herajs Documentation

aergo team and contributors

Mar 17, 2020

User Documentation

Herajs is the JavaScript implementation of hera. What's hera? Hera is a goddess from ancient Greek mythology who protects the Argo ship. We use the name hera for all the aergo clients. Currently, there is herajs, heraj, and herapy. They all implement similar API but with language-specific different styles.

Getting Started

1.1 Installation

Herajs is available on NPM. You can install it using

```
npm install --save @herajs/client
```

1.2 Quick start

```javascript
import { AergoClient } from '@herajs/client';

const aergo = new AergoClient();

aergo.blockchain().then((blockchainState) => {
    console.log(blockchainState.bestHeight, blockchainState.bestBlockHash);
});
```

All async functions return standard promises, so you can also use them like this:

```javascript
import { AergoClient } from '@herajs/client';

const aergo = new AergoClient();

async function update() {
    const blockchainState = await aergo.blockchain();
    console.log(blockchainState.bestHeight, blockchainState.bestBlockHash);
    setTimeout(update, 1000);
}
update();
```

1.3 Supported platforms

Herajs supports both Node.js and Web platforms. However, there is one major implementation difference that you should be aware of.

The Aergo API uses GRPC, which is designed around HTTP 2. Many of its performance advantages come from utilizing HTTP 2. However, currently most browsers do not fully support this protocol. There is a standard called GRPC-WEB that gets around that limitation.

Node.js supports the plain GRPC standard, so that's what you should use in that case. When you import herajs in Node.js environments, it will automatically pick up that implementation.

To use herajs in browser environments either explicitly `import ... from '@herajs/client/dist/herajs.js'` or create an alias in your bundler configuration. Many bundlers like Webpack automatically pick the browser version, so you don't need to configure anything and can just `import ... from '@herajs/client'`.

1.4 Setting a custom provider

By default, AergoClient attempts to connect to the network using a default provider. On Node.js, that is a GRPC connection at localhost:7845. On Web, that is a GRPC-WEB connection at http://localhost:7845.

You can also set a provider manually. In this case, you should choose between GRPC and GRPC-WEB providers.

For Node.js:

```
import { AergoClient, GrpcProvider } from '@herajs/client';

const aergo = new AergoClient({}, new GrpcProvider({url: 'localhost:12345'}));
```

For Web:

```
import { AergoClient, GrpcWebProvider } from '@herajs/client';

const aergo = new AergoClient({}, new GrpcWebProvider({url: 'http://localhost:12345'}
↪));
```

The web transport also supports https.

> **Warning:** To repeat, `GrpcProvider` does not work in browser environments and `GrpcWebProvider` does not work in node environments. When you configure a custom provider, use the correct class.

Reading from blockchain

These are the main methods to read state from the blockchain.

The examples all assume an instance of the AergoClient named aergo.

```
import { AergoClient } from '@herajs/client';
let aergo = new AergoClient();
```

2.1 Get block height

```
let currentHeight;
aergo.blockchain().then(blockchainState => {
    currentHeight = blockchainState.bestHeight;
    console.log(currentHeight);
})

// 3924
```

2.2 Get block

```
aergo.getBlock(currentHeight).then(block => {
    console.log(block);
})
```

```
Block {
    hash: 'VhkA9tF5jHnTC3HJpqKQWovxXDhpnU6PsZUbTV8v3r9',
    header: {
        prevblockhash: '6u1HVdsEhQVwWeP1fZy9U8BDRz9pMpQhTMo6vSKs236g',
        blockno: 3924,
```

```
        timestamp: 1540795459777847800,
        blocksroothash: 'K6MhDUqkA6kTIDw1TLDFQeMxboHEAZ73KfVD3e1H+Ig=',
        txsroothash: 'NNYSS5G0dTPu7/qbk8TlcDiPZ2s/HC5o9GL5MVN+Q7Y=',
        receiptsroothash: 'nyhdK6qewAKRJlxLanoPOoZthnXbhFJsgAe57CpXZGI=',
        confirms: 0,
        pubkey: '',
        sign: '',
        coinbaseaccount: ''
    },
    body: {
        txsList: [
            Tx {
                hash: '9YBYY1onL9RLeEFxV9AdnCReV8i1k689KKsosS3eAx3X',
                nonce: 1,
                from: 'AmNrsAqkXhQfE6sGxTutQkf9ekaYowaJFLekEm8qvDr1RB1AnsiM',
                to: 'AmNrsAqkXhQfE6sGxTutQkf9ekaYowaJFLekEm8qvDr1RB1AnsiM',
                amount: 123,
                payload: '',
                sign:
→'MEUCIQDP3ywVXX1DP42nTgM6cF95GFfpoEcl4D9ZP+MHO7SgoQIgdq2UiEiSp23lcPFzCHtDmh7pVzsow5x1s8p5Kz0aN7I=
→',
                type: 0
            }
            ...
        ]
    }
}
```

