# medjs Documentation

***Release 0.2.0***

**ggomma**

**Aug 06, 2018**

# User Documentation

**Note:** This documentation is for a work currently in progress and medjs 1.0 is not yet released.

medjs is a client-side JavaScript library for medibloc blockchain.

**Note:** This documentation is for a work currently in progress and medjs 1.0 is not yet released.

CHAPTER 1

# Getting Started

medjs is a client-side JavaScript library for medibloc blockchain.

## 1.1 Adding medjs

**npm**

```
npm install medjs
```

**Note:** This documentation is for a work currently in progress and medjs 1.0 is not yet released.

CHAPTER 2

# Tutorial

This section handles several examples to use medjs.

```
var Medjs = require('medjs');
var medjs = Medjs.init(['http://localhost:9921']);
var Account = medjs.local.Account;
var Client = medjs.client;
var HealthData = medjs.healthData;
var Transaction = medjs.local.transaction;
```

## 2.1 send data upload transaction

```
// create a new account
var account = new Account();
// get account state
Client.getAccountState(account.pubKey, '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
    });
  });
});
```

**Note:** This documentation is for a work currently in progress and medjs 1.0 is not yet released.

# CHAPTER 3

# medjs

```
var Medjs = require('medjs');
var medjs = Medjs.init(['http://localhost:9921']);
```

The `medjs` 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 medjs.

## 3.1 Parameters

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

**Hint:** The medjs 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.

**Note:** This documentation is for a work currently in progress and medjs 1.0 is not yet released.

CHAPTER 4

# medjs.client

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

```
var Medjs = require('medjs');
var Client = Medjs.client(['http://localhost:9921']);
//
// Instead, you can import from medjs like below.
//
// var Medjs = require('medjs');
// var medjs = Medjs.init(['http://localhost:9921']);
// var Client = medjs.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 getAccountState

```
Client.getAccountState(address, height)
```

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|String`: The height of the block. Or the string `'genesis'`, `'confirmed'`, `'tail'`.

### 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-8 MED of the account at the block.
- `certs_issued` - `Array`: Account addresses certificated by the account.
- `certs_received` - `Array`: Account addresses that have certificated the account.
- `nonce` - `String`: The nonce of the account at the block.
- `records` - `Array`: Hash list of records.
- `vesting` - `String`: The vesting in 1e-8 MED of the account at the block.
- `voted` - `String`: Voted address.

---

**Note:** `balance` and `vesting` '1' indicates '0.00000001' (1e-8) MED.

---

### 4.1.3 Example

```
Client.getAccountState(
↪'02fc22ea22d02fc2469f5ec8fab44bc3de42dda2bf9ebc0c0055a9eb7df579056c', 1)
.then(console.log);
> {
  address: '02fc22ea22d02fc2469f5ec8fab44bc3de42dda2bf9ebc0c0055a9eb7df579056c',
  balance: '100000000000000000',
  certs_issued: [],
  certs_received: [],
  nonce: '0',
  records: [],
  vesting: '0',
  voted: ''
}


Client.getAccountState(
↪'02fc22ea22d02fc2469f5ec8fab44bc3de42dda2bf9ebc0c0055a9eb7df579056c', 'latest')
.then(console.log);
> {
  address: '02fc22ea22d02fc2469f5ec8fab44bc3de42dda2bf9ebc0c0055a9eb7df579056c',
  balance: '99999999000000000',
  certs_issued: [],
  certs_received: [],
  nonce: '1',
  records: [],
  vesting: '0',
  voted: ''
}
```

---

## 4.2 getBlock

