
blockchain-core Documentation

even.network GmbH

Jan 17, 2020

Contents

1	Getting Started	1
1.1	Adding blockchain core	1
1.2	Configuring and initializing blockchain core	1
1.3	Create a new profile on evan.network via API	3
2	Blockchain	5
2.1	Executor	5
2.2	ExecutorAgent	10
2.3	ExecutorWallet	18
2.4	Signer Identity	22
2.5	Account Store	27
2.6	Event Hub	28
2.7	Name Resolver	31
2.8	Description	44
2.9	Wallet	51
2.10	Votings	56
2.11	Payments	66
3	Common	77
3.1	Logger	77
3.2	Validator	79
3.3	Utils	81
4	Contracts	83
4.1	Contract Loader	83
4.2	Rights and Roles	85
4.3	Sharing	91
4.4	Base Contract	104
4.5	Data Contract	110
4.6	Service Contract	124
4.7	Digital Twin Usage Examples	133
4.8	Digital Twin	141
4.9	Container	155
5	DFS (Distributed File System)	181
5.1	DFS Interface	181
5.2	IPFS	184

5.3	IPLD	188
6	Encryption	197
6.1	Encryption Wrapper	197
6.2	Key Provider	204
6.3	Crypto Provider	206
6.4	Cryptor - AES CBC	209
6.5	Cryptor - AES ECB	211
6.6	Cryptor - AES Blob	214
6.7	Cryptor - Unencrypted	217
7	Profile	221
7.1	Profile	221
7.2	Business Center Profile	241
7.3	Onboarding	245
7.4	Key Exchange	248
7.5	Mailbox	254
7.6	Verifications	261
7.7	Verifications Examples	286
7.8	Verifications Examples - Full Output	299
7.9	DID	309
7.10	VC	314

CHAPTER 1

Getting Started

The blockchain core is a helper library, that offers helpers for interacting with the `evan.network` blockchain. It is written in TypeScript and offers several (up to a certain degree) stand-alone modules, that can be used for

- creating and updating contracts
- managing user profiles
- en- and decryption
- distributed filesystem file handling
- key exchange and key handling
- ENS domain handling
- sending and receiving bmails

1.1 Adding blockchain core

First you need to get blockchain core and its dependencies into your project. This can be done using the following methods:

- `npm: npm install @evan.network/api-blockchain-core web3@1.0.0-beta.55`

After that you need to create a blockchain core runtime with a predefined configuration.

Node.js version `>= 10` is supported. The framework has been tested with Node.js 10, 11 and 12.

1.2 Configuring and initializing blockchain core

```
// require blockchain-core dependencies
const Web3 = require('web3');
```

(continues on next page)

(continued from previous page)

```
// require blockchain-core
const { Ipfs, createDefaultRuntime } = require('@evan.network/api-blockchain-core');

const runtimeConfig = {
  // account map to blockchain accounts with their private key
  accountMap: {
    'ACCOUNTID':
      'PRIVATE KEY',
  },
  // key configuration for private data handling
  keyConfig: {
    'ACCOUNTID': 'PASSWORD',
  },
  // ipfs configuration for evan.network storage
  ipfs: {host: 'ipfs.test.evan.network', port: '443', protocol: 'https'},
  // web3 provider config (currently evan.network testcore)
  web3Provider: 'wss://testcore.evan.network/ws',
};

async function init() {
  // initialize dependencies
  const provider = new Web3.providers.WebsocketProvider(
    runtimeConfig.web3Provider,
    { clientConfig: { keepalive: true, keepaliveInterval: 5000 } });
  const web3 = new Web3(provider, null, { transactionConfirmationBlocks: 1 });
  const dfs = new Ipfs({ dfsConfig: runtimeConfig.ipfs });

  // create runtime
  const runtime = await createDefaultRuntime(web3, dfs, { accountMap: runtimeConfig.
    ↪accountMap, keyConfig: runtimeConfig.keyConfig });
  console.dir(runtime);
}

init();
```

or you can initialize the api-blockchain-core runtime with your mnemonic and your password previously created on evan.network

```
// require blockchain-core dependencies
const Web3 = require('web3');

// require blockchain-core
const { Ipfs, createDefaultRuntime } = require('@evan.network/api-blockchain-core');

// ipfs configuration for evan.network testnet storage
const ipfsConfig = {host: 'ipfs.test.evan.network', port: '443', protocol: 'https'};
// web3 provider config (currently evan.network testcore)
const web3Provider = 'wss://testcore.evan.network/ws'

async function init() {
  // initialize dependencies
  const provider = new Web3.providers.WebsocketProvider(
    web3Provider,
    { clientConfig: { keepalive: true, keepaliveInterval: 5000 } });
  const web3 = new Web3(provider, null, { transactionConfirmationBlocks: 1 });
```

(continues on next page)

(continued from previous page)

```

const dfs = new Ipfs({ dfsConfig: ipfsConfig });

// create runtime
const runtime = await createDefaultRuntime(
  web3,
  dfs,
  {
    mnemonic: 'YOUR_MNEMONIC',
    password: 'YOUR_PASSWORD'
  }
);
console.dir(runtime);
}

init();

```

That's it! Now you can use the `runtime` object and interact with the `evan.network` blockchain.

The `blockchain-core` api is a set of modules which can be plugged in individually. So the above `runtime` is a full blown entry point to the api. You can also plug your own runtime with needed modules together.

1.3 Create a new profile on `evan.network` via API

When you want to create profiles programatically via our API, you can use the “Onboarding” class on the `api-blockchain-core` with the function `createNewProfile`. You can also generate your custom mnemonic from the `Onboarding` class as well. The only thing that should be defined is a password for the profile

```

// require blockchain-core
const { Onboarding } = require('@evan.network/api-blockchain-core');

async function createProfile() {
  // generate a new random mnemonic
  const mnemonic = Onboarding.createMnemonic();
  // create a profile for a mnemonic and a given password
  const profile = await Onboarding.createNewProfile(mnemonic, 'CUSTOM_PASSWORD');

  console.dir(profile);
}

createProfile();

```


2.1 Executor

Class Name	Executor
Extends	Logger
Source	executor.ts
Examples	executor.spec.ts

The executor is used for

- making contract calls
- executing contract transactions
- creating contracts
- send EVEs to another account or contract

The signer requires you to have a contract instance, either by

- loading the contract via [Description](#) helper (if the contract has an abi at its description)
- loading the contract via [ContractLoader](#) helper (if the contract has not abi at its description)
- directly via [web3.js](#).

2.1.1 constructor

```
new Executor(options);
```

Creates a new Executor instance.

The Executor allows to pass the `defaultOptions` property to its constructor. This property contains options for transactions and calls, that will be used if no other properties are provided in calls/transactions. Explicitly passed options always overwrite default options.

Parameters

1. **options - ExecutorOptions:** options for ServiceContract constructor.

- config - any: configuration object for the executor instance
- defaultOptions - any (optional): default options for web3 transactions/calls
- eventHub - `EventHub`: `EventHub` instance
- signer - `SignerInterface`: `SignerInterface` instance
- web3 - `Web3`: `Web3` instance
- log - Function (optional): function to use for logging: (message, level) => {...}
- logLevel - `LogLevel` (optional): messages with this level will be logged with log
- logLog - `LogLogInterface` (optional): container for collecting log messages
- logLogLevel - `LogLevel` (optional): messages with this level will be pushed to logLog

Returns

Executor instance

Example

```
const executor = new Executor({
  config,
  eventHub,
  signer,
  web3
});
```

2.1.2 init

```
executor.init(name);
```

initialize executor

Parameters

1. **options - any:** object with the property “eventHub” (of the type `EventHub`)

- eventHub - `EventHub`: The initialized `EventHub` Module.

Returns

void.

Example

```
runtime.executor.init({eventHub: runtime.eventHub})
```

2.1.3 executeContractCall

```
executor.executeContractCall(contract, functionName, ...args);
```

run the given call from contract

Parameters

1. **contract** - any: the target contract
2. **functionName** - string: name of the contract function to call
3. **...args** - any[]: optional array of arguments for contract call. if last arguments is {Object}, it is used as the options parameter

Returns

Promise resolves to any: contract calls result.

Example

```
const greetingMessage = await runtime.executor.executeContractCall(  
  contract, // web3.js contract instance  
  'greet' // function name  
);
```

2.1.4 executeContractTransaction

```
executor.executeContractTransaction(contract, functionName, inputOptions, ...  
  ↪functionArguments);
```

execute a transaction against the blockchain, handle gas exceeded and return values from contract function

Parameters

1. **contract** - any: contract instance
2. **functionName** - string: name of the contract function to call
3. **inputOptions** - any: options object
 - **from** - string (optional): The address the call “transaction” should be made from.
 - **gas** - number (optional): The amount of gas provided with the transaction.

- `event` - any (optional): object with two properties `target` (contract name) and `eventName`
- `getEventResult` - function (optional): callback function which will be called when the event is triggered. First argument is the full event info, second argument is an options with the event arguments.
- `eventTimeout` - number (optional): timeout (in ms) to wait for a event result before the transaction is marked as error
- `estimate` - boolean (optional): Should the amount of gas be estimated for the transaction (overwrites `gas` parameter)
- `force` - string (optional): Forces the transaction to be executed. Ignores estimation errors
- `autoGas` - number (optional): enables `autoGas 1.1` ==> adds 10% to estimated gas costs. value capped to current block.
- `value` - number (optional): Allows to specify the amount of funds for transfer

4. ...`functionArguments` - any[]: optional arguments to pass to contract transaction

Returns

Promise resolves to: no result (if no event to watch was given), the `event` (if event but no `getEventResult` was given), the value returned by `getEventResult(eventObject, arguments)`.

Because an estimation is performed, even if a fixed gas cost has been set, failing transactions are rejected before being executed. This protects users from executing transactions, that consume all provided gas and fail, which is usually not intended, especially if a large amount of gas has been provided. To prevent this behavior for any reason, add a `force: true` to the options, though it is **not advised to do so**.

To allow to retrieve the result of a transaction, events can be used to receive values from a transaction. If an event is provided, the transaction will only be fulfilled, if the event is triggered. To use this option, the executor needs to have the `eventHub` property has to be set. Transactions, that contain event related options and are passed to an executor without an `eventHub` will be rejected immediately.

Example

```
const accountId = '0x...';
const greetingMessage = await runtime.executor.executeContractTransaction(
  contract,                                // web3.js contract instance
  'setData',                               // function name
  { from: accountId, },                    // perform transaction with this account
  123,                                     // arguments after the options are passed
  // to the contract
);
```

Provided gas is estimated automatically with a fault tolerance of 10% and then used as `gas` limit in the transaction. For a different behavior, set `autoGas` in the transaction options:

```
const greetingMessage = await runtime.executor.executeContractTransaction(
  contract,                                // web3.js contract instance
  'setData',                               // function name
  { from: accountId, autoGas: 1.05, },      // 5% fault tolerance
  123,                                     // arguments after the options are passed
  // to the contract
);
```

or set a fixed gas limit:

```
const greetingMessage = await runtime.executor.executeContractTransaction(
  contract,                                // web3.js contract instance
  'setData',                              // function name
  { from: accountId, gas: 100000, },       // fixed gas limit
  123,                                    // arguments after the options are passed
  // to the contract
);
```

Using events for getting return values:

```
const contractId = await runtime.executor.executeContractTransaction(
  factory,
  'createContract', {
    from: accountId,
    autoGas: 1.1,
    event: { target: 'FactoryInterface', eventName: 'ContractCreated', },
    getResult: (event, args) => args.newAddress,
  },
);
```

2.1.5 executeSend

```
executor.executeSend(options);
```

send EVEs to target account

Parameters

1. options - any: the target contract

- from - string: The address the call “transaction” should be made from.
- to - string: The address where the eve’s should be send to.
- value - number: Amount to send in Wei

Returns

Promise resolves to void: resolved when done.

Example

```
await runtime.executor.executeSend({
  from: '0x...', // send from this account
  to: '0x...',   // receiving account
  value: web3.utils.toWei('1'), // amount to send in Wei
});
```

2.1.6 createContract

```
executor.createContract(contractName, functionArguments, options);
```

creates a contract by constructing creation transaction and signing it with private key of options.from

Parameters

1. `contractName` - string: contract name (must be available withing contract loader module)
2. `functionArguments` - any[]: arguments for contract creation, pass empty Array if no arguments
3. **options** - any: options object
 - `from` - string: The address the call “transaction” should be made from.
 - `gas` - number: Provided gas amout for contract creation.

Returns

Promise resolves to any: new contract.

Example

```
const newContractAddress = await runtime.executor.createContract(
  'Greeter', // contract name
  ['I am a demo greeter! :3'], // constructor arguments
  { from: '0x...', gas: 100000, }, // gas has to be provided with a fixed value
);
```

2.2 ExecutorAgent

Class Name	ExecutorAgent
Extends	Executor
Source	executor-agent.ts
Examples	executor-agent.spec.ts

The `ExecutorAgent` module is designed to cover basically the same tasks as the `Executor` module. While the last one performs the transactions directly with an account, that is given as `inputOptions`, the `ExecutorAgent` module wraps those transactions by submitting them to a smart agent.

The smart agent receives those transactions and performs them with its own account, if a valid tokens has been passed to it alongside the transaction data. Tokens for transactions can be issued at the smart agent as well (a password is required for this).

Using the `ExecutorAgent` allows to delegate transactions to users that do not have their own blockchain accounts. Delegating those transactions requires that smart agent user to be invited into the contract instead of the users, that will interact with the contract.

Users without accounts are then interacting with the front-end the same way as regular users, but do not submit their transactions themselves, they make a REST request against a smart agent server. To prevent spamming and scamming,

the users use tokens for their transactions. Tokens are basically like prepaid telephone card and allows to perform a limited set of functions for a limited amount of times. Tokens can only be created with a valid password.

Let's say, we have created a `DataContract`, that contains three questions, that should be answered by someone without an own blockchain account (think of a customer survey or something similar).

To allow that, first invite the corresponding smart agent account into the contract. Smart agent accounts for this should be known to the party, that wants to delegate transactions and their funds are maintained by this party, we'll use an example address for this here.

For abbreviation we assume, we have the following values:

```
let dataContractInstance;           // our data contract with the survey
↪ questions
let contractOwner;                 // 'we' - the contract creator and owner
let businessCenterDomain = 'testbc.evan'; // contracts business center
let smartAgentId;                 // account id of our smart agent
let password;                     // password, for token generation
```

So let's invite the smart agent to our contract:

```
await dataContract.inviteToContract(
  businessCenterDomain,
  dataContractInstance.options.address,
  contractOwner,
  smartAgentId,
);
```

Now with the smart agent being able to perform transactions, create a token for transaction delegation. We want the user to be able to:

1. accept the invitation
2. write three answers
3. mark their state as "done"

```
const txToken = await executor.generateToken(
  password, [
    { contract: dataContractInstance, functionName: 'changeConsumerState', count: 2, },
    { contract: dataContractInstance, functionName: 'addListEntry', count: 3, },
  ]);
```

Allowing `addListEntry` three time is to allow them to write three answers. One of the `changeConsumerState` times is for accepting the contract, which is done by setting the own state to `Active`. The other one is for marking editing as done (setting the own state to `Terminated`).

The resulting token `txToken` is a string, that can be handed over to our users without an account. This is usually done by sending them an email with a link, that contains the token and skips the login step for them and routes them directly to the contract, they should respond to.

Let's switch sides. The next steps are performed in the front-end by a user without a blockchain account, that has received the token.

To make transaction via a smart agent, create an instance of the `ExecutorWallet` and assign the token from before as token.

```
const executor = new ExecutorAgent({
  agentUrl
  config,
```

(continues on next page)

(continued from previous page)

```

    eventHub,
    signer,
    web3
  });
  executor.token = txToken;

```

To use the `ExecutorWallet` instance created this way in your other modules, hand it over to their constructor like instead of a normal `Executor` instance. Then use your modules, that have the `ExecutorWallet` instance like you would use them for making transactions with your own account.

```

const surveyAnswer = {
  foo: 'sample',
  bar: 123,
};
await dataContract.addListEntries(dataContractInstance, ['surveyAnswers'], ↵
  ↵[sampleValue], smartAgentId);

```

Note, that the last sample uses the `smartAgentId` as the performing account. Because transactions are estimated before being executed and in some cases the underlying modules require an “active” account, that is used as the users identity, this has to match the smart agents account id. The smart agent account id is passed alongside the token via the link in the email for users without blockchain accounts. References, that would point to a users account have to be replaced with this smart agent account id.

