
panaceaajs Documentation

Release 1.0.1

ggomma

Mar 12, 2019

1	Getting Started	3
1.1	Adding panacea-js	3
2	Tutorial	5
2.1	send data upload transaction	5
3	panacea.js	7
3.1	Parameters	7
4	panacea.js.client	9
4.1	getAccount	9
4.2	getBlock	10
4.3	getBlocks	12
4.4	getMedState	13
4.5	getCandidates	13
4.6	getDynasty	14
4.7	getTransaction	15
4.8	sendTransaction	16
5	panacea.js.cryptography	17
5.1	encryptData	17
5.2	decryptData	18
5.3	encryptKey	19
5.4	decryptKey	20
5.5	getKeyPair	20
5.6	getKeyPairFromPassphrase	21
5.7	getPubKey	21
5.8	getSharedSecretKey	22
5.9	recoverPubKeyFromSignature	22
5.10	sign	23
5.11	verifySignature	24
6	panacea.js.local.Account	25
6.1	new Account	25
6.2	createCertificate	27
6.3	getDecryptedPrivateKey	28
6.4	signTx	28

6.5	signTxAsPayer	29
6.6	signDataPayload	30
7	panacea.js.local.transaction	33
7.1	Transaction types	33
7.2	valueTransferTx	34
7.3	dataUploadTx	35
7.4	vestTx	37
7.5	withdrawVestingTx	38
7.6	becomeCandidateTx	39
7.7	quitCandidacyTx	41
7.8	voteTx	42
7.9	addCertificationTx	43
7.10	revokeCertificationTx	45
7.11	createDefaultPayload	46
7.12	createDataPayload	47
7.13	createVotePayload	47
7.14	createAddCertificationPayload	48
7.15	createRevokeCertificationPayload	49
7.16	recoverPayload	49
7.17	recoverPayloadWithType	50
8	panacea.js.healthData	53
8.1	MediBloc Health Data(MHD) Format	54
8.2	decodeData	54
8.3	decodeDataFromFile	55
8.4	encodeData	56
8.5	encodeDataFromFile	57
8.6	hashData	57
8.7	hashDataFromFile	58
9	panacea.js.identification	61
9.1	createCertificate	61
9.2	verifyCertificate	62
10	panacea.js.utils	65
10.1	genHexBuf	65
10.2	isAddress	66
10.3	isHexadecimal	66
10.4	padLeftWithZero	67
10.5	randomHex	67
10.6	sha3	68
10.7	sha3Stream	68

panacea-js is a client-side JavaScript library for [medibloc](#) blockchain.

CHAPTER 1

Getting Started

`panacea-js` is a client-side JavaScript library for `medibloc` blockchain.

1.1 Adding panacea-js

npm

```
npm install @medibloc/panacea-js
```


This section handles several examples to use panaceaajs.

```
var Panaceaajs = require('@medibloc/panacea-js');
var panaceaajs = Panaceaajs.init(['http://localhost:9921']);
var Account = panaceaajs.local.Account;
var Client = panaceaajs.client;
var HealthData = panaceaajs.healthData;
var Transaction = panaceaajs.local.transaction;
```

2.1 send data upload transaction

```
// create a new account
var account = new Account();
// get account state
Client.getAccount(account.pubKey, null, 'tail').then((res) => {
  var nonce = parseInt(res.nonce, 10);

  // calculate hash of the medical data file
  HealthData.hashDataFromFile('/file/path', 'medical-fhir', 'observation').
  →then((hash) => {
    // creating a medical data payload
    var healthDataPayload = Transaction.createDataPayload(hash);

    // creating a medical data upload transaction
    var tx = Transaction.dataUploadTx({
      from: account.pubKey,
      payload: healthDataPayload,
      nonce: nonce + 1
    });
  });
});
```

(continues on next page)

(continued from previous page)

```
// sign transaction
account.signTx(tx);

// send transaction
Client.sendTransaction(tx).then((res2) => {
  // .. do something
});
});
});
```

```
var Panaceaajs = require('@medibloc/panacea-js');  
var panaceaajs = Panaceaajs.init(['http://localhost:9921']);
```

The `panaceaajs` object has following objects.

- The *client* object allows you to interact with MediBloc blockchain.
- The *cryptography* object contains cryptographic functions.
- The *local.Account* object contains functions to generate MediBloc accounts, which contain encrypted private key and public key and can induce public key from the private key.
- The *local.transaction* object contains functions to generate transaction.
- The *healthData* object helps to encode and decode the health data as *MHD format*.
- The *identification* contains identification functions.
- The *utils* object provides utility functions for `panaceaajs`.

3.1 Parameters

`nodes` - Array: The array of node URL that will be used for the request.

Hint: The `panaceaajs` client sends a request to one of the nodes. If the request fails, it automatically retries the request to another node.

Note: You can test the library by running the MediBloc blockchain on a local machine as Testnet or Mainnet are not yet launched. Please refer to [go-medibloc](#) to run MediBloc blockchain on a local machine.

CHAPTER 4

panaceajs.client

The `panaceajs.client` object allows you to interact with the MediBloc blockchain.

```
var Panaceajs = require('@medibloc/panacea-js');
var Client = Panaceajs.client(['http://localhost:9921']);
//
// Instead, you can import from panaceajs like below.
//
// var Panaceajs = require('@medibloc/panacea-js');
// var panaceajs = Panaceajs.init(['http://localhost:9921']);
// var Client = panaceajs.client;
```

Note: You can test the library by running the MediBloc blockchain on a local machine as Testnet or Mainnet are not yet launched. Please refer to [go-medibloc](#) to run MediBloc blockchain on a local machine.

4.1 getAccount

```
Client.getAccount(address, height, type)
```

Returns the state of the account at a given block height.

4.1.1 Parameters

1. `address` - String: The address of the account of which to get the state.
2. `height` - Number: The height of the block.
3. `type` - String: The string 'genesis', 'confirmed', 'tail'.

Note: If type is set among 'genesis', 'confirmed' and 'tail', height is ignored.

4.1.2 Returns

Promise returns Object - The object of the account state:

- address - String: The address of the account.
- balance - String: The balance in 1e-12 MED of the account at the block.
- bandwidth - String: The consumed bandwidth.
- nonce - String: The nonce of the account at the block.
- unstaking - String: The amount of MED which is in progress of unstaking.
- vesting - String: The vesting in 1e-12 MED of the account at the block.
- voted - String: Voted address.

Note: balance and vesting '1' indicates '0.000000000001' (1e-12) MED.

4.1.3 Example

```
Client.getAccount('02fc22ea22d02fc2469f5ec8fab44bc3de42dda2bf9ebc0c0055a9eb7df579056c
→', 1, 'tail')
.then(console.log);
> {
  address: "02fc22ea22d02fc2469f5ec8fab44bc3de42dda2bf9ebc0c0055a9eb7df579056c",
  balance: "50000069000000000000",
  bandwidth: "0",
  nonce: "1",
  unstaking: "0",
  vesting: "50000000000000000000",
  voted: ["02fc22ea22d02fc2469f5ec8fab44bc3de42dda2bf9ebc0c0055a9eb7df579056c"],
}
```

4.2 getBlock

```
Client.getBlock(hash, height, type)
```

Returns a block matching the given block hash.

4.2.1 Parameters

1. hash - String: The hash of the block.
2. height - Number: The height of the block.

3. type - String: The string 'genesis', 'confirmed', 'tail'.

Note: If hash is sent, other fields are ignored.

4.2.2 Returns

Promise returns Object - The Block object:

- accs_root - String: The root hash of the accounts trie at the block.
- alg - Number: The signature algorithm of the block.
- chain_id - Number: The chain id of the block.
- coinbase - String: The coinbase address of the block.
- dpos_root - String: The root hash of the dpos trie at the block.
- hash - String: The hash of the block.
- height - String: The height of the block.
- parent_hash - String: The parent block's hash of the block.
- reward - String: The block producing reward.
- sign - String: The signature of the block.
- supply - String: The total supply of MED.
- timestamp - String: The timestamp of the block.
- transactions - Array: The transaction objects array of the block.
- txs_root - String: The root hash of the transactions trie at the block.

4.2.3 Example

```
Client.getBlock('4341669cbe7ed0c04640ebb2882e2290abdfbc4b9d162474826e5bd2a6efd259')
.then(console.log);
> {
  accs_root: "829745260cde8156131871962eae28db7a57736053c40d49261656e91a49062a"
  alg: 1,
  chain_id: 1,
  coinbase: "02fc22ea22d02fc2469f5ec8fab44bc3de42dda2bf9ebc0c0055a9eb7df579056c",
  dpos_root:
    ↪ "0a2017e57591d063ce1bc6bef82e1067f3e331037be811f3a192e346bd74a6ae06ae1220cd68b8e3a73670bd261712c76"
    ↪ ",
  hash: "4341669cbe7ed0c04640ebb2882e2290abdfbc4b9d162474826e5bd2a6efd259",
  height: "361",
  parent_hash: "0049aecc65a6383003d7dd392f925c0dfbd79c5e61ca789246f511b59c3970a7",
  reward: "46000000000000",
  sign:
    ↪ "649b96816d3dbc4d3df536f72cf1baf9886d00cd8b91b95132d1103f1308ae1f13101aca4a5f8b9193cb4ac23d19056ea"
    ↪ ",
  supply: "1000001656000000000000",
  timestamp: "1536044076",
  transactions: [],
```

(continues on next page)

(continued from previous page)

```
txs_root: "7a3d9780b7d68a05fb96ac180b67041e1b4c1ca68b8c2772566fce2dadd28dcb"
}
```

4.3 getBlocks

```
Client.getBlocks(from, to)
```

Returns blocks in given range.

4.3.1 Parameters

1. `from` - Number: The from height of the block.
2. `to` - Number: The to height of the block.