```
Client.getBlock(hash)
```

Returns a block matching the given block hash.

### 4.2.1 Parameters

`hash` - `String`: The hash of the block. Or the string `'genesis'`, `'confirmed'`, `'tail'`.

### 4.2.2 Returns

`Promise` returns `Object` - The Block object:

- `hash` - `String`: The hash of the block.
- `parent_hash` - `String`: The parent block's hash of the block.
- `coinbase` - `String`: The coinbase address of the block.
- `timestamp` - `String`: The timestamp of the block.
- `chain_id` - `Number`: The chain id of the block.
- `alg` - `Number`: The signature algorithm of the block.
- `sign` - `String`: The signature of the block.
- `accs_root` - `String`: The root hash of the accounts trie at the block.
- `txs_root` - `String`: The root hash of the transactions trie at the block.
- `usage_root` - `String`: The root hash of the usage trie at the block.
- `records_root` - `String`: The root hash of the records trie at the block.
- `consensus_root` - `String`: The root hash of the consensus trie at the block.
- `transactions` - `Array`: The transaction objects array of the block.
- `height` - `String`: The height of the block.

### 4.2.3 Example

```
Client.getBlock('1091173fe2bc7087e559bedf871a04e99927c92dad42d6270ae22c1bba720c30')
.then(console.log);
> {
  hash: '1091173fe2bc7087e559bedf871a04e99927c92dad42d6270ae22c1bba720c30',
  parent_hash: 'eb71569022ead2d290123bae4563a361a207109c1ef18969646570b566aa02e2',
  coinbase: '02fc22ea22d02fc2469f5ec8fab44bc3de42dda2bf9ebc0c0055a9eb7df579056c',
  timestamp: '1526033040',
  chain_id: 1,
  alg: 1,
  sign:
↪'8844c0ab33338906f64c45bd4849b7916a458dd9d8a960428b3e5d27369054cd3250fc08133cceeac4f2d75220e3fa8c3
↪',
  accs_root: 'f70f08d05514f549748620aa7cf677ae5303b8489f872e81078d09a538fcbec6',
  txs_root: '0362e767ab9fe76d940368cf97ae0318a99fb38dce60dd0bb56d23e28b86c3d7',
```

(continues on next page)

(continued from previous page)

```
  usage_root: '7788b87f9b2be5b10e27cc080cf662e144b5f78d7222bd265b5721c7481ba39a',
  records_root: '7788b87f9b2be5b10e27cc080cf662e144b5f78d7222bd265b5721c7481ba39a',
  consensus_root: 'bc28b8ef7f709b7457f5459db562a011e481232148dcbcb44b1e9f3d0eefdfbc',
  transactions: [],
  height: '5093'
}
```

---

## 4.3 getMedState

```
Client.getMedState()
```

Returns the current state of a node.

### 4.3.1 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.

### 4.3.2 Example

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

---

## 4.4 getTransaction

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

Returns the transaction matching a given transaction hash.

### 4.4.1 Parameters

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

### 4.4.2 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-8 MED.
- `timestamp` - `String`: The timestamp of the transaction.
- `data` - `Object`: The transaction data object corresponding to each *transaction types*.
- `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 '0.00000001' (1e-8) MED.

---

### 4.4.3 Example

```
Client.getTransaction(
↪'e6e2cbd69c32604f4a5195bbc0063876611c36d42a64ec95c6005bb1f3234d88')
.then(console.log);
> {
  hash: 'e6e2cbd69c32604f4a5195bbc0063876611c36d42a64ec95c6005bb1f3234d88',
  from: '02b83999492119eeea90a44bd621059e9a2f0b8219e067fb040473754a1821da07',
  to: '02b83999492119eeea90a44ad621059e9a2f0b8219e067fb040473754a1821da07',
  value: '100000000',
  timestamp: '1530853255670',
  data: { type: 'binary', payload: '' },
  nonce: '3',
  chain_id: 1,
  alg: 1,
  sign:
↪'ca4b60467a75c53a95f7f85578c7e01a4e72598e6bc10866cd2db54daa59f7786ad07467b45164b47147039d2969863a7b
↪',
  payer_sign: '',
  executed: true
}
```

---

## 4.5 sendTransaction

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

Returns a transaction hash.

### 4.5.1 Parameters

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

### 4.5.2 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.

### 4.5.3 Example

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

**Note:** This documentation is for a work currently in progress and medjs 1.0 is not yet released.

CHAPTER 5

# medjs.cryptography

The `medjs.cryptography` contains cryptographic functions.

To use this package in a standalone use:

