## Espresso API

Reference for REST APIs served by Espresso nodes and query services

### Modularity

The Espresso API comprises several independent modules serving different purposes and requiring different resources. A given node may serve one, or all, or any combination of these modules, depending on its role in the system and the resources available to it. To see a list of the modules available from a particular node, navigate to the root URL of that node's API.

In brief, the available API modules are:

- Status API: node-specific state and consensus metrics
- Catchup API: serves recent consensus state to allow peers to catch up with the network
- Availability API: serves data recorded by the Tiramisu DA layer, such as committed blocks
- Node API: complements the availability API by serving eventually consistent data that is not (yet) necessarily agreed upon by all nodes
- State API: serves consensus state derived from finalized blocks
- Events API: streams events from HotShot
- Submit API: allows users to submit transactions to the public mempool

### Content Types

All APIs support JSON and binary formats in both request and response bodies.

- The JSON format is a straightforward serialization of the data types used internally by consensus nodes. In the future, a formal specification will be published, and the API will conform to that specification.
- The binary format is the bincode serialization of consensus data types, prefixed with an 8-byte version header. In the future, this will be replaced with a binary format with better cross-language support, and the data types will be defined by a published schema, rather than generated from code.

In requests and responses, the JSON format is denoted by the MIME type application/json, and the binary format by application/octet-stream. For requests with a body, the content type of the body must be set via the Content-Type header. The desired content type of the response can be controlled via the Accept header of the request. If the Accept header does not preference either format, JSON will be used by default.

### Server-Hosted Documentation

All Espresso API servers provide self-hosted API documentation, which makes it easy to see exactly what APIs the server supports, and can be easier to browse than these docs. The root URL of an application, e.g. my-server.xyz, lists the supported API modules and versions. Clicking on any API module, or navigating to the root of that API, e.g. my-server.xyz/status, documents the endpoints available in that module.

**Versioning and Stability**

A node may serve multiple major versions of a given API at the same time. The desired version can be selected via a URL prefix. For example, my-server.xyz/v0/status/metrics and my-server.xyz/v1/status/metrics both hit the same endpoint, but in different API versions. A URL with no version segment will get a permanent redirect response to the latest supported version. In this case, /status/metrics would redirect to /v1/status/metrics.

Whenever a breaking change is made to an API, a new major version will be created, and the old version will continue to be served for some time, giving clients time to upgrade to the new version whenever it is convenient for them. Note that non-breaking changes, such as adding new endpoints, may be made in place to existing versions.

To see a list of versions of an API supported by a server, visit the root URL of that server.

**Types**

These types are used in requests and responses across many of the API modules.

**Primitives**

**Integer**

We use integer to represent any JSON integer, with a maximum size of at least 2^63 - 1. Of special note, byte arrays are sometimes represented as arrays of integers ([integer]). When the type [integer] is used as a byte array, each integer therein is restricted to the range [0, 255].

**Hex**

In the following, we use the type hex to indicate a hex-encoded binary string, with 0x prefix.

**Base 64**

In the following, we use the type base64 to indicate a base64-encoded binary string, using the standard base 64 alphabet with padding.

**Tagged Base 64**

Some types use an enhanced tagged base 64 encoding, which consists of a prefix identifying the type of the encoded object, a ~, and then a base 64 string using the URL-safe base 64 alphabet without padding. The base 64 string encodes the binary representation of a typed object, plus a checksum. Because the encoding is URL-safe, these strings can be used not only in request and response bodies, but also in URL paths. The checksum allows the server to provide useful errors if a tagged base 64 string is mistyped or corrupted. The tag makes it easy for a human to tell different types of objects apart.

For example, a transaction hash might be encoded like TX~QDuwVkmexu1fWgJbjxshXcGqXku838Pa9cTn0d-v3hZ-, while a block hash could look like BLOCK~00ISpu2jHbXD6z-BwMkwR4ijGdgUSoXLp_2jIStmqBrD.