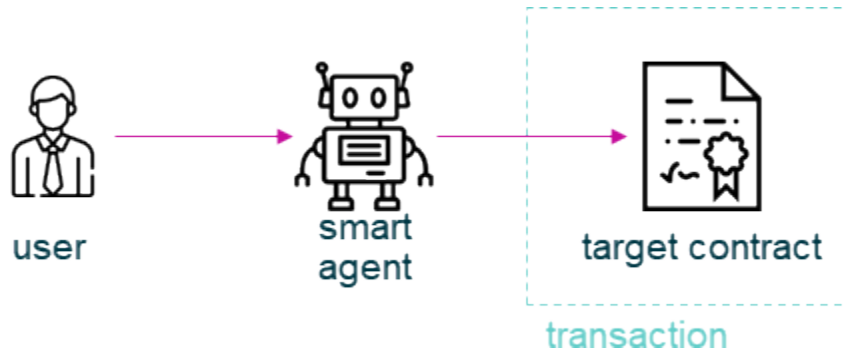


Fig. 1: transaction flow in agent based transactions

2.2.1 constructor

```

new ExecutorAgent(options);

```

Creates a new `ExecutorAgent` instance.

The `ExecutorAgent` allows to pass the `defaultOptions` property to its constructor. This property contains options for transactions and calls, that will be used if no other properties are provided in calls/transactions. Explicitly passed options always overwrite default options.

Parameters

1. `options - ExecutorAgentOptions`: options for `ServiceContract` constructor.

- `config` - any: configuration object for the executor instance
- `defaultOptions` - any (optional): default options for web3 transactions/calls

- `eventHub` - `EventHub`: `EventHub` instance
- `signer` - `SignerInterface`: `SignerInterface` instance
- `web3` - `Web3`: `Web3` instance
- `agentUrl` - `string` (optional): agent url, including protocol, host and port, defaults to `'http://localhost:8080'`
- `log` - `Function` (optional): function to use for logging: `(message, level) => {...}`
- `logLevel` - `LogLevel` (optional): messages with this level will be logged with `log`
- `logLog` - `LogLogInterface` (optional): container for collecting log messages
- `logLogLevel` - `LogLevel` (optional): messages with this level will be pushed to `logLog`

Returns

`ExecutorAgent` instance

Example

```
const executor = new ExecutorAgent({
  agentUrl
  config,
  eventHub,
  signer,
  web3
});
```

2.2.2 init

```
executor.init(name);
```

initialize executor

Parameters

1. **options** - **any**: object with the property “eventHub” (of the type `EventHub`)
 - `eventHub` - `EventHub`: The initialized `EventHub` Module.

Returns

`void`.

Example

```
runtime.executor.init({eventHub: runtime.eventHub})
```

2.2.3 executeContractCall

```
executor.executeContractCall(contract, functionName, ...args);
```

run the given call from contract

this is done as a rest call against the smart agent

a token is not required for performing calls

Parameters

1. `contract` - any: the target contract
2. `functionName` - string: name of the contract function to call
3. `...args` - any[]: optional array of arguments for contract call. if last arguments is {Object}, it is used as the options parameter

Returns

Promise resolves to any: contract calls result.

Example

```
const greetingMessage = await runtime.executor.executeContractCall(  
  contract, // web3.js contract instance  
  'greet' // function name  
);
```

2.2.4 executeContractTransaction

```
executor.executeContractTransaction(contract, functionName, inputOptions, ...  
  ↪functionArguments);
```

execute a transaction against the blockchain, handle gas exceeded and return values from contract function

this is done as a rest call against the smart agent

transactions, that transfer EVEs to a target account, will be rejected

this requires a valid token issued with `generateToken` beforehand, tokens can be set at the executor with:

```
executor.token = someToken;
```

Parameters

1. `contract` - any: contract instance
2. `functionName` - string: name of the contract function to call
3. **`inputOptions` - any: options object**

- `from - string` (optional): The address the call “transaction” should be made from.
- `gas - number` (optional): The amount of gas provided with the transaction.
- `event - string` (optional): The event to wait for a result of the transaction,
- `getEventResult - function` (optional): callback function which will be called when the event is triggered.
- `eventTimeout - number` (optional): timeout (in ms) to wait for a event result before the transaction is marked as error
- `estimate - boolean` (optional): Should the amount of gas be estimated for the transaction (overwrites `gas` parameter)
- `force - string` (optional): Forces the transaction to be executed. Ignores estimation errors
- `autoGas - number` (optional): enables `autoGas 1.1` ==> adds 10% to estimated gas costs. value capped to current block.

4. `...functionArguments - any[]`: optional arguments to pass to contract transaction

Returns

Promise resolves to: no result (if no event to watch was given), the `event` (if event but no `getEventResult` was given), the value returned by `getEventResult(eventObject)`.

Because an estimation is performed, even if a fixed gas cost has been set, failing transactions are rejected before being executed. This protects users from executing transactions, that consume all provided gas and fail, which is usually not intended, especially if a large amount of gas has been provided. To prevent this behavior for any reason, add a `force: true` to the options, though it is **not advised to do so**.

To allow to retrieve the result of a transaction, events can be used to receive values from a transaction. If an event is provided, the transaction will only be fulfilled, if the event is triggered. To use this option, the executor needs to have the `eventHub` property has to be set. Transactions, that contain event related options and are passed to an executor without an `eventHub` will be rejected immediately.

Example

```
const accountId = '0x...';
const greetingMessage = await runtime.executor.executeContractTransaction(
  contract,                                // web3.js contract instance
  'setData',                               // function name
  { from: accountId, },                    // perform transaction with this account
  123,                                     // arguments after the options are passed
  // to the contract
);
```

Provided gas is estimated automatically with a fault tolerance of 10% and then used as `gas` limit in the transaction. For a different behavior, set `autoGas` in the transaction options:

```
const greetingMessage = await runtime.executor.executeContractTransaction(
  contract,                                // web3.js contract instance
  'setData',                               // function name
  { from: accountId, autoGas: 1.05, },      // 5% fault tolerance
  123,                                     // arguments after the options are passed
  // to the contract
);
```

or set a fixed gas limit:

```
const greetingMessage = await runtime.executor.executeContractTransaction(
  contract,                                // web3.js contract instance
  'setData',                              // function name
  { from: accountId, gas: 100000, },      // fixed gas limit
  123,                                    // arguments after the options are passed
  // to the contract
);
```

Using events for getting return values:

```
const contractId = await runtime.executor.executeContractTransaction(
  factory,
  'createContract', {
    from: accountId,
    autoGas: 1.1,
    event: { target: 'FactoryInterface', eventName: 'ContractCreated', },
    getResult: (event, args) => args.newAddress,
  },
);
```

2.2.5 executeSend

```
executor.executeSend(options);
```

creating contracts directly is not supported by the walled based executor, use a regular Executor for such tasks

2.2.6 createContract

```
executor.createContract(contractName, functionArguments, options);
```

creates a contract by constructing creation transaction and signing it with private key of options.from

this is done as a rest call against the smart agent

transactions, that transfer EVEs to a target account, will be rejected

this requires a valid token issued with generateToken beforehand, tokens can be set at the executor with:

```
executor.token = someToken;
```

Parameters

1. **contractName** - string: contract name (must be available withing contract loader module)
2. **functionArguments** - any[]: arguments for contract creation, pass empty Array if no arguments
3. **options** - any: options object
 - **from** - string: The address the call “transaction” should be made from.
 - **gas** - number: Provided gas amout for contract creation.

Returns

Promise resolves to any: new contract.

Example

```
const newContractAddress = await runtime.executor.createContract(
  'Greeter', // contract name
  ['I am a demo greeter! :3'], // constructor arguments
  { from: '0x...', gas: 100000, }, // gas has to be provided with a fixed value
);
```

2.2.7 generateToken

```
executor.generateToken(password, functions);
```

generate a new token for transactions (or contract creations)

this generates a new token for the given functions, this token can be used for each requested function (either only once or multiple times, if a count > 1 has been requested)

Parameters

1. password - string: password for token creation
2. functions - any[]: array of function signatures as abi objects

Returns

Promise returns string: token for given transactions

Example

Tokens can be created for transactions by passing the contract and the function name to it:

```
const txToken = await executor.generateToken(
  password, [{ contract: contract, functionName: 'addListEntry', }]);
```

When the token should be able to perform those transactions multiple times, pass a count alongside:

```
const txToken = await executor.generateToken(
  password, [{ contract: contract, functionName: 'addListEntry', count: 3, }]);
```

When wanting to perform a contract creation without a factory contract, this contract has to be known to the smart agent. Then the contract name can be passed as 'signature':

```
const ccToken = await executor.generateToken(
  password, [{ signature: 'Owned', }]);
```

2.3 ExecutorWallet

Class Name	ExecutorWallet
Extends	Executor
Source	executor-wallet.ts
Examples	executor-wallet.spec.ts

The `ExecutorWallet` module is designed to cover basically the same tasks as the `Executor` module. While the last one performs the transactions directly with an account, that is given as `inputOptions`, the `ExecutorWallet` module wraps those transactions by submitting them to a configured wallet contract.

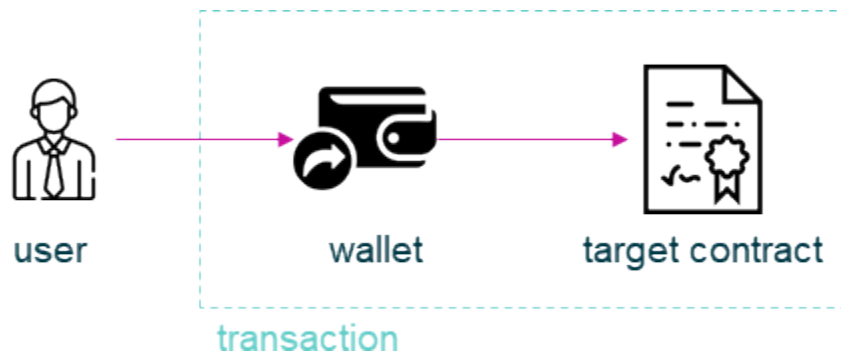


Fig. 2: transaction flow in wallet based transactions

The wallet is a smart contract and the account, that performs the transactions against the target contracts on the blockchain. Transactions are wrapped by using the `Wallet` module, see details on how to transactions are performed internally at the documentation page of the `Wallet` module.

Because transactions are performed via the `Wallet` module, they have the same limitations in regards to en- and decryption as described in the introduction section of the `Wallet`.

2.3.1 constructor

```
new ExecutorWallet(options);
```

Creates a new `ExecutorWallet` instance.

The `ExecutorWallet` allows to pass the `defaultOptions` property to its constructor. This property contains options for transactions and calls, that will be used if no other properties are provided in calls/transactions. Explicitly passed options always overwrite default options.

Parameters

1. options - `ExecutorWalletOptions`: options for `ServiceContract` constructor.

- `accountId` - string: account, that is used for making transactions against wallet contract
- `config` - any: configuration object for the executor instance
- `eventHub` - `EventHub`: `EventHub` instance
- `signer` - `SignerInterface`: `SignerInterface` instance

- `wallet` - `Wallet: Wallet` instance with a loaded wallet contract
- `web3` - `Web3: Web3` instance
- `defaultOptions` - any (optional): default options for web3 transactions/calls
- `log` - Function (optional): function to use for logging: `(message, level) => {...}`
- `logLevel` - `LogLevel` (optional): messages with this level will be logged with `log`
- `logLog` - `LogLogInterface` (optional): container for collecting log messages
- `logLogLevel` - `LogLevel` (optional): messages with this level will be pushed to `logLog`

Returns

`ExecutorWallet` instance

Example

```
const executor = new ExecutorWallet({
  accountId,
  config,
  eventHub,
  signer,
  wallet,
  web3
});
```

2.3.2 init

```
executor.init(name);
```

initialize executor

Parameters

1. **options** - any: object with the property “eventHub” (of the type `EventHub`)
 - `eventHub` - `EventHub`: The initialized `EventHub` Module.

Returns

void.

Example

```
runtime.executor.init({eventHub: runtime.eventHub})
```

2.3.3 executeContractCall

```
executor.executeContractCall(contract, functionName, ...args);
```

run the given call from contract

note, that if a from is used, this from is replaced with the wallets address

Parameters

1. `contract` - any: the target contract
2. `functionName` - string: name of the contract function to call
3. `...args` - any[]: optional array of arguments for contract call. if last arguments is {Object}, it is used as the options parameter

Returns

Promise resolves to any: contract calls result.

Example

```
const greetingMessage = await runtime.executor.executeContractCall(  
  contract, // web3.js contract instance  
  'greet' // function name  
);
```

2.3.4 executeContractTransaction

```
executor.executeContractTransaction(contract, functionName, inputOptions, ...  
  ↪functionArguments);
```

execute a transaction against the blockchain, handle gas exceeded and return values from contract function,

note, that a from passed to this function will be replaced with the wallets address and that transactions, that transfer EVEs to a target account, will be rejected

Parameters

1. `contract` - any: contract instance
2. `functionName` - string: name of the contract function to call
3. **inputOptions** - any: options object
 - `from` - string (optional): The address the call “transaction” should be made from.
 - `gas` - number (optional): The amount of gas provided with the transaction.
 - `event` - string (optional): The event to wait for a result of the transaction,

- `getEventResult` - function (optional): callback function which will be called when the event is triggered.
- `eventTimeout` - number (optional): timeout (in ms) to wait for a event result before the transaction is marked as error
- `estimate` - boolean (optional): Should the amount of gas be estimated for the transaction (overwrites `gas` parameter)
- `force` - string (optional): Forces the transaction to be executed. Ignores estimation errors
- `autoGas` - number (optional): enables `autoGas 1.1` ==> adds 10% to estimated gas costs. value capped to current block.

4. `...functionArguments` - any[]: optional arguments to pass to contract transaction

Returns

Promise resolves to: no result (if no event to watch was given), the event (if event but no `getEventResult` was given), the value returned by `getEventResult(eventObject)`.

Because an estimation is performed, even if a fixed gas cost has been set, failing transactions are rejected before being executed. This protects users from executing transactions, that consume all provided gas and fail, which is usually not intended, especially if a large amount of gas has been provided. To prevent this behavior for any reason, add a `force: true` to the options, though it is **not advised to do so**.

To allow to retrieve the result of a transaction, events can be used to receive values from a transaction. If an event is provided, the transaction will only be fulfilled, if the event is triggered. To use this option, the executor needs to have the `eventHub` property has to be set. Transactions, that contain event related options and are passed to an executor without an `eventHub` will be rejected immediately.

Example

```
const accountId = '0x...';
const greetingMessage = await runtime.executor.executeContractTransaction(
  contract,                                // web3.js contract instance
  'setData',                               // function name
  { from: accountId, },                    // perform transaction with this account
  123,                                     // arguments after the options are passed
  ↪to the contract
);
```

Provided gas is estimated automatically with a fault tolerance of 10% and then used as *gas* limit in the transaction. For a different behavior, set *autoGas* in the transaction options:

```
const greetingMessage = await runtime.executor.executeContractTransaction(
  contract,                                // web3.js contract instance
  'setData',                               // function name
  { from: accountId, autoGas: 1.05, },      // 5% fault tolerance
  123,                                     // arguments after the options are passed
  ↪to the contract
);
```

or set a fixed gas limit:

```
const greetingMessage = await runtime.executor.executeContractTransaction(
  contract, // web3.js contract instance
  'setData', // function name
  { from: accountId, gas: 100000, }, // fixed gas limit
  123, // arguments after the options are passed
  // to the contract
);
```

Using events for getting return values:

```
const contractId = await runtime.executor.executeContractTransaction(
  factory,
  'createContract', {
    from: accountId,
    autoGas: 1.1,
    event: { target: 'FactoryInterface', eventName: 'ContractCreated', },
    getResult: (event, args) => args.newAddress,
  },
);
```

2.3.5 executeSend

```
executor.executeSend(options);
```

sending funds is not supported by the walled based executor, use a regular Executor for such tasks

2.3.6 createContract

```
executor.createContract(contractName, functionArguments, options);
```

creating contracts directly is not supported by the walled based executor, use a regular Executor for such tasks

2.4 Signer Identity

Class Name	SignerIdentity
Implements	SignerInterface
Extends	Logger
Source	signer-identity.ts

The signers are used to create contract transactions and are used internally by the [Executor](#). The default runtime uses the [SignerInternal](#) helper to sign transaction.

In most cases, you won't have to use the Signer objects directly yourself, as the [Executor](#) is your entry point for performing contract transactions. *SignerIdentity* may be an exception to this rule, as it can be used to check currently used identity and account.

Note, that this signer supports using accounts and identities. If the *.from* property in the options is given as configured *activeIdentity*, transaction will be made via this identity, which requires the *underlyingAccount* to be in control of this

identity. If *.from* is given as *underlyingAccount*, transactions will be made directly from this account. Also keep in mind, that in both cases *underlyingAccount* needs to have enough funds to pay for the transactions, as this account is used to pay for them.

2.4.1 Public Properties

1. *activeIdentity* - string: identity used for transactions, usually controlled by *underlyingAccount*
 2. *underlyingAccount* - string: account, that pays for transactions used for transactions, usually controlling *activeIdentity*
-

2.4.2 constructor

```
new SignerIdentity(options, config);
```

Creates a new *SignerInternal* instance. *config* can be set up later on with *updateConfig*, if required (e.g. when initializing a circular structure).