2.3 Get transaction

```
aergo.getTransaction('9YBYY1onL9RLeEFxV9AdnCReV8i1k689KKsosS3eAx3X').then(txInfo => {
    console.log(txInfo);
})
```

```
{
    block: { // key only present if transaction has been mined
        hash: '2dR66zrZfo9Je2mavs5RVPXFG4FBpGwarE9PXKyA5bSo',
        idx: 0
    },
    tx: Tx {
        hash: '9YBYY1onL9RLeEFxV9AdnCReV8i1k689KKsosS3eAx3X',
        nonce: 1,
        from: 'AmNrsAqkXhQfE6sGxTutQkf9ekaYowaJFLekEm8qvDr1RB1AnsiM',
        to: 'AmNrsAqkXhQfE6sGxTutQkf9ekaYowaJFLekEm8qvDr1RB1AnsiM',
        amount: 123,
        payload: '',
        sign:
→'MEUCIQDP3ywVXX1DP42nTgM6cF95GFfpoEcl4D9ZP+MHO7SgoQIgdq2UiEiSp23lcPFzCHtDmh7pVzsow5x1s8p5Kz0aN7I=
→',
        type: 0
    }
}
```

2.4 Get account state

```
aergo.getState('AmNrsAqkXhQfE6sGxTutQkf9ekaYowaJFLekEm8qvDr1RB1AnsiM').then(state => {
    console.log(state);
})
```

```
{
    nonce: 1,
    balance: 99999999,
    codehash: ''
}
```

Accounts

There are several methods to create and access accounts.

The examples all assume an instance of the AergoClient named aergo.

```
import { AergoClient } from '@herajs/client';
let aergo = new AergoClient();
```

3.1 Address encoding

A note about address encoding: Internally, addresses are stored and sent as raw bytes. On the client side, they should always be displayed using Base58-check encoded strings (with the prefix 0x42).

Regarding return values, all methods in herajs return *Address* objects that encapsulate raw bytes. String encoding happens automatically when you convert these objects as a string (or call toString() manually).

Regarding parameters, all methods in herajs can take either Base58-check encoded strings or raw byte arrays. It is recommended to just stick to using strings everywhere, but in some cases it may be more efficient to skip the encoding-decoding process.

3.2 Server-side

When you are connected to a local node, you can store your accounts in the node.

3.2.1 Create a new account

```
const password = 'testpass';
aergo.accounts.create(password).then(address => {
    console.log(address);
})
```

```
// Address {}
```

3.2.2 Unlock account

```
const password = 'testpass';
address = 'AmNrsAqkXhQfE6sGxTutQkf9ekaYowaJFLekEm8qvDr1RB1AnsiM';
aergo.accounts.unlock(address, password).then(unlockedAddress => {
    console.log(unlockedAddress);
})

// Address {}
```

Warning: Only create and unlock accounts on access-controlled servers. This method is not safe to use on publicly accessible nodes.

3.3 Client-side

You can also create accounts entirely on the client side without any interaction with the server. For that, you need to create a public-private key pair. As these cryptographic functions take up a lot of space in code, they are located in a separate package from the main client.

You can install it like this:

```
npm install --save @herajs/crypto
```

This package is mostly a thin wrapper around https://github.com/indutny/elliptic for herajs compatability. Please refer to the documentation of elliptic for details.

3.3.1 Create a new account

```
import { createIdentity } from '@herajs/crypto';
const identity = createIdentity();
console.log(identity);
```

```
{
    address: 'AmPwShSoZbAFWWT58yimPBctajMrbjaLWAv88PbgxjvAftPAdLv7',
    publicKey: <EC Point x: ... y: ...>,
    privateKey: <BN: ...>,
    keyPair: <Key priv: ... pub: <EC Point x: ... y: ...> >
}
```

Warning: Be aware that if you keep private keys in the browser, any program that has access to your Javascript environment (for example browser extensions) can potentially read your keys which is dangerous. Take care of storing private keys in a safe way.

Sending transactions

These are the main methods to send transactions to the blockchain.

The examples all assume an instance of the AergoClient named aergo.

```javascript
import { AergoClient } from '@herajs/client';
let aergo = new AergoClient();
```

4.1 Using a server-side account

The easiest way to send transactions is when you are connected to a local node that stores the account you want to use. First unlock an account, then send a transaction using that account.

```javascript
const address = 'AmNrsAqkXhQfE6sGxTutQkf9ekaYowaJFLekEm8qvDr1RB1AnsiM';
aergo.accounts.unlock(address, 'testpass').then(unlockedAddress => {
    const testtx = {
        from: unlockedAddress,
        to: unlockedAddress,
        amount: 123
    };
    aergo.accounts.sendTransaction(testtx).then(txhash => {
        console.log(txhash);
    });
});

// 9YBYY1onL9RLeEFxV9AdnCReV8i1k689KKsosS3eAx3X
```

4.2 Using a client-side account

If you created a public-private key pair on the client side, you can sign and send transactions directly. Remember you need the optional package *@herajs/crypto* for this.

```
import { createIdentity, signTransaction, hashTransaction } from '@herajs/crypto';
const identity = createIdentity();
const tx = {
    nonce: 1,
    from: identity.address,
    to: identity.address,
    amount: 100
};
tx.sign = await signTransaction(tx, identity.keyPair);
tx.hash = await hashTransaction(tx, 'bytes');
aergo.sendSignedTransaction(tx);
```

4.3 Getting transaction status

After submitting a transaction, if it is valid it will be picked up from a mempool and included in a block.