We use the type tagged<TAG> to indicate a tagged base 64 object with the given tag, as in tagged<TX> or tagged<BLOCK>.

**NamespaceTable**

Copy

```
{
  "bytes": base64
}
```

**ChainConfig**

A chain config determines properties of consensus, such as the base fee for sequencing and the chain ID. To save space, it can be represented either as the full config object (Left variant below) or as a commitment to the chain config (Right variant). The genesis header will always contain the full config, so clients can fetch the full config from genesis and then compare its commitment against any other header.

Copy

```
{
  "chain_config":
    { "Left": { "chain_id": hex, "max_block_size": integer, "base_fee": hex } }
    | { "Right": tagged<CHAIN_CONFIG> }
}
```

**Header**

Copy

```
{
  "height": integer,
  "timestamp": integer,
  "l1_head": integer,
  "l1_finalized": null | {
    "number": integer,
    "timestamp": hex,
    "hash": hex
  },
  "payload_commitment": tagged<HASH>,
  "builder_commitment": tagged<BUILDER_COMMITMENT>,
  "ns_table": NamespaceTable,
  "block_merkle_tree_root": tagged<MERKLE_COMM>,
  "fee_merkle_tree_root": tagged<MERKLE_COMM>,
  "fee_info": { "account": hex, "amount": hex },
  "chain_config": ChainConfig
}
```

**Payload**

Copy

```
{
  "raw_payload": base64,
  "ns_table": NamespaceTable
}
```

**VidCommon**

Copy

```
{
  "poly_commits": tagged<FIELD>,
  "all_evals_digest": tagged<FIELD>,
  "payload_byte_len": integer,
  "num_storage_nodes": integer,
  "multiplicity": integer
}
```

**Leaf**

Copy

```
{
  "view_number": integer,
  "justify_qc": QC,
  "parent_commitment": string,
  "block_header": Header,
  "proposer_id": string,
}
```

**Transaction**

Copy

```
{
  "namespace": integer,
  "payload": base64
```

}

If using the Rust API, you may notice that namespace is represented by a u64. However, some internal sub-protocols represent the namespace as a u32, and thus the maximum allowable namespace ID is 4294967295 (2^32 - 1). Larger namespace IDs will be rejected on transaction submission.

**MerkleProof**

The low-level proof type for a Merklized data structure. The specific format of this type is not currently specified, but it can be deserialized and interpreted in Rust using the MerkleProof type.

**NsIndex**

The 0-based position of a namespace in a NamespaceTable. The index is a little-endian byte-encoded 4-byte integer, as in [3, 2, 1, 255] (0xff010203).

**NsProof**

A proof that a certain list of transactions corresponds to a certain namespace in a block.

Copy

```
{
  "ns_index": NsIndex,
  "ns_payload": base64, // binary encoding of the namespace data
  "ns_proof": {
    "prefix_elems": tagged<FIELD>,
    "suffix_elems": tagged<FIELD>,
    "prefix_bytes": [integer],
    "suffix_bytes": [integer]
  }
}
```

ns_proof is a low-level range proof in the Espresso ADVZ VID scheme. The details of this proof are out of scope of this document, but this JSON object corresponds to the LargeRangeProofType, and can be manipulated in Rust using that type.

**Status API**

Node-specific state and consensus metrics

This API provides insight into the inner workings of consensus. It is primarily useful to the operator of the node, as many of the metrics provided here make for useful alerts.

**Endpoints**

**GET /status/block-height**

The last known block height of the chain.

**Returns integer**

**GET /status/success-rate**

The view success rate since genesis. This equals the number of views completed divided by the number of successful views, i.e. the block height.

**Returns float**

**GET /status/time-since-last-decide**

The elapsed time, in seconds, since consensus last decided on a block. Useful to alert when consensus is stalled or this node has been disconnected from consensus.

**Returns integer**

**GET /status/metrics**

Exposes all metrics recorded by consensus in Prometheus format.