Parameters

1. **options - SignerIdentityOptions: options for SignerIdentity constructor (runtime like object)**
 - *contractLoader* - *ContractLoader*: *ContractLoader* instance
 - *verifications* - *Verifications*: *Verifications* instance
 - *web3* - *Web3*: *Web3* instance
 - *log* - Function (optional): function to use for logging: (message, level) => {...}
 - *logLevel* - *LogLevel* (optional): messages with this level will be logged with *log*
 - *logLog* - *LogLogInterface* (optional): container for collecting log messages
 - *logLogLevel* - *LogLevel* (optional): messages with this level will be pushed to *logLog*
2. **config - SignerIdentityConfig (optional): custom config for SignerIdentity instance**
 - *activeIdentity* - string: identity used for transactions, usually controlled by *underlyingAccount*
 - *underlyingAccount* - string: account, that pays for transactions used for transactions, usually controlled by *underlyingAccount*
 - *underlyingSigner* - *SignerInterface*: an instance of a *SignerInterface* implementation; usually a *SignerInternal* instance

Returns

SignerIdentity instance

Example

```
const signer = new SignerIdentity({
  {
    contractLoader,
    verifications,
    web3,
  },
  {
    activeIdentity: await verifications.getIdentityForAccount(accounts[0], true),
    underlyingAccount: accounts[0],
    underlyingSigner,
  },
});
```

2.4.3 signAndExecuteSend

```
signer.signAndExecuteSend(options, handleTxResult);
```

Performs a value transfer transaction. This will send specified funds to identity, which will send it to target. Funds are returned if transaction fails.

Parameters

1. **options - any:**

- **from - string:** The address the call “transaction” should be made from.
- **to - string:** The address where the eve’s should be send to.
- **value - number:** Amount to send in Wei

2. **handleTxResult - function(error, receipt):** callback when transaction receipt is available or error

Example

```
const patchedInput = runtime.signer.signAndExecuteSend({
  from: '0x...', // send from this identity/account
  to: '0x...', // receiving account
  value: web3.utils.toWei('1'), // amount to send in Wei
}, (err, receipt) => {
  console.dir(arguments);
});
```

2.4.4 signAndExecuteTransaction

```
signer.signAndExecuteTransaction(contract, functionName, functionArguments, options, ↵
↵handleTxResult);
```

Create, sign and submit a contract transaction.

Parameters

1. `contract` - any: contract instance from `api.eth.loadContract(...)`
 2. `functionName` - string: function name
 3. `functionArguments` - any[]: arguments for contract creation, pass empty Array if no arguments
 4. **options** - any:
 - `from` - string: The address (identity/account) the call “transaction” should be made from.
 - `gas` - number: Amount of gas to attach to the transaction
 - `to` - string (optional): The address where the eve’s should be send to.
 - `value` - number (optional): Amount to send in Wei
 5. `handleTxResult` - function(error, receipt): callback when transaction receipt is available or error
-

2.4.5 createContract

```
signer.createContract(contractName, functionArguments, options);
```

Creates a smart contract.

Parameters

1. `contractName` - any: `contractName` from `contractLoader`
2. `functionArguments` - any[]: arguments for contract creation, pass empty Array if no arguments
3. **options** - any:
 - `from` - string: The address the call “transaction” should be made from.
 - `gas` - number: Amount of gas to attach to the transaction

Returns

Promise resolves to any: web3 instance of new contract.

2.4.6 signMessage

```
signer.signMessage(accountId, message);
```

Sign given message with accounts private key, does not work for identity.

Parameters

1. `accountId` - string: `accountId` to sign with, **cannot be done with activeIdentity**
2. `message` - string: message to sign

Returns

Promise resolves to string: signature

Example

```
const signature = await signer.signMessage(accountId, messageToSign);
```

2.4.7 updateConfig

```
signer.updateConfig(partialOptions, config);
```

Update config of *SignerIdentity* can also be used to setup verifications and accounts after initial setup and linking with other modules.

Parameters

1. **partialOptions** - { verifications: Verifications }: object with *verifications* property, e.g. a runtime
2. **config** - **SignerIdentityConfig**: custom config for *SignerIdentity* instance
 - **activeIdentity** - string: identity used for transactions, usually controlled by *underlyingAccount*
 - **underlyingAccount** - string: account, that pays for transactions used for transactions, usually controlled by *underlyingAccount*
 - **underlyingSigner** - *SignerInterface*: an instance of a *SignerInterface* implementation; usually a *SignerInternal* instance

Returns

Promise returns void: resolved when done

Example

```
// create new instance
const signer = new SignerIdentity(
  {
    contractLoader,
    verifications,
    web3,
  },
);

// use instance, e.g. reference it in other components like `verifications`
// ...

// now set verifications instance and account in signer
```

(continues on next page)

(continued from previous page)

```

signer.updateConfig(
  { verifications },
  {
    activeIdentity,
    underlyingAccount,
    underlyingSigner: signerInternal,
  },
);

```

2.5 Account Store

Class Name	AccountStore
Implements	KeyStoreInterface
Extends	Logger
Source	account-store.ts

The [AccountStore](#) implements the [KeyStoreInterface](#) and is a wrapper for a storage, where `evan.network` account ids are stored. The default [AccountStore](#) takes an account → private key mapping as a pojo as its arguments and uses this to perform lookups, when the `getPrivateKey` function is called. This lookup needs to be done, when transactions are signed by the [InternalSigner](#) (see [Signer](#)).

Note that the return value of the `getPrivateKey` function is a promise. This may not be required in the default [AccountStore](#), but this allows you to implement own implementations of the [KeyStoreInterface](#), which may enforce a more strict security behavior or are able to access other sources for private keys.

2.5.1 constructor

```
new AccountStore(options);
```

Creates a new [AccountStore](#) instance.

Parameters

1. **options - AccountStoreOptions:** options for [AccountStore](#) constructor.

- `accounts` - any: object with `accountid/privatekey` mapping
- `log` - Function (optional): function to use for logging: `(message, level) => {...}`
- `logLevel` - [LogLevel](#) (optional): messages with this level will be logged with `log`
- `logLog` - [LogLogInterface](#) (optional): container for collecting log messages
- `logLogLevel` - [LogLevel](#) (optional): messages with this level will be pushed to `logLog`

Returns

[AccountStore](#) instance

Example

```
const accountStore = new AccountStore({
  accounts: {
    '0x1234': '12479abc3df'
  }
});
```

2.5.2 getPrivateKey

```
accountStore.getPrivateKey(accountId);
```

get private key for given account

Parameters

1. accountId - string: eth accountId

Returns

Promise resolves to string: private key for this account

Example

```
const privateKey = await runtime.accountStore.getPrivateKey(
  ↪ '0x0000000000000000000000000000000000000000000000000000000000000002');
```

2.6 Event Hub

Class Name	EventHub
Extends	Logger
Source	event-hub.ts

The [EventHub](#) helper is wrapper for using contract events. These include - contract events (e.g. contract factory may trigger an event, announcing the address of the new contract) - global events (some contracts in the [evan.network](#) economy, like the *MailBox* use such global events)

2.6.1 constructor

```
new EventHub(options);
```

Creates a new EventHub instance.

Parameters

1. options - **EventHubOptions**: options for **EventHub** constructor.

- config - any: configuration object for the eventhub module
- nameResolver - **NameResolver**: **NameResolver** instance
- contractLoader - **ContractLoader**: **ContractLoader** instance
- eventWeb3 - **Web3** (optional): **Web3** instance used for event handling (metamask web3 can't handle events correct)
- log - Function (optional): function to use for logging: (message, level) => {...}
- logLevel - **LogLevel** (optional): messages with this level will be logged with log
- logLog - **LogLogInterface** (optional): container for collecting log messages
- logLogLevel - **LogLevel** (optional): messages with this level will be pushed to logLog

Returns

EventHub instance

Example

```
const eventHub = new EventHub({
  config,
  nameResolver,
  contractLoader,
});
```

2.6.2 subscribe

```
eventHub.subscribe(contractName, contractAddress, eventName, filterFunction, onEvent, ↳fromBlock);
```

subscribe to a contract event or a global EventHub event

Parameters

1. contractName - string: target contract name (must be available within **ContractLoader**)
2. contractAddress - string: target contract address
3. eventName - string: name of the event to subscribe to
4. filterFunction - function: a function that returns true or a Promise that resolves to true if onEvent function should be applied
5. onEvent - function: executed when event was fired and the filter matches, gets the event as its parameter
6. fromBlock - string (optional): get all events from this block, defaults to latest

The **NameResolver** is a collection of helper functions, that can be used for ENS interaction. These include:

- setting and getting ENS addresses
- setting and getting ENS content flags, which is used when setting data in distributed file system, especially in case of setting a description for an *ENS* address

2.7.1 constructor

```
new NameResolver(options);
```

Creates a new NameResolver instance.

Parameters

1. **options - NameResolverOptions:** options for NameResolver constructor.

- config - any: configuration object for the NameResolver instance
- executor - **Executor**: **Executor** instance
- contractLoader - **ContractLoader**: **ContractLoader** instance
- web3 - **Web3**: **Web3** instance
- log - Function (optional): function to use for logging: (message, level) => {...}
- logLevel - **LogLevel** (optional): messages with this level will be logged with log
- logLog - **LogLogInterface** (optional): container for collecting log messages
- logLogLevel - **LogLevel** (optional): messages with this level will be pushed to logLog

Returns

NameResolver instance

Example

```
const nameResolver = new NameResolver({
  cryptoProvider,
  dfs,
  executor,
  keyProvider,
  nameResolver,
  contractLoader,
  web3,
});
```

2.7.2 getAddressOrContent

```
nameResolver.getAddressOrContent(name, type);
```

get address or content of an ens entry

Parameters

1. name - string: ens domain name (plain text)
2. type - string: content type to get (address or content)

Returns

Promise resolves to string: address, returns null if not available

Example

```
const testEvanAddress = await runtime.nameResolver.getAddressOrContent('test.evan',  
  ↪ 'address');  
// returns 0x9c0Aaa728Daa085Dfe85D3C72eE1c1AdF425be49
```

2.7.3 getAddress

```
nameResolver.getAddress(name);
```

get address of an ens entry

Parameters

1. name - string: ens domain name (plain text)

Returns

Promise resolves to string: address, returns null if not available

Example

```
const testEvanAddress = await runtime.nameResolver.getAddress('test.evan');  
// returns 0x9c0Aaa728Daa085Dfe85D3C72eE1c1AdF425be49
```

2.7.4 getContent

```
nameResolver.getContent(name);
```

get content of an ens entry

Parameters

1. name - string: ens domain name (plain text)

Returns

Promise resolves to string: content, returns null if not available

Example

```
const testEvanAddress = await runtime.nameResolver.getContent('test.evan');  
// returns (encoded ipfs hash) 0x9c0Aaa728Daa085Dfe85D3C72eE1c1AdF425be49
```

2.7.5 setAddressOrContent

```
nameResolver.setAddressOrContent(name, value, accountId, domainOwnerId, type);
```

set ens name. this can be a root level domain domain.test or a subdomain sub.domain.test

Parameters

1. name - string: ens domain name (plain text)
2. value - string: ethereum address
3. accountId - string: owner of the parent domain
4. domainOwnerId - string: owner of the address to set
5. type - string: content type to set

Returns

Promise resolves to void: resolves when done

Example

```
const testEvanAddress = await runtime.nameResolver  
  .setAddressOrContent(  
    'test.evan',  
    '0x9c0Aaa728Daa085Dfe85D3C72eE1c1AdF425be49',  
    '0x00000000000000000000000000000000beef',  
    '0x00000000000000000000000000000000beef',  
    'address'  
  );  
// returns (encoded ipfs hash) 0x9c0Aaa728Daa085Dfe85D3C72eE1c1AdF425be49
```

2.7.6 setAddress

```
nameResolver.setAddress(name, address, accountId, domainOwnerId);
```

set address for ens name. this can be a root level domain domain.test or a subdomain sub.domain.test

Parameters

1. name - string: ens domain name (plain text)
2. address - string: ethereum address
3. accountId - string: owner of the parent domain
4. domainOwnerId - string: owner of the address to set

Returns

Promise resolves to void: resolves when done

Example

```
const testEvanAddress = await runtime.nameResolver
  .setAddress(
    'test.evan',
    '0x9c0Aaa728Daa085Dfe85D3C72eE1c1AdF425be49',
    '0x00000000000000000000000000000000beef',
    '0x00000000000000000000000000000000beef'
  );
```

2.7.7 setContent

```
nameResolver.setContent(name, content, accountId, domainOwnerId);
```

set content for ens name. this can be a root level domain domain.test or a subdomain sub.domain.test

Parameters

1. name - string: ens domain name (plain text)
2. content - string: ethereum address
3. accountId - string: owner of the parent domain
4. domainOwnerId - string: owner of the address to set

Returns

Promise resolves to void: resolves when done

Example

```
const testEvanAddress = await runtime.nameResolver
  .setContent (
    'test.evan',
    '0x9c0Aaa728Daa085Dfe85D3C72eE1c1AdF425be49',
    '0x00000000000000000000000000000000beef',
    '0x00000000000000000000000000000000beef'
  );
```

2.7.8 claimAddress

```
nameResolver.claimAddress(name, accountId[, domainOwnerId, price]);
```

Tries to claim node ownership from parent nodes owner, this assumes, that the parent node owner is a registrar, that supports claiming address from it (FIFS registrar or PayableRegistrar).

Parameters

1. name - string: domain name to set (plain text)
2. accountId - string: account, that executes the transaction
3. domainOwnerId - string (optional): owner of the new domain, defaults to accountId
4. value - string|number (optional): value to send (if registrar is payable)

Returns

Promise returns void: resolved when done

Example

```
// claim '123test.fifs.registrar.test.evan' with account[0] for account[1] from FIFS_
↳registrar
const domain = '123test.fifs.registrar.test.evan';
await nameResolver.claimAddress(domain, accounts[0], accounts[1]);

// claim '123test.payable.registrar.test.evan' with account[0] for account[1] from_
↳payable registrar
const domain = '123test.fifs.registrar.test.evan';
const price = await nameResolver.getPrice(domain);
await nameResolver.claimAddress(domain, accounts[0], accounts[1], price);
```

2.7.9 claimPermanentAddress

```
nameResolver.claimPermanentAddress(name, accountId[, domainOwnerId]);
```

Registers a permanent domain via registrar, can only be done by registrar owner.

Parameters

1. name - string: domain name to set (plain text)
2. accountId - string: account, that executes the transaction, has to be registrar owner
3. domainOwnerId - string (optional): owner of the new domain, defaults to accountId

Returns

Promise returns void: resolved when done

Example

```
// claim '123sample.evan' with account[0] for account[1] from registrar
const domain = '123sample.evan';
await nameResolver.claimPermanentAddress(domain, accounts[0], accounts[1]);
```

2.7.10 setPrice

```
nameResolver.setPrice(name, accountId, newPrice);
```

Set price for a registrar at a domain.

Parameters

1. name - string: ENS address of a domain owned by a registrar (e.g. 'sample.payable.test.evan')
2. accountId - string: account that performs the action (needs proper permissions for registrar)
3. newPrice - number|string (optional): new price in Wei

Returns

Promise returns void: resolved when done

Example

```
await nameResolver.setPrice(
  'payable.registrar.test.evan',
  '0x1111111111111111111111111111111111111111111111111111111111111111',
  web3.utils.toWei('5', 'ether'),
);
```

2.7.11 getPrice

```
nameResolver.getPrice(name);
```

Get price for domain (if domain is payable).

Parameters

1. name - string: a domain to check price for (e.g. 'sample.payable.test.evan')

Returns

Promise returns string: price in Wei

Example

```
console.log(await nameResolver.getPrice('payable.registrar.test.evan'));  
// Output:  
// 5000000000000000000
```

2.7.12 setValidUntil

```
nameResolver.setValidUntil(name, accountId, newPrice);
```

Set duration, that an address is valid; resoval stops after this, depending on configuration of the ENS an extra period, where owner is still available, can be granted; notice that this can only be done by parent owner of given domain.

Parameters

1. `name` - string: ENS address of a domain owned by a registrar (e.g. `'sample.payable.test.ivan'`)
2. `accountId` - string: account that performs the action; must be parent owner of given domain
3. `validUntil` - number|string: js timestamp, when name resolution stops

Returns

Promise returns void: resolved when done

Example

[illegible]

2.7.13 getValidUntil

```
nameResolver.getValidUntil(name);
```

Get timestamp, when domain will stop resolval.

Parameters

1. name - string: domain to get valid until for

Returns

Promise returns string: js timestamp, when resolver lookup will expire

Example

```
console.log(await nameResolver.getValidUntil('payable.registrar.test.evan'));  
// Output:  
// 1544630375417
```

2.7.14 claimFunds

```
namerResolver.claimFunds(name, accountId);
```

Verification funds for domain.