4.3.2 Returns

Promise returns Array - The array of the Blocks:

- `accs_root` - String: The root hash of the accounts trie at the block.
- `alg` - Number: The signature algorithm of the block.
- `chain_id` - Number: The chain id of the block.
- `coinbase` - String: The coinbase address of the block.
- `dpos_root` - String: The root hash of the dpos trie at the block.
- `hash` - String: The hash of the block.
- `height` - String: The height of the block.
- `parent_hash` - String: The parent block's hash of the block.
- `reward` - String: The block producing reward.
- `sign` - String: The signature of the block.
- `supply` - String: The total supply of MED.
- `timestamp` - String: The timestamp of the block.
- `transactions` - Array: The transaction objects array of the block.
- `txs_root` - String: The root hash of the transactions trie at the block.

4.3.3 Example


```
Client.getBlocks(3, 4)
  .then(console.log);
> [
  {height: '3', ...},
  {height: '4', ...}
]
```

4.4 getMedState

```
Client.getMedState()
```

Returns the current state of a node.

4.4.1 Returns

Promise returns Object - The object of the node state:

- `chain_id` - Number: The chain id of the node.
- `tail` - String: The hash of the most recent block.
- `height` - String: The height of the most recent block.
- `LIB` - String: The hash of the last irreversible block.

4.4.2 Example

```
Client.getMedState()
  .then(console.log);
> {
  chain_id: 1,
  tail: 'e2bd04e46ffd8ee1226d8ad8577a414ae57e226512d38d6b422e0413df36dfc0',
  height: '541',
  LIB: '432492182c8be7f30b552bceafe3f6bdaacd77a16bd396a9feaa112dbd52b5d5'
}
```

4.5 getCandidates

```
Client.getCandidates()
```

Returns the list of candidates.

4.5.1 Returns

Promise returns Array - The array of the candidates:

- address - String: The address of the candidate.
- collatral - String: The collatral of the candidate.
- votePower - String: The vote power of the candidate.

4.5.2 Example

```
Client.getCandidates()  
.then(console.log);  
> [  
  {  
    address: "03528fa3684218f32c9fd7726a2839cff3ddef49d89bf4904af11bc12335f7c939",  
    collatral: "0",  
    votePower: "50000000000000000000",  
  },  
  ...  
]
```

4.6 getDynasty

```
Client.getDynasty()
```

Returns the list of block producers.

4.6.1 Returns

Promise returns Array - The array of the candidates:

- address - String: The address of the block producer.

4.6.2 Example

```
Client.getDynasty()  
.then(console.log);  
> [  
  "03528fa3684218f32c9fd7726a2839cff3ddef49d89bf4904af11bc12335f7c939",  
  "02fc22ea22d02fc2469f5ec8fab44bc3de42dda2bf9ebc0c0055a9eb7df579056c",  
  ...  
]
```

4.7 getTransaction

```
Client.getTransaction(hash)
```

Returns the transaction matching a given transaction hash.

4.7.1 Parameters

hash - String: The hash of the transaction.

4.7.2 Returns

Promise returns Object - The object of the transaction:

- hash - String: The hash of the transaction.
- from - String: The address which use it's bandwidth. Or the address which to send this value from.
- to - String: The address which to take this value.
- value - String: The transferred value in 1e-12 MED.
- payload - String: The payload hex string made from protoBuffer.
- tx_type - String: The type of the transaction.
- nonce - String: The nonce indicates the number of transactions that this account has made.
- chain_id - Number: The chain id of the blockchain which this transaction belong to.
- alg - Number: The signature algorithm of the transaction.
- sign - String: The signature of the transaction.
- payer_sign - String: The signature of the payer.
- executed - Boolean: True if the transaction is executed and included in the block. otherwise, false.:

Note: value '1' indicates '0.000000000001' (1e-12) MED.

4.7.3 Example

```
Client.getTransaction(
  ↪ 'e6e2cbd69c32604f4a5195bbc0063876611c36d42a64ec95c6005bb1f3234d88')
.then(console.log);
> {
  hash: 'e6e2cbd69c32604f4a5195bbc0063876611c36d42a64ec95c6005bb1f3234d88',
  from: '02b83999492119eeee90a44bd621059e9a2f0b8219e067fb040473754a1821da07',
  to: '02b83999492119eeee90a44ad621059e9a2f0b8219e067fb040473754a1821da07',
  value: '100000000',
  payload: '',
  tx_type: 'transfer',
  nonce: '3',
  chain_id: 1,
```

(continues on next page)

(continued from previous page)

```
alg: 1,
sign:
↪ 'ca4b60467a75c53a95f7f85578c7e01a4e72598e6bc10866cd2db54daa59f7786ad07467b45164b47147039d2969863a7b
↪ ',
payer_sign: '',
executed: true
}
```

4.8 sendTransaction

```
Client.sendTransaction(transaction)
```

Returns a transaction hash.

4.8.1 Parameters

transaction - Object: The *Transaction* object created and signed.

4.8.2 Returns

Promise returns Object - The object contains the transaction hash:

- hash - String: The hash of the transaction.

Note: Receiving hash of the transaction **does not** mean it is valid or it is uploaded to the blockchain.

4.8.3 Example

```
Client.sendTransaction(tx)
  .then(console.log);
> {
  hash: '2edfc32b61528cedd3cafe7a794020d32ae3bcbfbc45fb810e169f34a4a30208'
}
```

panaceaajs.cryptography

The `panaceaajs.cryptography` contains cryptographic functions.

To use this package in a standalone use:

```
var Cryptography = require('@medibloc/panacea-js').cryptography;
//
// Instead, you can import from panaceaajs like below.
//
// var Panaceaajs = require('@medibloc/panacea-js');
// var panaceaajs = Panaceaajs.init(['http://localhost:9921']);
// var Cryptography = panaceaajs.cryptography;
```

5.1 encryptData

```
Cryptography.encryptData(accessKey, data);
```

To encrypt data you can use `Cryptography.encryptData(accessKey, data)`. This function generates an encrypted string using AES-256-CTR algorithm. Initialization vector(IV) is randomly generated and prepended with the delimiter `:`.

Note: Encryption algorithm can be added or changed.

5.1.1 Parameters

1. `accessKey` - String : The access key to encrypt data using symmetric key algorithm. If not given, empty string is used.
2. `data` - String : The data to encrypt with the access key.

5.1.2 Returns

String - The encrypted data.

5.1.3 Example

```
Cryptography.encryptData('this is access key !', 'hello medibloc!');  
> 'cc3ecbfc39c59fcab796d63458ff27fb:a32ae9c5c19068c6a3c90f57cc8662'
```

5.2 decryptData

```
Cryptography.decryptData(accessKey, encryptedData);
```

To decrypt data you can use `Cryptography.decryptData(accessKey, encryptedData)`. This function decrypts the encrypted data using AES-256-CTR algorithm. Initialization vector(IV) is parsed from the encrypted data.

Note: Decrypt algorithm can be added or changed.

5.2.1 Parameters

1. `accessKey` - String : The access key to decrypt data using symmetric key algorithm. If not given, empty string is used.
2. `encryptedData` - String : The encrypted data.

Note: In decryption, `encryptedData` must be the string generated through `Cryptography.encryptData`. If not, this function returns a wrong result.

5.2.2 Returns

String - The original data.

5.2.3 Example

```
Cryptography.decryptData('this is access key !',  
  ↪ 'cc3ecbfc39c59fcab796d63458ff27fb:a32ae9c5c19068c6a3c90f57cc8662');  
> 'hello medibloc!'
```

5.3 encryptKey

```
Cryptography.encryptKey(password, privKey, options);
```

To encrypt key you can use `Cryptography.encryptKey(password, privKey, options)`. This function generates an encrypted key defined as same as [Ethereum web3 secret storage definition](#).

5.3.1 Parameters

1. `password` - `String`: The password to encrypt key. It should be long and complicated.
2. `privKey` - `String`: The private key to encrypt with the password.
3. `options` - `Object` : (optional) Encryption options such as salt, iv, kdf, etc.

5.3.2 Returns

`Object` - The encrypted key.

5.3.3 Example

```
var encryptedKey = Cryptography.encryptKey('medibloc12345^&*()',
→ '337c72130334930ee3ad42a9fe323648bc33f4b4ff8fd2a7d71ea816078f7a27');
console.log(encryptedKey);
> {
  version: 3,
  id: '8a58575c-4367-4be8-a5de-cdf82931b53e',
  address: '0256b32f907826155408d41662b51e77878ef9bb58f8dfcdae98eb2eaf4dc3ce7a',
  crypto: {
    ciphertext:
→ '303460753670210a53f023aac4e7252e04d0cd93b79f0aa3889f6735de729919fea0fdf0c22be8d5d427c09d519022104
→ ',
    cipherparams: {
      iv: 'd1f1f735155432b640b5229c8a5de5d1'
    },
    cipher: 'aes-128-ctr',
    kdf: 'scrypt',
    kdfparams: {
      dklen: 32,
      salt: '932db3b379fb2a3762e876a5714079a89188f686c85c3b880775eebbd3c3b0c8',
      n: 8192,
      r: 8,
      p: 1
    },
    mac: 'a59dcc2e6dfaadb5aca6d1270e6341207817a64618323cb95f23d7b027318f54'
  }
}
```

5.4 decryptKey

```
Cryptography.decryptKey(password, encryptedKey);
```

To decrypt encrypted key you can use `Cryptography.decryptKey(password, encryptedKey)`. This function decrypts the encrypted key.

5.4.1 Parameters

1. `password - String` : The password used to encrypt `encryptedKey`.
2. `encryptedKey - Object | String` : The encrypted key to decrypt.

Note: In decryption, `encryptedKey` must be the object or string generated through `Cryptography.encryptKey`. If not, this function returns a error.

5.4.2 Returns

`String` - The original key.