```
var Cryptography = require('medjs').cryptography;
//
// Instead, you can import from medjs like below.
//
// var Medjs = require('medjs');
// var medjs = Medjs.init(['http://localhost:9921']);
// var Cryptography = medjs.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: '891ce3a670e6680bd5055bb2eb9ac6e134bcac827658172ef62655e6a7f75d87',
  pubKey: '02f5ef5b17dc0e40fa390c1cfe6a04eb9d3d111f82693cd52dc97335f0c008b492'
}
```

---

## 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
```

**Note:** This documentation is for a work currently in progress and medjs 1.0 is not yet released.

CHAPTER 6

# medjs.local.Account

The `medjs.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('medjs').local.Account;
//
// Instead, you can import from medjs like below.
//
// var Medjs = require('medjs');
// var medjs = Medjs.init(['http://localhost:9921']);
// var Account = medjs.local.Account;
```

## 6.1 new Account

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

To generate account, you can use `medjs.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 encrypted-PrivateKey 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.

**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 never be shared or stored anywhere.
- `getDecryptedPrivateKey(passphrase)` - `Function`: The function to decrypt an account's `encryptedPrivKey` and return a private key.

### 6.1.3 Example

```
new Account();
> {
  encryptedPrivKey:
↪'6cd4d5aa9385c9897f7b143adc104e8c7d185a4c87eb21c828...fefa6b2a087f47445908b766bebe9c5f05c2551901c0e
↪',
  pubKey: '037d91596727bc522553510b34815f382c2060cbb776f2765deafb48ae528d324b',
  getDecryptedPrivateKey: function(passphrase){...}
}

new Account('123456789abcdeABCDE!@#');
> {
  encryptedPrivKey:
↪'6cd4d5aa9385c9897f7b143adc104e8c7d185a4c87eb21c828...fefa6b2a087f47445908b766bebe9c5f05c2551901c0e
↪',
  pubKey: '037d91596727bc522553510b34815f382c2060cbb776f2765deafb48ae528d324b',
  getDecryptedPrivateKey: function(passphrase){...}
}

new Account('123456789abcdeABCDE!@#',
↪'6cd4d5aa9385c9897f7b143adc104e8c7d185a4c87eb21c828...fefa6b2a087f47445908b766bebe9c5f05c2551901c0e
↪');
> {
  encryptedPrivKey:
↪'6cd4d5aa9385c9897f7b143adc104e8c7d185a4c87eb21c828...fefa6b2a087f47445908b766bebe9c5f05c2551901c0e
↪',
  pubKey: '037d91596727bc522553510b34815f382c2060cbb776f2765deafb48ae528d324b',
  getDecryptedPrivateKey: function(passphrase){...}
}
```

**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
};
```

(continues on next page)

(continued from previous page)

```
var transaction = Transaction.valueTransferTx(transactionData);
owner.signTx(transaction);
console.log(transaction);
> {
  rawTx: {...},
  hash: '15be7e844e19ecdbad46894bf310e7c15bb315837baf4aac82991d0c531b02d8',
  sign:
↪'882c24751521bae53bff1673b896b3d0cce2b81a03fea9563323975b79955cbe134744cbd21913955093e60c8d56d3884c
↪'
}
```

## 6.5 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`.

### 6.5.1 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`.

### 6.5.2 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',
};
owner.signDataPayload(dataPayload);
console.log(dataPayload);
> {
  hash: 'eb36d0606ff84bba5ae84e2af0f2197b2ff4272c3d22c46ffa27ca17851cea7f',
```

(continues on next page)

(continued from previous page)