Parameters

1. `name` - string: ENS address of a domain owned by a registrar (e.g. `'sample.payable.test.evan'`)
2. `accountId` - string: account that performs the action (needs proper permissions for registrar)

Returns

Promise returns void: resolved when done

Example

[illegible]

2.7.15 getFactory

```
nameResolver.getFactory(contractName);
```

helper function for retrieving a factory address (e.g. 'tasks.factory.evan')

Parameters

1. `contractName` - string: name of the contract that is created by the factory

Returns

string: address of the contract factory

Example

```
const taskFactory = await runtime.nameResolver.getFactory('tasks');  
// returns '0x9c0Aaa728Daa085Dfe85D3C72eE1c1AdF425be49';
```

2.7.16 getDomainName

```
nameResolver.getDomainName(domainConfig, ...subLabels);
```

builds full domain name based on the provided domain config a module initialization.

Parameters

1. `domainConfig` - string[] | string: The domain configuration
2. `...subLabels` - string[]: array of domain elements to be looked up and added at the lefthand

Returns

string: the domain name

Example

```
const domain = runtime.nameResolver.getDomainName(['factory', 'root', 'task'];  
// returns 'task.factory.evan';
```

2.7.17 getArrayFromIndexContract

```
nameResolver.getArrayFromIndexContract(indexContract, listHash, retrievers, chain, ↵
↵triesLeft);
```

retrieve an array with all values of a list from an index contract.

Parameters

1. `indexContract` - any: Ethereum contract address (DataStoreIndex)
2. `listHash` - string: bytes32 namehash like `api.nameResolver.sha3('ServiceContract')`
3. `retrievers` - any (optional): overwrites for index or index like contract property retrievals defaults to:

```
{
  listEntryGet: 'listEntryGet',
  listLastModified: 'listLastModified',
  listLength: 'listLength',
}
```

1. `chain` - Promise: Promise, for chaining multiple requests (should be omitted when called 'from outside', defaults to `Promise.resolve()`)
2. `triesLeft` - number: tries left before quitting defaults to 10

Returns

Promise resolves to `string[]`: list of addresses

2.7.18 getArrayFromListContract

```
nameResolver.getArrayFromListContract(indexContract, count, offset, reverse, chain, ↵
↵triesLeft);
```

retrieve an array with all values of a list from an index contract.

Parameters

1. `indexContract` - any: Ethereum contract address (DataStoreIndex)
2. `count` - number (optional): how many items should be returned, defaults to 10
3. `offset` - number (optional): how many items should be skipped, defaults to 0
4. `reverse` - boolean (optional): should the list be iterated reverse, defaults to `false`
5. `chain` - Promise (optional): Promise, for chaining multiple requests (should be omitted when called 'from outside', defaults to `Promise.resolve()`)
6. `triesLeft` - number (optional): tries left before quitting defaults to 10

Returns

Promise resolves to `string[]`: list of addresses

2.7.19 `getArrayFromUintMapping`

```
nameResolver.getArrayFromUintMapping(contract, countRetriever, elementRetriever[,   
↪count, offset, reverse]);
```

retrieve elements from a contract using a count and element retriever function.

Parameters

1. `contract` - any: Ethereum contract address (`DataStoreIndex`)
2. `countRetriever` - `Function`: function which returns the count of the retrieved elements
3. `elementRetriever` - `Function`: function which returns the element of the retrieved elements
4. `count` - number (optional): number of elements to retrieve, defaults to 10
5. `offset` - number (optional): skip this many items when retrieving, defaults to 0
6. `reverse` - boolean (optional): retrieve items in reverse order, defaults to `false`

Returns

Promise resolves to `string[]`: list of addresses

2.7.20 `sha3`

```
nameResolver.sha3(input);
```

`sha3` hashes an input, substitutes `web3.utils.sha3` from geth console

Parameters

1. `input` - string | buffer: input text or buffer to hash

Returns

string: hashed output

2.7.21 `soliditySha3`

```
nameResolver.soliditySha3(...args);
```

Will calculate the sha3 of given input parameters in the same way solidity would. This means arguments will be ABI converted and tightly packed before being hashed.

Parameters

1. `args - string | buffer`: arguments for hashing

Returns

`string`: hashed output

2.7.22 namehash

```
nameResolver.namehash(inputName);
```

hash ens name for usage in contracts

Parameters

1. `inputName - string`: inputName ens name to hash

Returns

`string`: name hash

2.7.23 bytes32ToAddress

```
nameResolver.bytes32ToAddress(hash);
```

converts a bytes32 hash to address

Parameters

1. `hash - string`: bytes32 hash

Returns

`string`: converted address

2.8 Description

Class Name	Description
Extends	Description
Source	description.ts
Examples	description.spec.ts

The Description module is the main entry point for interacting with contract descriptions. It allows you to:

- get and set descriptions
- work with contracts and ENS descriptions
- create web3.js contract instances directly from an Ethereum address and its description
- The main use cases for interacting with a contracts description in your application will most probably be reading a contracts description and loading contracts via their description.

The examples folder contains some samples for getting started. With consuming or setting contract descriptions.

When setting descriptions to contracts, these contracts have to support the [Executor](#) interface.

A simple flow for working with description may look like following:

We already have smart contract, that supports the [zz](#) interface and set a description to it.

```
const address = '0x...'; // or 'test.evan' as ens name
const accountId = '0x...';
const description = {
  "public": {
    "name": "DBCP sample contract",
    "description": "DBCP sample contract description",
    "author": "dbcp test",
    "tags": [
      "example",
      "greeter"
    ],
    "version": "0.1.0",
    "dbcpVersion": 2
  }
};
await runtime.description.setDescription(address, description, accountId);
```

Now we have made some updates to our contract and we want to update its version to 0.2.0.

```
// get description const retrieved = await runtime.description.getDescription(address, accountId);
// update version const accountId = '0x0000000000000000000000000000000000000000000000000000000000000000beef';
retrieved.public.version = '0.2.0'; await runtime.description.setDescription(address, retrieved, accountId);
```

2.8.1 constructor

```
new Description(options);
```

Creates a new Description instance.

Parameters

1. options - DescriptionOptions: options for Description constructor.

- cryptoProvider - `CryptoProvider`: `CryptoProvider` instance
- dfs - `DfsInterface`: `DfsInterface` instance
- executor - `Executor`: `Executor` instance
- keyProvider - `KeyProvider`: `KeyProvider` instance
- nameResolver - `NameResolver`: `NameResolver` instance
- contractLoader - `ContractLoader`: `ContractLoader` instance
- web3 - `Web3`: `Web3` instance
- log - `Function` (optional): function to use for logging: (message, level) => {...}
- logLevel - `LogLevel` (optional): messages with this level will be logged with log
- logLog - `LogLogInterface` (optional): container for collecting log messages
- logLogLevel - `LogLevel` (optional): messages with this level will be pushed to logLog

Returns

Description instance

Example

```
const description = new Description({
  cryptoProvider,
  dfs,
  executor,
  keyProvider,
  nameResolver,
  contractLoader,
  web3,
});
```

2.8.2 getDescription

```
description.getDescription(address, accountId);
```

loads description envelope from ens or contract if an ENS address has a contract set as well and this contract has a definition, the contract definition is preferred over the ENS definition and therefore returned

Parameters

1. address - string: The ens address or contract address where the description is stored
2. accountId - string: Account id to load the contract address for

Returns

Promise returns Envelope: description as an Envelope.

Example

```
const address = '0x9c0Aaa728Daa085Dfe85D3C72eE1c1AdF425be49';
const accountId = '0x000000000000000000000000000000000000000000000000beef';
const description = await runtime.description.getDescription(address, accountId);
console.dir(description);
// Output:
// { public:
//   { name: 'DBCP sample greeter',
//     description: 'smart contract with a greeting message and a data property',
//     author: 'dbcp test',
//     tags: [ 'example', 'greeter' ],
//     version: '0.1.0',
//     dbcpVersion: 2,
//     abis: { own: [Array] } } }
```

2.8.3 setDescription

```
description.setDescription(address, envelope, accountId);
```

set description, can be used for contract addresses and ENS addresses

Parameters

1. address - string: contract address or ENS address
2. envelope - Envelope: description as an envelope
3. accountId - string: ETH account id

Returns

Promise returns void: resolved when done.

Example

```
const address = '0x...'; // or 'test.evan' as ens name
const accountId = '0x...';
const description = {
  "public": {
    "name": "DBCP sample contract",
    "description": "DBCP sample contract description",
    "author": "dbcp test",
    "tags": [
      "example",
```

(continues on next page)

(continued from previous page)

```

    "greeter"
  ],
  "dbcpVersion": 2,
  "version": "0.1.0"
}
};
await runtime.description.setDescription(address, description, accountId);

```

2.8.4 validateDescription

Descriptions are validated when setting them. A list of known DBCP definition schemas is maintained in [description.schema.ts](#). If a description is set, its property *dbcpVersion* will be used for validating the description, if *dbcpVersion* is not provided, version 1 is used and a warning is logged.

Descriptions can be checked against the validator before setting them.

```
description.validateDescription(envelope);
```

try to validate description envelope; throw Error if validation fails

Parameters

1. envelope - Envelope: envelop with description data; private has to be unencrypted

Returns

Promise returns `boolean|any[]`: true if valid or array of issues.

Example

```

const brokenDescription = {
  "public": {
    "name": "DBCP sample contract with way to few properties",
  }
};
console.log(runtime.description.validateDescription(brokenDescription));
// Output:
// [ { keyword: 'required',
//     dataPath: '',
//     schemaPath: '#/required',
//     params: { missingProperty: 'description' },
//     message: 'should have required property \'description\' },
//   { keyword: 'required',
//     dataPath: '',
//     schemaPath: '#/required',
//     params: { missingProperty: 'author' },
//     message: 'should have required property \'author\' },
//   { keyword: 'required',
//     dataPath: '',
//     schemaPath: '#/required',

```

(continues on next page)

(continued from previous page)

```
//      params: { missingProperty: 'version' },
//      message: 'should have required property \'version\'' } ]
```

```
const workingDescription = {
  "public": {
    "name": "DBCP sample contract",
    "description": "DBCP sample contract description",
    "author": "dbcp test",
    "tags": [
      "example",
      "greeter"
    ],
    "version": "0.1.0"
  }
};

console.log(runtime.description.validateDescription(workingDescription));
// Output:
// true
```

2.8.5 = Contract =

2.8.6 getDescriptionFromContract

```
description.getDescriptionFromContract(address, accountId);
```

loads description envelope from contract

Parameters

1. `address - string`: The ens address or contract address where the description is stored
2. `accountId - string`: Account id to load the contract address for

Returns

Promise returns Envelope: description as an Envelope.

Example

```
const address = '0x9c0Aaa728Daa085Dfe85D3C72eE1c1AdF425be49';
const accountId = '0x00000000000000000000000000000000beef';
const description = await runtime.description.getDescriptionFromContract(address,
  accountId);
console.dir(description);
// Output:
// { public:
//   { name: 'DBCP sample greeter',
//     description: 'smart contract with a greeting message and a data property',
//     author: 'dbcp test',
```

(continues on next page)

(continued from previous page)

```
//      tags: [ 'example', 'greeter' ],  
//      version: '0.1.0',  
//      dbcpVersion: 2,  
//      abis: { own: [Array] } } }
```

2.8.7 setDescriptionToContract

```
description.setDescriptionToContract(contractAddress, envelope, accountId);
```

store description at contract

Parameters

1. contractAddress - string: The contract address where description will be stored
2. envelope - Envelope: description as an envelope
3. accountId - string: ETH account id

Returns

Promise returns void: resolved when done.

Example

```
const address = '0x...';  
const accountId = '0x...';  
const description = {  
  "public": {  
    "name": "DBCP sample contract",  
    "description": "DBCP sample contract description",  
    "author": "dbcp test",  
    "tags": [  
      "example",  
      "greeter"  
    ],  
    "version": "0.1.0",  
    "dbcpVersion": 2  
  }  
};  
await runtime.description.setDescriptionToContract(address, description, accountId);
```

2.8.8 = ENS =

ENS addresses are able to hold multiple values at once. So they may be holding a contract address and a description. If this is the case and the contract at the ENS address has another description, the contracts description is preferred over the ENS description. If you explicitly intend to retrieve an ENS endpoints description and want to ignore the contracts description, use the function *getDescriptionFromEns*.

2.8.9 getDescriptionFromEns

```
description.getDescriptionFromEns(address);
```

loads description envelope from ens

Parameters

1. ensAddress - string: The ens address where the description is stored

Returns

Promise returns Envelope: description as an Envelope.

Example

```
const address = '0x9c0Aaa728Daa085Dfe85D3C72eE1c1AdF425be49';
const accountId = '0x000000000000000000000000000000000000beef';
const description = await runtime.description.getDescriptionFromContract(address, ↵
↵accountId);
console.dir(description);
// Output:
// { public:
//   { name: 'DBCP sample greeter',
//     description: 'smart contract with a greeting message and a data property',
//     author: 'dbcp test',
//     tags: [ 'example', 'greeter' ],
//     version: '0.1.0',
//     dbcpVersion": 2,
//     abis: { own: [Array] } } }
```

2.8.10 setDescriptionToEns

```
description.setDescriptionToEns(ensAddress, envelope, accountId);
```

store description at contract

Parameters

1. `contractAddress` - string: The ens address where description will be stored
2. `envelope` - Envelope: description as an envelope
3. `accountId` - string: ETH account id

Returns

Promise `returns` void: resolved when done.

Example

```
const address = '0x...';
const accountId = '0x...';
const description = {
  "public": {
    "name": "DBCP sample contract",
    "description": "DBCP sample contract description",
    "author": "dbcp test",
    "tags": [
      "example",
      "greeter"
    ],
  },
  "version": "0.1.0",
  "dbcpVersion": 2
};
await runtime.description.setDescriptionToEns(address, description, accountId);
```

2.9 Wallet

Class Name	Wallet
Extends	Logger
Source	wallet.ts
Examples	wallet.spec.ts

The `Wallet` module is a wrapper for the `MultiSigWallet` and allows to create wallets, manage owners and execute transactions with it. One of the common use cases is setting confirmation count to 1 (which is the default value). This basically makes the wallets a multiowned account, where all parties are able to perform transactions on their own.

The `Wallet` module is bound to a wallet contract instance. This instance can be loaded, which requires an existing `MultiSigWallet` contract and a user, that is able to work with it:

```
wallet.load('0x0123456789012345678901234567890123456789');
```

If no wallet exists, a new wallet can be created with:

```
// account id, that creates the wallet
const accountId = '0x0000000000000000000000000000000000000000000000000000000000000001';
// account, that will be able to manage the new wallet
```

(continues on next page)

(continued from previous page)

```
const manager = '0x0000000000000000000000000000000000000000000000000000000000000001';
// wallet owners
const owners = [
  '0x0000000000000000000000000000000000000000000000000000000000000001',
  '0x0000000000000000000000000000000000000000000000000000000000000002',
];
await wallet.create(accountId, manager, owners);
```

The last example creates a wallet

- with the account `0x0001` (used as `accountId`)
- that can be managed (adding/removing owners) by `0x0001` (used as `manager`)
- that allows the accounts `0x0001` and `0x0002` to perform transactions (used as `owners`)

Creating wallets via the `Wallet` module loads them to the module as well, so there is no need to call the `load` function after that.

So now we have a valid and working wallet loaded to our module and start performing calls. Let's say, we have an instance of an owned contract and want to transfer its ownership, we can do:

```
const accountId = '0x0000000000000000000000000000000000000000000000000000000000000001';
const newOwner = '0x0000000000000000000000000000000000000000000000000000000000000002';
await wallet.submitTransaction(ownedContract, 'transferOwnership', { from: accountId, ↵
↵ }, newOwner);
```

Looks pretty simple, but comes with a few things to consider:

- `accountId` is the account, that performs the transaction from the “outside perspective” (if you look into a chain explorer like the [evan.network Test-Explorer](#) you'll see, that account `accountId` is actually performing the transaction and not the wallet contract)
- `accountId` pays the gas cost for the transaction (obviously, when considering the last point)
- from the perspective of the target contract `ownedContract`, the wallet contract (either loaded or created beforehand) is performing the transaction
- taken our example transaction `transferOwnership`, **the wallet contract has to be the current owner** and not the account `accountId`

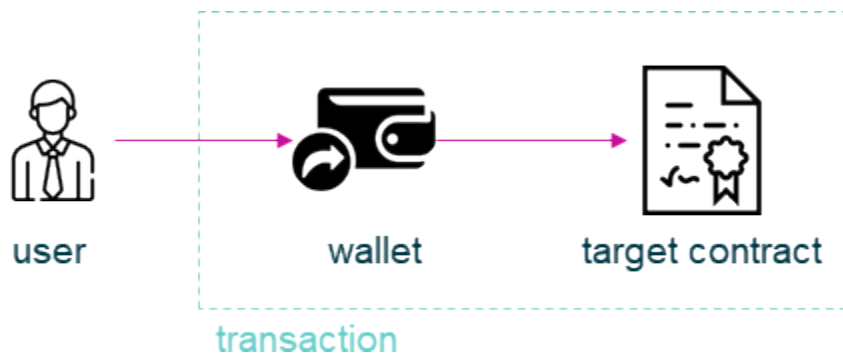


Fig. 3: transaction flow in wallet based transactions

An implementation of an `Executor` module, that uses a wallet for contract execution, has been created as `ExecutorWallet`. Because the profiles are (externally owned) account based and many modules rely on pro-

files for data encryption and decryption, the `ExecutorWallet` module should be used with care, when dealing with operations, that require en- or decryption.