To retrieve the status of a transaction, refer to *Reading from blockchain#Get transaction*. At the moment, you need to poll *getTransaction* until it returns the block that includes the transaction.

Interacting with contracts

For more general information about smart contracts, please refer to the Aergo documentation.

This guide uses Javascript's await/async syntax. Of course, you can always write code like `await aergo.getTransactionReceipt()` as `aergo.getTransactionReceipt().then(callbackFunction)`.

5.1 Deployment

To interact with a smart contract, it needs to be deployed on the blockchain. To deploy a contract, you send a transaction with the contract code as the payload to the null (empty) account.

1. Compile your contract source code into the payload format. This is not supported by Herajs, please refer to the Aergo documentation. For example, you can run the command `aergoluac --payload contract.lua > contract.txt`.

2. Setup the contract object and build a deployment transaction:

```
import { AergoClient, Contract } from '@herajs/client';
const aergo = new AergoClient();

const myAddress = 'Am....'; // Enter your account address or name
const contractCode = 'output from aergoluac --payload';
const contract = Contract.fromCode(contractCode);
const tx = {
    from: myAddress,
    to: null,
    payload: contract.asPayload(),
};
```

3. Unlock account and deploy contract:

```
await aergo.accounts.unlock(myAddress, 'your password');

const deployTxhash = await aergo.accounts.sendTransaction(tx);
```

4. Check the transaction receipt for the created contract address or any error:

```
const receipt = await aergo.getTransactionReceipt(deployTxhash);
const contractAddress = receipt.contractaddress;
console.log(receipt);
/*
{
    status: 'CREATED',
    contractaddress: 'Am.....'
}
*/
```

5.1.1 How can I pass constructor paremeters?

The `Contract.asPayload()` function accepts an optional parameter to pass a list of constructor parameters to the contract. Example:

```
const tx = {
    from: myAddress,
    to: null,
    payload: contract.asPayload([1, 2, 3]),
};
```

5.2 Setup

If you have the address of a contract instance and its ABI, you can setup communication with the contract like this.

```
import { AergoClient, Contract } from '@herajs/client';
const aergo = new AergoClient();

import contractAbi from './contract.abi.json';
const contract = Contract.fromAbi(contractAbi).setAddress(contractAddress);
```

If you don't have the ABI, it is possible to retrieve it from the blockchain like this:

```
const abi = await aergo.getABI(contractAddress);
const contract = Contract.atAddress(contractAddress);
contract.loadAbi(await aergo.getABI(contractAddress));
```

5.3 Call

Calls are contract executions on the blockchain, i.e. transactions with a payload and a result.

Once you have your `contract` instance set up, you can call contract methods like this.

```
// Build a transaction
const callTx = contract.someContractMethod().asTransaction({
    from: myAddress
});

// Send the transaction
```

```
const calltxhash = await aergo.accounts.sendTransaction(callTx);

// Wait until the transaction is executed and included in a block, then get the␣
↪receipt
const calltxreceipt = await aergo.getTransactionReceipt(calltxhash);
console.log(calltxreceipt);
/*
{
    status: "SUCCESS",
    result: "json string"
}
*/
```

5.4 Query

Queries are static contract executions, i.e. they return a result from your local node without creating changes on the blockchain. Thus, they don't require a transaction.

```
const result = await aergo.queryContract(contract.someContractMethod());
console.log(result);
```

5.5 Events

Contracts can log events during execution. This is the preferred way to notify the outside world of important state changes. It is easy to request events using the *AergoClient.getEvents()* method.

```
const result = await aergo.getEvents({
    address: contractAddress
});
/*
[
    {
        eventName: '..',
        address: 'Am....',
        args: [ 1, 2, 3 ]
    }
]
*/
```

5.5.1 Filter events

You can also filter events in a fine grained way. Check *FilterInfo()* for all available options.

```
const result = await aergo.getEvents({
    address: contractAddress,
    args: [1] // or new Map([[1, 2]]) to only filter for the second argument
});
/*
[
```

```
    {
        eventName: '..',
        address: 'Am....',
        args: [ 1, 2, 3 ]
    }
]
*/
```

5.5.2 Stream events

Events can also be streamed in real time using *AergoClient.getEventStream()*. The options are the same as for getEvents, but instead of retrieving all previous events, this creates a stream that receives all future events as they get created.

```javascript
const stream = aergo.getEventStream({
    address: contractAddress
});
stream.on('data', (event) => {
    console.log(event);
    /*
    {
        eventName: '..',
        address: 'Am....',
        args: [ 1, 2, 3 ]
    }
    */
});
// Call stream.cancel(); when you don't need it any more to free resources on the
→full node.
```

Examples

Looking at example code can be a great way to learn.

- [aergoio/herajs-example](#)
- [herajs tests](#)

More examples will be added in the future.

Top-level exports

You can import these using `import { x } from '@herajs/client'`.

- AergoClient (also exported as default)
- GrpcProvider (only from Node.js build)
- GrpcWebProvider (only from Web build)
- Contract
- Address
- Amount
- Tx
- constants

AergoClient

class AergoClient(*config*, *provider*)
: *exported from* `client/index`

  Main aergo client controller.

