_address).balance == 900
```

Important: If you create a ModelInstance without using a query method, then you must provide a Model to the `save()` method:

```
account_address = "13AyivXK7hYaBeLVRShrZDwpUQ7nFeWaZg"
all_accounts = AccountModel(state_slice)

# This works as expected; the ModelInstance knows which Model
# created it.
linked_account = all_accounts.require(pk=account_address)
linked_account.balance = 900
linked_account.save()

# This will not work because the ModelInstance doesn't have a reference
# to the Model!
unlinked_account = Account(pk=account_address, balance=1000)
unlinked_account.balance = 900
unlinked_account.save()

# To save the unlinked ModelInstance, provide the Model to
# ``save()``.
unlinked_account.save(using=all_accounts)
```

4.2.4 Activating Models

Don't forget to register your Models in your node's YAML configuration file!

See *Running Plug Nodes* for more information.

Note: You only have to register your Models. ModelInstances do not need to be registered.

CHAPTER

FIVE

GLOSSARY

5.1 Address

A truncated, base-58-encoded hash plus a checksum that represents a lookup key for state on a given blockchain. Often, an Address serves as a reference to an identity (person, device or other autonomous actor) on a blockchain network.

Each address is derived from a private key, and thus it can be used to verify the correctness of a Proof (for example, used to authorize a Transaction).

An Addresses can represent a single public key, a static group of public keys, or a dynamic membership group assembled under a Namespace.

5.2 Block

A list of ordered and valid Transactions, built off a base State.

When applied to a State, the Transactions are applied in order, returning an updated State reflecting the changes detailed in the individual Transactions.

See *Blocks* for more information.

5.3 Blockchain

A cryptographically-verifiable data structure, maintained by a distributed set of Validators.

5.4 Cross-Chain

A Blockchain that utilizes information or functionality maintained in a separate Blockchain network, for example, using SPV proofs or Multi-Signature Addresses.

5.5 Cryptocurrency

A censorship-resistant, usually pseudonymous, form of virtual cash that uses a Blockchain to maintain accounts and process transactions.

5.6 Commit

The second half of the two-phase commit consensus process. Occurs when when the current Voting Window has selected a winning Proposal from a group of candidate Proposals.

See *Plug's Consensus Algorithm* for more information.

5.7 ĐApp

Short for *Decentralized Application*. Also known as a *Smart Contract*.

A collection of Models and Transforms that define the business logic for an application deployed to Nodes in a Plug network.

See *ĐApp Basics* for more information.

5.8 ED25519

A digital signature scheme using a variant of Schnorr signatures, based on Twisted Edwards curves. It is designed to be faster than existing digital signature schemes without sacrificing security.

5.9 Genesis State

The initial starting point for a blockchain network.

See *States* for more information.

5.10 Height

When used in reference to Blocks or States, this number represents the unique count of these items that exist in the full, un-pruned blockchain history in a given network.

For example, a blockchain with a Block Height of 1 means there is only one Block in existence since the Genesis State.

See *States* for more information.

5.11 Interoperability

The ability to transport value, information, or identity from one blockchain network to another, for example, using SPV proofs and/or Multi-Signature Transactions.

Interoperability may take many forms:

- The ability to trigger State changes in one Network as a result of events that occur in another Network.
- The ability to have one Network act as an autonomous user of a second Network.
- The ability to process a set of Transactions atomically across multiple Networks.

5.12 MerkleMap

An ordered dictionary that maintains a Merkle Tree of the items it holds.

5.13 Multi-Key Addresses

An Address that is controlled by 2 or more Public Keys.

There are two types of Multi-Key Addresses:

- Static Multi-Key Addresses are made up of a fixed set of participants.
- Dynamic Multi-Key Addresses, or Namespaces, can add or remove participants over time.

5.14 Network

A group of Nodes collectively maintaining a Blockchain according to a predefined set of rules for verifying data and authorizing updates.

5.15 Node

A participant in a Network, running the Plug software.

Note: Node is a logical distinction, not necessarily a physical one.

For example, if a single computer is running 3 instances of the Plug software, then there are 3 Nodes running on that computer.

5.16 Nonce

An integer that demarcates an ordering of Transactions authorized by a given address. Nonces must be used in order.

Nonces are stored and incremented globally in the Blockchain State.

5.17 Proof

A combination of Signatures, Public Keys, and Metadata that signify that the entity controlling an Address (or entities, in the case of Multi-Key Addresses) has authorized the action detailed in the corresponding Transaction.

5.18 Proposal

A Block published by a Node as a candidate for the next Block in the Blockchain, during the Two-Phase Commit process.

5.19 Signature

A sequence of bytes (typically encoded using hexadecimal notation) generated using a Private Key. The Signature is used to confirm the authenticity or authority of a message (usually a Transaction).

5.20 State

A collection of key-value pairs that represent all of the data stored in the Blockchain at a particular Height.

The initial State in a Blockchain is often called the "Genesis State", and the latest State is often called the "Head State".

See *States* for more information.

5.21 Transaction

A structured set of data that represents an action being authorized on behalf of a network participant.

See *Transactions* for more information.

5.22 Transform

A functor that can be verified against, and applied to, the state of a blockchain network.

See *Transforms* for more information.

5.23 Two Phase Commit

A consensus algorithm that uses two phases to achieve consensus:

1. In the first phase, Validators create and vote for Proposals.
2. Once a Proposal receives a quorum of voting power, a second round of voting begins, wherein Validators vote to "commit" the proposed block.

See *Plug's Consensus Algorithm* for more information.

5.24 Validator

A Node that controls a private key whose Address has been allocated Voting Power, and is therefore authorized to participate in the consensus process.

See *Voting and Voting Power* for more information.

5.25 Vote

A Signature made by a Validator during a Voting Window on a candidate Proposal.

See *Plug's Consensus Algorithm* for more information.

5.26 Voting Power

The relative weight given to Votes issued by a Validator (or more precisely, by that Validator's private key).

The records defining how Voting Power is allocated are stored in the Blockchain State.

See *Voting and Voting Power* for more information.

5.27 Voting Window

A time-boxed window wherein Validators propagate Votes for a candidate Proposal.

During a Voting Window, each Validator may vote for at most one Proposal.

At the end of each Voting Window, the Validators reset their list of Votes sent and received.

See *Plug's Consensus Algorithm* for more information.

CHAPTER

SIX

INTERNAL DOCUMENTATION

This section contains documentation intended for developers contributing to the Plug codebase.

6.1 Interfaces

To support loose coupling, the Plug SDK defines a number of interfaces that components must implement.

6.1.1 Packable

The *plug.abstract.Packable* interface provides the necessary methods to convert fully-realized instances into more primitive maps and to reverse the process by converting these maps back into instances.

FQDN

Packable objects must contain a Fully Qualified Domain Name (FQDN), this is used by the framework to resolve objects from serialized representations.

The FQDN should be added to the implementing class as either a class-level attribute or a property.

The following example shows two valid ways to specify a FQDN:

```
class Foo(Model):
    fqdn = "com.example.models.Foo"

class Bar(Model):
    @property
    def fqdn(self):
        return "com.example.models.Bar"
```

pack

Packing is the process of converting an instance into a map of primitive variables which can be then serialized. This allows libraries like `json` and `msgpack` to use their `dumps` methods to convert the output of `pack` to a string or series of bytes suitable for transport over the network.

The `fqdn` is used by the framework to resolve the implementor's class and convert the output of `pack()` back into an instance. Note that a Packable object should also pack any of its sub-components.

```
def marshall(self) -> dict:
    return {
        "fqdn": self.fqdn,
        "variable": 49,
        "sub_component": self.sub_component.pack(),
    }
```

unpack

Once a packed object has been received, the `unpack()` method may be used to reverse the process. Like packing, unpacking should also be performed recursively on all sub-components to resolve the object fully.

```
@classmethod
def unpack(cls, registry: Registry, payload: dict) -> cls:
    return cls(
        variable=payload["variable"],
        sub_component=registry.unpack(payload["sub_component"]),
    )
```

6.1.2 Applicable

The *plug.abstract.Applicable* interface provides the necessary methods for objects to perform manipulations against the state.

verify

The `verify()` method should accept a state slice and verify if the implementor is capable of manipulating state.

If verification fails, `verify()` must raise an exception (any type).

