herajs Documentation

aergo team and contributors

Mar 13, 2020

API Reference

CHAPTER 1

Keys

1.1 Generating random private key

createIdentity()
Shortcut function to create a new random private key and return keys and address as encoded strings.

Returns **keys.Identity** – identity including address and keys

1.2 Importing private key

identityFromPrivateKey(*privKeyBytes*)
Returns identity associated with private key

Arguments

- **privKeyBytes** (*Uint8Array*) –

Returns **keys.Identity** – identity including address and keys

1.3 Encryption

decryptPrivateKey(*encryptedBytes*, *password*)
Decrypt an AES_GCM encrypted private key

Arguments

- **encryptedBytes** (*Uint8Array*) –
- **password** (*string*) –

Returns **Uint8Array** – decrypted private key bytes

encryptPrivateKey(*clearBytes*, *password*)
Encrypt a private key using AES_GCM

Arguments

- **clearBytes** (*Uint8Array*) –
- **password** (*string*) –

Returns **Uint8Array** – encrypted private key bytes

1.4 Public keys and addresses

In Aergo, addresses are generated directly from public keys. Because of that, it is easy to convert between the two.

publicKeyFromAddress(*address*)
Retrieve public key from address

Arguments

- **address** (*string*) –

Returns **KeyPair** – key pair (with missing private key)

addressFromPublicKey(*publicKey*)
Encode public key as address

Arguments

- **publicKey** (*any*) –

Returns **string** – base58check encoded address

CHAPTER 2

Keys from seed

Key generation from seeds follows BIP39/BIP44.

The derivation path used here by default is `m/44'/441'/0'/0/n`, but you can supply a custom one.

2.1 Mnemonic seed

BIP39 mnemonic seed phrases

privateKeysFromMnemonic (*mnemonic*, *options*)
Returns n private keys derived from mnemonic

Arguments

- **mnemonic** (*string*) –
- **options** (*seed.Options*) – (optional) { count: number, hdpath: string }

Returns **Promise<Buffer[]>** –

privateKeyFromMnemonic (*mnemonic*, *options*)
Returns the first private key derived from mnemonic

Arguments

- **mnemonic** (*string*) –
- **options** (*seed.Options*) – (optional) { hdpath: string }

Returns **Promise<Buffer>** –

generateMnemonic (*strength*, *rng*, *wordlist*)
Generate random mnemonic

Arguments

- **strength** (*undefined|number*) – in bits, default 128
- **rng** (*undefined|<TODO>*) – optional, function to generate random bots

- **wordlist** (*string[]*) – optional, custom wordlist

Returns **string** –

mnemonicToSeed(*mnemonic*, *password*)
Convert mnemonic string to seed

Arguments

- **mnemonic** (*string*) –
- **password** (*undefined|string*) – optional

Returns **Promise<Buffer>** –

2.2 Raw seed

privateKeysFromSeed(*seed*, *options*)
Returns n private keys derived from seed

Arguments

- **seed** (*Buffer*) –
- **options** (*seed.Options*) – (optional) { count: number, hdpath: string }

Returns **Buffer[]** –

privateKeyFromSeed(*seed*, *options*)
Returns the first private key derived from seed

Arguments

- **seed** (*Buffer*) –
- **options** (*seed.Options*) – (optional) { hdpath: string }

Returns **Promise<Buffer>** –

class Options()
interface, *exported from* seed

Key derivation options

Options.**count**
type: undefined|number

Options.**hdpath**
type: undefined|string

CHAPTER 3

Hashing

3.1 Arbitrary messages

hash(*data*)
Calculate hash of transaction

Arguments

- **data** (*Buffer*) –

Returns **Buffer** – transaction hash

3.2 Transactions

hashTransaction(*tx*)
Calculate hash of transaction

Arguments

- **tx** (*hashing.TxBody*) – Transaction

Returns **Promise<string>** – transaction hash. If encoding is bytes, the result is a Buffer, otherwise a string.

class TxBody()
interface, *exported from* hashing

Transaction body. All fields except nonce, from, and chainIdHash are optional and will assume sensible defaults.

TxBody.**amount**
type: string|number|JSBI|Record<string,any>

TxBody.**chainIdHash**
type: Uint8Array|string

TxBody.**from**
type: string|Record<string,any>

TxBody.**limit**
type: undefined|number

TxBody.**nonce**
type: number

TxBody.**payload**
type: null|Uint8Array

TxBody.**price**
type: string|number|JSBI|Record<string,any>

TxBody.**sign**
type: undefined|string

TxBody.**to**
type: null|string|Record<string,any>

TxBody.**type**
type: undefined|number

CHAPTER 4

Signing

4.1 Arbitrary messages

signMessage (*msgHash*, *key*, *enc*)
Sign transaction with key.

Arguments

- **msgHash** (*Buffer*) – hash of a message. Can technically be any Buffer, but it really is only secure if using a hash.
- **key** (*KeyPair*) – key pair or private key
- **enc** (*signing.Encoding*) –

Returns **Promise<string>** –

verifySignature (*msg*, *key*, *signature*, *enc*)
Verify that a signature for msg was generated by key

Arguments

- **msg** (*Buffer*) –
- **key** (*KeyPair*) – key pair or public key
- **signature** (*string*) –
- **enc** (*signing.Encoding*) –

Returns **Promise<boolean>** –

4.2 Transactions

signTransaction (*tx*, *key*, *enc*)
Sign transaction with key.