## Catchup API

Serves recent consensus state to allow peers to catch up with the network

The primary customer of this API is peer consensus nodes who may have recently joined the network or were temporarily disconnected. These nodes need the very latest state, one which hasn't even been finalized yet, in order to start voting and proposing in consensus.

In HotShot, all state required to participate is represented in the form of Merkle trees or Merkle tries, so this API is able to provide select segments of the state with a proof that will convince the client that the returned segment is accurate, as long as they know the corresponding state *commitment*.

### Types

#### Account

Copy

```
{
  // Account balance in WEI. Serialized as a hex string so as not to exceed the
  // range of JSON integers.
  "balance": hex,
  // Merkle proof justifying "balance" relative to the state commitment.
  "proof": {
    "account": hex,
    "proof":
      { "Presence": MerkleProof },
     | { "Absence": MerkleProof }
    }
  }
}
```

**Endpoints**

**GET /catchup/account/:address**

Get the balance of the requested fee account in the latest finalized state.

**Parameters**

| Name | Type | Description |
|---|---|---|
| address | hex | The account (Ethereum address) to look up |

**Returns Account**

**GET /catchup/:view/account/:address**

Get the balance of the requested fee account in the state from the requested consensus view. This is used to fetch the state for *unfinalized* views, to facilitate rapid catchup. If :view has already been finalized, this endpoint may fail with error 404.

**Parameters**

| Name | Type | Description |
|---|---|---|
| view | integer | The view from which to look up the account balance |
| address | hex | The account (Ethereum address) to look up |

**Returns Account**

**GET /catchup/blocks**

Get the Merkle frontier (path to most recently inserted element) of the accumulator of blocks, from the most recently finalized state. This frontier is sufficient to append new blocks, so it is all that is needed for state catchup.

**Returns MerkleProof**

**GET /catchup/:view/blocks**

Get the Merkle frontier (path to most recently inserted element) of the accumulator of blocks, from the requested consensus view. This is used to fetch the state for *unfinalized* views, to facilitate rapid catchup. If :view has already been finalized, this endpoint may fail with error 404.

**Parameters**

| Name | Type | Description |
|---|---|---|
| view | integer | The view from which to look up the frontier |

**Returns MerkleProof**

**Availability API**

Serves data recorded by the Tiramisu DA layer, such as committed blocks

The availability API is the place to get onchain data, like blocks and transactions. It is the primary interface for downstream components like rollups and end users.

The API is designed to be *robust* and *pure*. Robust means that if the node hosting the API misses some data, for example from being offline when a certain block was finalized, it will automatically fetch the missing data from a peer, and will eventually fetch and store all finalized data. Pure means that each endpoint is a pure function -- with the exception of occasionally returning 404 for missing data, each endpoint will always give the same response given the same parameters.

Due to purity, this API provides no aggregate queries, like block or transaction counts, which might change as missing data is fetched. Likewise, every endpoint takes some specification of the exact point in the chain the client is looking for, like a block height or hash. There is no "latest block" query. Thus, most real-world use cases will need to complement the availability API with use of the node API.

**Organization**

While this API has many endpoints, don't be intimidated -- there is a method to the madness. The API is organized around collections of different resources, each of which corresponds to blocks and can be indexed by block height or hash.

**Resources**

- Leaves

- Headers
- Blocks
- Block summaries
- Payloads
- VID common

**Indices**

Each of these resources can be addressed in the following ways:

- <resource>/:height
- <resource>/hash/:hash
- <resource>/payload-hash/:payload-hash

Not all of the indices are implemented for all resources, although in principle they can be. The supported indices are documented below for each endpoint. Future releases will fill in the missing functionality.

Leaves are currently indexed slightly differently from other resources. See documentation on leaf endpoints. Future versions of this API will merge the concept of a leaf and a header, resolving this discrepancy.

In addition, there are endpoints to fetch a range of each resource (<resource>/:form/:until) and to subscribe to a WebSockets stream (/stream/<resource>/:from).