5.4.3 Example

```
var encryptedKey = {
  version: 3,
  id: '8a58575c-4367-4be8-a5de-cdf82931b53e',
  address: '0256b32f907826155408d41662b51e77878ef9bb58f8dfcdae98eb2eaf4dc3ce7a',
  ...
};

var decryptedKey = Cryptography.decryptData('medibloc12345^&*()', encryptedKey);
console.log(decryptedKey);
> '337c72130334930ee3ad42a9fe323648bc33f4b4ff8fd2a7d71ea816078f7a27'
```

5.5 getKeyPair

```
Cryptography.getKeyPair();
```

To get a new private, public key pair, you can use `Cryptography.getKeyPair()`. Secp256k1 is used in generating a random key pair.

5.5.1 Returns

`Object` - The key pair

- `privKey - String` : The private key string in hexadecimal format.
- `pubKey - String` : The public key string in hexadecimal format.

5.5.2 Example

```
Cryptography.getKeyPair();  
> {  
  privKey: '1719e598983d472efbd3303cc3c4a619d89aef27a6d285443efe8e07f8100cbd',  
  pubKey: '03aa5632864e042271c375c95d1a7418407f986a45d36829879d106883a2e03cb3'  
}
```

5.6 getKeyPairFromPassphrase

```
Cryptography.getKeyPairFromPassphrase(passphrase);
```

To get a new private, public key pair from passphrase, you can use `Cryptography.getKeyPairFromPassphrase(passphrase)`.

5.6.1 Parameters

`passphrase - String`: The passphrase using as a seed to generate the key pair.

5.6.2 Returns

`Object` - The key pair

- `privKey - String`: The private key string in hexadecimal format.
- `pubKey - String`: The public key string in hexadecimal format.

5.6.3 Example

```
Cryptography.getKeyPairFromPassphrase('med med med med med med med med med med med med  
→');  
> {  
  privKey: 'c16d3c171c37faf691ec87814296fe8aa9b5e53390b06c548d325b2c0aa8668c',  
  pubKey: '02406b3e2fd06f97b822cd3c6bb661c9ecf13c0c13f1bf2117faf6ea1695dd9876'  
}
```

5.7 getPubKey

```
Cryptography.getPubKey(privKey);
```

To get the public key induced from the private key, you can use `Cryptography.getPubKey(privKey)`.

5.7.1 Parameters

`privKey - String`: The private key in hexadecimal format.

5.7.2 Returns

String - The public key induced from the private key.

5.7.3 Example

```
Cryptography.getPubKey (  
  ↪ '1719e598983d472efbd3303cc3c4a619d89aef27a6d285443efe8e07f8100cbd');  
> '03aa5632864e042271c375c95d1a7418407f986a45d36829879d106883a2e03cb3'
```

5.8 getSharedSecretKey

```
Cryptography.getSharedSecretKey(privKey, pubKey);
```

To get the shared secret key using ECDH, you can use `Cryptography.getSharedSecretKey(privKey, pubKey)`.

5.8.1 Parameters

1. `privKey` - String : The private key in hexadecimal format.
2. `pubKey` - String : The public key in hexadecimal format.

5.8.2 Returns

String - The public key from the private key.

5.8.3 Example

```
Cryptography.getSharedSecretKey (  
  ↪ '1719e598983d472efbd3303cc3c4a619d89aef27a6d285443efe8e07f8100cbd',  
  ↪ '03aa5632864e042271c375c95d1a7418407f986a45d36829879d106883a2e03cb3');  
> '21175492259a998204538e66d9cd3cd099147329683e601c408edff9e1e7f93f'
```

5.9 recoverPubKeyFromSignature

```
Cryptography.recoverPubKeyFromSignature(msgHash, signature);
```

To recover the public key from the signature, you can use `Cryptography.recoverPubKeyFromSignature(msgHash, signature)`.

5.9.1 Parameters

1. msgHash - String : The hash string of the message.
2. signature - String : The signature for the msgHash.

5.9.2 Returns

String - The public key from the message hash and signature.

5.9.3 Example

```
Cryptography.recoverPubKeyFromSignature(  
  ↪ '9e2d90f1ebc39cd7852973c6bab748579d82c93e4a2aa5b44a7769e51a606fd9',  
  ↪ 'c4f3d2cc47d999cfff0eb6845fb41cab3a0735afecd1fa178235d10e3d9aac835fdea24640626f6bae8795594f82c7ad8',  
  ↪ ');  
> '03047cd865161c3243c7b7a4d389ff407befbb3dd23f520152bc2a6ff2e2f0463d'
```

5.10 sign

```
Cryptography.sign(privKey, msgHash);
```

To make a signature for a message, you can use `Cryptography.sign(privKey, msgHash)`.

5.10.1 Parameters

1. privKey - String : The private key.
2. msgHash - String : The hash string of the message.

5.10.2 Returns

String - The signature for the msgHash.

5.10.3 Example

```
Cryptography.sign('aaca80d340f0cc94ea3baf128994376b2de7343f46e9c78efebea9c587edc7d3',  
  ↪ '9e2d90f1ebc39cd7852973c6bab748579d82c93e4a2aa5b44a7769e51a606fd9');  
>  
  ↪ 'c4f3d2cc47d999cfff0eb6845fb41cab3a0735afecd1fa178235d10e3d9aac835fdea24640626f6bae8795594f82c7ad8',  
  ↪ '
```

5.11 verifySignature

```
Cryptography.verifySignature(pubKey, msgHash, signature);
```

To verify the signature, you can use `Cryptography.verifySignature(pubKey, msgHash, signature)`.

5.11.1 Parameters

1. `pubKey - String`: The public key of the signature.
2. `msgHash - String`: The hash string of the message.
3. `signature - String`: The signature for the `msgHash`.

5.11.2 Returns

`Boolean` - If a signature is made from a valid public key and message hash pair, `true` is returned.

5.11.3 Example

```
Cryptography.verifySignature(  
  → '03047cd865161c3243c7b7a4d389ff407befbb3dd23f520152bc2a6ff2e2f0463d',  
  → '9e2d90f1ebc39cd7852973c6bab748579d82c93e4a2aa5b44a7769e51a606fd9',  
  → 'c4f3d2cc47d999cfff0eb6845fb41cab3a0735afecd1fa178235d10e3d9aac835fdea24640626f6bae8795594f82c7ad8'  
  → ');  
> true
```

panaceajs.local.Account

The `panaceajs.local.Account` contains functions to generate MediBloc accounts, which contain encrypted private key and public key pair and can induce public key from the private key.

To use this package in a standalone use:

```
var Account = require('@medibloc/panacea-js').local.Account;
//
// Instead, you can import from panaceajs like below.
//
// var Panaceajs = require('@medibloc/panacea-js');
// var panaceajs = Panaceajs.init(['http://localhost:9921']);
// var Account = panaceajs.local.Account;
```

6.1 new Account

```
new Account(passphrase, encryptedPrivateKey, pubKey);
```

To generate account, you can use `panaceajs.local.Account()`. Basically, account is just a pair of private and public key that has several functions described as below.

Note: MediBloc uses public key as an address.

6.1.1 Parameters

1. `passphrase - String` : (optional) If `encryptedPrivateKey` is not given, `passphrase` works as a key to encrypt private key. If `encryptedPrivateKey` is given, `passphrase` works as a key to decrypt `encryptedPrivateKey` and it must be used in encryption of the `encryptedPrivateKey`. If not given, `passphrase` is set with an empty string.

2. `encryptedPrivateKey` - `String`:(optional) Restore account is matched with the given encrypted private key. If not given, it will generate a new keypair.
3. `pubKey` - `String`:(optional) Restore account is matched with the given public key. If not given, it will be decrypted from the given passphrase and the given `encryptedPrivateKey`.

Note: If `passphrase` does not match with `encryptedPrivateKey`, it will return a different private key.

6.1.2 Returns

Object - The account object with the following structure:

- `pubKey` - `String`: The account's public key.
- `encryptedPrivKey` - `String`: The account's encrypted private key. This should be carefully shared or stored.
- And other following functions...

6.1.3 Example

```
var accountNoPassphrase = new Account();
console.log(accountNoPassphrase);
> Account {
  encryptedPrivKey: {
    version: 3,
    id: '6402eabe-db3f-497e-8d22-134d3690c349',
    address: '02ed468d1c8e4ee91b889b3e8fe79cd024df5ef4087c4ab1141d365a7b8d218ca4',
    ...
  },
  pubKey: '02ed468d1c8e4ee91b889b3e8fe79cd024df5ef4087c4ab1141d365a7b8d218ca4',
  ...
}

var account = new Account('123456789abcdeABCDE!@#');
console.log(account);
> Account {
  encryptedPrivKey: {
    version: 3,
    id: 'ca05cb03-7f84-4251-9c8c-e468fda47f0f',
    address: '0387e6dd9576a9bc792658bcdcb257311739c88b8a3ca68eef980316bb73a45d9',
    ...
  },
  pubKey: '0387e6dd9576a9bc792658bcdcb257311739c88b8a3ca68eef980316bb73a45d9',
  ...
}

var account1 = new Account('123456789abcdeABCDE!@#', account.encryptedPrivKey);
console.log(account1);
> // same with the previous result

var account2 = new Account('', account.encryptedPrivKey, account.pubKey);
console.log(account2);
> // same with the previous result
```

Note: SDK doesn't hold or share unencrypted private key. Account object holds encrypted private key and only the right passphrase can retrieve the unencrypted private key.

6.2 createCertificate

```
var account = new Account(passphrase, encryptedPrivateKey);
account.createCertificate(expireDate, issuer, issueDate, passphrase);
```

To create the certificate of the account, use `account.createCertificate(expireDate, issuer, issueDate, passphrase)`.

6.2.1 Parameters

1. `expireDate` - Number : The unix timestamp when certificate is expired.
2. `issuer` - String : The issuer's url to check certificate authenticity.
3. `issueDate` - Number : The unix timestamp when issuing certificate.
4. `passphrase` - String : (optional) The passphrase to decrypt encrypted private key. If not given, empty string is used to decrypt.

Note: `Account.createCertificate` doesn't return anything but assign the certificate object to the account. After signing, `account.cert` is changed from `Null` to `Object`.