```
  sign:
↪'e04c9c20093d686224bd759e8ca272772ed0528251a80c43502a8e21d3dcbfea21827b37f199132fef58a0fd2325f0ed4a
↪',
  cert: {
    expireDate: 1558759447996,
    issuer: 'https://medibloc.org',
    issueDate: 1527223447996,
    pubKey: '02a980d3064c6135e75eb4843c5a15382d3dd4fa277625dea86f3fc97864eae288',
    signature:
↪'9199402de763728112c68ddde02b06fbdb2745b0539ba5e981cb9a5233935c5e1e6f814fafe88f752e63635c77d48f58ee
↪'
  }
}
```

**Note:** This documentation is for a work currently in progress and medjs 1.0 is not yet released.

CHAPTER 7

# medjs.local.transaction

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

To use this package in a standalone use:

```
var Transaction = require('medjs').local.transaction;
//
// Instead, you can import from medjs like below.
//
// var Medjs = require('medjs');
// var medjs = Medjs.init(['http://localhost:9921']);
// var Transaction = medjs.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 one of the candidate.(It could be change to multiple voting.)

## 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.
- timestamp - Number :(optional) The unix timestamp. If not given, current timestamp is automatically set.

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

**Note:** value '1' indicates '0.00000001' (1e-8) MED. If you want to send 1MED, you need to use '100000000' (1e+8).

### 7.2.2 Returns

Object - The transaction object with the following structure:

- rawTx - Object : The rawTx contains transaction elements.
  - alg - Number : The algorithm that is used in transaction.
  - 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.
  - timestamp - Number : The unix timestamp.
  - value - String : The amount of value to transfer.
  - data - Object
    - type - String : The transaction type. For the value transfer transaction, it must be transfer.
- 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',
  value: '55',
  nonce: 3
}
var tx = Transaction.valueTransferTx(transactionData);
console.log(tx);
> {
  hash: 'e7e838973c9ee679cfc34d950304d3b3ce1ad539a4f3a9946ad289ac19aa2bb1',
  rawTx:
  { alg: 1,
    chain_id: 1,
    from: '0367e7dee7bb273147991cb1d2b99a4daf069064fb77bd9a70c7998c5f1a00d58c',
    nonce: 3,
    data: { type: 'transfer' },
    timestamp: 1530854902566,
    to: '037d91596727bc522553510b34815f382c2060cbb776f2765deafb48ae528d324b',
    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.
- `medicalData` - `Object` : The medical data object generated from `Data.createDataPayload(dataObject)`.
  - `Hash` - `String` : The encrypted data's hash.
- `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.
- `timestamp` - `Number` :(optional) The unix timestamp. If not given, current timestamp is automatically set.

**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.
  - `alg` - `Number` : The algorithm used in transaction.
  - `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.
  - `timestamp` - `Number` : The unix timestamp.
  - `value` - `String` : '0'
  - `data` - `Object`
    - `type` - `String` : The transaction type. For the data upload transaction, it must be `add_record`
    - `payload` - `String` : The payload for the data uploading. It is a string from json object. (Will be changed to protoBuffer)
- `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

```
medjs.healthData.hashData('hello MediBloc!', 'pghd').then((hash) => {
  var payload = Transaction.createDataPayload(hash);
  var transactionData = {
    from: '0367e7dee7bb273147991cb1d2b99a4daf069064fb77bd9a70c7998c5f1a00d58c',
    medicalData: payload,
    nonce: 4,
  };
  var tx = Transaction.dataUploadTx(transactionData);
  console.log(tx);
});
> {
  hash: 'd668ba9f62542e882e8b1699b4a678d79f8bddc82f0be03861abfcf1b9a4dda9',
  rawTx:
   { alg: 1,
     chain_id: 1,
     from: '0367e7dee7bb273147991cb1d2b99a4daf069064fb77bd9a70c7998c5f1a00d58c',
     nonce: 4,
     data:
      { payload: '{"Hash":"boWTsoo+YOIZ0tz5P6kwfbswEwkI6OKIDk9UaaaRskw="}',
        type: 'add_record' },
     timestamp: 1530855002759,
     to: null,
```

(continues on next page)