**Types**

**BlockSummary**

Copy

```
{
  "header": Header,
  "hash": tagged<BLOCK>,
  "size": integer,
  "num_transactions": integer
}
```

**BlockResponse**

Copy

```
{
  "header": Header,
  "payload": Payload,
  "hash": tagged<BLOCK>,
  "size": integer,
  "num_transactions": integer
}
```

**LeafResponse**

Copy

```
{
  "leaf": Leaf,
  "qc": QC,
}
```

**PayloadResponse**

Copy

```
{
  "data": Payload,
  "height": integer,
  "size": integer,
  "block_hash": tagged<BLOCK>,
  "hash": tagged<HASH>
}
```

**VidCommonResponse**

Copy

```
{
  "common": VidCommon,
  "block_hash": tagged<BLOCK>,
  "payload_hash": tagged<HASH>
}
```

**Endpoints**

**GET /availability/leaf**

**Paths**

- /availability/leaf/:height
- /availability/leaf/hash/:hash

**Parameters**

| Name | Type | Description |
|---|---|---|
| height | integer | Height of the leaf to fetch |
| hash | tagged<COMMIT> | Hash of the leaf to fetch |

**Returns LeafResponse**

**GET /availability/leaf/:from/:until**

Retrieve a range of consecutive leaves.

**Parameters**

| Name | Type | Description |
|---|---|---|
| from | integer | Height of the first leaf to fetch |
| until | integer | Height just after the last leaf to fetch |

**Returns [LeafResponse]**

**GET /availability/stream/leaves/:height**

This is a WebSockets endpoint. The client must be prepared to upgrade the connection to a WebSockets connection, including the proper headers.

Subscribe to a stream of leaves, in order, starting from the given height.

**Parameters**

| Name | Type | Description |
|---|---|---|

height

integer

Height of the first leaf to yield

**Yields LeafResponse**

**GET /availability/header**

**Paths**

- /availability/header/:height
- /availability/header/hash/:hash
- /availability/header/payload-hash/:payload-hash

**Parameters**

Name

Type

Description

height

integer

Height of the header to fetch

hash

tagged<BLOCK>

Hash of the header to fetch

payload-hash

tagged<HASH>

Hash of the payload of the header to fetch. Note that block payloads are not necessarily unique. If there are multiple blocks whose payload matches this hash, it is unspecified which one is returned.

**Returns Header**

**GET /availability/header/:from/:until**

Retrieve a range of consecutive headers.

**Parameters**

Name

Type

Description

from

integer

Height of the first header to fetch

until

integer

Height just after the last header to fetch

**Returns [Header]**

**GET /availability/stream/headers/:height**

This is a WebSockets endpoint. The client must be prepared to upgrade the connection to a WebSockets connection, including the proper headers.

Subscribe to a stream of headers, in order, starting from the given height.

**Parameters**

| Name | Type | Description |
|---|---|---|
| height | integer | Height of the first header to yield |

**Yields Header**

**GET /availability/block**

**Paths**

- /availability/block/:height
- /availability/block/hash/:hash
- /availability/block/payload-hash/:payload-hash

**Parameters**

| Name | Type | Description |
|---|---|---|
| height | integer | Height of the block to fetch |
| hash | | |

tagged<BLOCK>

Hash of the block to fetch

payload-hash

tagged<HASH>

Hash of the payload of the block to fetch. Note that block payloads are not necessarily unique. If there are multiple blocks whose payload matches this hash, it is unspecified which one is returned.

**Returns BlockResponse**

**GET /availability/block/:from/:until**

Retrieve a range of consecutive blocks.

**Parameters**

Name

Type

Description

from

integer

Height of the first block to fetch

until

integer

Height just after the last block to fetch

**Returns [BlockResponse]**

**GET /availability/stream/blocks/:height**

This is a WebSockets endpoint. The client must be prepared to upgrade the connection to a WebSockets connection, including the proper headers.