6.2.2 Example

```
var owner = new Account();
owner.createCertificate({
  expireDate: Date.now() + (365 * 24 * 60 * 60 * 1000),
  issuer: 'https://medibloc.org',
  issueDate: Date.now(),
  passphrase: '',
});
console.log(owner.cert);
> {
  expireDate: 1558759043199,
  issuer: 'https://medibloc.org',
  issueDate: 1527223043199,
  pubKey: '020505d5ce655f7651eddfc6ee8bc96a78c40a622c5e28b1b8dfe1cf0f3af6c448',
  signature:
    '1d7b003afb947bcb6e8f27f1366a34d27f473c398e98c7cc36a8720dbfda064e03cfd35cf352057a23194da874afbe9a0'
}
```

6.3 getDecryptedPrivateKey

```
var account = new Account(passphrase, encryptedPrivateKey);
account.getDecryptedPrivateKey(passphrase);
```

To decrypt encrypted private key with the passphrase from the account object, you can use `account.getDecryptedPrivateKey(passphrase)`.

6.3.1 Parameters

`passphrase - String` : (optional) Passphrase is used to decrypt encrypted private key. If not given, empty string is used to decrypt.

Note: If `passphrase` does not match with `encryptedPrivateKey`, it will return a different private key.

6.3.2 Returns

`String` - Decrypted private key in hexadecimal format.

6.3.3 Example

```
var account = new Account('123456789abcdeABCDE!@#');
account.getDecryptedPrivateKey('123456789abcdeABCDE!@#');
> '960d2ea9a19b2b939b2ecbdbba75ffb50aafa0b63a73cd1b614cb53c50482d26'
```

6.4 signTx

```
var account = new Account(passphrase, encryptedPrivateKey);
account.signTx(tx, passphrase);
```

To sign a transaction with the private key, you can use `account.signTx(tx, passphrase)`. It assigns signature string to `tx.sign`.

6.4.1 Parameters

1. `tx - Object` : Transaction object created from one of the *transaction creation functions*.
2. `passphrase - String` : (optional) The passphrase to decrypt encrypted private key. If not given, empty string is used to decrypt the encrypted private key.

Note: `account.signTx` doesn't return anything but assign a signature string to the transaction object. After signing, `transaction.sign` is changed from null to `String`.

6.4.2 Example

```
var owner = new Account();
var transactionData = {
  from: owner.pubKey,
  to: '0266e30b34c9b377c9699c026872429a0fa582ac802759a3f35f9e90b352b8d932',
  value: '5',
  nonce: 3
};
var transaction = Transaction.valueTransferTx(transactionData);
owner.signTx(transaction);
console.log(transaction);
> {
  rawTx: {...},
  hash: '15be7e844e19ecdbad46894bf310e7c15bb315837baf4aac82991d0c531b02d8',
  sign:
    ↪ '882c24751521bae53bff1673b896b3d0cce2b81a03fea9563323975b79955cbe134744cbd21913955093e60c8d56d3884'
    ↪
}
```

6.5 signTxAsPayer

```
var account = new Account(passphrase, encryptedPrivateKey);
account.signTxAsPayer(tx, passphrase);
```

To sign a transaction as payer with the private key, you can use `account.signTxAsPayer(tx, passphrase)`. It assigns signature string to `tx.payerSign`.

6.5.1 Parameters

1. `tx` - Object : Transaction object created from one of the *transaction creation functions* and signed by a requester.
2. `passphrase` - String : (optional) The passphrase to decrypt encrypted private key. If not given, empty string is used to decrypt the encrypted private key.

Note: `account.signTxAsPayer` doesn't return anything but assign a signature string to the transaction object. After signing, `transaction.payerSign` is set as String.

6.5.2 Example

```
var requester = new Account('MediBloc1!');
var payer = new Account('MediBloc2@');
var transactionData = {
  from: requester.pubKey,
  to: '0266e30b34c9b377c9699c026872429a0fa582ac802759a3f35f9e90b352b8d932',
  value: '5',
  nonce: 3
}
```

(continues on next page)

(continued from previous page)

```
};
var transaction = Transaction.valueTransferTx(transactionData);
requester.signTx(transaction, 'MediBloc1!');
payer.signTxAsPayer(transaction, 'MediBloc2@');
console.log(transaction);
> {
  rawTx: {...},
  hash: 'd04bd6cb8eef9f59e9cab7fcb253303d003ce34f30bd8e0f59c09fba5281c303',
  sign:
    ↪ '5351c2e3375570ab15c4f1ef36b4d854516a368a4e0ad3e1c26d32df854083bb3bf9cc188fdc242484183edff5aeb8615d',
    ↪ ',
  payerSign:
    ↪ '0b0b28624ab007538c7044624f7a1f90c5e7b215118e8894288378c8895f2a585d10087d4be3d20a22bf381360fb70a5d',
    ↪ ',
}
```

6.6 signDataPayload

```
var account = new Account(passphrase, encryptedPrivateKey);
account.signDataPayload(dataPayload, passphrase);
```

To sign a data payload with the private key, you can use `account.signDataPayload(dataPayload, passphrase)`. It assigns signature string to `dataPayload.sign`.

6.6.1 Parameters

1. `dataPayload` - Object : data payload object:
 - `hash` - String: The hash string of the data payload.
2. `passphrase` - String:(optional) The passphrase to decrypt encrypted private key. If not given, empty string is used to decrypt.

Note: `account.signDataPayload` doesn't return anything but assign the signature string and the certificate to the data payload object. After signing, `dataPayload.sign` is changed from `Null` to `String` and `dataPayload.cert` is changed from `Null` to `Object`.

6.6.2 Example

```
var owner = new Account();
owner.createCertificate({
  expireDate: Date.now() + (365 * 24 * 60 * 60 * 1000),
  issuer: 'https://medibloc.org',
  issueDate: Date.now(),
  passphrase: '',
});
var dataPayload = {
  hash: 'eb36d0606ff84bba5ae84e2af0f2197b2ff4272c3d22c46ffa27ca17851cea7f',
```

(continues on next page)

(continued from previous page)

```
};
owner.signDataPayload(dataPayload);
console.log(dataPayload);
> {
  hash: 'eb36d0606ff84bba5ae84e2af0f2197b2ff4272c3d22c46ffa27ca17851cea7f',
  sign:
    ↪ 'e04c9c20093d686224bd759e8ca272772ed0528251a80c43502a8e21d3dcbfea21827b37f199132fef58a0fd2325f0ed4
    ↪ ',
  cert: {
    expireDate: 1558759447996,
    issuer: 'https://medibloc.org',
    issueDate: 1527223447996,
    pubKey: '02a980d3064c6135e75eb4843c5a15382d3dd4fa277625dea86f3fc97864eae288',
    signature:
    ↪ '9199402de763728112c68ddde02b06fbdb2745b0539ba5e981cb9a5233935c5e1e6f814fafe88f752e63635c77d48f58e
    ↪ '
  }
}
```

panaceajs.local.transaction

The `panaceajs.local.transaction` contains functions to generate transaction.

To use this package in a standalone use:

```
var Transaction = require('@medibloc/panacea-js').local.transaction;
//
// Instead, you can import from panaceajs like below.
//
// var Panaceajs = require('@medibloc/panacea-js');
// var panaceajs = Panaceajs.init(['http://localhost:9921']);
// var Transaction = panaceajs.local.transaction;
```

7.1 Transaction types

MediBloc blockchain has below transaction types.

- value transfer transaction : To transfer MED from one account to another.
- data upload transaction : To upload the hash of the data on the blockchain.
- vest transaction: To vest MED from balance of the account.
- withdraw vesting transaction: To withdraw vesting MED.
- become candidate transaction: To become candidate of the delegate.
- quit candidacy transaction: To quit candidacy for the delegate.
- vote transaction: To vote multiple candidates.
- add certification transaction: To issue certification to the subject.
- revoke certification transaction: To revoke certification from the subject.

7.2 valueTransferTx

```
Transaction.valueTransferTx(transactionData);
```

Returns a transaction which type is "value transfer".

7.2.1 Parameters

transactionData - Object

- from - String : The address from which to send the value.
- to - String : The address to which to send the value.
- value - String : The amount of value to transfer. It must not exceed the amount that the sender address has.
- nonce - Number : The nonce indicates how many transactions that this account has made. It should be exactly 1 larger than the current account's nonce. Highly recommend getting an account's latest nonce before making any transaction.
- chain_id - Number : (optional) The chain id of the blockchain. If not given, default chainId is automatically set. (For the testnet, 181112)
- payload - Object : (optional) The payload object. Recommend to use Transaction.createDefaultPayload(data).
 - message - Any Type : Any type of message which to store on the blockchain.

Note: value must be an integer between 0 and 340282366920938463463374607431768211455. And it's type should be a string.

Note: value '1' indicates '0.000000000001' (1e-12) MED. If you want to send 1MED, you need to use '1000000000000' (1e+12).

7.2.2 Returns

Object - The transaction object with the following structure:

- rawTx - Object : The rawTx contains transaction elements.
 - chain_id - Number : The chain id of the blockchain.
 - from - String : The address from which to send this value.
 - to - String : The address to which to send this value.
 - nonce - Number : The nonce.
 - value - String : The amount of value to transfer.
 - tx_type - String : The transaction type. For the value transfer transaction, it must be transfer.
 - payload - String : The transaction payload from protoBuffer. For the transaction which does not include payload, it must be undefined.
- hash - String : The hash to the transaction.

- `sign - String` : The signature to the transaction hash. Default is `null`.