```
def verify(self, state: dict) -> Result:
    if self.from_address not in state["whitelist"]:
        raise Exception(
            "from_address {addy} not found in whitelist.".format(
                addy = self.from_address,
            ),
        )

    return state
```

apply

The `apply()` method should accept a state slice and modify it accordingly.

Tip: `apply()` does not need to return anything; any changes it makes to the state slice are applied automatically to the blockchain state.

```
def apply(self, state: dict) -> Result:
    state["foo.bar"][self.from_address] -= self.amount
```

(continues on next page)

(continued from previous page)

```
    state["foo.bar"][self.to_address] += self.amount
    return state
```

6.1.3 Signable

The :py:class‘plug.abstract.Signable‘ interface provides a single method to take an unsigned object and return a signed version.

sign

The `sign()` method should accept a *plug.abstract.SigningKey* instance and sign some sort of challenge stored within the implementor.

It must return a new instance of the implementor with both the signature and the associated *plug.abstract.VerifyingKey* attached.

These additional variables can then, for example, be used in conjunction with the `Applicable.verify()` method to check if the implementor has been signed correctly.

```
def sign(self, signing_key: SigningKey):
    verifying_key = signing_key.get_verifying_key()
    signature = signing_key.sign(self.challenge)

    return self.__class__(
        address=self.address,
        nonce=self.nonce,
        challenge=self.challenge,
        verifying_key=verifying_key,
        signature=signature,
    )
```

6.1.4 Hashable

The *plug.abstract.Hashable* interface provides a single method to provide a cryptographic hash of the implementor.

hash

Given a hash function, construct some sort of string and hash it.

```
def _generate_hash(self, hashfn: Callable[[bytes], bytes]) -> bytes:
    challenge = (self.name + str(self.age) + str(self.value)).encode('utf-8)
    return hashfn(challenge)
```

Important: This method must return a `bytes` object, **not** a `str`.

6.2 Supporting Multiple Languages

Plug provides internationalization (I18N) support using the `gettext` module from the standard library. Locale files are generated using Babel.

Tip: Babel configuration can be found in `setup.cfg`.

6.2.1 Workflow

I18N-ifying an application works like this:

1. Mark strings to be translated in the Python code.
2. Extract messages into a template file.
3. Create/update locale-specific catalogs.
4. Translate strings in the catalogs.
5. Compile catalogs.

6.2.2 Mark Strings to Be Translated

In order for Babel to know which strings to translate, you must first "mark" them in the Python code using the `gettext` function (often aliased as "_"):

```
from gettext import gettext as _
...
my_string = _("This is a String to be translated.")
```

Any strings that might be displayed to an end user are good candidates for translation:

- Log messages
- Exception messages
- CLI output messages (i.e., created by the `plug` command)
- Error/success messages returned by the API
- Etc.

6.2.3 Extract Messages

Extracting messages is the process of finding all of the marked strings in the Python code and creating a template. This template will be used to create/update locale-specific catalogs.

To extract messages, use the following command:

```
$ python setup.py extract_messages
```

This will generate a file at `plug/locale/plug.pot`.

Note: The `plug.pot` file does not need to be edited, nor should it be committed to the repository (i.e., add it to `.gitignore`). Instead, it will be used by Babel in the subsequent steps.

Think of it like a cache, so that Babel doesn't have to parse all of the Python code every time you want to update the translation catalogs.

6.2.4 Create/Update Catalogs

The next step is to produce catalogs. Catalogs store the actual locale-specific translations.

Create a New Catalog

To create a new catalog, use the following command:

```
$ python setup.py init_catalog --locale "<locale>"
```

Where `<locale>` is the IETF Language Tag that corresponds to the locale that you want to create translations for.

For example, this command will create a new catalog for Spanish:

```
$ python setup.py init_catalog --locale "es"
```

And this command will create a new catalog specifically for Mexican Spanish:

```
$ python setup.py init_catalog --locale "es_MX"
```

Update Existing Catalogs

If you have any existing catalogs, you will need to update them, to add any new strings that were found when you last did an `extract_messages`.

This command updates all of the existing catalogs at once (you don't have to specify the locale):

```
$ python setup.py update_catalog
```

Tip: Don't worry; this won't delete any existing strings in those catalogs. It only adds new strings.

6.2.5 Make With the Translating

Now for the reason we're all here!

After running `init_catalog` and/or `update_catalog`, you will find the locale-specific catalogs in `plug/locale/*/LC_MESSAGES/plug.po`.

For example, the catalog for Mexican Spanish can be found at `plug/locale/es_MX/LC_MESSAGES/plug.po`.

Adding translations is a simple process:

1. Identify a string by its `msgid` value.
2. Enter the translated string next to `msgstr` on the subsequent line.

If a string doesn't need to be translated, you can leave `msgstr` blank.

6.2.6 Compile Catalogs

Once you've translated the strings in the catalogs, you must then compile them into a format that is efficient for Babel to use at runtime.

To compile catalogs, execute the following command:

```
$ python setup.py compile_catalog
```

This command will load each `plug.po` file and create a corresponding `plug.mo` file in the same directory.

For example, the compiled catalog for Mexican Spanish will be stored at `plug/locale/es_MX/LC_MESSAGES/plug.mo`.

Note: Just like `plug.pot`, you do not need (nor want) to edit these `plug.mo` files. Additionally, they should not be committed to the repository (i.e., add them to `.gitignore`).

However, they should be included in any distributions of the software (i.e., when running `python setup.py sdist`).

To facilitate this, ensure that `recursive-include plug/locale *.mo *.po` appears in `MANIFEST.in`, and be sure to compile catalogs during the build process.

6.3 ACLs

We provide the ability to configure read and write permissions using Access Control Lists (ACLs). These permissions allow to define which users (or addresses) are allowed to perform which Transforms (write permissions) and who is allowed to read specific models or specific keys (read permissions).

6.3.1 Example

In the following example, we define an `admin` group. The members of that group are allowed to read any key from the `BalanceModel`. Anonymous read requests are not allowed to read any key. Authenticated requests are allowed to read the balance belonging to its own address.

Also, the members of the `admin` group are the only ones who can perform `ACLWriteTransform` and `ACLReadTransform` (which are used to modify read and write permissions in the blockchain).

As part of the example, the `sender` of a `BalanceTransfer` transform needs to be a member of the `admin` group too.

Members of any group staring with `foo_` are allowed to access any address that starts with either `foo_` or `bar_`. Note the use of regular expressions to define those rules. Similarly the rules applying to the `admin` group, will apply to the `staff` group too.

```
plug.initial_state:

  plug.builtin.BalanceModel:
    127crBgG5YWvvexNL1bGrgLp9Abr8U5HLZ:
      balance: 1000
    1wyJBsjXzpQvfaRWdXKFNTYow3PzSJumM:
      balance: 4242

  plug.ext.acl.ACLGroupModel:
```

(continues on next page)

(continued from previous page)