Arguments

- **tx** (*any*) – transaction
- **key** (*KeyPair*) – key pair or private key
- **enc** (*signing.Encoding*) –

Returns **Promise<string>** –

verifyTxSignature(*tx*, *key*, *signature*, *enc*)

Verify that a signature for tx was generated by key

Arguments

- **tx** (*any*) –
- **key** (*KeyPair*) –
- **signature** (*string*) –
- **enc** (*signing.Encoding*) –

Returns **Promise<boolean>** –

CHAPTER 5

Encoding

5.1 Hexadecimal strings

fromHexString(*hexString*)
Converts hex string to Uint8Array.

Arguments

- **hexString** (*string*) –

Returns **Uint8Array** –

5.2 Uint8Arrays

fromNumber(*d*, *bitLength*)
Converts number to Uint8 array.

Arguments

- **d** (*number*) –
- **bitLength** (*number*) – default 64, can also use 32

Returns **Uint8Array** –

fromBigInt(*d*)
Converts BigInt to Uint8 array.

Arguments

- **d** (*JSBI|string|number*) –

Returns **Uint8Array** –

5.3 Addresses

encodeAddress (*byteArray*)
Encodes address or name from byte array to string.

Arguments

- **byteArray** (*Uint8Array*) –

Returns **string** – base58check encoded address or character bytes of name

decodeAddress (*address*)
Decodes address from string to byte array.

Arguments

- **address** (*string*) – base58check encoded address or name

Returns **Uint8Array** – byte array

5.4 Hashes

encodeTxHash (*bytes*)
Encodes data as base58 encoded string.

Arguments

- **bytes** (*Uint8Array|number[]*) – data

Returns **string** – base58 encoded string

decodeTxHash (*bs58string*)
Decodes base58 encoded data.

Arguments

- **bs58string** (*string*) – base58 encoded string

Returns **Uint8Array** – decoded data

5.5 Private keys

encodePrivateKey (*byteArray*)
Encodes address form byte array to string.

Arguments

- **byteArray** (*Uint8Array*) –

Returns **string** –

decodePrivateKey (*key*)
Decodes address from string to byte array.

Arguments

- **key** (*string*) –

Returns **Uint8Array** – byte array

CHAPTER 6

Keystore

Keystore is a specification to store private keys in a secure way. Please see Aergo documentation for format specification.

6.1 Generating keystore from private key (encryption)

keystoreFromPrivateKey(*key*, *password*, *kdfParams*)

Encrypt private key and return keystore data.

```
import { keystoreFromPrivateKey, createIdentity } from '@herajs/crypto';
const identity = createIdentity();
const keystore = await keystoreFromPrivateKey(identity.privateKey, 'password');
console.log(JSON.stringify(keystore, null, 2));
```

Arguments

- **key** (*Buffer*) –
- **password** (*string*) –
- **kdfParams** (*Partial<keystore.ScryptParams>*) –

Returns **Promise<keystore.Keystore>** –

6.2 Reading private key from keystore (decryption)

identityFromKeystore(*keystore*, *password*)

Decrypt keystore and return identity information.

```
import { identityFromKeystore } from '@herajs/crypto';
const keystore = JSON.parse('keystore file contents');
const identity = await identityFromKeystore(keystore, 'password');
console.log(identity);
```

Arguments

- **keystore** (*keystore.Keystore*) –
- **password** (*string*) –

Returns Promise<keys.Identity> –

class Keystore()
interface

Keystore.**aergo_address**
type: string

Keystore.**cipher**
type: keystore.KeystoreCipher

Keystore.**kdf**
type: keystore.KeystoreKdf

Keystore.**ks_version**
type: keystore.Version

class KeystoreCipher()
interface

KeystoreCipher.**algorithm**
type: keystore.CipherAlgorithm

KeystoreCipher.**ciphertext**
type: keystore.HexString

KeystoreCipher.**params**
type: keystore.CipherParams

class CipherParams()
interface

CipherParams.**iv**
type: keystore.HexString

class KeystoreKdf()
interface

KeystoreKdf.**algorithm**
type: "scrypt"

KeystoreKdf.**mac**
type: keystore.HexString

KeystoreKdf.**params**
type: keystore.ScryptParams

class ScryptParams()
interface

ScryptParams.**dklen**
type: number

ScryptParams.**n**
type: number

ScryptParams.**p**
type: number

ScryptParams.**r**
type: number

ScryptParams.**salt**
type: keystore.HexString

CHAPTER 7

Using with React Native

To use @herajs/crypto with React Native, you need to shim a few Node internal packages.

Otherwise, you may get an error like `Module `crypto` does not exist in the Haste module map`.

The following guide uses rn-nodeify.

1. Installation

When using Yarn:

```
// Install dependencies
yarn add react-native-crypto react-native-randombytes

// Fix integration
react-native link react-native-randombytes
yarn add -D tradle/rn-nodeify
./node_modules/.bin/rn-nodeify --install --hack --yarn
```

When using NPM:

```
// Install dependencies
npm install --save react-native-crypto react-native-randombytes

// Fix integration
react-native link react-native-randombytes
npm install --save-dev tradle/rn-nodeify
./node_modules/.bin/rn-nodeify --install --hack
```

Note: You have to run the final command every time you add packages. It is a good idea to add it as a post-install script to your package.json:

```
"scripts": {
    "postinstall": "rn-nodeify --install --hack"
}
```

2. Add shim to index.js

Import these at the top of the file.

```
import './shim.js'
import crypto from 'crypto'
```

If you are using a simulator, you may also need to add this line to shim.js:

```
self = undefined
```

3. Use normally

Now you can use @herajs/crypto normally. Add the dependency `@herajs/crypto` and use it, for example:

```
import { createIdentity } from '@herajs/crypto';

const identity = createIdentity();
```

CHAPTER 8

Indices and tables

- genindex

Index

T

V