7.2.3 Example

```
var transactionData = {
  from: '0367e7dee7bb273147991cb1d2b99a4daf069064fb77bd9a70c7998c5f1a00d58c',
  to: '037d91596727bc522553510b34815f382c2060cbb776f2765deafb48ae528d324b',
  chain_id: 1,
  value: '55',
  nonce: 3,
  payload: Transaction.createDefaultPayload('Hello World'),
};
var tx = Transaction.valueTransferTx(transactionData);
console.log(tx);
> {
  hash: '8ed74038236b0465cb16a6f687ed63f1d525599c4f6968906f91381242a6960f'
  rawTx:
  { chain_id: 1,
    from: '0367e7dee7bb273147991cb1d2b99a4daf069064fb77bd9a70c7998c5f1a00d58c',
    nonce: 3,
    payload: '0a0b48656c6c6f20576f726c64',
    to: '037d91596727bc522553510b34815f382c2060cbb776f2765deafb48ae528d324b',
    tx_type: 'transfer',
    value: "55" },
    sign: null
  }
```

7.3 dataUploadTx

```
Transaction.dataUploadTx(transactionData);
```

Returns a transaction which type is "data upload".

7.3.1 Parameters

`transactionData` - Object

- `from - String` : The address that spends bandwidth to upload data.
- `nonce - Number` : The nonce indicates the number of transactions that this account has made. It should be exactly 1 larger than current account's nonce. Highly recommend getting account's latest nonce before making transaction.
- `payload - Object` : The medical data payload which to store on the blockchain. Recommend to use `Transaction.createDataPayload(dataHash)`
 - `hash - Buffer|Uint8Array` : The hash byte array generated from entered data hash.
- `chain_id - Number` : (optional) The chain id of the blockchain. If not given, default chainId is automatically set. (For the testnet, 181112)

Note: Assigned writer can send transaction using owner(from)'s bandwidth. To use owner's bandwidth, use owner's address as `from` and sign the transaction with assigned writer's private key.

7.3.2 Returns

Object - The transaction object with the following structure:

- `rawTx` - Object : The `rawTx` contains transaction elements.
 - `chain_id` - Number : The chain id of the blockchain.
 - `from` - String : The address which uses its bandwidth to send transaction.
 - `to` - String : null
 - `nonce` - Number : The nonce.
 - `value` - String : '0'
 - `tx_type` - String : The transaction type. For the data upload transaction, it must be `add_record`
 - `payload` - String : The transaction payload from `protoBuffer` for the data uploading.
- `hash` - String : The hash to the transaction
- `signature` - String : The signature to the transaction hash. Default is null.
- `sign` - Function : The function for signing the transaction. It assigns signature string to signature.

Note: Data upload transaction does not send any value to any address. Hence, it has null in `to` parameter.

7.3.3 Example

```
panacea.js.healthData.hashData('hello MediBloc!', 'pghd').then((hash) => {
  var payload = Transaction.createDataPayload(hash);
  var transactionData = {
    from: '0367e7dee7bb273147991cb1d2b99a4daf069064fb77bd9a70c7998c5f1a00d58c',
    payload: payload,
    nonce: 4,
    chain_id: 1,
  };
  var tx = Transaction.dataUploadTx(transactionData);
  console.log(tx);
});
> {
  hash: '839f8fea1a35937c8e5367f68fcc1e47939311edf5c67c67aa39c372a95dfbe8',
  rawTx:
    { chain_id: 1,
      from: '0367e7dee7bb273147991cb1d2b99a4daf069064fb77bd9a70c7998c5f1a00d58c',
      nonce: 4,
      payload: '0a206e8593b28a3e60e219d2dcf93fa9307dbb30130908e8e2880e4f5469a691b24c',
      to: null,
      tx_type: 'add_record',
      value: '0' },
}
```

(continues on next page)

(continued from previous page)

```
sign: null
}
```

7.4 vestTx

```
Transaction.vestTx(transactionData);
```

Returns a transaction which type is "vest".

7.4.1 Parameters

transactionData - Object

- from - String : The address from which to vest the value.
- value - String : The amount of value to vest. It must not exceed the amount that the address has.
- nonce - Number : The nonce indicates how many transactions that this account has made. It should be exactly 1 larger than the current account's nonce. Highly recommend getting an account's latest nonce before making any transaction.
- chain_id - Number : (optional) The chain id of the blockchain. If not given, default chainId is automatically set. (For the testnet, 181112)

Note: value must be an integer between 0 and 340282366920938463463374607431768211455. And it's type should be a string.

Note: value '1' indicates '0.0000000000001' (1e-12) MED. If you want to vest 1MED, you need to use '1000000000000' (1e+12).

7.4.2 Returns

Object - The transaction object with the following structure:

- rawTx - Object : The rawTx contains transaction elements.
 - chain_id - Number : The chain id of the blockchain.
 - from - String : The address from which to send this value.
 - to - String : null
 - nonce - Number : The nonce.
 - value - String : The amount of value to vest.
 - tx_type - String : The transaction type. For the vest transaction, it must be vest.
- hash - String : The hash to the transaction.
- sign - String : The signature to the transaction hash. Default is null.

7.4.3 Example

```
var transactionData = {
  from: '0367e7dee7bb273147991cb1d2b99a4daf069064fb77bd9a70c7998c5f1a00d58c',
  value: '100',
  nonce: 3,
  chain_id: 1,
};
var tx = Transaction.vestTx(transactionData);
console.log(tx);
> {
  hash: '6d0f9b6bf9e765cd4fb5b9533202fee41f9e1382aaabf049609bd42b8ab842ae',
  rawTx:
    { chain_id: 1,
      from: '0367e7dee7bb273147991cb1d2b99a4daf069064fb77bd9a70c7998c5f1a00d58c',
      nonce: 3,
      payload: undefined,
      to: null,
      tx_type: 'stake',
      value: '100' },
  sign: null
}
```

7.5 withdrawVestingTx

```
Transaction.withdrawVestingTx(transactionData);
```

Returns a transaction which type is "withdraw vesting".

7.5.1 Parameters

transactionData - Object

- from - String: The address from which to withdraw the vesting value.
- value - String: The amount of value to withdraw vesting. It must not exceed the amount that the address vesting.
- nonce - Number: The nonce indicates how many transactions that this account has made. It should be exactly 1 larger than the current account's nonce. Highly recommend getting an account's latest nonce before making any transaction.
- chain_id - Number : (optional) The chain id of the blockchain. If not given, default chainId is automatically set. (For the testnet, 181112)

Note: value must be an integer between 0 and 340282366920938463463374607431768211455. And it's type should be a string.

Note: value '1' indicates '0.0000000000001' (1e-12) MED. If you want to withdraw vesting 1MED, you need to use '1000000000000' (1e+12).

7.5.2 Returns

Object - The transaction object with the following structure:

- rawTx - Object : The rawTx contains transaction elements.
 - chain_id - Number : The chain id of the blockchain.
 - from - String : The address from which to send this value.
 - to - String : null
 - nonce - Number : The nonce.
 - value - String : The amount of value to withdraw vesting.
 - tx_type - String : The transaction type. For the withdraw vesting transaction, it must be withdraw_vesting.
- hash - String : The hash to the transaction.
- sign - String : The signature to the transaction hash. Default is null.

7.5.3 Example

```
var transactionData = {
  from: '0367e7dee7bb273147991cb1d2b99a4daf069064fb77bd9a70c7998c5f1a00d58c',
  value: '100',
  nonce: 3,
  chain_id: 1,
};
var tx = Transaction.withdrawVestingTx(transactionData);
console.log(tx);
> {
  hash: '5f9a2ad7a759c6a5e754d8c7381a35ef62fdde92e347302458aeffb845d9b484',
  rawTx:
    { chain_id: 1,
      from: '0367e7dee7bb273147991cb1d2b99a4daf069064fb77bd9a70c7998c5f1a00d58c',
      nonce: 3,
      payload: undefined,
      to: null,
      tx_type: 'unstake',
      value: '100' },
  sign: null
}
```

7.6 becomeCandidateTx

```
Transaction.becomeCandidateTx(transactionData);
```

Returns a transaction which type is "become candidate".

7.6.1 Parameters

transactionData - Object

- from - String : The address from which to withdraw the vesting value.
- value - String : The amount of collateral. It must not exceed the amount that the address has.
- nonce - Number : The nonce indicates how many transactions that this account has made. It should be exactly 1 larger than the current account's nonce. Highly recommend getting an account's latest nonce before making any transaction.
- chain_id - Number : (optional) The chain id of the blockchain. If not given, default chainId is automatically set. (For the testnet, 181112)

Note: value must be an integer between 0 and 340282366920938463463374607431768211455. And it's type should be a string.

Note: value '1' indicates '0.000000000001' (1e-12) MED. If you want to guarantee collateral 1MED, you need to use '1000000000000' (1e+12).

7.6.2 Returns

Object - The transaction object with the following structure:

- rawTx - Object : The rawTx contains transaction elements.
 - chain_id - Number : The chain id of the blockchain.
 - from - String : The address from which to become a candidate for delegate.
 - to - String : null
 - nonce - Number : The nonce.
 - value - String : The amount of collateral.
 - tx_type - String : The transaction type. For the become candidate transaction, it must be become_candidate.
- hash - String : The hash to the transaction.
- sign - String : The signature to the transaction hash. Default is null.

7.6.3 Example

```
var transactionData = {
  from: '0367e7dee7bb273147991cb1d2b99a4daf069064fb77bd9a70c7998c5f1a00d58c',
  value: '10000',
  nonce: 2,
  chain_id: 1,
};
var tx = Transaction.becomeCandidateTx(transactionData);
console.log(tx);
> {
```

(continues on next page)

(continued from previous page)

```

hash: 'd5e2f04d8313b3489f51e41292d724f348ae8b4febccc46bae14304cb0f94c45',
rawTx:
  { chain_id: 1,
    from: '0367e7dee7bb273147991cb1d2b99a4daf069064fb77bd9a70c7998c5f1a00d58c',
    nonce: 2,
    payload: undefined,
    to: null,
    tx_type: 'become_candidate',
    value: '10000' },
  sign: null
}

```

7.7 quitCandidacyTx

```
Transaction.quitCandidacyTx(transactionData);
```

Returns a transaction which type is "quit candidacy".