(continued from previous page)

```
    value: '0' },
  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.
- `timestamp` - `Number` :(optional) The unix timestamp. If not given, current timestamp is automatically set.

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

**Note:** `value` '1' indicates '0.00000001' (1e-8) MED. If you want to vest 1MED, you need to use '100000000' (1e+8).

### 7.4.2 Returns

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

- `rawTx` - `Object` : The rawTx contains transaction elements.
  - `alg` - `Number` : The algorithm that is used in transaction.
  - `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.
  - `timestamp` - `Number` : The unix timestamp.
  - `value` - `String` : The amount of value to vest.
  - `data` - `Object`

    * `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
}
var tx = Transaction.vestTx(transactionData);
console.log(tx);
> {
  hash: '108dcbd0eb0c72f4e42220191acbe572853a92a9c94fc8bf5693894f98728823',
  rawTx:
   { alg: 1,
     chain_id: 1,
     from: '0367e7dee7bb273147991cb1d2b99a4daf069064fb77bd9a70c7998c5f1a00d58c',
     nonce: 3,
     data: { type: 'vest' },
     timestamp: 1531800108373,
     to: null,
     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.
- `timestamp` - `Number` :(optional) The unix timestamp. If not given, current timestamp is automatically set.

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

**Note:** `value` '1' indicates '0.00000001' (1e-8) MED. If you want to withdraw vesting 1MED, you need to use '100000000' (1e+8).

### 7.5.2 Returns

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

- `rawTx` - `Object` : The rawTx contains transaction elements.
  - `alg` - `Number` : The algorithm that is used in transaction.
  - `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.
  - `timestamp` - `Number` : The unix timestamp.
  - `value` - `String` : The amount of value to withdraw vesting.
  - `data` - `Object`
    - `type` - `String` : The transaction type. For the vest 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
}
var tx = Transaction.withdrawVestingTx(transactionData);
console.log(tx);
> {
  hash: '92fc4a56a34d9b67990c7e5b238e30920f13d4d353065336a9296e3440b3d5c2',
  rawTx:
   { alg: 1,
     chain_id: 1,
     from: '0367e7dee7bb273147991cb1d2b99a4daf069064fb77bd9a70c7998c5f1a00d58c',
     nonce: 3,
     data: { type: 'withdraw_vesting' },
     timestamp: 1531800486773,
     to: null,
     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.
- timestamp - Number :(optional) The unix timestamp. If not given, current timestamp is automatically set.

---

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

---

---

**Note:** value '1' indicates '0.00000001' (1e-8) MED. If you want to guarantee collateral 1MED, you need to use '100000000' (1e+8).

---

### 7.6.2 Returns

Object - The transaction object with the following structure:

- rawTx - Object : The rawTx contains transaction elements.
  - alg - Number : The algorithm that is used in transaction.
  - 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.
  - timestamp - Number : The unix timestamp.
  - value - String : The amount of collateral.
  - data - Object
    - 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
}
var tx = Transaction.becomeCandidateTx(transactionData);
console.log(tx);
> {
  hash: '4f5acb2f6ae8cf57e1625fc6d6e10c56d9a2de5a6bb284d38b59b23a9383fa30',
  rawTx:
   { alg: 1,
     chain_id: 1,
     from: '0367e7dee7bb273147991cb1d2b99a4daf069064fb77bd9a70c7998c5f1a00d58c',
     nonce: 2,
     data: { type: 'become_candidate' },
     timestamp: 1531801596611,
     to: null,
     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.
- `timestamp` - `Number` :(optional) The unix timestamp. If not given, current timestamp is automatically set.

### 7.7.2 Returns

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

- `rawTx` - `Object` : The rawTx contains transaction elements.
  - `alg` - `Number` : The algorithm that is used in transaction.
  - `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.
  - `timestamp` - `Number` : The unix timestamp.
  - `value` - `String` : `null`
  - `data` - `Object`
    * `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
}
var tx = Transaction.quitCandidacyTx(transactionData);
console.log(tx);
> {
  hash: 'd11b5c325e5ae5d8eefa1602a3862971a5af99fa8dac013b011007741c34cf8e',
  rawTx:
   { alg: 1,
     chain_id: 1,
     from: '0367e7dee7bb273147991cb1d2b99a4daf069064fb77bd9a70c7998c5f1a00d58c',
     nonce: 3,
     data: { type: 'quit_candidate' },
     timestamp: 1531801875679,
     to: null,
     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.
- `to` - `String` : The address of candidate to vote.
- `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.

- `timestamp` - `Number` :(optional) The unix timestamp. If not given, current timestamp is automatically set.

### 7.8.2 Returns

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

- `rawTx` - `Object` : The rawTx contains transaction elements.
  - `alg` - `Number` : The algorithm that is used in transaction.
  - `chain_id` - `Number` : The chain id of the blockchain.
  - `from` - `String` : The address of voter.
  - `to` - `String` : The address of candidate to vote.
  - `nonce` - `Number` : The nonce.
  - `timestamp` - `Number` : The unix timestamp.
  - `value` - `String` : `null`
  - `data` - `Object`
    - `type` - `String` : The transaction type. For the value transfer transaction, it must be `vote`.
- `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',
  to: '03528fa3684218f32c9fd7726a2839cff3ddef49d89bf4904af11bc12335f7c939',
  nonce: 4
}
var tx = Transaction.voteTx(transactionData);
console.log(tx);
> {
  hash: '5e42da19675f22c9523642735dadbd5c2bc70f95ef43a741dd6ace5f691189fe',
  rawTx:
   { alg: 1,
     chain_id: 1,
     from: '0367e7dee7bb273147991cb1d2b99a4daf069064fb77bd9a70c7998c5f1a00d58c',
     nonce: 4,
     data: { type: 'vote' },
     timestamp: 1531802070310,
     to: '03528fa3684218f32c9fd7726a2839cff3ddef49d89bf4904af11bc12335f7c939',
     value: '0' },
  sign: null
}
```