  Create a new auto-configured client with:

  ```
  import AergoClient from '@herajs/client';
  const aergo = new AergoClient();
  ```

  Arguments

  - **config** (*<TODO>*) –

  - **provider** (*any*) –

AergoClient.**accounts**
: **type:** accounts/index.Accounts

AergoClient.**client**
: **type:** any

AergoClient.**config**
: **type:** object

AergoClient.**target**
: **type:** string

AergoClient.**blockchain**()
: Request current status of blockchain.

  Returns **Promise<client/types.BlockchainResult>** – an object detailing the current status

AergoClient.**defaultProvider**()

  Returns **any** –

`AergoClient.`**`getABI`**(*address*)

> Query contract ABI
>
> > **Arguments**
> >
> > - **address** (`models/address.AddressInput`) – of contract
> >
> > **Returns** **Promise<GrpcABI.AsObject>** – abi

`AergoClient.`**`getAccountVotes`**(*address*)

> Return the top voted-for block producer or system parameter
>
> > **Arguments**
> >
> > - **address** (`models/address.AddressInput`) – string
> >
> > **Returns** **Promise<any>** –

`AergoClient.`**`getBlock`**(*hashOrNumber*)

> Retrieve information about a block.
>
> > **Arguments**
> >
> > - **hashOrNumber** (`string|number`) – either 32-byte block hash encoded as a bs58 string or block height as a number.
> >
> > **Returns** **Promise<models/block.Block>** – block details

`AergoClient.`**`getBlockBody`**(*hashOrNumber*, *offset*, *size*)

> Get the transactions of a block in a paged manner
>
> > **Arguments**
> >
> > - **hashOrNumber** (`string|number`) –
> > - **offset** (`number`) –
> > - **size** (`number`) –
> >
> > **Returns** **Promise<client/types.BlockBodyPaged>** –

`AergoClient.`**`getBlockHeaders`**(*hashOrNumber*, *size*, *offset*, *desc*)

> Retrieve the last n blocks, beginning from given block
>
> > **Arguments**
> >
> > - **hashOrNumber** (`string|number`) – either 32-byte block hash encoded as a bs58 string or block height as a number.
> > - **size** (`number`) – number of blocks to return
> > - **offset** (`number`) – number of blocks to skip
> > - **desc** (`boolean`) – order of blocks
> >
> > **Returns** **Promise<models/block.Block[]>** – list of block headers (blocks without body)

`AergoClient.`**`getBlockMetadata`**(*hashOrNumber*)

> Retrieve block metadata (excluding body).
>
> > **Arguments**
> >
> > - **hashOrNumber** (`string|number`) – either 32-byte block hash encoded as a bs58 string or block height as a number.
> >
> > **Returns** **Promise<models/blockmetadata.BlockMetadata>** – block metadata

AergoClient.**getBlockMetadataStream**()
    Returns a stream of block metadata

        Returns   client/types.Stream<models/blockmetadata.BlockMetadata> –

AergoClient.**getBlockStream**()

        Returns   client/types.Stream<models/block.Block> –

AergoClient.**getChainIdHash**(*enc*)
    Request chain id hash. This automatically gathers the chain id hash from the current node if not specified.

        Arguments

            • **enc**(*string*) – set to 'base58' to retrieve the hash encoded in base58. Otherwise returns
              a Uint8Array.

        Returns   Promise<Uint8Array|string> – Uint8Array by default, base58 encoded string if enc
              = 'base58'.

AergoClient.**getChainInfo**()
    Request current status of blockchain.

        Returns   Promise<models/chaininfo.ChainInfo> – an object detailing the current status

AergoClient.**getConfig**()

        Returns   any –

AergoClient.**getConsensusInfo**()
    Return consensus info. The included fields can differ by consensus type.

        Returns   Promise<client/types.ConsensusInfoResult> –

AergoClient.**getEventStream**(*filter*)
    Returns a stream that yields new events matching the specified filter in real-time.

```
const stream = aergo.getEventStream({
    address: 'Am....'
});
stream.on('data', (event) => {
   console.log(event);
   stream.cancel();
});
```

        Arguments

            • **filter**(*Partial<models/filterinfo.FilterInfo>*) – *FilterInfo()*

        Returns   client/types.Stream<models/event.Event> – event stream

AergoClient.**getEvents**(*filter*)
    Query contract state This only works vor variables explicitly defines as state variables.

        Arguments

            • **filter**(*Partial<models/filterinfo.FilterInfo>*) – *FilterInfo()*

        Returns   Promise<models/event.Event[]> – list of events

AergoClient.**getNameInfo**(*name*)
    Return information for account name

        Arguments

- **name** (*string*) –

    Returns **Promise<client/types.NameInfoResult>** –

`AergoClient.getNodeState`(*component*, *timeout*)

    Request current status of node.

    Arguments

    - **component** (*string*) –

    - **timeout** (*number*) –

    Returns **Promise<any>** – an object detailing the state of various node components

`AergoClient.getNonce`(*address*)

    Retrieve account's most recenlty used nonce. This is a shortcut function as the same information is also returned by getState.