2.9.1 constructor

```
new Wallet(options);
```

Creates a new `Wallet` instance.

Parameters

1. **options - `WalletOptions`:** options for `Wallet` constructor.

- `contractLoader` - `ContractLoader`: `ContractLoader` instance
- `description` - `Description`: `Description` instance
- `eventHub` - `EventHub`: `EventHub` instance
- `executor` - `Executor`: `Executor` instance
- `nameResolver` - `NameResolver`: `NameResolver` instance

Returns

`Wallet` instance

Example

```
const wallet = new Wallet({
  contractLoader,
  description,
  eventHub,
  executor,
  nameResolver,
});
```

2.9.2 = Contract Management =

2.9.3 create

```
wallet.create(accountId, manager, owners);
```

Create a new wallet contract and uses it as its wallet contract.

Parameters

1. `accountId - string`: account id, that creates the wallet
2. `manager - string`: account, that will be able to manage the new wallet
3. `owners - string[]`: wallet owners
4. `confirmations - number` (optional): number of confirmations required to complete a transaction, defaults to 1

Returns

Promise returns void: resolved when done

Example

```
await wallet.create(accounts[0], accounts[0], [accounts[0]]);
```

2.9.4 load

```
wallet.load(contractId[, walletType]);
```

Load wallet contract from address and uses it as its wallet contract.

Parameters

1. `contractid - string`: a wallet contract address
2. `walletType - string` (optional): wallet contract type, defaults to `MultiSigWallet`

Returns

Promise returns void: resolved when done

Example

```
wallet.load('0x0123456789012345678901234567890123456789');
```

2.9.5 = Transactions =

2.9.6 submitTransaction

```
wallet.submitTransaction(target, functionName, inputOptions[, ...functionArguments]);
```

Submit a transaction to a wallet, as required is fixed to 1, this will immediately execute the transaction.

Parameters

1. `target` - any: contract of the submitted transaction
2. `functionName` - string: name of the contract function to call
3. `inputOptions` - any: currently supported: `from`, `gas`, `value`, `event`, `getEventResult`, `eventTimeout`, `estimate`, `force`
4. `functionArguments` - any[]: optional arguments to pass to contract transaction

Returns

Promise returns any: status information about transaction

Example

```
await wallet.submitTransaction(testContract, 'transferOwnership', { from: accounts[0],
↪ }, accounts[1]);
```

2.9.7 = Account Management =

2.9.8 addOwner

```
wallet.addOwner(accountId, toAdd);
```

Function description

Parameters

1. `accountId` - string: account with management permissions on wallet
2. `toAdd` - string: account to add as an owner

Returns

Promise returns void: resolved when done

Example

```
await wallet.addOwner(accounts[0], accounts[1]);
```

2.9.9 removeOwner

```
initializedModule.removeOwner(arguments);
```

Remove an owner from a wallet contract.

Parameters

1. `accountId` - string: account with management permissions on wallet
2. `toAdd` - string: account to remove from wallet owners

Returns

Promise returns void: resolved when done

Example

```
await wallet.removeOwner(accounts[0], accounts[1]);
```

2.9.10 getOwners

```
wallet.getOwners();
```

Get all owners of a wallet.

Parameters

(none)

Returns

Promise returns `string[]`: array of account ids

Example

```
console.dir(await wallet.getOwners())
// Output:
// [ '0x0123456789012345678901234567890123456789' ]
```

2.10 Votings

Class Name	Wallet
Extends	Logger
Source	votings.ts
Examples	votings.spec.ts

The votings helper allows to hold votes over decisions and execute them, if enough votes have been submitted.

The usual flow for proposals has the following steps:

0. if not already created, create a voting contract with *createContract* (this doesn't have to be done when using a central voting contract (e.g. from a well known ENS address))
 1. create a proposal for a change with *createProposal*
 2. let participants vote with *vote*
 3. if enough time has passed and you have enough votes, you can finally execute your proposal with *execute*
-

2.10.1 constructor

```
new Votings(options);
```

Creates a new Votings instance.

Parameters

1. **options - VotingsOptions:** options for Votings constructor.
 - `contractLoader` - `ContractLoader`: `ContractLoader` instance
 - `executor` - `Executor`: `Executor` instance

Returns

Votings instance

Example

```
const votings = new Votings({
  contractLoader,
  executor,
});
```

2.10.2 = Contract =

2.10.3 createContract

```
votings.createContract(accountId, votingsContractOptions);
```

Create new voting contract instance.


```
await votings.addMember(
  '0x0000000000000000000000000000000000000000000000000000000000000000c0274ac7',
  '0x1111111111111111111111111111111111111111111111111111111111111111',
  '0x2222222222222222222222222222222222222222222222222222222222222222',
  { name: 'Member Number 2' },
);
```

2.10.6 removeMember

```
votings.removeMember(votingsContract, votingsOwner, toRemove);
```

Remove member from votings contract.

Parameters

1. contract - string|any: web3 voting contract instance or contract address
2. accountId - string: account that performs the action
3. targetAccount - string: account to remove from votings contract

Returns

Promise returns void: resolved when done

Example

```
await votings.removeMember(
  '0x0000000000000000000000000000000000000000000000000000000000000000c0274ac7',
  '0x1111111111111111111111111111111111111111111111111111111111111111',
  '0x2222222222222222222222222222222222222222222222222222222222222222',
);
```

2.10.7 getMemberInfo

```
votings.getMemberInfo(votingsContract, accountId);
```

Get info of a member.

Parameters

1. contract - string|any: web3 voting contract instance or contract address
2. targetAccount - string: account to get info for

Returns

Promise returns *MemberInfo*: member info

Example

```
console.dir(await votings.getMemberInfo(
  '0x000000000000000000000000000000000000000000000000c0274ac7',
  '0x2222222222222222222222222222222222222222222222222222222',
));
// Output:
// {
//   address: '0x2222222222222222222222222222222222222222222222222',
//   name: 'Member Number 2',
//   memberSince: 1544092270556
// }
```

2.10.8 isMember

```
votings.isMember(votingsContract, accountId);
```

Checks if a given account is member in voting contract.

Parameters

1. contract - string|any: web3 voting contract instance or contract address
2. targetAccount - string: account to get info for

Returns

Promise returns bool: true if member

Example

```
console.dir(await votings.isMember(
  '0x000000000000000000000000000000000000000000000000c0274ac7',
  '0x2222222222222222222222222222222222222222222222222222222',
));
// Output:
// true
```


2.10.9 = Proposals =

2.10.10 createProposal

```
votings.createProposal(votingsContract, accountId, proposalOptions);
```

Create a new proposal in votings contract.

Parameters

1. contract - string|any: web3 voting contract instance or contract address
2. accountId - string: account that performs the action
3. proposalOptions - *ProposalOptions*: options for proposal

Returns

Promise returns string: id of new proposal

Example

```
// make a proposal about a suggestion (text only)
const textProposal = await votings.createProposal(
  '0x0000000000000000000000000000000000000000000000000000000000000000c0274ac7',
  '0x1111111111111111111111111111111111111111111111111111111111111111',
  { description: 'Change voting time to 2 hours.' },
);

// propose a transaction
const txProposal = await votings.createProposal(
  '0x0000000000000000000000000000000000000000000000000000000000000000c0274ac7',
  '0x1111111111111111111111111111111111111111111111111111111111111111',
  {
    description: 'set data of this contract to "def"',
    data: '0x47064d6a' +
      '000000000000000000000000000000000000000000000000000000000000000020' +
      '000000000000000000000000000000000000000000000000000000000000000003' +
      '646566000000000000000000000000000000000000000000000000000000000000',
    to: '0x0000000000000000000000000000000000000000000000000000000000000000a2074340c0274ac7',
  },
);
```

2.10.11 getProposalCount

```
votings.getProposalCount(contract);
```

Get number of proposals in votings contract.

Parameters

1. `contract - string|any`: web3 voting contract instance or contract address

Returns

Promise returns `number`: number of proposals

Example

```
await votings.createProposal(
  '0x0000000000000000000000000000000000000000000000000000000000000000c0274ac7',
  '0x1111111111111111111111111111111111111111111111111111111111111111',
  {
    description: 'set data of this contract to "def"',
    data: '0x47064d6a' +
      '000000000000000000000000000000000000000000000000000000000000000020' +
      '000000000000000000000000000000000000000000000000000000000000000003' +
      '646566000000000000000000000000000000000000000000000000000000000000',
    to: '0x0000000000000000000000000000000000000000000000000000000000000000a2074340c0274ac7',
  },
);
const count = await votings.getProposalCount(
  ↪ '0x0000000000000000000000000000000000000000000000000000000000000000c0274ac7');
console.log(count);
// Output:
// 1
```

2.10.12 getProposalInfo

```
votings.getProposalInfo(votingsContract, proposalId);
```

Gets info about a given proposal in contract.

Parameters

1. `contract - string|any`: web3 voting contract instance or contract address
2. `proposalId - string`: id of proposal to retrieve info for

Returns

Promise returns *ProposalInfo*: info about proposal

Example

```

console.dir(await votings.getProposalInfo(
  '0x0000000000000000000000000000000000000000000000000000000000000000c0274ac7',
  '0',
));
// Output:
// {
//   currentResult: '0',
//   description: 'Change voting time to 2 hours.',
//   executed: false,
//   minExecutionDate: 1544093505000,
//   numberOfVotes: '0',
//   proposalHash: '0xa86d54e9aab41ae5e520ff0062ff1b4cbd0b2192bb01080a058bb170d84e6457
→',
//   proposalPassed: false,
//   to: '0x0000000000000000000000000000000000000000000000000000000000000000',
//   value: '0'
// }

```

2.10.13 getProposalInfos

```
votings.getProposalInfos(contract[, count, offset, reverse]);
```

Get multiple proposals from votings contract.

Parameters

1. contract - string|any: web3 voting contract instance or contract address
2. count - number (optional): number of items to retrieve, defaults to 10
3. offset - number (optional): skip this many entries, defaults to 0
4. reverse - boolean (optional): fetch entries, starting with last entry, defaults to true

Returns

Promise returns *ProposalInfos*: proposals listing

Example

```

await votings.createProposal(
  '0x0000000000000000000000000000000000000000000000000000000000000000c0274ac7',
  '0x1111111111111111111111111111111111111111111111111111111111111111',
  {
    description: 'set data of this contract to "def"',
    data: '0x47064d6a' +
      '000000000000000000000000000000000000000000000000000000000000000020' +
      '000000000000000000000000000000000000000000000000000000000000000003' +
      '646566000000000000000000000000000000000000000000000000000000000000',
    to: '0x0000000000000000000000000000000000000000000000000000000000000000a2074340c0274ac7',
  },
);

```

(continues on next page)

(continued from previous page)

```
);  
const proposals = await votings.getProposalInfos(  
  ↪ '0x00000000000000000000000000000000000000000000c0274ac7');  
console.log(proposals.results.length);  
// Output:  
// 1
```

2.10.14 vote

```
votings.vote(votingsContract, accountId, proposal, accept[, comment]);
```

Vote for a proposal.

Parameters

1. contract - string|any: web3 voting contract instance or contract address
2. accountId - string: account that performs the action
3. proposal - string: id of proposal to vote for
4. accept - boolean: accept proposal or not
5. comment - string (optional): comment for vote, left empty if omitted

Returns

Promise returns void: resolved when done

Example

```
await votings.vote(  
  '0x00000000000000000000000000000000000000000000c0274ac7',  
  '0x222222222222222222222222222222222222222222222222222',  
  '1',  
  true,  
);
```

2.10.15 execute

```
votings.execute(votingsContract, accountId, proposal[, data]);
```

Execute a proposal.

ProposalInfo

1. `currentResult - number`: current number of positive votes
2. `description - string`: description text
3. `executed - boolean`: true if already executed
4. `minExecutionData - number`: earliest day of execution
5. `numberOfVotes - number`: number of submitted votes
6. `proposalHash - string`: checksum of proposal: keccak256(beneficiary, weiAmount, transactionBytecode)
7. `proposalPassed - boolean`: true if executed and proposal passed
8. `to - string`: target of proposal (contract/account to send transaction to)
9. `value - string`: amount of Wei to send to target

ProposalInfos

1. `results - ProposalInfo[]`: proposals of current page (length is 10)
2. `totalCount - number`: total number of results

ProposalOptions

1. `description - string`: description text
2. `data - string (optional)`: input data for proposal, defaults to 0x
3. `to - string (optional)`: target of proposal (contract/account to send transaction to), defaults to 0x00
4. `value - string (optional)`: amount of Wei to send to target, defaults to 0

VotingsContractOptions

1. `minimumQuorumForProposals - number`: votes that must have been given before any proposal is accepted; updates to this may affect running proposals
2. `minutesForDebate - number`: time to have passed before a proposal can be accepted; updates to this do not affect running proposals
3. `marginOfVotesForMajority - number`: accepting votes that must have been given before any proposal is accepted; updates to this may affect running proposals

2.11 Payments

Class Name	Payments
Extends	Logger
Source	payments.ts
Examples	payments.spec.ts

Payments are a Service to open unidirectional payment channels to other accounts. You can open a payment channel to another account and do some micropayments offchain.

The heart of the system lies in its sender -> receiver off-chain transactions. They offer a secure way to keep track of the last verified channel balance. The channel balance is calculated each time the sender pays for a resource. He is prompted to sign a so-called balance proof, i.e., a message that provably confirms the total amount of transferred tokens. This balance proof is then sent to the receiver's server. If the balance proof checks out after comparing it with the last received balance and verifying the sender's signature, the receiver replaces the old balance value with the new one.

2.11.1 constructor

```
new Payments(options);
```

Creates a new Payments instance.

Parameters

1. **options - PaymentOptions:** options for Votings constructor.

- `accountStore` - `AccountStore`: `AccountStore` instance
- `contractLoader` - `ContractLoader`: `ContractLoader` instance
- `executor` - `Executor`: `Executor` instance
- `web3` - `Web3`: `Web3` instance

Returns

Payments instance

Example

```
const payments = new Payments({
  accountStore,
  contractLoader,
  executor,
  web3,
});
```

2.11.2 closeChannel

```
payments.closeChannel(closingSig);
```

Closes a given payment channel, when a closing signature is available, the channel will be closed cooperately, otherwise the channel will be close uncooperately and the sender or receiver has to wait a given amount of blocks (500) to get the funds out of the payment channel

Parameters

1. `closingSig` - string (optional): Cooperative-close signature from receiver

Returns

Promise returns void: resolved when done

Example

```
await payments.closeChannel('0x00000000000000000000000000000000c0274ac7');
```

2.11.3 confirmPayment

```
payments.confirmPayment(proof);
```

Persists `next_proof` to `proof`. This method must be used after successful payment request, or right after `signNewProof` is resolved, if implementation don't care for request status

Parameters

1. `proof` - *MicroProof*: given microproof object after calling `incrementBalanceAndSign`

Returns

Promise returns void: resolved when done

Example

```
payments.confirmPayment({  
  balance: 1,  
  sig: '0x1234567899'  
});
```

2.11.4 getChannelInfo

```
payments.getChannelInfo(channel);
```

Get channel details such as current state (one of opened, closed or settled), block in which it was set and current deposited amount

Parameters

1. `channel` - *MicroChannel*: Channel to get info from. Default to channel

Returns

Promise returns *MicroChannelInfo*: member info

Example

```
await payments.getChannelInfo();
```

2.11.5 getChallengePeriod

```
payments.getChallengePeriod();
```

Get contract's configured challenge's period. As it calls the contract method, can be used for validating that contract's address has code in current network

Parameters

Returns

Promise returns number: challenge period number, in blocks

Example

```
console.dir(await payments.getChallengePeriod());  
// Output:  
// 500
```

2.11.6 getClosingSig

```
payments.getClosingSig(account);
```

Get the closing balance signature signed from the defined account. This signature can be used to transfer it from the receiver to the sender when the sender wants to close the payment channel. Otherwise when the receiver wants to close the channel cooperative he uses the closign signature to close th channel directly.

Parameters

1. `account - string`: AccountId which should sign the closing signature (mostly the current active account)

Returns

Promise returns string: signed closing signature

Example

```
console.dir(await payments.getClosingSig(account));  
// Output:  
// 0x1234567890ABCDEF
```

2.11.7 isChannelValid

```
payments.isChannelValid(channel);
```

Health check for currently configured channel info

Parameters

1. `channel` - *MicroChannel*: Channel to get info from. Default to channel

Returns

boolean: True if channel is valid, false otherwise

Example

```
console.dir(payments.isChannelValid(channel));  
// Output:  
// True
```

2.11.8 incrementBalanceAndSign

```
payments.incrementBalanceAndSign(amount);
```

Ask user for signing a payment, which is previous balance incremented of amount. Warnings from `signNewProof` applies.