## 7.9 createDataPayload

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

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

### 7.9.1 Parameters

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

### 7.9.2 Returns

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

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

### 7.9.3 Example

```
medjs.healthData.hashData('hello MediBloc!', 'pghd').then((hash) => {
  var payload = Transaction.createDataPayload(hash);
  console.log(payload);
});
> {
  Hash: 'boWTsoo+YOIZ0tz5P6kwfbswEwkI6OKIDk9UaaaRskw=',
}
```

**Note:** This documentation is for a work currently in progress and medjs 1.0 is not yet released.

# CHAPTER 8

## medjs.healthData

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

```
var HealthData = require('medjs').healthData;
//
// Instead, you can import from medjs like below.
//
// var Medjs = require('medjs');
// var medjs = Medjs.init(['http://localhost:9921']);
// var HealthData = medjs.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<br>( unknown: 0, medical-fhir: 1,<br>claim-fhir: 2, dicom: 3, . . . ) |
| 8 | 2 | subtype code of each type<br>( for medical-fhir type,<br>null: 0, patient: 1,<br>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'
}
```

**Note:** This documentation is for a work currently in progress and medjs 1.0 is not yet released.

CHAPTER 9

# medjs.identification

The `medjs.identification` contains identification functions.

To use this package standalone use:

```
var Id = require('medjs').identification;
//
// Instead, you can import from medjs like below.
//
// var Medjs = require('medjs');
// var medjs = Medjs.init(['http://localhost:9921']);
// var Id = medjs.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:
↪'520282dce69b18f2dfefad8345a68e26a7b84ded32bc64e5a43cf0743e35a946629bc4245fe814f40acd196d19d5f9afce
↪',
}
```

## 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
```

**Note:** This documentation is for a work currently in progress and medjs 1.0 is not yet released.

CHAPTER 10

# medjs.utils

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

To use this package standalone use:

```
var Utils = require('medjs').utils;
//
// Instead, you can import from medjs like below.
//
// var Medjs = require('medjs');
// var medjs = Medjs.init(['http://localhost:9921']);
// var Utils = medjs.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();
> 'baab6c02ce89592e03b8f9bbea8eb553'
```

---

## 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'
```