    Arguments

    - **address** (*models/address.AddressInput*) – Account address encoded in Base58check

    Returns **Promise<number>** – account state

`AergoClient.getPeers`(*showself*, *showhidden*)

    Get list of peers of connected node

    Arguments

    - **showself** (*boolean*) –

    - **showhidden** (*boolean*) –

    Returns **Promise<models/peer.Peer[]>** –

`AergoClient.getServerInfo`(*keys*)

    Return server info

    Arguments

    - **keys** (*string[]*) –

    Returns **Promise<client/types.ServerInfoResult>** –

`AergoClient.getStaking`(*address*)

    Return information for account name

    Arguments

    - **address** (*models/address.AddressInput*) – Account address encoded in Base58check

    Returns **Promise<Staking.AsObject>** –

`AergoClient.getState`(*address*)

    Retrieve account state, including current balance and nonce.

    Arguments

    - **address** (*models/address.AddressInput*) – Account address encoded in Base58check

    Returns **Promise<models/state.State>** – account state

`AergoClient.getTopVotes`(*count*, *id*)

    Return the top {count} result for a vote

Arguments

- **count** (*number*) – number

- **id** (*string*) – vote identifier, default: voteBP

Returns **Promise<any>** –

AergoClient.**getTransaction**(*txhash*)

Get transaction information in the aergo node. If transaction is in the block return result with block hash and index.

Arguments

- **txhash** (*string*) – transaction hash

Returns **Promise<client/types.GetTxResult>** – transaction details, object of tx: <Tx> and block: { hash, idx }

AergoClient.**getTransactionReceipt**(*txhash*)

Retrieve the transaction receipt for a transaction

Arguments

- **txhash** (*string*) – transaction hash

Returns **Promise<client/types.GetReceiptResult>** – transaction receipt

AergoClient.**grpcMethod**(*method*)

Arguments

- **method** (*Function*) –

Returns **<TODO>** –

AergoClient.**isConnected**()

Returns **boolean** –

AergoClient.**queryContract**(*functionCall*)

Query contract ABI

Arguments

- **functionCall** (*models/contract.FunctionCall*) – call details

Returns **Promise<any>** – result of query

AergoClient.**queryContractState**(*stateQuery*)

Query contract state. This only works for variables explicitly defines as state variables. Throws when contract do not exist, or when variable does not exist when requesting single key.

Arguments

- **stateQuery** (*models/contract.StateQuery*) – query details obtained from contract.queryState()

Returns **Promise<client/types.JsonData|models/statequeryproof.BasicType>** – result of query: single value if requesting one key, list of values when requesting multiple keys.

AergoClient.**queryContractStateProof**(*stateQuery*)

Query contract state, including proofs. This only works vor variables explicitly defines as state variables.

Arguments

- **stateQuery** (*models/contract.StateQuery*) – query details obtained from contract.queryState()

> > **Returns** **Promise<models/statequeryproof.StateQueryProof>** – result of query, including account and var proofs

`AergoClient.`**`sendSignedTransaction`**(*tx*)

  Send a signed transaction to the network.

>   **Arguments**
>
>   * **`tx`** (*any*) – signed transaction
>
>   **Returns** **Promise<string>** – transaction hash

`AergoClient.`**`setChainIdHash`**(*hash*)

  Set the chain id hash to use for subsequent transactions.

>   **Arguments**
>
>   * **`hash`** (*string|Uint8Array*) – string (base58 encoded) or byte array

`AergoClient.`**`setDefaultLimit`**(*limit*)

  Set the default gas limit to use for transactions that do not define their own.

>   **Arguments**
>
>   * **`limit`** (*number*) –

`AergoClient.`**`setProvider`**(*provider*)

  Set a new provider

>   **Arguments**
>
>   * **`provider`** (*any*) –

`AergoClient.`**`defaultProviderClass`**

  type: <TODO>

`AergoClient.`**`platform`**

  type: string

class `Stream`(*T*)

  interface, *exported from* `client/types`

>   **Arguments**
>
>   * **`T`** –

`Stream.`**`_stream`**

  type: any

`Stream.`**`cancel`**()

`Stream.`**`on`**(*eventName*, *callback*)

>   **Arguments**
>
>   * **`eventName`** (*string*) –
>
>   * **`callback`** (*<TODO>*) –

AergoClient.accounts

class Accounts(*aergo*)
   exported from `accounts/index`

   Accounts controller. It is exposed at *aergoClient.accounts*.

   Arguments

   - **aergo** (`any`) –

`Accounts.`**aergo**
   type: any

`Accounts.`**client**
   type: any

`Accounts.`**create**(*passphrase*)
   Create a new account in the node.

   Arguments

   - **passphrase** (`string`) –

   Returns **Promise<models/address.Address>** – newly created account address

`Accounts.`**get**()
   Get list of accounts.

   Returns **Promise<models/address.Address[]>** – list of account addresses

`Accounts.`**lock**(*address*, *passphrase*)
   Lock account.

   Arguments

   - **address** (`models/address.Address|string`) –

   - **passphrase** (`string`) –

   Returns **Promise<models/address.Address>** – locked account address

Accounts.**sendTransaction**(*tx*)

> Convenience method to send transaction from account. This method automatically retrieves the nonce, signs the transaction, and sends it to the network.