```
    admin:
      members:
        - 127crBgG5YWvvexNL1bGrgLp9Abr8U5HLZ
        - 1wyJBsjXzpQvfaRWdXKFNTYow3PzSJumM

  plug.ext.acl.ACLReadModel:
    demo.BalanceModel:
      _anonymous:
        - _none
      _authenticated:
        - _self
      admin|staff:
        - _all
      ^foo_.*$:
        - ^foo_.*$
        - ^bar_.*$
    plug.model.VotingPowerModel:
      _anonymous:
        - _none
      _authenticated:
        - _none

  plug.ext.acl.ACLWriteModel:
    demo.BalanceTransfer:
      sender:
        - admin
    plug.ext.acl.ACLWriteTransform:
      author:
        - admin
    plug.ext.acl.ACLReadTransform:
      author:
        - admin
```

CHAPTER

SEVEN

PYTHON API DOCUMENTATION

7.1 plug package

7.1.1 Submodules

plug.abstract module

class plug.abstract.**Applicable**
Bases: object

Applicable objects interact with State slices by causing mutations. Note that implementors of Applicable are typically containers, whereas objects that directly mutate State slices must implement Mutator

apply(*state_slice*)
Apply this object to state_slice. Ensure that application is performed atomically. Use util.state_slice and util.state_merge to extract a copy of the state that will be mutated, perform and verify the mutation, and merge the slice back into state.

verify(*state_slice*)
Verify that this object can successfully mutate state_slice. This method is used to verify business logic.

class plug.abstract.**Block**
Bases: *plug.abstract.Applicable*, *plug.abstract.PreflightEnabled*

CACHE_NAME = 'blocks'

class plug.abstract.**Cacheable**
Bases: object

CACHE_CLASS
alias of builtins.dict

CACHE_NAME

exception DoesNotExist(*fqdn*, *key*)
Bases: Exception

classmethod get_cached(*caches*, *key*)

class plug.abstract.**Hashable**
Bases: object

Hashable objects can be reduced to a hash, this hash can then be used to make the object immutable as well as provide consistency checks.

hash(*hashfn*) → bytes

class plug.abstract.**Middleware**
Bases: object

class plug.abstract.**Model**(*data*)
Bases: typing.MutableMapping, *plug.abstract.Packable*, *plug.abstract.Hashable*

Defines the structure for a collection of values stored in blockchain state.

static default_factory()

fqdn = None

pack(*registry*)

to_dict()
Return a dict with the content of the internal 'data' structure.

classmethod unpack(*registry*, *payload*)

validator = None

class plug.abstract.**ModelMeta**(**args*, ***kwds*)
Bases: typing.GenericMeta, abc.ABCMeta

Validates model definitions at class creation.

schema = None

class plug.abstract.**Mutator**
Bases: *plug.abstract.Applicable*

A specific Applicable object that may directly interact with State. Use a Mutator to actually change state.

required_keys()
Return a set of keys (from the models) that this mutator requires to verify and apply correctly.

static required_models()
Return a set of models that this mutator requires to verify and apply correctly.

class plug.abstract.**Packable**
Bases: object

pack(*registry*)

classmethod unpack(*registry*, *payload*)

class plug.abstract.**PayloadMiddleware**
Bases: *plug.abstract.Middleware*

static on_payload(*proxy*, *runner*, *ident*)
Process this inbound payload before the runner acts upon it. You may mutate *payload* to decorate the inbound request in any way your application requires. To abort the request, raise an exception.

class plug.abstract.**PeerDiscovery**
Bases: object

discover_peers(*current_address*)

register(*address*)

total_nodes

class plug.abstract.**Preflight**(*fqdn*, *hash_*)
Bases: *plug.abstract.Packable*

fqdn = 'plug.abstract.Preflight'

pack(*registry=None*)

classmethod unpack(*registry*, *payload*)

class plug.abstract.**PreflightEnabled**
Bases: *plug.abstract.Hashable*, *plug.abstract.Packable*, *plug.abstract.Cacheable*

PreflightEnabled objects should not be sent around the network unless explicitly requested. Instead a hash is sent, which the receiver must then decide what to do with.

classmethod make_preflight(*hash_*)

class plug.abstract.**Proof**
Bases: *plug.abstract.Signable*, *plug.abstract.Mutator*, *plug.abstract.Hashable*, *plug.abstract.Packable*

fqdn = None

pack(*registry*)

represented_address()

classmethod unpack(*registry*, *payload*)

class plug.abstract.**Proxy**
Bases: *plug.abstract.Runnable*

address

connect(*endpoint*, *ident=None*)

name

send(*ident*, *payload*) → str

class plug.abstract.**RPC**(*request_id=None*)
Bases: *plug.abstract.Packable*

fqdn = None

handle(*runner*, *proxy*, *registry*, *ident*)

version = None

class plug.abstract.**RequestMiddleware**
Bases: *plug.abstract.Middleware*

static on_request(*proxy*, *runner*, *ident*)
Process this inbound payload before the runner acts upon it. You may mutate *payload* to decorate the inbound request in any way your application requires. To abort the request, raise an exception.

static on_response(*response*, *proxy*, *runner*, *ident*)
Process this outbound payload after the runner has acted upon it. You may mutate *payload* to decorate the outbound request in any way your application requires. To abort the response, raise an exception.

class plug.abstract.**Runnable**
Bases: object

run()

setup()

stop_flag = False

teardown()

tick()

class plug.abstract.**Runner**
Bases: *plug.abstract.Runnable*

on_connection(*ident*, *proxy*)

on_payload(*payload*, *ident*, *proxy*)

class plug.abstract.**Signable**
Bases: object

sign(*key*) → plug.abstract.Signable

class plug.abstract.**SigningKey**
Bases: *plug.abstract.Packable*

Base implementation for any signing keys used by the system.

fqdn = None

classmethod from_string(*data: str*) → plug.abstract.SigningKey
Construct a signing key from a given string.

get_verifying_key() → plug.abstract.VerifyingKey
Return the associated verifying key for this signing key.

classmethod new()
Create a new instance of this signing key.

sign(*data: bytes*) → bytes
Sign some data using this key and return the signature.

to_string() → str
Extract the signing key.

class plug.abstract.**State**
Bases: *plug.abstract.Hashable*, *plug.abstract.Packable*

get_node_by_path(*path: str*)

upgrade(*block: plug.abstract.Block*, *hashfn*)

class plug.abstract.**Storage**(*registry*)
Bases: object

add_unconfirmed_transaction(*transaction*)
Persists an unconfirmed transaction to storage.

Note: Unlike state/block, the order that unconfirmed transactions are added does not matter.

---

confirm_transactions(*transaction_hashes*)
Marks a collection of unconfirmed Transactions as confirmed, ignoring any that are already confirmed.

Tip: If you can confirm a single transaction more efficiently, handle that in a local single confirm method, if a list of one hash is provided

---

get_block(*height_or_hash*)
Retrieve a block from storage by its height or hash.

Raise

- plug.storage.BlockNotFound if the referenced block does not exist.

get_keyring()
Retrieve the node's keyring from storage.

get_state(*height_or_hash*)
Retrieve a state from storage by its height or hash.

Raise

- `plug.storage.StateNotFound` if the referenced state does not exist.

get_transaction(*transaction_hash*)
Retrieve a transaction from storage by its hash.

Raises:

- `plug.storage.TransactionNotFound` if the referenced transaction does not exist.

remove_unconfirmed_transaction(*transaction*)
Removes the specified transaction from storage.

If the transaction does not exist, this method is a no-op.

set_block(*block*)
Persists a new head block to storage.

set_keyring(*keyring*)
Persists the node's keyring to storage.

set_state(*state*)
Persists a new head state to storage.

unconfirmed_transactions()
Iterates over unconfirmed transactions and the corresponding hashes.

Each iteration yields a tuple of `(hash, transaction)`.

Important: Order is undefined.

Note: This method usually returns an AsyncGenerator.

See https://www.python.org/dev/peps/pep-0525/ for more information.

---

class plug.abstract.**Transaction**
Bases: *plug.abstract.Applicable*, *plug.abstract.PreflightEnabled*

CACHE_NAME = 'transactions'

class plug.abstract.**Transform**
Bases: *plug.abstract.Mutator*, *plug.abstract.Hashable*, *plug.abstract.Packable*

fqdn = None

pack(*registry*)

required_authorizations()

classmethod unpack(*registry*, *payload*)

class plug.abstract.**TypeCheckedABCMeta**
Bases: abc.ABCMeta

ABCMeta with mandatory type-checking on the subclasses' __init__ method.

class plug.abstract.**VerifyingKey**
Bases: *plug.abstract.Packable*

fqdn = None

classmethod from_string(*data: str*) → plug.abstract.VerifyingKey
Construct a new verifying key from a given string

to_string() → str
Export this verifying key to a string

verify(*data: bytes*, *signature: bytes*) → bool
Validate a given signature for some data then return TruelFalse

class plug.abstract.**WebSocketPage**
Bases: object

actions
Supported websocket actions, override if needed

render()
Return rendered json for the requested page

plug.abstract.**instance_to_dict**(*obj*, *registry*)
Pack the provided object into a primitive representation. The output should be serializable with json.dumps

plug.builtin module

class plug.builtin.**ACLTransform**(*author: str*, *address: str*, *add: Collection[str] = None*, *remove: Collection[str] = None*)
Bases: *plug.abstract.Transform*

Transform the permissions in the ACL Model. Permissions are constant strings stored in a collection against an address

An example view of the model is shown below:

```
{
    "1FG3LHD7kmQeZnCUXQJ756H2sGv9L8oRJ": ["ADMIN"]
}
```

apply(*state*)
Apply the additions and subtractions to the permissions. Do this as sets.

fqdn = 'plug.builtin.ACLTransform'

pack(*registry*)

required_authorizations()

required_keys()
Return a set of keys (from the models) that this mutator requires to verify and apply correctly.

static required_models()
Return a set of models that this mutator requires to verify and apply correctly.

classmethod unpack(*registry*, *payload*)

verify(*state*)
For an ACL transformation there is no logical check as permissions are enforced at the transaction level.

class plug.builtin.**BalanceModel**(*data*)
Bases: *plug.abstract.Model*

static default_factory()

fqdn = 'demo.BalanceModel'

validator = None

class plug.builtin.**BalanceTransfer**(*sender: str*, *receiver: str*, *amount: numbers.Number*)
Bases: *plug.abstract.Transform*

Send an amount of crypto currency from one address to another.

apply(*state*)
Apply this object to state_slice. Ensure that application is performed atomically. Use util.state_slice and util.state_merge to extract a copy of the state that will be mutated, perform and verify the mutation, and merge the slice back into state.

fqdn = 'demo.BalanceTransfer'

required_authorizations()

required_keys()
Return a set of keys (from the models) that this mutator requires to verify and apply correctly.

static required_models()
Return a set of models that this mutator requires to verify and apply correctly.

verify(*state*)
Verify that this object can successfully mutate state_slice. This method is used to verify business logic.

class plug.builtin.**PersonModel**(*data*)
Bases: *plug.abstract.Model*

static default_factory()

fqdn = 'demo.PersonModel'

validator = None

class plug.builtin.**UpdatePerson**(*person: str*, *data: Dict*)
Bases: *plug.abstract.Transform*

apply(*state*)
Apply this object to state_slice. Ensure that application is performed atomically. Use util.state_slice and util.state_merge to extract a copy of the state that will be mutated, perform and verify the mutation, and merge the slice back into state.

fqdn = 'demo.UpdatePerson'

required_authorizations()

required_keys()
Return a set of keys (from the models) that this mutator requires to verify and apply correctly.

static required_models()
Return a set of models that this mutator requires to verify and apply correctly.

verify(*state_slice*)
Verify that this object can successfully mutate state_slice. This method is used to verify business logic.

plug.consensus module

class plug.consensus.**Block**(*height*, *state_hash*, *timestamp*, *transactions*, *previous_commits*, *commits=None*)
Bases: *plug.abstract.Block*

Block is a collection of transactions. There is no special behavior other than wrapping the verify and apply, and exposing a hash for the contents.

apply(*state_slice*)
Apply this object to state_slice. Ensure that application is performed atomically. Use util.state_slice and util.state_merge to extract a copy of the state that will be mutated, perform and verify the mutation, and merge the slice back into state.

fqdn = 'plug.consensus.Block'

pack(*registry*)

classmethod unpack(*registry*, *payload*)

verify(*state_slice*)
Verify that this object can successfully mutate state_slice. This method is used to verify business logic.

class plug.consensus.**Commit**(*block_hash: bytes*, *height: int*, *signature: str = None*, *verifying_key=None*) → None
Bases: *plug.abstract.Hashable*, *plug.abstract.Packable*, *plug.abstract.Signable*

Commit is the second part of the two phase commit. It indicates that an address believes this block has quorum and therefore wants to commit to making it the future.

You must only commit once per height. There should never be a case where you can or will commit more than once.

fqdn = 'plug.consensus.Commit'

pack(*registry*)

sign(*key*)

classmethod unpack(*registry*, *payload*)

class plug.consensus.**State**(*models*, *height=0*, *previous_block_hash=None*, *timestamp=None*)
Bases: *plug.abstract.State*

State is the core of the default plug consensus mechanism. It stores a reference to the previous block hash, all of the models that contain the keys and values.

fqdn = 'plug.consensus.State'

get_node_by_path(*path*)
We traverse a state following the given path and return the node.

Args: path (str): The path to use.

Returns: The node pointed by *path*.

Raises: ValidationError: If the path is invalid or unknown.

pack(*registry*)

classmethod unpack(*registry*, *payload*)

upgrade(*block*, *hashfn*)
Given a block and its hash value upgrade the state of the world. Apply the block directly to state and then increment the height.

class plug.consensus.**Transaction**(*transform: T*, *proofs: Union[typing.Mapping[str, plug.abstract.Proof], NoneType] = None*) → None
Bases: typing.Generic, *plug.abstract.Transaction*

Transaction is a wrapper that binds together a Transformation and then some Proofs. This is more of a container. It enforces that an action can happen and that it is authenticated. It will also enforce that a transform has the required set of proofs from the required addresses.

Tip: When using *Transaction* in type hints, you can also specify the transform type.

Example:

```
txn: Transaction[BalanceTransfer] = ...
```

In the above code, most IDEs will recognize `txn.transform` as an instance of `BalanceTransfer` automatically.

---

apply(*state_slice: MutableMapping*) → None

- Slice up the state slice passed in.
- Apply all of the proofs to the state slice.
- Apply the transform.
- Return the new state slice.

fqdn = 'plug.consensus.Transaction'

pack(*registry: plug.registry.Registry*) → Dict

required_keys() → Set[str]

required_models() → Set[str]

classmethod unpack(*registry: plug.registry.Registry*, *payload: Mapping*) → plug.consensus.Transaction[T]

verify(*state_slice: Mapping*) → None

Verify...

- ... that all required proofs are attached.
- ... every proof against state slice.
- ... the transform against state slice.

class plug.consensus.**Vote**(*block_hash: bytes*, *timestamp: str*, *height: int*, *signature=None*, *verifying_key=None*) → None

Bases: *plug.abstract.Hashable*, *plug.abstract.Packable*, *plug.abstract.Signable*

Vote object is the first part of the two phase commit. It indicates that an address that has voting power wants this block to be the next future block.

You can vote once per window until a winner is found, but voting for two different blocks in the same window is considered naughty.

fqdn = 'plug.consensus.Vote'

pack(*registry*)

sign(*key*)

classmethod unpack(*registry*, *payload*)

plug.digest module

plug.digest.**digest**(*obj: Any*) → bytes

Take a primitive or a collection of primitives and produce a repeatable digest for cryptographic hashing. Base supported types are:

None Bool Unsigned Int (0 .. 18446744073709551615) Signed Int (-9223372036854775808 .. 9223372036854775807) UTF8 String Bytes

The following collections containing primitive types are also supported

List (tuple) Map

Map keys *must* be strings.

plug.digest.**digest_bool**(*obj: bool*) → bytes

plug.digest.**digest_bytes**(*obj: bytes*) → bytes

plug.digest.**digest_int**(*obj: int*) → bytes

plug.digest.**digest_mapping**(*obj: Mapping[str, Any]*) → bytes

plug.digest.**digest_none**(*obj: None*) → bytes

plug.digest.**digest_sequence**(*obj: Sequence[Any]*) → bytes

plug.digest.**digest_str**(*obj: str*) → bytes

plug.error module

exception plug.error.**AuthorizationError**(*message*, *status_code=None*)

Bases: *plug.error.HttpException*

Indicates that a Read or Write operation requested through an Proxy was not properly Authenticated or that the authentiated user doesn't have the proper permissions to perform such operation.

status_code = 403

exception plug.error.**BadValueError**(*message*)

Bases: *plug.error.PlugException*

exception plug.error.**BlockNotFoundError**(*message*)

Bases: *plug.error.StorageException*

Used by storage backends to indicate that the block with the specified height/hash was not found.

exception plug.error.**ConfigurationError**(*message*)

Bases: *plug.error.PlugException*

Indicates that there is a problem with the Plug configuration (e.g., loaded from the node's `config.yaml` file).

exception plug.error.**HttpException**(*message*, *status_code=None*)

Bases: *plug.error.PlugException*

exception plug.error.**NoFQDNError**(*message*)

Bases: *plug.error.RegistryException*

Used by registries to indicate that a value could not be processed because it has no FQDN.

exception plug.error.**NotPackableError**(*message*)

Bases: *plug.error.RegistryException*

Indicates that a *plug.registry.Registry* tried to pack an object that it does not know how to pack.

This can occur if, for example, the object's type does not implement the *plug.abstract.Packable* interface.

exception plug.error.**PlugException**(*message*)
Bases: Exception

Base class for exceptions related to Plug functionality.

exception plug.error.**RegistryException**(*message*)
Bases: *plug.error.PlugException*

Base class for exceptions related to Plug registry operations.

exception plug.error.**RemovedInNextVersionWarning**
Bases: DeprecationWarning

A warning indicating that a reference/feature has been deprecated.

Note that this class does not derive from *PlugException*, because it's not an exception.

exception plug.error.**SetupError**(*message*)
Bases: *plug.error.PlugException*

Indicates that there is a problem setting up an object.setup()

exception plug.error.**StateNotFoundError**(*message*)
Bases: *plug.error.StorageException*

Used by storage backends to indicate that the state with the specified height/hash was not found.

exception plug.error.**StorageException**(*message*)
Bases: *plug.error.PlugException*

Base class for exceptions related to Plug storage operations.

exception plug.error.**TransactionNotFoundError**(*message*)
Bases: *plug.error.StorageException*

Used by storage backends to indicate that the transaction with the specified hash was not found.

exception plug.error.**UnhandledRequestTypeError**(*message*)
Bases: *plug.error.PlugException*

Used to signal that a proxy tried to pass an invalid Request to a Runner using the *on_request* method.

exception plug.error.**UnknownPayloadError**(*message*)
Bases: *plug.error.RegistryException*

Indicates that a *plug.registry.Registry* tried to unpack a payload containing an unregistered FQDN.

exception plug.error.**ValidationError**(*message*, *status_code=None*)
Bases: *plug.error.HttpException*

Indicates that a value in an API request failed validation.

This exception is used by HTTP proxies; the message is accompanied by an HTTP status code that should be used for the response sent back to the client (usually 400, but 404 is also commonly used).

status_code = 400

exception plug.error.**VerificationError**(*message*)
Bases: *plug.error.PlugException*

Indicates that a transaction's `verify()` method failed during a `propose()` operation.

This exception is used internally; client applications will not encounter this exception.

plug.key module

class plug.key.**ED25519SigningKey**(*signing_key*)
Bases: *plug.abstract.SigningKey*

Default plug key. ED25519 provided by Nacl.

fqdn = 'plug.key.ED25519SigningKey'

classmethod from_string(*data: str*) → plug.abstract.SigningKey
Import nacl ed25519 signing key in hex encoding.

get_verifying_key() → plug.abstract.VerifyingKey
Return the associated verifying key for this signing key.

classmethod new()
Create a new instance of this signing key.

pack(*registry*)

sign(*data: bytes*) → bytes
Sign some data using this key and return the signature.

to_string() → str
Export nacl ed25519 signing key as hex.

classmethod unpack(*registry*, *payload*)

class plug.key.**ED25519VerifyingKey**(*verifying_key*)
Bases: *plug.abstract.VerifyingKey*

fqdn = 'plug.key.ED25519VerifyingKey'

classmethod from_string(*data: str*) → plug.abstract.VerifyingKey
Export nacl ed25519 verifying key in hex encoding.

pack(*registry*)

to_string() → str
Import nacl ed25519 verifying key in hex encoding.

classmethod unpack(*registry*, *payload*)

verify(*data: bytes*, *signature: bytes*) → bool
Validate a given signature for some data then return TruelFalse

class plug.key.**Keyring**
Bases: dict, *plug.abstract.Packable*

add_key(*key*)

fqdn = 'plug.key.Keyring'

pack(*registry*)

classmethod unpack(*registry*, *payload*)

plug.merkle module

class plug.merkle.**MerkleMap**(*data=None*, *order=None*)
Bases: *plug.abstract.Hashable*, *plug.abstract.Packable*

A MerkleMap is a map like data structure. The MerkleMap is designed to store key+value as leafs in a true Merkle tree where root hashes can be calculated. This tree works using binary children (single left and right) And will fill any uneven levels with hash("").

fqdn = 'plug.MerkleMap'

classmethod from_dict(*data*, *order=None*)
Helper method to build from a dictionary.

get(*key*, *default=None*)
Get this key from the container. If no default is provided then use the underlying default value (same as doing thing[key])

Otherwise if a default is explicitly provided override the underlying default.

get_proof(*key*)

hash(*hashfn*) → bytes

keys()
Return the keys (in order) of the merkle map.

pack(*registry*)
Pack the data into a naive dict.

classmethod unpack(*registry*, *payload*)
From a given payload restore into an actual instance.

values()
Return the unsorted values of this container.

verify_proof(*proof*)

plug.merkle.**create_pairs**(*items*, *fill*)
Take a list of items and pair them together. If there are not enough elements then use the fill value.

plug.model module

class plug.model.**NonceModel**(*data*)
Bases: *plug.abstract.Model*

static default_factory()

fqdn = 'plug.model.NonceModel'

validator = None

class plug.model.**TransformPermissionModel**(*data*)
Bases: *plug.abstract.Model*

static default_factory()

fqdn = 'plug.model.TransformPermissionModel'

validator = None

class plug.model.**UserPermissionModel**(*data*)
Bases: *plug.abstract.Model*

static default_factory()

fqdn = 'plug.model.UserPermissionModel'

validator = None

class plug.model.**VotingPowerModel**(*data*)
Bases: *plug.abstract.Model*

static default_factory()

fqdn = 'plug.model.VotingPowerModel'

validator = None

plug.proof module

class plug.proof.**SingleKeyProof**(*address*, *nonce*, *challenge*, *verifying_key=None*, *signature=None*)
Bases: *plug.abstract.Proof*

apply(*state_slice*)
Apply this object to state_slice. Ensure that application is performed atomically. Use util.state_slice and util.state_merge to extract a copy of the state that will be mutated, perform and verify the mutation, and merge the slice back into state.

fqdn = 'plug.proof.SingleKeyProof'

pack(*registry*)

represented_address()

required_keys()
Return a set of keys (from the models) that this mutator requires to verify and apply correctly.

static required_models()
Return a set of models that this mutator requires to verify and apply correctly.

sign(*key*)

classmethod unpack(*registry*, *payload*)

verify(*state_slice*)
Verify that this object can successfully mutate state_slice. This method is used to verify business logic.

plug.registry module

class plug.registry.**Registry** → None
Bases: `object`

Maintains references to important classes, keyed by (usually) their corresponding `fqdn` attributes.

Registries are used primarily when packing (serializing) and unpacking (deserializing) values, for example, when preparing values for transport or storage.

classmethod default() → plug.registry.Registry
Creates a new Registry instance and populates it with core Plug components.

import_class(*ref: str*, *use_class_fqdn: bool = False*) → Type
Attempts to import and register a class, treating the fqdn as a Python path or Entry Point.

Parameters

- **ref** – Dotted path to the Python class, or entry point syntax.

  If entry point syntax is used, the name of the entry point will be used as the FQDN instead of the class' `fqdn` attribute (unless `use_class_fqdn=True`).

  Examples:

```
# Using dotted path:
registry.import_class("plug.model.NonceModel")
assert registry["plug.model.NonceModel"] is NonceModel

# Using entry point syntax:
registry.import_class("custom_fqdn=plug.model:NonceModel")
assert registry["custom_fqdn"] is NonceModel
```

- **use_class_fqdn** – Whether to register the class using that class' `fqdn` attribute.

  If `ref` is an Entry Point, then this will override the Entry Point name.

  Example:

```
ref = "custom_fqdn=plug.model:NonceModel"