Parameters

1. `amount` - `BigNumber|string`: Amount to increment in current balance

Returns

Promise returns string: signed signature

Example

```
payments.loadChannelFromBlockchain(account, receiver);
```

Parameters

- ## Returns

[illegible]

```
payments.openChannel(account, receiver, deposit);
```

Parameters

- ## Returns

71

Example

```
await payments.openChannel('0x222222222222222222222222222222222222',
↪ '0x2222222222222222222222222222222222223', new BigNumber(5));
```

2.11.11 setChannelManager

```
payments.setChannelManager(channelManager);
```

sets a new channelmanager contract to the current instance

Parameters

1. channelManager - string: the new channelmanager address

Returns

void

Example

```
payments.setChannelManager('0x222222222222222222222222222222222222');
```

2.11.12 setChannel

```
payments.setChannel(channel);
```

Set channel info. Can be used to externally [re]store an externally persisted channel info

Parameters

1. channelManager - *MicroChannel*: Channel info to be set

Returns

void

Example

```
payments.setChannel({
  account: '0x1234',
  receiver: '0x1234'
  block: 12346,
  proof: {
    balance: 1,
    sig: '0x12345677899'
  }
});
```

2.11.13 signNewProof

```
payments.signNewProof(proof);
```

Ask user for signing a channel balance. Notice it's the final balance, not the increment, and that the new balance is set in `next_proof`, requiring a `confirmPayment` call to persist it, after successful request.

Implementation can choose to call `confirmPayment` right after this call resolves, assuming request will be successful after payment is signed.

Parameters

1. `proof` - *MicroProof* (optional): Balance proof to be signed

Returns

Promise returns *MicroProof*: signature

Example

```
payments.signNewProof({
  balance: 1,
  sig: '0x12345677899'
});
```

2.11.14 signMessage

```
payments.signMessage(msg);
```

Ask user for signing a string with `eth_accounts_sign`

Parameters

1. `msg` - string: Data to be signed

Returns

Promise returns string: signed data

Example

```
await payments.signMessage('This is a message');
```

2.11.15 topUpChannel

```
payments.topUpChannel(deposit);
```

Top up current channel, by depositing some [more] EVE to it

Parameters

1. deposit - `BigNumber`|string: EVE (in wei) to be deposited in the channel

Returns

Promise returns void: resolved when done

Example

```
await payments.topUpChannel(new BigNumber(5));
```

2.11.16 = Additional Components =

2.11.17 Interfaces

MicroProof

1. balance - `BigNumber`: balance value
2. sig - string (optional): balance signature

MicroChannel

1. account - string: Sender/client's account address
2. receiver - string: Receiver/server's account address
3. block - number: Open channel block number
4. proof - *MicroProof*: Current balance proof

5. `next_proof` - *MicroProof* (optional): Next balance proof, persisted with `confirmPayment`
6. `closing_sig` - string (optional): Cooperative close signature from receiver

MicroChannelInfo

1. `state` - string: Current channel state, one of 'opened', 'closed' or 'settled'
2. `block` - number: Block of current state (opened=open block number, closed=channel close requested block number, settled=settlement block number)
3. `deposit` - BigNumber: Current channel deposited sum
4. `withdrawn` - BigNumber: Value already taken from the channel

This section contains modules, that deal with basic blockchain interactions, like

- performing calls and transaction to contracts
- signing transactions
- handle account private keys
- use on-chain services, like ENS name resolver and event emitting contracts
- describe contracts and ENS addresses

3.1 Logger

Class Name	Logger
Source	logger.ts

The `Logger` class is used throughout the package for logging events, updates and errors. Logs can be written by classes, that inherit from the `Logger` class, by using the `this.log` function. A log level can be set by its second parameter:

```
this.log('hello log', 'debug');
```

All log messages without a level `default` to level `'info'`. If not configured otherwise,
 ↳ the following behavior is used:

- drop all log messages but errors
- log errors to console.error

It can be useful for analyzing issues to increase the log level. You can do this in two ways:

- Set the environment variable `DBCP_LOGLEVEL` to a level matching your needs, which increases the log level for all modules and works with the default runtime. For example:

```
export DBCP_LOGLEVEL=info
```

- When creating a custom runtime, set the `logLevel` property to a value matching your needs, when creating any module instance. This allows you to change log level for single modules, but requires you to create a custom runtime, e.g.:

```
const { ContractLoader } = require('@evan.network/dbcp');
const Web3 = require('web3');

// web3 instance for ContractLoader
```

(continues on next page)

(continued from previous page)

```
const provider = new Web3.providers.WebsocketProvider({ '...' });
const web3 = new Web3(provider, null, { transactionConfirmationBlocks: 1 });

// custom log level 'info'
const contractLoader = new ContractLoader({ web3, logLevel: 'info', });
```

All loggers can have a custom LogLog storage where all logmessages with a given level will be stored. You can access the storage for the current logger module at the property logLog. All messages are stored with the following markup:

```
{
  timestamp, // current date in millis
  level, // loglevel of the message
  message // log message
}
```

You can configure the current LogLogLevel at the property logLogLevel at instantiation of your module.

3.1.1 constructor

```
new Logger(options);
```

Creates a new Logger instance.

Parameters

1. options - LoggerOptions: options for Logger constructor.

- log - Function (optional): function to use for logging: (message, level) => {...}
- logLevel - LogLevel (optional): messages with this level will be logged with log
- logLog - LogLogInterface (optional): container for collecting log messages
- logLogLevel - LogLevel (optional): messages with this level will be pushed to logLog

Returns

Logger instance

Example

```
const logger = new Logger();
```

3.1.2 log

```
logger.log(message, level);
```

log message with given level

Parameters

1. message - string: log message
2. level - string: log level as string, defaults to 'info'

Example

```
runtime.executor.log('test', 'error');
```

3.1.3 = Additional Components =

Interfaces

LogLogInterface

A different LogLog storage can be attached to the logger instance of the module. The storage must implement the following functions (default array like instance)

```
export interface LogLogInterface {
  push: Function;
  map: Function;
  filter: Function;
};
```

Enums

LogLevel

Available LogLevels for the logger instance, free definable between error and gasLog

```
export enum LogLevel {
  debug,
  info,
  notice,
  warning,
  error,

  gasLog = 100,
  disabled = 999,
};
```

3.2 Validator

Class Name	Validator
Extends	Logger
Source	validator.ts
Examples	validator.spec.ts

The Validator module can be used to verify given JSON schemas.

3.2.1 constructor

```
new Validator(options);
```

Creates a new Validator instance.

Parameters

1. **options - ValidatorOptions:** options for Validator constructor.

- schema - any: the validation schema definition
- log - Function (optional): function to use for logging: (message, level) => {...}
- logLevel - LogLevel (optional): messages with this level will be logged with log
- logLog - LogLogInterface (optional): container for collecting log messages
- logLogLevel - LogLevel (optional): messages with this level will be pushed to logLog

Returns

Validator instance

Example

```
const nameResolver = new Validator({  
  schema  
});
```

3.2.2 validate

```
validator.validate(data);
```

validate a given data object with the instantiated schema

Parameters

1. data - any: to be validated data

Returns

bool | strings[]: true if data is valid, array of object if validation is failed

3.2.3 getErrorsAsText

```
validator.getErrorsAsText();
```

returns errors as text if previous validation was failed

Returns

string: all previous validation errors concatenated as readable string

3.3 Utils

Source	utils.ts
--------	--------------------------

Utils contain helper functions which are used across the whole project

3.3.1 promisify

```
utils.promisify(funThis, functionName, ...args);
```

run given function from this, use function(error, result) {...} callback for promise resolve/reject

Parameters

1. funThis - any: the functions 'this' object
2. functionName - string: name of the contract function to call
3. ...args - any: any additional parameters that should be passed to the called function

Returns

Promise resolves to any: the result from the function(error, result) {...} callback.

Example

```
runtime.utils
  .promisify(fs, 'readFile', 'somefile.txt')
  .then(content => console.log('file content: ' + content))
```

3.3.2 obfuscate

```
utils.obfuscate(text);
```

obfuscates strings by replacing each character but the last two with 'x'

Parameters

1. text - string: text to obfuscate

Returns

string: obfuscated text

Example

```
const obfuscated = runtime.utils.obfuscate('sample text');  
// returns 'sample texx'
```

3.3.3 getSmartAgentAuthHeaders

```
utils.getSmartAgentAuthHeaders(runtime[, message]);
```

create auth header data to authenticate with current account against a smart agent server

Parameters

1. runtime - Runtime: an initialized runtime
2. message - string (optional): message to sign, defaults to current timestamp

Returns

Promise resolves to string: auth header value as string

Example

```
const authData = await getSmartAgentAuthHeaders(runtime);  
console.log(authData);  
// Output:  
// EvanAuth 0x001De828935e8c7e4cb56Fe610495cAe63fb2612,EvanMessage 1566569193297,  
↳EvanSignedMessage↳  
↳0x4ce5c94b3fb77e6fbd7dcbbbedc564058d841c849020f11514b7e525776b033eb6cb54f480b604ae7dccb9858eb1162670
```

This section contains modules, that are used throughout the code, the `Logger` for example is the base class for almost every module and enables them to log in a structured manner.

4.1 Contract Loader

Class Name	ContractLoader
Extends	Logger
Source	contract-loader.ts
Examples	contract-loader.spec.ts

The [ContractLoader](#) is used when loading contracts without a DBCP description or when creating new contracts via bytecode. In both cases additional information has to be passed to the [ContractLoader](#) constructor.

Loading contracts requires an abi interface as a JSON string and creating new contracts requires the bytecode as hex string. Compiling Ethereum smart contracts with [solc](#) provides these.

Abis, that are included by default are:

- AbstractENS
- Described
- EventHub
- Owned
- PublicResolver

Bytecode for these contracts is included by default:

- Described
- Owned

Following is an example for loading a contract with a custom abi. The contract is a [Greeter Contract](#) and a shortened interface containing only the *greet* function is used here.

They can be side-loaded into an existing contract loader instance, e.g. into a runtime:

```
runtime.contractLoader.contracts['Greeter'] = {
  "interface": "[{"constant":true,"inputs":[],"name":"greet","outputs":[{"
↪ "name":"","type":"string"}],"payable":false,"stateMutability":"view","
↪ "type":"function"}]",
};
```

4.1.1 getCompiledContract

```
contractLoader.getCompiledContract (name);
```

gets contract from a solc compilation

Parameters

1. name - string: Contract name

Returns

any: The compiled contract.

Example

```
const address = '0x9c0Aaa728Daa085Dfe85D3C72eE1c1AdF425be49';
const accountId = '0x000000000000000000000000000000000000beef';
const description = await runtime.description.getDescription(address, accountId);
console.dir(description);
// Output:
// { public:
//   { name: 'DBCP sample greeter',
//     description: 'smart contract with a greeting message and a data property',
//     author: 'dbcp test',
//     tags: [ 'example', 'greeter' ],
//     version: '0.1.0',
//     abis: { own: [Array] } } }
```

4.1.2 loadContract

```
contractLoader.loadContract (name, address);
```

creates a contract instance that handles a smart contract at a given address

Parameters

1. name - string: Contract name
2. address - string: Contract address

Returns

any: contract instance.

Example

```
const greeter = runtime.contractLoader.loadContract('Greeter',  
↪ '0x9c0Aaa728Daa085Dfe85D3C72eE1c1AdF425be49');
```

4.2 Rights and Roles

Class Name	RightsAndRoles
Extends	Logger
Source	rights-and-roles.ts
Examples	rights-and-roles.spec.ts

The [RightsAndRoles](#) module follows the approach described in the [evan.network](#) wik at:

- [Function Permissions](#)
- [Operation Permissions](#)

It allows to manage permissions for contracts, that use the authority [DSRolesPerContract.sol](#) for as its permission approach.

Contracts, that use [DSRolesPerContract](#) and therefore allow to configure its permissions with the [RightsAndRoles](#) module are:

- [BaseContract](#)
- [DataContract](#)
- [ServiceContract](#)
- [Shared](#)
- [Described](#)
- [BusinessCenter](#)

Also have a look at the [Smart Contract Permissioning](#) section in the [evan.network](#) wiki.

4.2.1 constructor

```
new RightsAndRole(options);
```

Creates new [RightsAndRole](#) instance.

Parameters

1. options - **RightsAndRolesOptions**: options for **RightsAndRole** constructor.

- `contractLoader` - `ContractLoader`: `ContractLoader` instance
- `executor` - `Executor`: `Executor` instance
- `nameResolver` - `NameResolver`: `NameResolver` instance
- `web3` - `Web3`: `Web3` instance
- `log` - `Function` (optional): function to use for logging: `(message, level) => {...}`
- `logLevel` - `LogLevel` (optional): messages with this level will be logged with `log`
- `logLog` - `LogLogInterface` (optional): container for collecting log messages
- `logLogLevel` - `LogLevel` (optional): messages with this level will be pushed to `logLog`

Returns

`RightsAndRoles` instance

Example

```
const rightsAndRoles = new RightsAndRoles({
  contractLoader,
  executor,
  nameResolver,
  web3,
});
```

4.2.2 `addAccountToRole`

```
rightsAndRoles.addAccountToRole(contract, accountId, targetAccountId, role);
```

Adds the target account to a specific role.

The main principle is that accounts can be assigned to roles and those roles can be granted capabilities. *Function Permissions* are basically the capability to call specific functions if the calling account belongs to a certain role. To add an account to the role 'member'.

Parameters

1. `contract` - `string|any`: `contractId` or `contract` instance
2. `accountId` - `string`: `executing accountId`
3. `targetAccountId` - `string`: `target accountId`
4. `role` - `number`: `roleId`

Returns

Promise returns void: resolved when done

Example

```

const contractOwner = '0x0000000000000000000000000000000000000000000000000000000000000001';
const newMember = '0x0000000000000000000000000000000000000000000000000000000000000002';
const memberRole = 1;
await rightsAndRoles.addAccountToRole(
  contract,                // contract to be updated
  contractOwner,           // account, that can change permissions
  newMember,               // add this account to role
  memberRole,              // role id, uint8 value
);

```

4.2.3 removeAccountFromRole

```
rightsAndRoles.removeAccountFromRole(contract, accountId, targetAccountId, role);
```

Removes target account from a specific role.

Parameters

1. contract - string|any: contractId or contract instance
2. accountId - string: executing accountId
3. targetAccountId - string: target accountId
4. role - number: roleId

Returns

Promise returns void: resolved when done

Example

```

const contractOwner = '0x0000000000000000000000000000000000000000000000000000000000000001';
const newMember = '0x0000000000000000000000000000000000000000000000000000000000000002';
const memberRole = 1;
await rightsAndRoles.removeAccountFromRole(
  contract,                // contract to be updated
  contractOwner,           // account, that can change permissions
  newMember,               // remove this account from role
  memberRole,              // role id, uint8 value
);

```

4.2.4 getMembers

```
rightsAndRoles.getMembers(contract);
```

Returns all roles with all members.

The `DSRolesPerContract` authority tracks used roles and their members and allows to retrieve an overview with all roles and their members. To get this information, you can use the `getMembers` function.

Parameters

1. `contract` - `string|any`: `contractId` or `contract` instance

Returns

Promise returns `any`: Object with mapping `roleId` -> [`accountId`, `accountId`,...]

Example

```
const members = await rightsAndRoles.getMembers(contract);
console.log(members);
// Output:
// {
//   "0": [
//     "0x0000000000000000000000000000000000000000000000000000000000000001"
//   ],
//   "1": [
//     "0x0000000000000000000000000000000000000000000000000000000000000001",
//     "0x0000000000000000000000000000000000000000000000000000000000000002"
//   ]
// }
```

The contract from this example has an owner (`0x0001`) and a member (`0x0002`). As the owner account has the member role as well, it is listed among the members.

4.2.5 setFunctionPermission

```
rightsAndRoles.setFunctionPermission(contract, accountId, role, functionSignature, ↵
allow);
```

Allows or denies contract function for the `accountId`.

“Function permissions” are granted or denied by allowing a certain role to execute a specific function. The function is specified as the unhashed `function selector` and must follow its guidelines (no spaces, property typenames, etc.) for the function to be able to generate valid hashes for later validations. E.g. to grant the role “member” the permission to use the function `addListEntries`, that has two arguments (a `bytes32` array and a `bytes32` value), the function permission for `addListEntries(bytes32[], bytes32[])` has to be granted.

Parameters

1. `contract - string|any`: `contractId` or contract instance
2. `accountId - string`: `executing accountId`
3. `role - number`: `role id`
4. `functionSignature - string`: 4 Bytes function signature
5. `allow - boolean`: `allow` or `deny` function

Returns

Promise returns void: resolved when done

Example

```
const contractOwner = '0x0000000000000000000000000000000000000000000000000000000000000001';
const memberRole = 1;
await rightsAndRoles.setFunctionPermission(
  contract,
  contractOwner,           // contract to be updated
  memberRole,              // account, that can change permissions
                           // role id, uint8 value
  'addListEntries(bytes32[],bytes32[])', // (unhashed) function selector
  true,                    // grant this capability
);
```

4.2.6 setOperationPermission

```
rightsAndRoles.setOperationPermission(contract, accountId, role, propertyName, ↵
  ↪propertyType, modificationType, allow);
```

Allows or denies setting properties on a contract.