> > **Arguments**

> > > • **tx** (*any*) – transaction data

> > **Returns** **Promise<string>** – transaction hash

Accounts.**signTransaction**(*_tx*)

> Sign transaction.

> > **Arguments**

> > > • **_tx** (*models/tx.Tx|object*) –

> > **Returns** **Promise<models/tx.Tx>** – transaction data including signature

Accounts.**unlock**(*address*, *passphrase*)

> Unlock account.

> > **Arguments**

> > > • **address** (*models/address.Address|string*) –

> > > • **passphrase** (*string*) –

> > **Returns** **Promise<models/address.Address>** – unlocked account address

Providers

class GrpcProvider(*config*)
: *exported from* `providers/grpc`

Provider for standard GRPC connections over HTTP2. This is only compatible with Node.js environments.

```
import { GrpcProvider } from '@herajs/client';
const provider = new GrpcProvider({url: 'localhost:7845'});
```

Arguments

- **config**(*providers/grpc.GrpcProviderConfig*) –

class GrpcWebProvider(*config*)
: *exported from* `providers/grpc-web`

Provider for GRPC-WEB connections over HTTP. This is compatible with Web browsers. Note that the transport is considerably slower than over standard GRPC.

```
import { GrpcWebProvider } from '@herajs/client';
const provider = new GrpcWebProvider({url: 'http://localhost:7845'});
```

Arguments

- **config**(*providers/grpc-web.GrpcWebProviderConfig*) –

Address

class Address(*address*)
    exported from `models/address`

    A wrapper around addresses. Internally addresses are stored and sent as raw bytes, but client-side they are displayed as base58-check encoded strings. The encoding requires some computation, so you should only convert address objects to strings when needed.

        Arguments

            • **address** (*models/address.AddressInput*) –

`Address.`**encoded**
    type: string

`Address.`**isName**
    type: boolean

`Address.`**value**
    type: Buffer

`Address.`**asBytes**()

        Returns Uint8Array –

`Address.`**decode**(*bs58string*)

        Arguments

            • **bs58string** (*string*) –

        Returns Buffer –

`Address.`**encode**(*byteArray*)

        Arguments

            • **byteArray** (*any*) –

        Returns string –

`Address.`**equal**(*_otherAddress*)

Arguments

- **_otherAddress** (*models/address.AddressInput*) –

Returns **boolean** –

Address.**isEmpty**()

Returns **boolean** –

Address.**isSystemAddress**()

Returns **boolean** –

Address.**isSystemName**(*name*)

Arguments

- **name** (*string*) –

Returns **boolean** –

Address.**setSystemAddresses**(*addresses*)

Arguments

- **addresses** (*string[]*) –

Address.**toJSON**()

Returns **string** –

Address.**toString**()

Returns **string** –

Address.**valueEqual**(*a*, *b*)

Arguments

- **a** (*Buffer*) –

- **b** (*Buffer*) –

Returns **boolean** –

Amount

class Amount(*value*, *unit*, *newUnit*)

exported from `models/amount`

A wrapper around amounts with units. Over the network, amounts are sent as raw bytes. In the client, they are exposed as BigInts, but also compatible with plain strings or numbers (if smaller than 2^31-1) Uses 'aergo' as default unit when passing strings or numbers. Uses 'aer' as default unit when passing BigInts, buffers or byte arrays. Whenever you pass amounts to other functions, they will try to coerce them to BigInt using this class.

> **Arguments**
>
> - **value** (`models/amount.AmountArg|Buffer|Uint8Array`) –
>
> - **unit** (`string`) –
>
> - **newUnit** (`string`) –

`Amount.`**unit**

> **type:** string

`Amount.`**value**

> **type:** Readonly<JSBI>

`Amount.`**_valueFromString**(*value*, *unit*)

> **Arguments**
>
> - **value** (`string`) –
>
> - **unit** (`string`) –
>
> **Returns JSBI** –

`Amount.`**add**(*otherAmount*)

> Add another amount to amount. If otherAmount has no unit, assumes unit of this amount. 10 aergo + 10 = 20 aergo 10 aer + 10 = 20 aer 10 aergo + 10 aer = 10.00000000000000001 aergo
>
> **Arguments**
>
> - **otherAmount** (`models/amount.AmountArg`) –

> **Returns models/amount.Amount** –

Amount.**asBytes**()
> Returns value as byte buffer

> > **Returns Buffer** –

Amount.**compare**(*otherAmount*)
> Compare this amount with other amount. If otherAmount has no unit, assumes unit of this amount. this > other -> +1 this < other -> -1 this == other -> 0

> > **Arguments**

> > > • **otherAmount** (*models/amount.AmountArg*) –

> > **Returns number** –

Amount.**div**(*otherAmount*)
> Divide amount by another amount. Warning: double check your units. The division is based on the aer value, so if your otherAmount has a unit, it will be converted to aer. This function tries to do the right thing in regards to dividing units: 10 aergo / 10 = 1 aergo (keep unit) 10 aergo / 10 aergo = 1 (unit-less) 1 aer / 2 aer = 0 (truncation of sub 1 aer amount)

> > **Arguments**

> > > • **otherAmount** (*models/amount.AmountArg*) –

> > **Returns models/amount.Amount** –

Amount.**equal**(*otherAmount*)
> Return true if otherAmount is equal to this amount.