7.7.1 Parameters

transactionData - Object

- from - String : The address from which to withdraw the vesting value.
- nonce - Number : The nonce indicates how many transactions that this account has made. It should be exactly 1 larger than the current account's nonce. Highly recommend getting an account's latest nonce before making any transaction.
- chain_id - Number : (optional) The chain id of the blockchain. If not given, default chainId is automatically set. (For the testnet, 181112)

7.7.2 Returns

Object - The transaction object with the following structure:

- rawTx - Object : The rawTx contains transaction elements.
 - chain_id - Number : The chain id of the blockchain.
 - from - String : The address from which to quit a candidate for delegate.
 - to - String : null
 - nonce - Number : The nonce.
 - value - String : null
 - tx_type - String : The transaction type. For the quit candidate transaction, it must be quit_candidate.
- hash - String : The hash to the transaction.
- sign - String : The signature to the transaction hash. Default is null.

7.7.3 Example

```
var transactionData = {
  from: '0367e7dee7bb273147991cb1d2b99a4daf069064fb77bd9a70c7998c5f1a00d58c',
  nonce: 3,
  chain_id: 1,
};
var tx = Transaction.quitCandidacyTx(transactionData);
console.log(tx);
> {
  hash: '5d97120c04bba28bb767c0bfdbcab7bcfe82dbaaae26a587e5ac7f891f9ea5d5',
  rawTx:
    { chain_id: 1,
      from: '0367e7dee7bb273147991cb1d2b99a4daf069064fb77bd9a70c7998c5f1a00d58c',
      nonce: 3,
      payload: undefined,
      to: null,
      tx_type: 'quit_candidacy',
      value: '0' },
  sign: null
}
```

7.8 voteTx

```
Transaction.voteTx(transactionData);
```

Returns a transaction which type is "vote".

7.8.1 Parameters

transactionData - Object

- from - String : The address of voter.
- nonce - Number : The nonce indicates how many transactions that this account has made. It should be exactly 1 larger than the current account's nonce. Highly recommend getting an account's latest nonce before making any transaction.
- payload - Object : The payload object from candidate list. Recommend to use `Transaction.createVotePayload([candidates])`
 - candidates - Array : The array of the candidates id byte array.
- chain_id - Number : (optional) The chain id of the blockchain. If not given, default chainId is automatically set. (For the testnet, 181112)

7.8.2 Returns

Object - The transaction object with the following structure:

- rawTx - Object : The rawTx contains transaction elements.
 - chain_id - Number : The chain id of the blockchain.

- from - String : The address of voter.
- to - String : null
- nonce - Number : The nonce.
- value - String : null
- tx_type - String : The transaction type. For the vote transaction, it must be vote.
- payload - String : The transaction payload from protoBuffer.
- hash - String : The hash to the transaction.
- sign - String : The signature to the transaction hash. Default is null.

7.8.3 Example

```
var transactionData = {
  from: '0367e7dee7bb273147991cb1d2b99a4daf069064fb77bd9a70c7998c5f1a00d58c',
  nonce: 4,
  chain_id: 1,
  payload: Transaction.createVotePayload([
    '5b803d26142699fa59ef51e5e39a6968f1fd1ef73908d2fdbdc1c51db723682f',
    '0e25e27e8f9ddb374d97f3c03664c1c5d044524e44145aa49ee33ae87ac1f118']),
};
var tx = Transaction.voteTx(transactionData);
console.log(tx);
> {
  hash: 'fc806da3d744d8a57a5aef28cd36025ded0690e37a740247ee43d8780fdfbcff',
  rawTx:
    { chain_id: 1,
      from: '0367e7dee7bb273147991cb1d2b99a4daf069064fb77bd9a70c7998c5f1a00d58c',
      nonce: 4,
      payload:
        ↪ '0a205b803d26142699fa59ef51e5e39a6968f1fd1ef73908d2fdbdc1c51db723682f0a200e25e27e8f9ddb374d97f3c03
        ↪ ',
      to: null,
      tx_type: 'vote',
      value: '0' },
  sign: null
}
```

7.9 addCertificationTx

```
Transaction.addCertificationTx(transactionData);
```

Returns a transaction which type is "add certification".

7.9.1 Parameters

transactionData - Object

- from - String : The address of issuer.

- `to - String` : The address of receiver.
- `nonce - Number` : The nonce indicates how many transactions that this account has made. It should be exactly 1 larger than the current account's nonce. Highly recommend getting an account's latest nonce before making any transaction.
- `payload - Object` : The payload object from the certificate data. Recommend to use `Transaction.createAddCertificationPayload(data)`.
 - `issueTime - Number` : The timestamp of the issuance. The unit must be seconds.
 - `expirationTime - Number` : The timestamp of the expiration. The unit must be seconds.
 - `hash - String` : The hash string of the certificate.
- `chain_id - Number` : (optional) The chain id of the blockchain. If not given, default chainId is automatically set. (For the testnet, 181112)

7.9.2 Returns

`Object` - The transaction object with the following structure:

- `rawTx - Object` : The rawTx contains transaction elements.
 - `chain_id - Number` : The chain id of the blockchain.
 - `from - String` : The address of voter.
 - `to - String` : The address of receiver.
 - `nonce - Number` : The nonce.
 - `value - String` : null
 - `tx_type - String` : The transaction type. For the vote transaction, it must be `add_certification`.
 - `payload - String` : The transaction payload from `protoBuffer`.
- `hash - String` : The hash to the transaction.
- `sign - String` : The signature to the transaction hash. Default is null.

7.9.3 Example

```
var transactionData = {
  from: '0367e7dee7bb273147991cb1d2b99a4daf069064fb77bd9a70c7998c5f1a00d58c',
  to: '037d91596727bc522553510b34815f382c2060cbb776f2765deafb48ae528d324b',
  nonce: 4,
  chain_id: 1,
  payload: Transaction.createAddCertificationPayload({
    issueTime: Math.floor(Date.now() / 1000),
    expirationTime: Math.floor(Date.now() / 1000),
    hash: '487b69767e201f485a67b915f1726e39a9d84d72ce3753dfdc824ebdf22e9b33',
  }),
};
var tx = Transaction.addCertificationTx(transactionData);
console.log(tx);
> {
  hash: '9b9b618603900a062959cec9a5b4be7f96303ac0f668f3e2cc3c4db4c42eb3ba',
  rawTx:
```

(continues on next page)

(continued from previous page)

```

{ chain_id: 1,
  from: '0367e7dee7bb273147991cb1d2b99a4daf069064fb77bd9a70c7998c5f1a00d58c',
  nonce: 4,
  payload:
→ '08a7b59ce10510a7b59ce1051a20487b69767e201f485a67b915f1726e39a9d84d72ce3753dfdc824ebdf22e9b33
→ ',
  to: '037d91596727bc522553510b34815f382c2060cbb776f2765deafb48ae528d324b',
  tx_type: 'add_certification',
  value: '0' },
  sign: null
}

```

7.10 revokeCertificationTx

```
Transaction.revokeCertificationTx(transactionData);
```

Returns a transaction which type is "revoke certification".

7.10.1 Parameters

transactionData - Object

- from - String : The address of issuer.
- nonce - Number : The nonce indicates how many transactions that this account has made. It should be exactly 1 larger than the current account's nonce. Highly recommend getting an account's latest nonce before making any transaction.
- payload - Object : The payload object from the certificate hash. Recommend to use Transaction.createRevokeCertificationPayload(data).
 - hash - Buffer|Uint8Array : The hash byte array generated from entered data hash.
- chain_id - Number : (optional) The chain id of the blockchain. If not given, default chainId is automatically set. (For the testnet, 181112)

7.10.2 Returns

Object - The transaction object with the following structure:

- rawTx - Object : The rawTx contains transaction elements.
 - chain_id - Number : The chain id of the blockchain.
 - from - String : The address of voter.
 - to - String : The address of receiver.
 - nonce - Number : The nonce.
 - value - String : null
 - tx_type - String : The transaction type. For the vote transaction, it must be add_certification.
 - payload - String : The transaction payload from protoBuffer.

- `hash - String`: The hash to the transaction.
- `sign - String`: The signature to the transaction hash. Default is `null`.

7.10.3 Example

```
var transactionData = {
  from: '0367e7dee7bb273147991cb1d2b99a4daf069064fb77bd9a70c7998c5f1a00d58c',
  nonce: 4,
  chain_id: 1,
  payload: Transaction.createRevokeCertificationPayload(
    '487b69767e201f485a67b915f1726e39a9d84d72ce3753dfdc824ebdf22e9b33'),
};
var tx = Transaction.revokeCertificationTx(transactionData);
console.log(tx);
> {
  hash: '92bala20afaafaab67ceffffe6909c2de0fc7d74d5dddc282bcc51980db3c4fdf',
  rawTx:
    { chain_id: 1,
      from: '0367e7dee7bb273147991cb1d2b99a4daf069064fb77bd9a70c7998c5f1a00d58c',
      nonce: 4,
      payload: '0a20487b69767e201f485a67b915f1726e39a9d84d72ce3753dfdc824ebdf22e9b33',
      to: null,
      tx_type: 'revoke_certification',
      value: '0' },
  sign: null
}
```

7.11 createDefaultPayload

```
Transaction.createDefaultPayload(message);
```

To generate default payload transaction, you can use `Transaction.createDefaultPayload(message)`. It is not executed on the blockchain, but only stored.

7.11.1 Parameters

`message - String`: String type of message can be entered.

7.11.2 Returns

`Object` - The payload object.

- `message - String`: The stringified data.

7.11.3 Example

```
var payload = Transaction.createDefaultPayload('MediBloc');
console.log(payload);
> { message: 'MediBloc' }
```

7.12 createDataPayload

```
Transaction.createDataPayload(hash);
```

To generate data payload transaction, you can use `Transaction.createDataPayload(hash)`. It returns data payload for `dataUploadTx`.

7.12.1 Parameters

`hash - String`: The hash of the data.

7.12.2 Returns

`Object` - The data payload object for data upload transaction.

- `hash - Buffer|Uint8Array`: The hash of the data.