“Operation Permissions” are capabilities granted per contract logic. They have a `bytes32` key, that represents the capability, e.g. in a [DataContract](#) a capability to add values to a certain list can be granted.

The way, those capability hashes are build, depends on the contract logic and differs from contract to contract. For example a capability check for validation if a member is allowed to add an item to the list “example” in a [DataContract](#) has four arguments, in this case:

- which role is allowed to do? (e.g. a member)
- what type of element is modified? (→ a list)
- which element is modified? (name of the list → “example”)
- type of the modification (→ “set an item” (== “add an item”))

These four values are combined into one `bytes32` value, that is used when granting or checking permissions, the `setOperationPermission` function takes care of that.

Parameters

1. `contract` - `string|any`: `contractId` or contract instance
2. `accountId` - `string`: executing `accountId`
3. `role` - `number`: `roleId`
4. `propertyName` - `string`: target property name
5. `propertyType` - `PropertyType`: list or entry
6. `modificationType` - `ModificationType`: set or remove
7. `allow` - `boolean`: allow or deny

Returns

Promise returns void: resolved when done

Example

```
// make sure, you have required the enums from rights-and-roles.ts
import { ModificationType, PropertyType } from '@evan.network/api-blockchain-core';
const contractOwner = '0x0000000000000000000000000000000000000000000000000000000000000001';
const memberRole = 1;
await rightsAndRoles.setOperationPermission(
  contract, // contract to be updated
  contractOwner, // account, that can change permissions
  memberRole, // role id, uint8 value
  'example', // name of the object
  PropertyType.ListEntry, // what type of element is modified
  ModificationType.Set, // type of the modification
  true, // grant this capability
);
```

4.2.7 hasUserRole

```
rightsAndRoles.hasUserRole(contract, accountId, targetAccountId, role);
```

Returns true or false, depending on if the account has the specific role.

Parameters

1. `contract` - `string|any`: `contractId` or contract instance
2. `accountId` - `string`: executing `accountId`
3. `targetAccountId` - `string`: to be checked `accountId`
4. `role` - `number`: `roleId`

Returns

Promise returns void: resolved when done

Example

```

const accountToCheck = '0x000000000000000000000000000000000002';
const memberRole = 1;
const hasRole = await rightsAndRoles.hashUserRole(contract, null, accountToCheck,
  ↳memberRole);
console.log(hasRole);
// Output:
// true

```

4.2.8 transferOwnership

```
rightsAndRoles.transferOwnership();
```

Function description

Parameters

1. contract - string|any: contractId or contract instance
2. accountId - string: executing accountId
3. targetAccountId - string: target accountId

Returns

Promise returns void: resolved when done

Example

```

const contractOwner = '0x0000000000000000000000000000000000000001';
const newOwner = '0x0000000000000000000000000000000000000002';
await rightsAndRoles.transferOwnership(
  contract,                // contract to be updated
  contractOwner,           // current owner
  newOwner,                // this account becomes new owner
);

```

4.3 Sharing

Class Name	Sharing
Extends	Logger
Source	sharing.ts
Examples	sharing.spec.ts

For getting a better understanding about how Sharings and Multikeys work, have a look at [Security](#) in the [evan.network](#) wiki.

Following is a sample for a sharing info with these properties:

- three users
 - 0x01 - owner of a contract
 - 0x02 - member of a contract
 - 0x03 - another member with differing permissions
- two timestamps
 - block 82745 - first sharing
 - block 90000 - splitting data, update sharings
- three sections
 - * generic “catch all” used in first sharing
 - secret area - available for all members
 - super secret area - available for 0x03

```
{
  "0x01": {
    "82745": {
      "*": {
        "private": "secret for 0x01, starting from block 82745 for all data",
        "cryptoInfo": {
          "originator": "0x01,0x01",
          "keyLength": 256,
          "algorithm": "aes-256-cbc"
        }
      }
    },
    "90000": {
      "secret area": {
        "private": "secret for 0x01, starting from block 90000 for 'secret area'",
        "cryptoInfo": {
          "originator": "0x01,0x01",
          "keyLength": 256,
          "algorithm": "aes-256-cbc"
        }
      },
      "super secret area": {
        "private": "secret for 0x01, starting from block 90000 for 'super secret area'
→",
        "cryptoInfo": {
          "originator": "0x01,0x01",
          "keyLength": 256,
          "algorithm": "aes-256-cbc"
        }
      }
    }
  },
  "0x02": {
    "82745": {
      "*": {
```

(continues on next page)

(continued from previous page)

```

        "private": "secret for 0x02, starting from block 82745 for all data",
        "cryptoInfo": {
            "originator": "0x01,0x02",
            "keyLength": 256,
            "algorithm": "aes-256-cbc"
        }
    },
    "90000": {
        "secret area": {
            "private": "secret for 0x02, starting from block 90000 for 'secret area'",
            "cryptoInfo": {
                "originator": "0x01,0x02",
                "keyLength": 256,
                "algorithm": "aes-256-cbc"
            }
        },
        "super secret area": {
            "private": "secret for 0x02, starting from block 90000 for 'super secret area'
↪",
            "cryptoInfo": {
                "originator": "0x01,0x02",
                "keyLength": 256,
                "algorithm": "aes-256-cbc"
            }
        }
    },
    "0x03": {
        "90000": {
            "secret area": {
                "private": "secret for 0x03, starting from block 90000 for 'secret area'",
                "cryptoInfo": {
                    "originator": "0x01,0x03",
                    "keyLength": 256,
                    "algorithm": "aes-256-cbc"
                }
            }
        }
    }
}

```

More information about sharings can be found at the [evan.network](#) wiki.

There are two functions to share keys with another user:

- *addSharing* is used for easily sharing keys to another user. There is no need to explicitly share hash keys to this other user as this is automatically covered here. This approach make up to two transaction (1 for hash key and 1 for the content key), which may sum up to a whole bunch of transactions when sharing multiple keys to multiple users.
- *extendSharing* is used to edit a sharings configuration that has been pulled or “checked out” with *getSharings-FromContract*. Hash keys have to be shared manually, if required. *extendSharing* make no transaction, so the contract isn’t updated - this has to be done with *saveSharingsToContract*. See function documentation [below](#) for an example with hash key and storing updates.

Be careful when performing multiple updates to sharings synchronously. As sharings are retrieved as a single file from a smart contract, updated and then saved back to it, doing two or more updates in parallel may overwrite each other

and lead to unexpected and most probably undesired results.

Perform sharing updates for the same contracts **one after another**, this goes for *addSharing* and for *extendSharing*. When wishing to speed things up, *extendSharing* can be used, but its updates need to be performed synchronously as well. Keep in mind, that single updates will be made off-chain and therefore be performed much faster than multiple updates with *addSharing*.

4.3.1 constructor

```
new Sharing(options);
```

Creates a new Sharing instance.

Parameters

1. **options - SharingOptions:** options for Sharing constructor.

- `contractLoader` - `ContractLoader`: `ContractLoader` instance
- `cryptoProvider` - `CryptoProvider`: `CryptoProvider` instance
- `description` - `Description`: `Description` instance
- `dfs` - `DfsInterface`: `DfsInterface` instance
- `executor` - `Executor`: `Executor` instance
- `keyProvider` - `KeyProvider`: `KeyProvider` instance
- `nameResolver` - `NameResolver`: `NameResolver` instance
- `defaultCryptoAlgo` - string (optional): crypto algorithm name from `CryptoProvider`, defaults to `aes`
- `log` - Function (optional): function to use for logging: `(message, level) => {...}`
- `logLevel` - `LogLevel` (optional): messages with this level will be logged with `log`
- `logLog` - `LogLogInterface` (optional): container for collecting log messages
- `logLogLevel` - `LogLevel` (optional): messages with this level will be pushed to `logLog`

Returns

Sharing instance

Example

```
const sharing = new Sharing({
  contractLoader,
  cryptoProvider,
  description,
  executor,
  dfs,
  keyProvider,
```

(continues on next page)

(continued from previous page)

```

    nameResolver,
    defaultCryptoAlgo: 'aes',
  });

```

4.3.2 addSharing

```

sharing.addSharing(address, originator, partner, section, block, sharingKey[, context,
↳ isHashKey, sharingId]);

```

Add a sharing to a contract or an ENS address.

This function is primarily used for sharing single keys with one other users, when sharing multiple keys and/or sharing with multiple users, have a look at *extendSharing*.

Parameters

1. address - string: contract address or ENS address
2. originator - string: Ethereum account id of the sharing user
3. partner - string: Ethereum account id for which key shall be added
4. section - string: data section the key is intended for or '*'
5. block - number|string: starting with this block, the key is valid
6. sharingKey - string: key to share
7. context - string (optional): context to share key in
8. isHashKey - bool (optional): indicates if given key already is a hash key, defaults to false
9. sharingId - string (optional): id of a sharing (when multi-sharings is used)

Returns

Promise returns void: resolved when done

Example

```

// two sample users, user1 wants to share a key with user2
const user1 = '0x0000000000000000000000000000000000000000000000000000000000000001';
const user2 = '0x0000000000000000000000000000000000000000000000000000000000000002';
// create a sample contract
// usually you would have an existing contract, for which you want to manage the
↳ sharings
const contract = await executor.createContract('Shared', [], { from: user1, gas:
↳ 500000, });
// user1 shares the given key with user2
// this key is shared for all contexts ('*') and valid starting with block 0
await sharing.addSharing(contract.options.address, user1, user2, '*', 0, 'i am the
↳ secured that will be shared');

```

4.3.3 extendSharing

```
sharing.extendSharing(address, originator, partner, section, block, sharingKey[,   
↪context, isHashKey]);
```

Extend an existing sharing info with given key.

This is done on a sharings object and does not perform a transaction on its own. This function extends a sharing object retrieved from *getSharingsFromContract* and does not update sharings at the smart contract. For updating smart contracts sharing use *saveSharingsToContract*.

This function is primarily used to prepare updates for multiple keys and/or multiple users and submitting the result in one single transaction. For simpler sharing scenarios have a look at *addSharing*.

Parameters

1. sharings - any: object with sharings info
2. originator - string: Ethereum account id of the sharing user
3. partner - string: Ethereum account id for which key shall be added
4. section - string: data section the key is intended for or '*'
5. block - number|string: starting with this block, the key is valid
6. sharingKey - string: key to share
7. context - string (optional): context to share key in

Returns

Promise returns any: updated sharings info

Example

```
// two sample users, user1 wants to share a key with user2
const user1 = '0x0000000000000000000000000000000000000000000000000000000000000001';
const user2 = '0x0000000000000000000000000000000000000000000000000000000000000002';

// get current sharings
const sharings = await sharing.getSharingsFromContract(contract);

// if receiver of sharing hasn't been added to the contract yet, share hash key as   
↪well
const hashKeyToShare = await sharing.getHashKey(contract.options.address, user1);
await sharing.extendSharings(sharings, user1, user2, '*', 'hashKey', hashKeyToShare,   
↪null);

// get current block number, keys will be available starting from this block
const blockNr = await web3.eth.getBlockNumber();

// get current key for field or in this case fallback '*'
const contentKey = sharing.getKey(contract.options.address, user1, '*', blockNr);

// share this key
```

(continues on next page)

(continued from previous page)

```
await sharing.extendSharings(sharings, user1, user2, '*', blockNr, contentKey);

// finally store to contract
await sharing.saveSharingsToContract(contract.options.address, sharings, user1);
```

4.3.4 trimSharings

```
sharing.trimSharings(sharings, partner[, partner, section, block]);
```

Removes properties from given sharing. If a block is given, the specific blocks key is removed, if no block is given, all keys for this section are removed. The same goes for section and partner. Note that only the last properties can be omitted and not properties in between can be set to null. So for example it is not possible to remove the same field for all accounts by just setting partner to null.

Parameters

1. sharings - any: sharings to trim
2. partner - string: Ethereum account id to remove keys for
3. section - string: data section the key is intended for or '*'
4. block - number|string: block to remove keys for

Returns

Promise returns void: resolved when done

Example

```
// this sample will undo the changes from the last example (extendSharings)
// two sample users, user1 wants to share a key with user2
const user1 = '0x0000000000000000000000000000000000000000000000000000000000000001';
const user2 = '0x0000000000000000000000000000000000000000000000000000000000000002';

// get current sharings
const sharings = await sharing.getSharingsFromContract(contract);

// remove key from last time
await sharing.trim(sharings, user2, '*', blockNr);

// finally store to contract
await sharing.saveSharingsToContract(contract.options.address, sharings, user1);
```

4.3.5 bumpSharings

```
sharing.bumpSharings(address, originator, partners, section, block, sharingKey);
```

Bump keys for given accounts by adding given key to their sharings. This is basically a shorthand version for adding the new key for every account in the `partners` array in a single transaction.

`context`, `hashKeys` and `sharingId` are currently not supported.

Parameters

1. `address` - string: contract address or ENS address
2. `originator` - string: Ethereum account id of the sharing user
3. `partner` - string: Ethereum account id for which key shall be added
4. `section` - string: data section the key is intended for or `*`
5. `block` - number|string: starting with this block, the key is valid
6. `sharingKey` - string: key to share

Returns

Promise returns void: resolved when done

Example

```
// two sample users, user1 wants to bump keys for user2 and user3
const user1 = '0x0000000000000000000000000000000000000000000000000000000000000001';
const user2 = '0x0000000000000000000000000000000000000000000000000000000000000002';
const user3 = '0x0000000000000000000000000000000000000000000000000000000000000003';
// assume we have a contract with sharings for those accounts
const contractId = '0x000000000000000000000000000000000000000000000000000000000000000c027rac7';
await sharing.bumpSharings(contractId, user1, [ user2, user3 ], '*', 0, 'i am a bump_
↪key');
```

4.3.6 getKey

```
sharing.getKey(address, partner, section[, block, sharingId]);
```

Get a content key from the sharing of a contract.

Parameters

1. `address` - string: contract address or ENS address
2. `partner` - string: Ethereum account id for which key shall be retrieved
3. `section` - string: data section the key is intended for or `*`
4. `block` - number|string (optional): starting with this block, the key is valid, defaults to `Number.MAX_SAFE_INTEGER`
5. `sharingId` - string (optional): id of a sharing (when multi-sharings is used), defaults to `null`

Returns

Promise returns string: matching key

Example

```
// a sample user
const user2 = '0x0000000000000000000000000000000000000000000000000000000000000002';
// user2 wants to read a key after receiving a sharing
// the key requested should be valid for all contexts ('*') and valid up to and
  ↳ including block 100
const key = await sharing.getKey(contract.options.address, user2, '*', 100);
```

4.3.7 getKeyHistory

```
sharing.getKeyHistory(address, partner, section[, sharingId]);
```

Get history of keys for an account and a section.

Parameters

1. address - string: contract address or ENS address
2. partner - string: Ethereum account id for which key shall be retrieved
3. section - string: data section the key is intended for or '*'
4. sharingId - string (optional): id of a sharing (when multi-sharings is used), defaults to null

Returns

Promise returns any: object with key: blockNr, value: key

Example

```
// a sample user
const user2 = '0x0000000000000000000000000000000000000000000000000000000000000002';
// user2 wants to retrieve all keys for '*'
const keyHistory = await sharing.getKeyHistory(contract.options.address, user2, '*');
```

4.3.8 ensureHashKey

```
sharing.ensureHashKey(address, originator, partner, hashKey[, context, sharingId]);
```

Give hash key “hashKey” to account “partner”, if this account does not have a hash key already.

Parameters

1. `address - string`: contract address or ENS address
2. `originator - string`: Ethereum account id of the sharing user
3. `partner - string`: Ethereum account id for which key shall be added
4. `hashKey - string`: key for DFS hashes
5. `context - string (optional)`: context to share key in
6. `sharingId - string (optional)`: id of a sharing (when multi-sharings is used)

Returns

Promise `returns void`: resolved when done

Example

```
const hashCrytor = cryptoProvider.getCryptorByCryptoAlgo('aesEcb');
const hashKey = await hashCrytor.generateKey();
await sharing.ensureHashKey(contract.options.address, accounts[0], accounts[1], ↵
↵hashKey);
```

4.3.9 getHashKey

```
sharing.getHashKey(address, partner[, sharingid]);
```

Function description

Parameters

1. `address - string`: contract address or ENS address
2. `partner - string`: Ethereum account id for which key shall be retrieved
3. `sharingId - string (optional)`: id of a sharing (when multi-sharings is used)

Returns

Promise `returns string`: matching key

Example

```
const hashCrytor = cryptoProvider.getCryptorByCryptoAlgo('aesEcb');
const hashKey = await hashCrytor.generateKey();
await sharing.ensureHashKey(contract.options.address, accounts[0], accounts[1], ↵
↵hashKey);
const retrieved = sharing.getHashKey(contract.options.address, accounts[1]);
```

(continues on next page)

(continued from previous page)

```
console.log(hashKey === retrieved);  
// Output:  
// true
```

4.3.10 getSharings

```
sharing.getSharings(address[, _partner, _section, _block, sharingId]);
```

Get sharing from a contract, if `_partner`, `_section`, `_block` matches.