> > **Arguments**

> > > • **otherAmount** (*models/amount.AmountArg*) –

> > **Returns boolean** –

Amount.**formatNumber**(*unit*)

> > **Arguments**

> > > • **unit** (*string*) –

> > **Returns string** –

Amount.**moveDecimalPoint**(*str*, *digits*)
> Move decimal point in string by digits, positive to the right, negative to the left. This extends the string if necessary. Example: ("0.0001", 4 => "1"), ("0.0001", -4 => "0.00000001")

> > **Arguments**

> > > • **str** (*string*) –

> > > • **digits** (*number*) –

> > **Returns string** –

Amount.**mul**(*otherAmount*)
> Multiply amount by another amount. Warning: double check your units. The multiplication is based on the aer value, so if your otherAmount has a unit, it will be converted to aer. However, while the value is correct, there's no way to display unit^2. 10 aergo * 10 aergo = 10 * 10^18 aer * 10 * 10^18 aer = 100 * 10^36 aer = 100 * 10^18 aergo 10 aergo * 10 = 10 * 10^18 aer * 10 = 100 * 10^18 aer = 100 aergo

> > **Arguments**

> > > • **otherAmount** (*models/amount.AmountArg*) –

> **Returns models/amount.Amount** –

`Amount.sub`(*otherAmount*)
> Subtract another amount from amount. If otherAmount has no unit, assumes unit of this amount. 10 aergo - 5 = 5 aergo 10 aer - 5 = 5 aer 1 aer - 1 aergo = -999999999999999999 aer

> **Arguments**

>> • **otherAmount** (`models/amount.AmountArg`) –

> **Returns models/amount.Amount** –

`Amount.toJSBI`(*arg*, *defaultUnit*)
> Convert arg into JSBI value Can optionally provide a defaultUnit that is used if arg does not contain a unit.

> **Arguments**

>> • **arg** (`models/amount.AmountArg`) –

>> • **defaultUnit** (`string`) –

> **Returns JSBI** –

`Amount.toJSON`()
> JSON.stringifes to string with unit aer, which can be easily deserialized by new Amount(x)

> **Returns string** –

`Amount.toString`()
> Returns formatted string including unit

> **Returns string** –

`Amount.toUnit`(*unit*)
> Convert to another unit

> **Arguments**

>> • **unit** (`string`) – string (aer, gaer, aergo)

> **Returns models/amount.Amount** –

Contract

class Contract(*data*)

  exported from `models/contract`

  Smart contract interface. You usually instantiante this class by using one of the static methods. Most of the instance methods return the contract so they can be chained. When an ABI is loaded, its functions will be added to the instance and can be called directly. ABI functions return *FunctionCall* objects that can be queried or called.

```
import { Contract } from '@herajs/client';
const contract = Contract.fromAbi(abi).setAddress(address);
aergo.queryContract(contract.someAbiFunction()).then(result => {
    console.log(result);
})
```

  Arguments

   - **data** (*Partial<models/contract.Contract>*) –

`Contract.`**address**

  type: models/address.Address

`Contract.`**code**

  type: Buffer

`Contract.`**functions**

  type: any

`Contract.`**asPayload**(*args*)

  Return contract code as payload for transaction

  Arguments

   - **args** (*Array<models/contract.PrimitiveType>*) –

  Returns Buffer – a byte buffer

`Contract.`**`atAddress`**(*address*)
Create contract instance and set address

> **Arguments**
>
> > • **address** (`models/address.Address`) –
>
> **Returns** **models/contract.Contract** – contract instance

`Contract.`**`decodeCode`**(*bs58checkCode*)

> **Arguments**
>
> > • **bs58checkCode** (`string`) –
>
> **Returns** **Buffer** –

`Contract.`**`encodeCode`**(*byteArray*)

> **Arguments**
>
> > • **byteArray** (`Buffer`) –
>
> **Returns** **string** –

`Contract.`**`fromAbi`**(*abi*)
Create contract instance from ABI

> **Arguments**
>
> > • **abi** (`any`) – parsed JSON ABI
>
> **Returns** **models/contract.Contract** – contract instance

`Contract.`**`fromCode`**(*bs58checkCode*)
Create contract instance from code

> **Arguments**
>
> > • **bs58checkCode** (`any`) – base58-check encoded code
>
> **Returns** **models/contract.Contract** – contract instance

`Contract.`**`loadAbi`**(*abi*)
Load contract ABI

> **Arguments**
>
> > • **abi** (`any`) – parsed JSON ABI
>
> **Returns** **models/contract.Contract** – contract instance

`Contract.`**`queryState`**(*keys*, *compressed*, *root*)
Create query object to query contract state.