# With ``use_class_fqdn=True``:
registry = Registry()
registry.import_class(ref, use_class_fqdn=True)

assert "custom_fqdn" not in registry
assert registry[NonceModel.fqdn] is NonceModel

# With ``use_class_fqdn=False``:
registry = Registry()
registry.import_class(ref, use_class_fqdn=False)

assert registry["custom_fqdn"] is NonceModel
assert NonceModel.fqdn not in registry
```

Returns The registered class.

models() → Generator[[Tuple[str, Type[plug.abstract.Model]], NoneType], NoneType]
Iterates over all registered models.

Yields the registered FQDN and the corresponding class (in case the class was registered using an alias).

pack(*instance*)

register(*cls: Type*, *fqdn: str = None*, *pack: Callable[plug.abstract.Packable, Mapping] = None*, *unpack: Callable[Mapping, plug.abstract.Packable] = None*, *legacy: bool = False*) → None
Adds a class to the registry.

Parameters

- **cls** – The class to register.
- **fqdn** – The corresponding FQDN (registry key).

  If not specified, `cls.fqdn` will be used.
- **pack** – Optional custom packer to use for this class.
- **unpack** – Optional custom unpacker to use for this class.
- **legacy** – Indicates whether this is a legacy/deprecated FQDN for the specified class.

  Legacy FQDNs will emit a warning when accessed.

safe_pack(*instance*)
Like *Registry.pack*, but we return the input instance if we don't know how to pack it.

transforms() → Generator[[Tuple[str, Type[plug.abstract.Transform]], NoneType], NoneType]
Iterates over all registered transforms.

Yields the registered FQDN and the corresponding class (in case the class was registered using an alias).

unpack(*payload*)

plug.rpc module

class plug.rpc.**AddTransactions**(*transactions*, *request_id=None*)
Bases: *plug.abstract.RPC*

fqdn = 'plug.rpc.AddTransactions'

handle(*runner*, *proxy*, *registry*, *ident*)

pack(*registry*)

classmethod unpack(*registry*, *payload*)

version = 'v0'

class plug.rpc.**Args**(*args*)
Bases: *plug.abstract.Hashable*, *plug.abstract.Packable*

fqdn = 'plug.consensus.Args'

get(**keys*)

pack(*registry*)

classmethod unpack(*registry*, *payload*)

class plug.rpc.**EnsureSynchronized**(*height*, *request_id=None*)
Bases: *plug.abstract.RPC*

fqdn = 'plug.rpc.EnsureSynchronized'

handle(*runner*, *proxy*, *registry*, *ident*)

pack(*registry*)

classmethod unpack(*registry*, *payload*)

version = 'v0'

class plug.rpc.**Event**(*topic*, *event*, *channel_id*, *payload*)
Bases: *plug.abstract.Hashable*, *plug.abstract.Packable*

fqdn = 'plug.rpc.Event'

pack(*registry*)

classmethod unpack(*registry*, *payload*)

class plug.rpc.**Handshake**(*advertise_endpoint*, *request_id=None*)
Bases: *plug.abstract.RPC*

fqdn = 'plug.rpc.Handshake'

handle(*runner*, *proxy*, *registry*, *ident*)

pack(*registry*)

classmethod unpack(*registry*, *payload*)

version = 'v0'

class plug.rpc.**HandshakeReply**(*peers*, *request_id=None*)
Bases: *plug.abstract.RPC*

fqdn = 'plug.rpc.HandshakeReply'

handle(*runner*, *proxy*, *registry*, *ident*)

pack(*registry*)

classmethod unpack(*registry*, *payload*)

version = 'v0'

class plug.rpc.**Heartbeat**(*request_id=None*)
Bases: *plug.abstract.RPC*

fqdn = 'plug.rpc.Heartbeat'

handle(*runner*, *proxy*, *registry*, *ident*)

pack(*registry*)

classmethod unpack(*registry*, *payload*)

version = 'v0'

class plug.rpc.**Reply**(*response_id*, *payload*, *exception=None*)
Bases: *plug.abstract.Packable*

fqdn = 'plug.rpc.Reply'

pack(*registry*)

classmethod unpack(*registry*, *payload*)

class plug.rpc.**Request**(*event*, *channel_id*, *payload*)
Bases: *plug.abstract.Hashable*, *plug.abstract.Packable*

fqdn = 'plug.rpc.Request'

pack(*registry*)

classmethod unpack(*registry*, *payload*)

class plug.rpc.**RequestPreflightedObject**(*payload: dict*, *request_id=None*)
Bases: *plug.abstract.RPC*

fqdn = 'plug.rpc.RequestPreflightedObject'

handle(*runner*, *proxy*, *registry*, *ident*)

pack(*registry*)

classmethod unpack(*registry*, *payload*)

version = 'v0'

class plug.rpc.**Subscribe**(*channel_id*, *topic*, *request_id=None*)
Bases: *plug.abstract.RPC*

fqdn = 'plug.rpc.Subscribe'

handle(*runner*, *proxy*, *registry*, *ident*)

pack(*registry*)

classmethod unpack(*registry*, *payload*)

version = 'v0'

class plug.rpc.**Topic**
Bases: object

BLOCK = '{}/block/{}'

COMMIT = '{}/commit/{}'

READ = '{}/read'

TRANSACTION = '{}/transaction/{}'

UPGRADE = '{}/upgrade/{}'

VOTE = '{}/vote/{}'

plug.runner module

plug.serializer module

class plug.serializer.**Serializer**(*loads*, *dumps*, *mime*, *ext*)
Bases: tuple

dumps
Alias for field number 1

ext
Alias for field number 3

loads
Alias for field number 0

mime
Alias for field number 2

plug.serializer.**all**()

plug.serializer.**get_serializer**(*name*)

plug.signals module

class plug.signals.**AsyncSignal**(**args*, ***kwargs*)
Bases: blinker.base.Signal

send(**sender*, ***kwargs*)
Emit this signal on behalf of *sender*, passing on **kwargs.

Returns a list of 2-tuples, pairing receivers with their return value. If receiver is a coroutine the return value is a Future. The ordering of receiver notification is undefined.

Parameters

- ***sender** – Any object or None. If omitted, synonymous with None. Only accepts one positional argument.
- ****kwargs** – Data to be sent to receivers.

class plug.signals.**NamedAsyncSignal**(*name*, *doc=None*)
Bases: *plug.signals.AsyncSignal*

A named generic notification emitter.

name = None
The name of this signal.

class plug.signals.**Namespace**
Bases: dict

A mapping of signal names to signals.

signal(*name*, *doc=None*)
Return the NamedSignal *name*, creating it if required.

Repeated calls to this function will return the same signal object.

plug.util module

plug.util.**block_to_transforms**(*block*)

plug.util.**eq_hashables**(*lhs: plug.abstract.Hashable*, *rhs: plug.abstract.Hashable*, *hash_fn: Callable[bytes, bytes] = <function sha256>*)
Performs an equality comparison on two *plug.abstract.Hashable* objects.

plug.util.**get_window**(*window_length: datetime.timedelta*, *step: int = 0*) → datetime.datetime
Get the floor of a voting window knowing the given window_length.

For example (at 12:00:53)