Subscribe to a stream of blocks, in order, starting from the given height.

**Parameters**

Name

Type

Description

height

integer

Height of the first block to yield

**Yields BlockResponse**

**GET /availability/block/summary**

**Paths**

- /availability/block/summary/:height

**Parameters**

Name

Type

Description

height

integer

Height of the block to fetch

**Returns BlockSummary**

**GET /availability/block/summaries/:from/:until**

Retrieve a range of consecutive block summaries.

**Parameters**

Name

Type

Description

from

integer

Height of the first block summary to fetch

until

integer

Height just after the last block summary to fetch

**Returns [BlockSummary]**

**GET /availability/payload**

**Paths**

- /availability/payload/:height
- /availability/payload/block-hash/:block-hash
- /availability/payload/hash/:hash

**Parameters**

| Name | Type | Description |
|---|---|---|
| height | integer | Height of the block whose payload should be fetched |
| block-hash | tagged<BLOCK> | Hash of the block whose payload should be fetched |
| hash | tagged<HASH> | Hash of the payload to fetch. Note that block payloads are not necessarily unique. If there are multiple payloads matching this hash, it is unspecified which one is returned. |

**Returns PayloadResponse**

**GET /availability/payload/:from/:until**

Retrieve a range of consecutive payloads.

**Parameters**

| Name | Type | Description |
|---|---|---|
| from | integer | Height of the first payload to fetch |
| until | integer | Height just after the last payload to fetch |

**Returns [PayloadResponse]**

**GET /availability/stream/payloads/:height**

This is a WebSockets endpoint. The client must be prepared to upgrade the connection to a WebSockets connection, including the proper headers.

Subscribe to a stream of payloads, in order, starting from the given height.

**Parameters**

| Name | Type | Description |
|---|---|---|
| height | integer | Height of the first payload to yield |

**Yields PayloadResponse**

**GET /availability/vid/common**

**Paths**

- /availability/vid/common/:height
- /availability/vid/common/hash/:hash
- /availability/vid/common/payload-hash/:payload-hash

**Parameters**

| Name | Type | Description |
|---|---|---|
| height | integer | Height of the block whose VID common data should be fetched |
| hash | tagged<BLOCK> | Hash of the block whose VID common data should be fetched |
| payload-hash | tagged<HASH> | Hash of the payload of the block whose VID common data should be fetched. Note that block payloads are not necessarily unique. If there are multiple blocks whose payload matches this hash, it is unspecified which one is returned. |

**Returns VidCommonResponse**

**GET /availability/stream/vid/common/:height**

This is a WebSockets endpoint. The client must be prepared to upgrade the connection to a WebSockets connection, including the proper headers.

Subscribe to a stream of VID common objects, in order, starting from the given height.

**Parameters**

| Name | Type | Description |
|---|---|---|
| height | integer | Height of the first VID common to yield |

**Yields VidCommonResponse**

**GET /availability/block/:height/namespace/:namespace**

Get the list of transactions in a block from a given namespace, along with a proof that these are only and all such transactions from that block. Note that the proof may be null if transactions is empty, in which case the caller should check the namespace table for the specified block to confirm that :namespace is not present.

**Parameters**

| Name | Type | Description |
|---|---|---|
| height | integer | Height of the block containing the desired namespace |
| namespace | integer | ID of the desired namespace |

**Returns**

Copy

```
{
  "transactions": [Transaction],
  "proof": NsProof | null
}
```

**GET /availability/transaction**

**Paths**

- /availability/transaction/:height/:index

- /availability/transaction/hash/:hash

**Parameters**

Name

Type

Description

height

integer

Height of the block containing the desired transaction

index

integer

0-based position of the desired transaction in its block

hash

tagged<TX>

Hash of the desired transaction. Note that transactions are not necessarily unique. If there are multiple transactions matching this hash, it is unspecified which one is returned.