> **Arguments**
>
> > • **keys** (`string|models/contract.BufferLike|string[]|models/contract.BufferLike[]`) – list of keys, either strings or Buffer-like byte arrays
> >
> > • **compressed** (`boolean`) – return compressed proof (default: false)
> >
> > • **root** (`Uint8Array`) – root hash
>
> **Returns** **models/contract.StateQuery** –

`Contract.`**`setAddress`**(*address*)
Set address of contract instance

Arguments

- **address** (*models/address.Address|string*) –

Returns **models/contract.Contract** – contract instance

class FunctionCall(*contractInstance*, *definition*, *args*)
exported from `models/contract`

Data structure for contract function calls. You should not need to build these yourself, they are returned from contract instance functions and can be passed to the client.

Arguments

- **contractInstance** (*any*) –

- **definition** (*any*) –

- **args** (*any*) –

`FunctionCall.`**args**
type: Array<models/contract.PrimitiveType>

`FunctionCall.`**contractInstance**
type: models/contract.Contract

`FunctionCall.`**definition**
type: Function.AsObject

`FunctionCall.`**asQueryInfo**()
Generate query info that can be passed to the API. You usually do not need to call this function yourself, *AergoClient.queryContract()* takes care of that.

```
import { Contract } from '@herajs/client';
const contract = Contract.fromAbi(abi).atAddress(address);
const functionCall = contract.someAbiFunction();
aergo.queryContract(functionCall).then(result => {
    console.log(result);
})
```

Returns **<TODO>** – queryInfo data

`FunctionCall.`**asTransaction**(*extraArgs*)
Generate transaction object that can be passed to `aergoClient.accounts.sendTrasaction()`

```
import { Contract } from '@herajs/client';
const contract = Contract.fromAbi(abi).atAddress(address);
const functionCall = contract.someAbiFunction();
aergo.accounts.sendTransaction(functionCall.asTransaction({
    from: myAddress
})).then(result => {
    console.log(result);
})
```

Arguments

- **extraArgs** (*any*) –

Returns **any** – transaction data

`FunctionCall.`**toGrpc**()

> **Returns Query** –

class StateQuery(*contractInstance*, *storageKeys*, *compressed*, *root*)
  exported from `models/contract`

Query contract state directlty without using ABI methods.

```
import { Contract } from '@herajs/client';
const contract = Contract.fromAbi(abi).atAddress(address);
const query = contract.queryState('stateVariableName');
aergo.queryContractState(query).then(result => {
    console.log(result);
})
```

Arguments

- **contractInstance** (*models/contract.Contract*) –

- **storageKeys** (*string[]|models/contract.BufferLike[]*) –

- **compressed** (*boolean*) –

- **root** (*Uint8Array*) –

StateQuery.**compressed**
  type: boolean

StateQuery.**contractInstance**
  type: models/contract.Contract

StateQuery.**root**
  type: Uint8Array|undefined

StateQuery.**storageKeys**
  type: string[]|models/contract.BufferLike[]

StateQuery.**toGrpc**()

> **Returns GrpcStateQuery** –

FilterInfo

class FilterInfo(*data*)
    exported from `models/filterinfo`

        Arguments

            • **data** (*Partial<models/filterinfo.FilterInfo>*) –

    `FilterInfo.`**address**
        type: models/address.Address

    `FilterInfo.`**args**
        type: Array<models/contract.PrimitiveType>|Map<number|string,models/contract.PrimitiveType>

    `FilterInfo.`**blockfrom**
        type: number

    `FilterInfo.`**blockto**
        type: number

    `FilterInfo.`**desc**
        type: boolean

    `FilterInfo.`**eventName**
        type: string

    `FilterInfo.`**fromGrpc**(*grpcObject*)

        Arguments

            • **grpcObject** (*GrpcFilterInfo*) –

        Returns models/filterinfo.FilterInfo –

    `FilterInfo.`**toGrpc**()

        Returns FilterInfo –

Tx

class Tx(*data*)
　　exported from `models/tx`

　　Class for converting transaction data to and from network representation. You usually don't need to interact with this class manually, you can pass simple JSON objects. This class is used when passing transaction data to client methods.

　　　　Arguments

　　　　　　• **data** (*Partial<models/tx.Tx>*) –

　　`Tx.`**amount**
　　　　type: models/amount.Amount

　　`Tx.`**chainIdHash**
　　　　type: string

　　`Tx.`**from**
　　　　type: models/address.Address

　　`Tx.`**hash**
　　　　type: string

　　`Tx.`**limit**
　　　　type: number

　　`Tx.`**nonce**
　　　　type: number

　　`Tx.`**payload**
　　　　type: Uint8Array

　　`Tx.`**price**
　　　　type: models/amount.Amount

　　`Tx.`**sign**
　　　　type: string

Tx.**to**
>   **type:** models/address.Address

Tx.**type**
>   **type:** models/tx.TxTypeValue

Tx.**fromGrpc**(*grpcObject*)

>   **Arguments**
>
>   • **grpcObject** (*GrpcTx*) –
>
>   **Returns models/tx.Tx** –

Tx.**toGrpc**()

>   **Returns Tx** –

Tx.**Type**
>   **type:** TxTypeMap

>   Map of tx types.   Use as Tx.Type.NORMAL, Tx.Type.GOVERNANCE, Tx.Type.REDEPLOY,
>   Tx.Type.FEEDELEGATION

CHAPTER 16

Indices and tables

- genindex

Index