7.12.3 Example

```
panaceaajs.healthData.hashData('hello MediBloc!', 'pghd').then((hash) => {
  var payload = Transaction.createDataPayload(hash);
  console.log(payload);
});
> {
  hash: [110, 133, 147, ...]
}
```

7.13 createVotePayload

```
Transaction.createVotePayload([candidateIds]);
```

To generate vote payload transaction, you can use `Transaction.createVotePayload([candidateIds])`. It returns payload for `voteTx`.

7.13.1 Parameters

`candidateIds - Array`: The array of the candidates id.

7.13.2 Returns

Object - The vote payload object for vote transaction.

- `candidates` - Array: The array of the candidate id byte array.

7.13.3 Example

```
var payload = Transaction.createVotePayload([
  '5b803d26142699fa59ef51e5e39a6968f1fd1ef73908d2fdbdc1c51db723682f',
  '0e25e27e8f9ddb374d97f3c03664c1c5d044524e44145aa49ee33ae87ac1f118',
]);
console.log(payload);
> {
  candidates: [
    <Buffer 5b 80 3d 26 14 26 99 fa 59 ef 51 e5 e3 9a 69 68 f1 fd 1e f7 39 08 d2 fd
    ↪bd c1 c5 1d b7 23 68 2f>,
    <Buffer 0e 25 e2 7e 8f 9d db 37 4d 97 f3 c0 36 64 c1 c5 d0 44 52 4e 44 14 5a a4
    ↪9e e3 3a e8 7a c1 f1 18>
  ]
}
```

7.14 createAddCertificationPayload

```
Transaction.createAddCertificationPayload(data);
```

To generate add certification payload transaction, you can use `Transaction.createAddCertificationPayload(data)`. It returns payload for `addCertificationTx`.

7.14.1 Parameters

Object - The data object for certification.

- `issueTime` - Number: The timestamp of the issuance. The unit must be seconds.
- `expirationTime` - Number: The timestamp of the expiration. The unit must be seconds.
- `hash` - String: The hash string of the certificate.

7.14.2 Returns

Object - The certification payload object for add certification transaction.

- `issueTime` - Number: The timestamp of the issuance. The unit must be seconds.
- `expirationTime` - Number: The timestamp of the expiration. The unit must be seconds.
- `hash` - Buffer|Uint8Array: The hash byte array generated from entered certification hash.

7.14.3 Example

```
var payload = Transaction.createAddCertificationPayload({
  issueTime: Math.floor(Date.now() / 1000),
  expirationTime: Math.floor(Date.now() / 1000),
  hash: '487b69767e201f485a67b915f1726e39a9d84d72ce3753dfdc824ebdf22e9b33',
});
console.log(payload);
> {
  expirationTime: 1536039205,
  hash: [72, 123, 105, 118, ...],
  issueTime: 1536039205,
}
```

7.15 createRevokeCertificationPayload

```
Transaction.createRevokeCertificationPayload(hash);
```

To generate revoke certification payload transaction, you can use `Transaction.createRevokeCertificationPayload(hash)`. It returns payload for `revokeCertificationTx`.

7.15.1 Parameters

`hash - String`: The hash string of the certificate.

7.15.2 Returns

`Object` - The certification payload object for revoke certification transaction.

- `hash - Buffer|Uint8Array`: The hash byte array generated from entered certification hash.

7.15.3 Example

```
var payload = Transaction.createRevokeCertificationPayload(
  '487b69767e201f485a67b915f1726e39a9d84d72ce3753dfdc824ebdf22e9b33');
console.log(payload);
> {
  hash: [72, 123, 105, 118, ...]
}
```

7.16 recoverPayload

```
Transaction.recoverPayload(transaction);
```

recover a payload of the transaction matched with its type.

7.16.1 Parameters

transaction - Object : The transaction contains the payload to recover.

7.16.2 Returns

Object - The recovered payload.

7.16.3 Example

```
var payload = Transaction.createDefaultPayload('Hello MediBloc!');
var fields = {
  from: '02bdc97dfc02502c5b8301ff46cbbb0dce56cd96b0af75edc50560630de5b0a472',
  nonce: 1,
  payload,
  to: '03e7b794e1de1851b52ab0b0b995cc87558963265a7b26630f26ea8bb9131a7e21',
  value: '10',
};
var tx = Transaction.valueTransferTx(fields);
var recoveredPayload = Transaction.recoverPayload(tx);
console.log(recoveredPayload);
> {
  message: 'Hello MediBloc!'
}
```

7.17 recoverPayloadWithType

```
Transaction.recoverPayloadWithType(payload, type);
```

recover a payload matched with its type.

7.17.1 Parameters

1. payload - Object : The payload to recover.
2. type - “string : transaction type matching with the payload

7.17.2 Returns

Object - The recovered payload.

7.17.3 Example

```
var payload = Transaction.createDefaultPayload('Hello MediBloc!');
var fields = {
  from: '02bdc97dfc02502c5b8301ff46cbbb0dce56cd96b0af75edc50560630de5b0a472',
  nonce: 1,
  payload,
  to: '03e7b794e1de1851b52ab0b0b995cc87558963265a7b26630f26ea8bb9131a7e21',
  value: '10',
};
var tx = Transaction.valueTransferTx(fields);
var recoveredPayload = Transaction.recoverPayloadWithType(tx.rawTx.payload, 'transfer
↔');
console.log(recoveredPayload);
> {
  message: 'Hello MediBloc!'
}
```

panaceajs.healthData

The `panaceajs.healthData` object helps to encode and decode the health data as *MHD format*.

```
var HealthData = require('@medibloc/panacea-js').healthData;
//
// Instead, you can import from panaceajs like below.
//
// var Panaceajs = require('@medibloc/panacea-js');
// var panaceajs = Panaceajs.init(['http://localhost:9921']);
// var HealthData = panaceajs.healthData;
```

Note: You can test the library by running the MediBloc blockchain on a local machine as Testnet or Mainnet are not yet launched. Please refer to [go-medibloc](#) to run MediBloc blockchain on a local machine.

8.1 MediBloc Health Data(MHD) Format

Offset	Bytes	Description
0	4	magic number(=0x004d4844)
4	2	protocol version
6	2	type code (unknown: 0, medical-fhir: 1, claim-fhir: 2, dicom: 3, ...)
8	2	subtype code of each type (for medical-fhir type, null: 0, patient: 1, observation: 2, ...)
10	32	hash of the encoded health data
42	6	encoded health data size(m)
48	m	encoded health data

We defined the MediBloc Health Data(MHD) format like above. More types of health data and its subtype will be supported.

User should upload the hash of the encoded health data to the MediBloc blockchain. By storing or transferring the health data as MHD type, it is easy to handle the health data with the blockchain.

Warning: This format can be changed.

8.2 decodeData

```
HealthData.decodeData(data)
```

Returns decoded health data.

8.2.1 Parameters

`data - Buffer|Uint8Array`: The MHD format health data.

8.2.2 Returns

`Promise returns Object` - The JSON object of the health data.

8.2.3 Example

```
var data = fs.readFileSync('/file/path');
console.log(data);
> <Buffer 00 4d 48 44 00 00 00 01 00 02 eb 36 d0 60 6f f8 4b ba 5a e8 4e 2a f0 f2 19
↳ 7b 2f f4 27 2c 3d 22 c4 6f fa 27 ca 17 85 1c ea 7f 00 00 00 00 01 15 0a 05 ... >

HealthData.decodeData(data)
.then(console.log);
> {
  status: 'final',
  category: [ { coding: [Array] } ],
  code: { coding: [ [Object], [Object] ] },
  resourceType: 'Observation',
  effectiveDateTime: '2017-05-15',
  id: 'f101',
  context: { reference: 'Encounter/f100' },
  subject: { reference: 'Patient/f100' },
  valueQuantity:
    { code: 'kg',
      unit: 'kg',
      value: 78,
      system: 'http://unitsofmeasure.org' } }
```

8.3 decodeDataFromFile

```
HealthData.decodeDataFromFile(filePath)
```

Returns decoded health data from the file path.

8.3.1 Parameters

`filePath` - String: The path of the MHD format file to read.

8.3.2 Returns

Promise returns Object - The JSON object of the health data.

8.3.3 Example

```
HealthData.decodeDataFromFile('/file/path')
.then(console.log);
> {
  status: 'final',
  category: [ { coding: [Array] } ],
  code: { coding: [ [Object], [Object] ] },
  resourceType: 'Observation',
  effectiveDateTime: '2017-05-15',
  id: 'f101',
```

(continues on next page)

(continued from previous page)

```
context: { reference: 'Encounter/f100' },
subject: { reference: 'Patient/f100' },
valueQuantity:
{ code: 'kg',
  unit: 'kg',
  value: 78,
  system: 'http://unitsofmeasure.org' } }
```

8.4 encodeData

```
HealthData.encodeData(data, type, subType)
```

Returns encoded Buffer|Uint8Array object as MHD format of the health data.

8.4.1 Parameters

1. data - Object|Uint8Array|Buffer: The health data to encode.
2. type - String: The type of the health data.
3. subType - String:(optional) The subtype of the health data.

8.4.2 Returns

Promise returns Buffer|Uint8Array - The MHD format object of the health data.

8.4.3 Example

```
var data = {
  resourceType: 'Observation',
  id: 'f101',
  status: 'final',
  category: [
    {
      coding: [
        {
          system: 'http://hl7.org/fhir/observation-category',
          code: 'vital-signs',
          display: 'Vital Signs',
        },
      ],
    },
  ],
  ...
};

HealthData.encodeData(data, 'medical-fhir', 'observation')
.then(console.log);
> {
```

(continues on next page)

(continued from previous page)

```

    <Buffer 00 4d 48 44 00 00 00 01 00 02 eb 36 d0 60 6f f8 4b ba 5a e8 4e 2a f0 f2 19
    ↪7b 2f f4 27 2c 3d 22 c4 6f fa 27 ca 17 85 1c ea 7f 00 00 00 00 01 15 0a 05 ... >
  }

```

8.5 encodeDataFromFile

```
HealthData.encodeDataFromFile(filePath, type, subType)
```

Returns encoded `Buffer|Uint8Array` object as MHD format object of the health data reading from the file path.