**Returns**

Copy

```
{
  "transaction": Transaction,
  "hash": tagged<TX>,
  "index": integer,
  "proof": TransactionInclusionProof,
  "block_hash": tagged<BLOCK>,
  "block_height": integer
}
```

The response contains the hash of the transaction, the hash and height of the block that contains it, and its index within that block. It also contains a TransactionInclusionProof, which proves inclusion of this transaction in the block with block_hash. The specific format of this type is not currently specified, but it can be deserialized and interpreted in Rust using the TxInclusionProof type.

**Node API**

Complements the availability API by serving eventually consistent data that is not necessarily agreed upon by all nodes

The availability API provides a pure view of snapshots of the Espresso blockchain at various points in time. Because it strives for robustness and purity, it does not include aggregate statistics like block or transaction counts, which may briefly return incorrect results and will gradually correct themselves as missing data is fetched. The node API does provide this data, making it a useful complement to the availability API.

In other words, while the availability API is a view of the blockchain abstractly, the node API provides information about *this node's* view of the chain, at the present moment in time.

**Endpoints**

**GET /node/block-height**

Get the height of the chain as known to this node. This is equal to one more than the height of the latest known block. It is *not* a count of the blocks in this node's database, as blocks earlier than the latest known block could be missing.

**Returns integer**

**GET /node/transactions/count**

Get the number of finalized transactions. This count may be too low if blocks are missing from the database.

**Returns integer**

**GET /node/payloads/total-size**

Get the total size, in bytes, of all finalized block payloads. This count may be too low if blocks are missing from the database.

**Returns integer**

**GET /node/vid/share**

Get the VID share belonging to this node for a given block.

**Paths**

- /node/vid/share/:height
- /node/vid/share/hash/:hash
- /node/vid/share/payload-hash/:payload-hash

**Parameters**

Name

Type

Description

height

integer

Height of the block whose VID share should be fetched hash

tagged<BLOCK>

Hash of the block whose VID share should be fetched

payload-hash

tagged<HASH>

Hash of the payload of the block whose VID share should be fetched. Note that block payloads are not necessarily unique. If there are multiple blocks whose payload matches this hash, it is unspecified which one is returned.

**Returns VidShare**

The specific format of this type is not currently specified, but it can be deserialized and interpreted in Rust using the VidShare type.

**GET /node/sync-status**

Get the node's progress in syncing with the latest state of blockchain.

If the node is fully synced (that is, all the missing counts are 0 and pruned_height is null or 0) other endpoints in this API should give accurate results.

**Returns**

Copy

```
{
  "missing_blocks": integer,
  "missing_leaves": integer,
  "missing_vid_common": integer,
  "missing_vid_shares": integer,
  "pruned_height": null | integer,
}
```

**GET /node/header/window**

Get a range of consecutive headers by timestamp window.

Returns all available headers, in order, whose timestamps fall between :start (inclusive) and :end (exclusive), or between the block indicated by :height or :hash (inclusive) and :end (exclusive). The response also includes one block before the desired window (unless the window includes the genesis block) and one block after the window. This proves to the client that the server has not omitted any blocks whose timestamps fall within the desired window.

It is possible that not all blocks in the desired window are available when this endpoint is called. In that case, whichever blocks are available are included in the response, and next is null to indicate that the response is not complete. The client can then use one of the /from/ forms of this endpoint to fetch the remaining blocks from where the first response left off, once they become available. If no blocks are available, not even prev, this endpoint will return an error.

**Paths**

- /node/header/window/:start/:end
- /node/header/window/from/:height/:end
- /node/header/window/from/hash/:hash/:end

**Parameters**

Name

Type

Description

start

integer

Timestamp in seconds where the window should start

end

integer

Timestamp in seconds where the window should end

height

integer

Block height where the window should start

hash

tagged<BLOCK>

Block hash where the window should start

**Returns**

Copy

```
{
  "window": Header,
  "prev": null | Header,
  "next": null | Header
}
```

**State API**

Serves consensus state derived from finalized blocks

All state derived from block data is represented in the form of Merkle trees or Merkle tries, so this API is able to provide select segments of the state with a proof that will convince the client that the returned segment is accurate, as long as they know the corresponding state *commitment* (part of each block header).