```
>>> get_window(datetime.timedelta(seconds=5))
>>> datetime.datetime(2017, 5, 25, 12, 00, 50)
```

Step allows for getting previous or future voting window floors. Negative numbers will compute previous windows with -1 being the last window just passed. Positive numbers will generate future windows.

plug.util.**import_dotted_class**(*dotted_path*)

plug.util.**import_dotted_func**(*dotted_path*)

plug.util.**obj_to_transforms**(*obj*)
Given a Plug object (Block, State etc.), yield all the transforms that are part of that object.

plug.util.**parse_duration**(*duration*)

plug.util.**plug_address**(*value*)

plug.util.**plug_address_from_key**(*key*)

plug.util.**sha256**(*data: bytes*) → bytes
Plug wrapper for hashlib.sha256

plug.util.**sha512**(*data: bytes*) → bytes
Plug wrapper for hashlib.sha512

plug.util.**state_merge**(*state*, *state_slice*)
Apply a state_slice to state.

plug.util.**state_slice**(*state*, *required_models*, *required_keys*)
Extract models and keys from state where they exist. Deep copy the values to stop accidental mutation.

plug.util.**strip_prefix**(*text*, *prefix*)

plug.util.**strip_suffix**(*text*, *suffix*)

plug.util.**transaction_to_transforms**(*transaction*)

7.2 plug.cli package

7.2.1 Submodules

plug.cli.new module

plug.cli.new.**DEFAULT_HASH_FN**(*data: bytes*) → bytes
Plug wrapper for hashlib.sha256

plug.cli.new.**generate_node_configurations**(*config: plug.cli.config.Config*, *node_count*, *hosts*) → Dict[int, plug.cli.config.Config]
Like new(), except that it only generates the node configurations; there are no side effects (e.g., initializing storages).

Returns A mapping of node configurations. Keys are node IDs.

plug.cli.new.**init_node_storages**(*output_directory: str, registry: plug.registry.Registry, state: plug.consensus.State, keys: Sequence[plug.key.ED25519SigningKey], node_configs: Mapping[int, plug.cli.config.Config], hash_fn: Callable[bytes, bytes] = <function sha256>, serializer: plug.serializer.Serializer = Serializer(loads=<built-in function loads>, dumps=<built-in function dumps>, mime='application/cbor', ext='cbor')*) → None
Initializes the storages for the new nodes.

plug.cli.new.**setup**(*num*, *config*, *output_directory*, *force*)

plug.cli.run module

plug.cli.run.**check_for_uvloop**()

plug.cli.run.**check_for_windows_loop**()

plug.cli.run.**init_caches**(*config*, *runner*)

plug.cli.run.**init_config**(*config_file*)

plug.cli.run.**init_hooks**(*config*)

plug.cli.run.**init_logging**(*log_level*)

plug.cli.run.**init_middlewares**(*config*)

plug.cli.run.**init_proxies**(*config*, *serializer*, *registry*, *loop*)

plug.cli.run.**init_registry**(*config*)

plug.cli.run.**init_storage**(*config*, *registry*, *serializer*, *path*)

7.3 plug.orm package

class plug.orm.**Model**(*state_slice: Union[typing.Mapping, plug.consensus.State]*) → None
Bases: *plug.abstract.Model*

Base functionality for ORM models.

DoesNotExist
alias of *Model.DoesNotExist*

IntegrityError
alias of *Model.IntegrityError*

create(*pk: str*, ***kwargs*) → T
Creates a new model record at the specified key.

Raise

- *IntegrityError* if the key already exists.

default_factory() → Dict

get_if_exists(*pk: str*) → Union[~T, NoneType]
Returns the model record at the specified key, if it exists.

If the key does not exist, this method returns `None`.

get_or_create(*pk: str*) → T
Returns the model record at the specified key, if it exists.

If the key does not exist, this method creates a new empty record and returns it.

items() → Iterator[Tuple[str, T]]
Iterates over stored key/value pairs for this model.

keys() → Iterator[str]
Iterates over stored keys for this model.

require(*pk: str*) → T
Finds the model record at the specified key.

If the key does not exist, a `DoesNotExist` exception will be raised.

update(*pk: str*, ***kwargs*) → T
Updates the model record at the specified key, if it exists.

If the key does not exist, a `DoesNotExist` exception will be raised.

Returns The updated model instance.

upsert(*pk*, ***kwargs*) → T
Updates the model record at the specified key, if it exists.

If the key does not exist, a new empty record will be created first.

Tip: Make sure that `kwargs` covers **all** of the fields you want to set; do not rely on default values for any field (i.e., assume that there is an existing record that will be updated, and some of its fields have unexpected values).

Returns The updated model instance.

validator = None

values() → Iterator[T]
Iterates over stored values for this model.

class plug.orm.**ModelInstance**(*pk: Union[str, NoneType] = None*, ***kwargs*)
Bases: `object`

A single instance of a model stored in state.

classmethod deserialize(*pk: Union[str, NoneType], payload: Mapping*) → T
Creates an instance from a dict representation.

Note: Not using `unpack` because that requires a Registry reference.

model = None

refresh() → None
Reloads the model instance from state, overwriting any local modifications.

save(*using: Union[_ForwardRef('Model'), NoneType] = None*) → None
Persists the model instance to state.

This will overwrite any existing model record with the same key.

Parameters **using** – If set, specify the *Model* instance to use.

Note: This will replace any model instance already attached (e.g., as a result of *Model.create()*).

serialize() → Dict
Returns a dict representation of the instance.

Note: Not using `pack` because that requires a Registry reference.

7.3.1 Submodules

plug.orm.fields module

class plug.orm.fields.**Collection**(*field_type: Union[plug.orm.fields.Field, typing.Type[plug.orm.fields.Field]] = <class 'plug.orm.fields.Unspecified'>, python_type: Union[typing.Type[typing.MutableSequence], typing.Callable[[], typing.MutableSequence]] = <class 'list'>*)
Bases: *plug.orm.fields.Field*

Models an ordered collection of homogeneous values (roughly equivalent to `typing.MutableSequence`).

coerce(*value*)
Ensures that an incoming value (i.e., provided to `__set__()`) has the correct type.

Caution: `value` might have any type.

Caution: `value` might be `None`.

deserialize(*value*)
Takes a serialized value and deserializes it.

Parameters **value** – Result of calling *serialize()* at some point.

> **Caution:** `value` might be `None`.

Note: Not using `unpack` because that requires a Registry reference.

serialize(*value*)
Returns a primitive version of the field value for hashing, packing, etc.

Parameters **value** – Result of calling *coerce()* at some point.

> **Caution:** `value` might be `None`.

Note: Not using `pack` because that requires a Registry reference.

class plug.orm.fields.**Field**(*default=None*)
Bases: `object`

A data descriptor that defines a field in a model's schema.

coerce(*value*)
Ensures that an incoming value (i.e., provided to `__set__()`) has the correct type.

> **Caution:** `value` might have any type.

> **Caution:** `value` might be `None`.

compare_equals(*lhs*, *rhs*) → bool
Compares the field value between two instances and returns whether they are equal.

deserialize(*value*)
Takes a serialized value and deserializes it.

Parameters **value** – Result of calling *serialize()* at some point.

> **Caution:** `value` might be `None`.

Note: Not using `unpack` because that requires a Registry reference.

serialize(*value*)
Returns a primitive version of the field value for hashing, packing, etc.

Parameters **value** – Result of calling *coerce()* at some point.

> **Caution:** `value` might be `None`.

Note: Not using `pack` because that requires a Registry reference.

class plug.orm.fields.**Mapping**(*field_type: Union[plug.orm.fields.Field, typing.Type[plug.orm.fields.Field]] = <class 'plug.orm.fields.Unspecified'>, python_type: Union[typing.Type[typing.MutableMapping], typing.Callable[[], typing.MutableMapping]] = <class 'dict'>*)

Bases: *plug.orm.fields.Field*

Models a mapping, where individual fields are each assigned a unique key (roughly equivalent to `typing.Mapping`).

coerce(*value*)

Ensures that an incoming value (i.e., provided to `__set__()`) has the correct type.

> **Caution:** `value` might have any type.

> **Caution:** `value` might be `None`.

deserialize(*value*)

Takes a serialized value and deserializes it.

Parameters **value** – Result of calling *serialize()* at some point.

> **Caution:** `value` might be `None`.

Note: Not using `unpack` because that requires a Registry reference.

serialize(*value*)

Returns a primitive version of the field value for hashing, packing, etc.

Parameters **value** – Result of calling *coerce()* at some point.

> **Caution:** `value` might be `None`.