8.5.1 Parameters

1. `filePath` - String: The path of the file to read.
2. `type` - String: The type of the health data.
3. `subType` - String:(optional) The subtype of the health data.

8.5.2 Returns

Promise returns `Buffer|Uint8Array` - The MHD format object of the health data.

8.5.3 Example

```

HealthData.encodeDataFromFile('/file/path', 'medical-fhir', 'observation')
.then(console.log);
> {
  <Buffer 00 4d 48 44 00 00 00 01 00 02 eb 36 d0 60 6f f8 4b ba 5a e8 4e 2a f0 f2 19
  ↪7b 2f f4 27 2c 3d 22 c4 6f fa 27 ca 17 85 1c ea 7f 00 00 00 00 01 15 0a 05 ... >
}

```

8.6 hashData

```
HealthData.hashData(data, type, subType)
```

Returns the hash `String` of the health data.

8.6.1 Parameters

1. `data` - Object|`Uint8Array`|`Buffer`: The health data to encode.
2. `type` - String: The type of the health data.
3. `subType` - String:(optional) The subtype of the health data.

8.6.2 Returns

Promise returns String - The hash of the health data.

8.6.3 Example

```
var data = {
  resourceType: 'Observation',
  id: 'f101',
  status: 'final',
  category: [
    {
      coding: [
        {
          system: 'http://hl7.org/fhir/observation-category',
          code: 'vital-signs',
          display: 'Vital Signs',
        },
      ],
    },
  ],
  ...
};

HealthData.hashData(data, 'medical-fhir', 'observation')
.then(console.log);
> {
  'eb36d0606ff84bba5ae84e2af0f2197b2ff4272c3d22c46ffa27ca17851cea7f'
}
```

8.7 hashDataFromFile

```
HealthData.hashDataFromFile(filePath, type, subType)
```

Returns the hash String of the health data reading from the file path.

8.7.1 Parameters

1. filePath - String: The path of the file to read.
2. type - String: The type of the health data.
3. subType - String:(optional) The subtype of the health data.

8.7.2 Returns

Promise returns String - The hash of the health data.

8.7.3 Example

```
HealthData.hashDataFromFile('/file/path', 'medical-fhir', 'observation')
.then(console.log);
> {
  'eb36d0606ff84bba5ae84e2af0f2197b2ff4272c3d22c46ffa27ca17851cea7f'
}
```

panaceaajs.identification

The `panaceaajs.identification` contains identification functions.

To use this package standalone use:

```
var Id = require('@medibloc/panacea-js').identification;
//
// Instead, you can import from panaceaajs like below.
//
// var Panaceaajs = require('@medibloc/panacea-js');
// var panaceaajs = Panaceaajs.init(['http://localhost:9921']);
// var Id = panaceaajs.identification;
```

9.1 createCertificate

```
Id.createCertificate({ expireDate, issuer, issuerAccount, issueDate, passphrase, ↵
↵pubKey });
```

To create certificate, you can use `Id.createCertificate({ expireDate, issuer, issuerAccount, issueDate, passphrase, pubKey })`. It generates certificate object which contains issuer's signature.

9.1.1 Parameters

Object

- `expireDate` - Number : The unix timestamp when certificate is expired.
- `issuer` - String : The issuer's url to check certificate authenticity.
- `issuerAccount` - Object : The certificate issuer's account object from `new Account()`.

- `issueDate - Number` : The unix timestamp when issuing certificate.
- `passphrase - String` : The passphrase for the issuerAccount. Passphrase is used to decrypt private key from issuerAccount's encryptedPrivKey.
- `pubKey - String` : The public key which to be certified by issuer.

9.1.2 Returns

`Object` : The certificate object.

- `expireDate - Number` : The unix timestamp when certificate is expired.
- `issuer - String` : The issuer's url to check certificate authenticity.
- `issueDate - Number` : The unix timestamp when issuing certificate.
- `pubKey - String` : The public key which certified by the certificate.
- `signature - String` : The signature of issuer to certificate object.

9.1.3 Example

```
var issuer = new Account();
Id.createCertificate({
  expireDate: Date.now() + (365 * 24 * 60 * 60 * 1000),
  issuer: 'https://medibloc.org',
  issuerAccount: issuer,
  issueDate: Date.now(),
  passphrase: '',
  pubKey: '031ae654051968bb57de12e36184fd9118c03d49e6c1d05ef99149074c31a8dcee',
});
> {
  expireDate: 1558588202729,
  issuer: 'https://medibloc.org',
  issueDate: 1527052202729,
  pubKey: '031ae654051968bb57de12e36184fd9118c03d49e6c1d05ef99149074c31a8dcee',
  signature:
    ↪ '520282dce69b18f2dfefad8345a68e26a7b84ded32bc64e5a43cf0743e35a946629bc4245fe814f40acd196d19d5f9afcf
    ↪ ',
}
```

9.2 verifyCertificate

```
Id.verifyCertificate(certificate, timeStamp, issuerPubKey);
```

To verify certificate, you can use `Id.verifyCertificate(certificate, timeStamp, issuerPubKey)`.

9.2.1 Parameters

1. `certificate` - Object : The certificate object from `createCertificate()`
2. `timeStamp` - Number : The `timeStamp` to check whether the certificate is valid in the target time.
3. `issuerPubKey` - String : The `issuerPubkey` is the public key of the certificate issuer.

9.2.2 Returns

Boolean : True if the certificate is valid.

9.2.3 Example

```
var certificate = {
  expireDate: 1558588202729,
  issuer: 'https://medibloc.org',
  issueDate: 1527052202729,
  pubKey: '031ae654051968bb57de12e36184fd9118c03d49e6c1d05ef99149074c31a8dcee',
  signature:
    '520282dce69b18f2dfefad8345a68e26a7b84ded32bc64e5a43cf0743e35a946629bc4245fe814f40acd196d19d5f9afce',
};
Id.verifyCertificate(certificate, Date.now(),
  '0253f338731d59180253be2a9eee8d8266948b23c71181a85df23b9883b19cb187')
> true
```


CHAPTER 10

panaceaajs.utils

The `panaceaajs.utils` provides utility functions for `panaceaajs`.

To use this package standalone use:

```
var Utils = require('@medibloc/panacea-js').utils;
//
// Instead, you can import from panaceaajs like below.
//
// var Panaceaajs = require('@medibloc/panacea-js');
// var panaceaajs = Panaceaajs.init(['http://localhost:9921']);
// var Utils = panaceaajs.utils;
```

10.1 genHexBuf

```
Utils.genHexBuf(str, bytesLen);
```

Returns `Buffer` or `Uint8Array` from a string with exact length in bytes.

10.1.1 Parameters

1. `str` - `String` : The string to generate a buffer.
2. `bytesLen` - `Number` : The target length in bytes.

10.1.2 Returns

`Buffer|Uint8Array` - The result `Buffer` or `Uint8Array`.

10.1.3 Example

```
Utils.genHexBuf('12ab', 5);  
> <Buffer 00 00 00 12 ab>
```

10.2 isAddress

```
Utils.isAddress(pubKey);
```

To validate the public key, you can use `Utils.isAddress(pubKey)`.

Note: MediBloc use public key as an address.

10.2.1 Parameters

`pubKey` - `String`: The public key to validate.

10.2.2 Returns

`Boolean` - It is `true` if the public key is valid.

10.2.3 Example

```
Utils.isAddress('037d91596727bc522553510b34815f382c2060cbb776f2765deafb48ae528d324b');  
> true
```

10.3 isHexadecimal

```
Utils.isHexadecimal(string);
```

To check the type of the string, you can use `Utils.isHexadecimal(string)`.

10.3.1 Parameters

`string` - `String`: The string to be validated.

10.3.2 Returns

`Boolean` - It is `true` if the string is in hexadecimal format.

10.3.3 Example

```
Utils.isHexadecimal('1234567890abcdef');  
> true
```

10.4 padLeftWithZero

```
Utils.padLeftWithZero(str, len);
```

Adds a '0' padding on the left of a string.

10.4.1 Parameters

1. `str` - `String`: The string to add padding on the left.
2. `len` - `Number`: The total length of the string should have.

10.4.2 Returns

`String` - The padded string.

10.4.3 Example

```
Utils.padLeftWithZero('12ab', 10);  
> '00000012ab'
```

10.5 randomHex

```
Utils.randomHex(length);
```

To get a random seed number, you can use `Utils.randomHex(length)`.

10.5.1 Parameters

`length` - `Number`: (optional) The byte size of a random seed number. If not given, 16 is used.

10.5.2 Returns

`String` - The random number in hexadecimal format.

10.5.3 Example

```
Utils.randomHex();  
> 'baab6c02ce89592e03b8f9bba8eb553'
```

10.6 sha3

```
Utils.sha3(msg);
```

To hash messages, you can use `Utils.sha3(msg)`. This function uses SHA3_256 algorithm and returns 256bit hexadecimal string.

10.6.1 Parameters

`msg` - `String|Object|Number` : The message is stringified.

10.6.2 Returns

`String` - The hash string in hexadecimal format.

10.6.3 Example

```
Utils.sha3('Hello MediBloc!!!');  
> '25cd0631574c642502446ace0c9c46811f1404e39d6d892771b346724851dd7e'
```

10.7 sha3Stream

```
Utils.sha3Stream(stream);
```

To hash stream, you can use `Utils.sha3Stream(stream)`. This function uses SHA3_256 algorithm and returns 256bit hexadecimal string.

10.7.1 Parameters

`stream` - `Stream` : The readable stream.

10.7.2 Returns

`String` - The hash string in hexadecimal format.

10.7.3 Example

```
Utils.sha3Stream(stream); // some readable stream  
> '8a1fb1154b917c9e3df4370008e0bf34c6de6babb1592225371731a71a9b2e13'
```