**Endpoints**

**GET /fee-state**

Get a Merkle proof proving the balance of a certain fee account in a given snapshot of the state. The element in the returned Merkle proof contains the balance of the requested account, or null if the account has no balance.

**Paths**

- /fee-state/:height/:account
- /fee-state/commit/:commit/:account

**Parameters**

Name

Type

Description

height

integer

Block height of the state snapshot to read from

commit

tagged<MERKLE_COMM>

Commitment of the state snapshot to read from

account

hex

Fee account to look up

**Returns MerkleProof**

**GET /fee-state/block-height**

The latest block height for which fee state is available.

Note that this may be less than the block height indicated by other APIs, such as status or node, since the fee state storage is updated asynchronously.

**Returns integer**

**GET /block-state**

Get a Merkle proof proving the inclusion of a certain block at a position in the history. The element in the returned Merkle proof contains the commitment of the block at the requested position in history.

**Paths**

- /block-state/:height/:index
- /block-state/commit/:commit/:index

**Parameters**

Name

Type

Description

height

integer

Block height of the state snapshot to read from

commit

tagged<MERKLE_COMM>

Commitment of the state snapshot to read from

index

integer

Height of the block to look up

**Returns MerkleProof**

**GET /block-state/block-height**

The latest block height for which block state is available.

Note that this may be less than the block height indicated by other APIs, such as status or node, since the block state storage is updated asynchronously.

**Returns integer**

**State API**

Serves consensus state derived from finalized blocks

All state derived from block data is represented in the form of Merkle trees or Merkle tries, so this API is able to provide select segments of the state with a proof that will convince the client that the returned segment is accurate, as long as they know the corresponding state *commitment* (part of each block header).

**Endpoints**

**GET /fee-state**

Get a Merkle proof proving the balance of a certain fee account in a given snapshot of the state. The element in the returned Merkle proof contains the balance of the requested account, or null if the account has no balance.

**Paths**

- /fee-state/:height/:account
- /fee-state/commit/:commit/:account

**Parameters**

Name

Type

Description

height

integer

Block height of the state snapshot to read from

commit

tagged<MERKLE_COMM>

Commitment of the state snapshot to read from

account

hex

Fee account to look up

**Returns MerkleProof**

**GET /fee-state/block-height**

The latest block height for which fee state is available.

Note that this may be less than the block height indicated by other APIs, such as status or node, since the fee state storage is updated asynchronously.

**Returns integer**

**GET /block-state**

Get a Merkle proof proving the inclusion of a certain block at a position in the history. The element in the returned Merkle proof contains the commitment of the block at the requested position in history.

**Paths**

- /block-state/:height/:index
- /block-state/commit/:commit/:index

**Parameters**

| Name | Type | Description |
|---|---|---|
| height | integer | Block height of the state snapshot to read from |
| commit | tagged<MERKLE_COMM> | Commitment of the state snapshot to read from |
| index | integer | Height of the block to look up |

**Returns MerkleProof**

**GET /block-state/block-height**

The latest block height for which block state is available.

Note that this may be less than the block height indicated by other APIs, such as status or node, since the block state storage is updated asynchronously.

**Returns integer**

**Submit API**

Submit transactions to the public mempool

**Endpoints**

**POST /submit/submit**

Returns the hash of the transaction if it was successfully submitted. This does not mean the transaction has yet been sequenced. The user can check for inclusion of the transaction using /availability/transaction/hash/:hash from the availability API.

This endpoint will fail with a 400 status code if the submitted transaction has a namespace ID larger than 4294967295 (2^32 - 1).

**Request Body Transaction**

**Returns tagged<TX>**