Note: Not using `pack` because that requires a Registry reference.

class plug.orm.fields.**NamedTuple**(*value_type: Type[Tuple]*)

Bases: *plug.orm.fields.Field*

A field that holds a named tuple value.

coerce(*value*)
Ensures that an incoming value (i.e., provided to `__set__()`) has the correct type.

> **Caution:** `value` might have any type.

> **Caution:** `value` might be `None`.

deserialize(*value*)
Takes a serialized value and deserializes it.

Parameters **value** – Result of calling *serialize()* at some point.

> **Caution:** `value` might be `None`.

Note: Not using `unpack` because that requires a Registry reference.

---

serialize(*value*)
Returns a primitive version of the field value for hashing, packing, etc.

Parameters **value** – Result of calling *coerce()* at some point.

> **Caution:** `value` might be `None`.

Note: Not using `pack` because that requires a Registry reference.

---

class plug.orm.fields.**Set**(*field_type: Union[plug.orm.fields.Field, typing.Type[plug.orm.fields.Field]] = <class 'plug.orm.fields.Unspecified'>, python_type: Union[typing.Type[typing.MutableSet], typing.Callable[[], typing.MutableSet]] = <class 'set'>*)
Bases: *plug.orm.fields.Field*

Models an unordered collection of unique homogeneous values (roughly equivalent to `typing.MutableSet`).

coerce(*value*)
Ensures that an incoming value (i.e., provided to `__set__()`) has the correct type.

> **Caution:** `value` might have any type.

> **Caution:** `value` might be `None`.

deserialize(*value*)
Takes a serialized value and deserializes it.

Parameters **value** – Result of calling *serialize()* at some point.

> **Caution:** `value` might be `None`.

Note: Not using `unpack` because that requires a Registry reference.

serialize(*value*)
Returns a primitive version of the field value for hashing, packing, etc.

Parameters **value** – Result of calling *coerce()* at some point.

> **Caution:** `value` might be `None`.

Note: Not using `pack` because that requires a Registry reference.

class plug.orm.fields.**Unspecified**(*default=None*)
Bases: *plug.orm.fields.Field*

A field of unspecified type (roughly equivalent to `typing.Any`).

coerce(*value*)
Ensures that an incoming value (i.e., provided to `__set__()`) has the correct type.

> **Caution:** `value` might have any type.

> **Caution:** `value` might be `None`.

deserialize(*value*)
Takes a serialized value and deserializes it.

Parameters **value** – Result of calling *serialize()* at some point.

> **Caution:** `value` might be `None`.

Note: Not using `unpack` because that requires a Registry reference.

serialize(*value*)
Returns a primitive version of the field value for hashing, packing, etc.

Parameters **value** – Result of calling *coerce()* at some point.

> **Caution:** `value` might be `None`.

Note: Not using `pack` because that requires a Registry reference.

7.4 plug.proxy package

7.4.1 Submodules

plug.proxy.http module

plug.proxy.zmq module

class plug.proxy.zmq.**Peer**(*dealer*, *endpoint*)
Bases: tuple

dealer
Alias for field number 0

endpoint
Alias for field number 1

class plug.proxy.zmq.**ZmqDealer**(*ident*, *endpoint*)
Bases: object

close()

create_stream()

write(*data*)

class plug.proxy.zmq.**ZmqProxy**(*bind*, *serializer*, *peer_discovery*, *registry*, *advertise_endpoint=None*, *loop=None*, *heartbeat_duration=None*)
Bases: *plug.abstract.Proxy*

address = None

check_pulses()

connect(*endpoint*, *ident=None*)

handle_meta_message(*ident*, *payload*)

handle_runner_message(*ident*, *payload*)

heartbeat_loop()

make_dealer(*endpoint*)

name = 'zmq'

on_handshake(*ident*, *instance*)

on_handshake_reply(*ident*, *instance*)

on_heartbeat(*ident*, *instance*)

receive_data()

request_object(*ident*, *event*, *request_id*)
When an object hash is specified in the payload instead of an object, we call request_object to request the full object.

send(*ident*, *payload*)

send_handshake(*dealer*)

send_heartbeat()

setup()

teardown()

tick()

class plug.proxy.zmq.**ZmqRouter**(*bind*)
Bases: object

bindings

close()

create_stream()

read()

7.5 plug.storage package

7.5.1 Submodules

plug.storage.file module

class plug.storage.file.**FileStorage**(*registry*, *hashfn*, *serializer*, *path*, *store_n_states=1*)
Bases: *plug.abstract.Storage*

add_unconfirmed_transaction(*transaction: plug.abstract.Transaction*) → None
Persists an unconfirmed transaction to storage.

> **Note:** Unlike state/block, the order that unconfirmed transactions are added does not matter.

block_exists(*block_hash*)

confirm_transactions(*transaction_hashes*)
Marks a collection of unconfirmed Transactions as confirmed, ignoring any that are already confirmed.

> **Tip:** If you can confirm a single transaction more efficiently, handle that in a local single confirm method, if a list of one hash is provided

exists(*filename*)

extract_hash_from_filename(*filename*, *prefix*)

get_block(*height_or_hash*)
Retrieve a block from storage by its height or hash.

Raise

- plug.storage.BlockNotFound if the referenced block does not exist.

get_keyring()
Retrieve the node's keyring from storage.

get_state(*height_or_hash*)
Retrieve a state from storage by its height or hash.

Raise

- `plug.storage.StateNotFound` if the referenced state does not exist.

get_transaction(*transaction_hash: str*, *confirmed: bool = None*)
Retrieve a transaction from storage by its hash.

Raises:

- `plug.storage.TransactionNotFound` if the referenced transaction does not exist.

get_unconfirmed_transaction(*transaction_hash: str*) → plug.abstract.Transaction

read(*filename*)

read_index()

remove_unconfirmed_transaction(*transaction: plug.abstract.Transaction*) → None
Removes the specified transaction from storage.

If the transaction does not exist, this method is a no-op.

set_block(*block*)
Persists a new head block to storage.

set_keyring(*keyring*)
Persists the node's keyring to storage.

set_state(*state*)
Persists a new head state to storage.

setup()

transaction_exists(*transaction_hash*)

unconfirmed_transactions() → AsyncGenerator[Tuple[str, plug.abstract.Transaction], NoneType]
Iterates over unconfirmed transactions and the corresponding hashes.

Each iteration yields a tuple of `(hash, transaction)`.

Important: Order is undefined.

Note: This method usually returns an AsyncGenerator.

See https://www.python.org/dev/peps/pep-0525/ for more information.

---

write(*filename*, *object_*)

write_index(*index*)

plug.storage.sqlite module

class plug.storage.sqlite.**SqliteStorage**(*registry*, *hashfn*, *serializer*, *path*, *store_n_states=1*)
Bases: *plug.abstract.Storage*

SQLITE_MAX_VARIABLE_NUMBER = 999

add_unconfirmed_transaction(*transaction*)
Persists an unconfirmed transaction to storage.

Note: Unlike state/block, the order that unconfirmed transactions are added does not matter.

---

block_exists(*block_hash*)

confirm_transactions(*transaction_hashes*)
Marks a collection of unconfirmed Transactions as confirmed, ignoring any that are already confirmed.

Tip: If you can confirm a single transaction more efficiently, handle that in a local single confirm method, if a list of one hash is provided

---

get_block(*height_or_hash*)
Retrieve a block from storage by its height or hash.

Raise

- `plug.storage.BlockNotFound` if the referenced block does not exist.

get_cursor()

get_keyring()
Retrieve the node's keyring from storage.

get_state(*height_or_hash*)
Retrieve a state from storage by its height or hash.

Raise

- `plug.storage.StateNotFound` if the referenced state does not exist.

get_transaction(*transaction_hash: str*, *confirmed: bool = None*)
Retrieve a transaction from storage by its hash.

Raises:

- `plug.storage.TransactionNotFound` if the referenced transaction does not exist.

remove_unconfirmed_transaction(*transaction*)
Removes the specified transaction from storage.

If the transaction does not exist, this method is a no-op.

static row_factory(*row*)

set_block(*block*)
Persists a new head block to storage.

set_keyring(*keyring*)
Persists the node's keyring to storage.

set_state(*state*)
Persists a new head state to storage.

setup()

transaction_exists(*transaction_hash*)

unconfirmed_transactions()
Iterates over unconfirmed transactions and the corresponding hashes.

Each iteration yields a tuple of `(hash, transaction)`.

Important: Order is undefined.

Note: This method usually returns an AsyncGenerator.

See https://www.python.org/dev/peps/pep-0525/ for more information.

PYTHON MODULE INDEX

p

INDEX

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z