Parameters

1. `address` - string: contract address or ENS address
2. `_partner` - string (optional): Ethereum account id for which key shall be retrieved
3. `_section` - string (optional): data section the key is intended for or `'*'`
4. `_block` - number (optional): starting with this block, the key is valid
5. `sharingId` - string (optional): id of a sharing (when multi-sharings is used)

Returns

Promise `returns` void: resolved when done

Example

```
const randomSecret = `super secret; ${Math.random()}`;  
await sharing.addSharing(testAddress, accounts[1], accounts[0], '*', 0, randomSecret);  
const sharings = await sharing.getSharings(testAddress);
```

4.3.11 removeSharing

```
sharing.removeSharing(address, originator, partner, section[, sharingId]);
```

Remove a sharing key from a contract with sharing info.

Parameters

1. `address` - string: contract address or ENS address
2. `originator` - string: Ethereum account id of the sharing user
3. `partner` - string: Ethereum account id for which key shall be removed
4. `section` - string: data section of the key
5. `sharingId` - string (optional): id of a sharing (when multi-sharings is used), defaults to `null`

Returns

Promise returns void: resolved when done

Example

```
await sharing.addSharing(contract.options.address, accounts[0], accounts[1], '*', 0, ↵
↵randomSecret);

let sharings = await sharing.getSharings(contract.options.address);
console.log(Object.keys(sharings[nameResolver.soliditySha3(accounts[1])]).length);
// Output:
// 1

await sharing.removeSharing(contract.options.address, accounts[0], accounts[1], '*');

let sharings = await sharing.getSharings(contract.options.address);
console.log(Object.keys(sharings[nameResolver.soliditySha3(accounts[1])]).length);
// Output:
// 0
```

4.3.12 getSharingsFromContract

```
sharing.getSharingsFromContract(contract[, sharingId]);
```

Get encrypted sharings from smart contract.

This can be used in combination with *getSharingsFromContract* to bulk editing sharing info.

Parameters

1. contract - any: contract with sharing info
2. sharingId - string (optional): id of a sharing in mutlisharings, defaults to null

Returns

Promise returns void: resolved when done

Example

```
// get sharings (encrypted)
const sharings = await sharing.getSharingsFromContract(serviceContract, callIdHash);

// make changes to sharing
await sharing.extendSharings(sharings, accountId, target, section, 0, ↵
↵contentKeyToShare, null);
await sharing.extendSharings(sharings, accountId, target, '*', 'hashKey', ↵
↵hashKeyToShare, null);
```

(continues on next page)

(continued from previous page)

```
// commit changes
await sharing.saveSharingsToContract(serviceContract.options.address, sharings,
  ↪accountId, callIdHash);
```

4.3.13 saveSharingsToContract

```
sharing.saveSharingsToContract(contract, sharings, originator[, sharingId]);
```

Save sharings object with encrypted keys to contract.

This can be used to pull sharings, edit them offline and commit changes in a bulk. See example section for usage.

Parameters

1. contract - string|any: contract address or instance
2. sharings - any: sharings object with encrypted keys
3. originator - string: Ethereum account id of the sharing user
4. sharingId - string (optional): id of a sharing (when multi-sharings is used)

Returns

Promise returns void: resolved when done

Example

```
// get sharings (encrypted)
const sharings = await sharing.getSharingsFromContract(serviceContract, callIdHash);

// make changes to sharing
await sharing.extendSharings(sharings, accountId, target, section, 0,
  ↪contentKeyToShare, null);
await sharing.extendSharings(sharings, accountId, target, '*', 'hashKey',
  ↪hashKeyToShare, null);

// commit changes
await sharing.saveSharingsToContract(serviceContract.options.address, sharings,
  ↪accountId, callIdHash);
```

4.3.14 addHashToCache

```
sharing.addHashToCache(address, sharingHash[, sharingId]);
```

Add a hash to to cache, can be used to speed up sharing key retrieval, when sharings hash is already known.

Parameters

1. `address - string`: contract address
2. `sharingHash - string`: bytes32 hash of a sharing
3. `sharingId - string (optional)`: id of a multisharing, defaults to `null`

Example

```
sharing.addHashToCache(contract.options.address, sharingHash, sharingId);
```

4.3.15 clearCache

```
sharing.clearCache();
```

Clear caches and fetch new hashes and sharing on next request.

When sharings are fetched and not all results could be read, the result would stay the same in following requests due to the internal caching mechanism, even if a proper key has been shared with the user later on. To prevent such old values from showing up, the cache can be cleared.

Example

```
sharing.clearCache();
```

4.4 Base Contract

Class Name	BaseContract
Extends	Logger
Source	base-contract.ts
Examples	base-contract.spec.ts

The [BaseContract](#) is the base contract class used for

- *DataContracts*
- *ServiceContractss*

Contracts, that inherit from `BaseContracts`, are able to:

- manage a list of contract participants (called “members”)
- manage the own state (a flag, that indicate its own life cycle status)
- manage members state (a flag, that indicate the members state in the contract)

What members can do, what non-members cannot do depends of the implementatin of the inheriting contracts.

4.4.1 constructor

```
new BaseContract(options);
```

Creates a new `BaseContract` instance.

Parameters

1. **options - BaseContractOptions:** options for `BaseContract` constructor.

- `executor` - `Executor`: `Executor` instance
- `loader` - `ContractLoader`: `ContractLoader` instance
- `nameResolver` - `NameResolver`: `NameResolver` instance
- `log` - `Function` (optional): function to use for logging: `(message, level) => {...}`
- `logLevel` - `LogLevel` (optional): messages with this level will be logged with `log`
- `logLog` - `LogLogInterface` (optional): container for collecting log messages
- `logLogLevel` - `LogLevel` (optional): messages with this level will be pushed to `logLog`

Returns

`BaseContract` instance

Example

```
const baseContract = new BaseContract({
  executor,
  loader,
  nameResolver,
});
```

4.4.2 createUninitialized

```
baseContract.createUninitialized(factoryName, accountId[, businessCenterDomain, ↵
↵descriptionDfsHash]);
```

Create new contract but do not initialize it yet.

The API supports creating contracts, that inherit from `BaseContract`. This is done by calling the respective factory. The factory calls are done via a function with this interface:

```
/// @notice create new contract instance
/// @param businessCenter address of the BusinessCenter to use or 0x0
/// @param provider future owner of the contract
/// @param _contractDescription DBCP definition of the contract
/// @param ensAddress address of the ENS contract
function createContract(
  address businessCenter,
  address provider,
```

(continues on next page)

(continued from previous page)

```
bytes32 contractDescription,  
address ensAddress) public returns (address);
```

The API supports creating contracts with this function. Contracts created this way may not be ready to use and require an additional function at the contract to be called before usage. This function is usually called `init` and its arguments and implementation depends of the specific contract.

The `createUninitialized` function performs a lookup for the respective factory contract and calls the `createContract` function at it.

Parameters

1. `factoryName` - string: contract factory name, used for ENS lookup; if the factory name contains periods, it is threaded as an absolute ENS domain and used as such, if not it will be used as `${factoryName}.factory.${businessCenterDomain}`
2. `accountId` - string: Ethereum account id
3. `businessCenterDomain` - string (optional): business center in which the contract will be created; use `null` when working without business center
4. `descriptionDfsHash` - string (optional): bytes32 hash for description in dfs

Returns

Promise returns string: Ethereum id of new contract

Example

```
const contractOwner = '0x...';  
const businessCenterDomain = 'testbc.evan';  
const contractId = await baseContract.createUninitialized(  
  'testdatacontract',           // factory name  
  contractOwner,                // account, that will be owner of the new_  
  ↪contract                     // business center, where the new contract_  
  businessCenterDomain,         ↪will be created  
  ↪will be created  
);
```

4.4.3 inviteToContract

```
baseContract.inviteToContract(businessCenterDomain, contract, inviterId, inviteeId);
```

Invite user to contract. To allow accounts to work with contract resources, they have to be added as members to the contract. This function does exactly that.

Parameters

1. `businessCenterDomain - string`: ENS domain name of the business center the contract was created in; use null when working without business center
2. `contract - string`: Ethereum id of the contract
3. `inviterId - string`: account id of inviting user
4. `inviteeId - string`: account id of invited user

Returns

Promise returns void: resolved when done

Example

```
const contractOwner = '0x0000000000000000000000000000000000000000000000000000000000000001';
const invitee = '0x0000000000000000000000000000000000000000000000000000000000000002';
const businessCenterDomain = 'testbc.evan';
const contract = loader.loadContract('BaseContractInterface', contractId);
await baseContract.inviteToContract(
  businessCenterDomain,
  contractId,
  contractOwner,
  invitee,
);
```

To check if an account is a member of a contract, the contract function `isMember` can be used:

```
const isMember = await executor.executeContractCall(contract, 'isConsumer', invitee);
console.log(isMember);
// Output:
// true
```

4.4.4 removeFromContract

```
baseContract.removeFromContract(businessCenterDomain, contract, accountId, ↵
↵idToBeRemoved);
```

Remove user from contract. To deny previously invited accounts to work with contract resources, they have to be removed as members from the contract. This function does exactly that.

Parameters

1. `businessCenterDomain - string`: ENS domain name of the business center the contract was created in; use null when working without business center
2. `contract - string`: Ethereum id of the contract
3. `accountId - string`: account id of executing user
4. `idToBeRemoved - string`: account id which should be removed

Returns

Promise returns void: resolved when done

Example

```
const contractOwner = '0x0000000000000000000000000000000000000000000000000000000000000001';
const idToBeRemoved = '0x0000000000000000000000000000000000000000000000000000000000000002';
const businessCenterDomain = 'testbc.evan';
const contract = loader.loadContract('BaseContractInterface', contractId);
await baseContract.removeFromContract(
  businessCenterDomain,
  contractId,
  contractOwner,
  idToBeRemoved,
);
```

To check if an account is a member of a contract, the contract function `isMember` can be used:

```
const isMember = await executor.executeContractCall(contract, 'isConsumer', ↵
↵idToBeRemoved);
console.log(isMember);
// Output:
// false
```

4.4.5 changeConsumerState

```
baseContract.changeContractState(contract, accountId, consumerId, state);
```

set state of a consumer. A members state reflects this members status in the contract. These status values can for example be be Active, Draft or Terminated.

Parameters

1. `contract` - `string|any`: contract instance or contract id
2. `accountId` - `string`: Ethereum account id
3. `consumerId` - `string`: Ethereum account id
4. `state` - `ConsumerState`: new state

Returns

Promise returns void: resolved when done

Example

```
await baseContract.changeConsumerState(contractId, accountId, consumerId, ↵
↵ConsumerState.Active);
```


`ConsumerState` is an enum in the `BaseContract` class, that holds the same state values as the `BaseContract.sol`. Alternatively integer values matching the enum in `BaseContractInterface.sol` can be used.

4.4.6 changeContractState

```
baseContract.changeContractState(contract, accountId, state);
```

Set state of the contract. The contracts state reflects the current state and how other members may be able to interact with it. So for example, a contract for tasks cannot have its tasks resolved, when the contract is still in Draft state. State transitions are limited to configured roles and allow going from one state to another only if configured for this role.

Parameters

1. `contract` - `string|any`: contract instance or contract id
2. `accountId` - `string`: Ethereum account id
3. `state` - `ContractState`: new state

Returns

Promise returns void: resolved when done

Example

```
await baseContract.changeContractState(contractId, contractOwner, ContractState.  
  ↪Active);
```

`ContractState` is an enum in the `BaseContract` class, that holds the same state values as the `BaseContract.sol`. Alternatively integer values matching the enum in `BaseContractInterface.sol` can be used.

4.4.7 Additional Components

Enums

ContractState

Describes contracts overall state.

In most cases, this property can only be set by the contract owner.

```
export enum ContractState {  
  Initial,  
  Error,  
  Draft,  
  PendingApproval,
```

(continues on next page)

(continued from previous page)

```
    Approved,  
    Active,  
    VerifyTerminated,  
    Terminated,  
};
```

ConsumerState

Describes the state of a consumer or owner in a contract.

In most cases, this can be set the the member, thats status is updated or by a more privileged role, like a contract owner.

```
export enum ConsumerState {  
    Initial,  
    Error,  
    Draft,  
    Rejected,  
    Active,  
    Terminated  
};
```

4.5 Data Contract

Class Name	DataContract
Extends	BaseContract
Source	data-contract.ts
Examples	data-contract.spec.ts

The [DataContract](#) is a secured data storage contract for single properties and lists. If created on its own, DataContracts cannot do very much. They rely on their authority to check which entries or lists can be used.

The following functions support the `encryptionContext` argument:

- *[addListEntries](#)*
- *[setEntry](#)*
- *[setMappingValue](#)*

If this argument is set, the data key in the data contracts sharing is encrypted by using a context key instead of the communication key between owner and contract member. This allows to omit key exchanges between contract owner and members and therefore enables the owner to write content to the smart contract, that can be used by a group of accounts, which only needs to hold the context key. So the `encryptionContext` can be used to address a group of accounts instead of single accounts.

For more information about DataContracts purpose and their authorities see [Data Contract](#) in the [evan.network](#) wiki.

4.5.1 constructor

```
new DataContract(options);
```

Creates a new `DataContract` instance.

Parameters

1. **options - DataContractOptions:** options for `DataContract` constructor.

- `cryptoProvider` - `CryptoProvider`: `CryptoProvider` instance
- `defaultCryptoAlgo` - string (optional): crypto algorithm name from `CryptoProvider`, defaults to `aes`
- `dfs` - `DfsInterface`: `DfsInterface` instance
- `executor` - `Executor`: `Executor` instance
- `loader` - `ContractLoader`: `ContractLoader` instance
- `nameResolver` - `NameResolver`: `NameResolver` instance
- `sharing` - `Sharing`: `Sharing` instance
- `log` - Function (optional): function to use for logging: `(message, level) => {...}`
- `logLevel` - `LogLevel` (optional): messages with this level will be logged with `log`
- `logLog` - `LogLogInterface` (optional): container for collecting log messages
- `logLogLevel` - `LogLevel` (optional): messages with this level will be pushed to `logLog`

Returns

`DataContract` instance

Example

```
const dataContract = new DataContract({
  cryptoProvider,
  description,
  dfs,
  executor,
  loader,
  nameResolver,
  sharing,
  web3,
});
```

4.5.2 create

```
dataContract.create(factoryName, accountId[, businessCenterDomain, ↵
↵contractDescription, allowConsumerInvite, sharingsHash]);
```

Create and initialize new contract.

Parameters

1. `factoryName` - string: contract factory name, used for ENS lookup; if the factory name contains periods, it is threaded as an absolute ENS domain and used as such, if not it will be used as `${factoryName}.factory.${businessCenterDomain}`
2. `accountId` - string: owner of the new contract and transaction executor
3. `businessCenterDomain` - string (optional): ENS domain name of the business center
4. `contractDescription` - string|any (optional): bytes32 hash of DBCP description or a schema object
5. `allowConsumerInvite` - bool (optional): true if consumers are allowed to invite other consumer
6. `sharingsHash` - string (optional): existing sharing to add, defaults to null

Returns

Promise returns any: contract instance

Example

Let's say, we want to create a `DataContract` for a business center at the domain "samplebc.evan" and this business center has a `DataContractFactory` named "testdatacontract". We want to have two users working in our `DataContract`, so we get these sample values:

```
const factoryName = 'testdatacontract';
const businessCenterDomain = 'samplebc.evan';
const accounts = [
  '0x0000000000000000000000000000000000000000000000000000000000000001',
  '0x0000000000000000000000000000000000000000000000000000000000000002',
];
```

Now create a contract with:

```
const contract = await dataContract.create(factoryName, accounts[0], ↵
↵businessCenterDomain);
```

Okay, that does not provide a description for the contract. Let's add a description to the process. The definition is a DBCP contract definition and is stored in an `Envelope` (see [Encryption](#)):

```
const definition: Envelope = {
  "public": {
    "name": "Data Contract Sample",
    "description": "reiterance oxynitrate sat alternize acurative",
    "version": "0.1.0",
    "author": "evan GmbH",
    "dataSchema": {
      "list_settable_by_member": {
        "$id": "list_settable_by_member_schema",
        "type": "object",
        "additionalProperties": false,
        "properties": {
          "foo": { "type": "string" },
          "bar": { "type": "integer" }
        }
      }
    }
  },
};
```

(continues on next page)

(continued from previous page)

```

    "entry_settable_by_member": {
      "$id": "entry_settable_by_member_schema",
      "type": "integer",
    }
  }
};
definition.cryptoInfo = cryptoProvider.getCryptorByCryptoAlgo('aes').
  ↪getCryptoInfo(accounts[0]);
const contract = await dataContract.create('testdatacontract', accounts[0], ↪
  ↪businessCenterDomain, definition);

```

Now we have a DataContract with a description. This contract is now able to be understood by other components, that understand the dbcp. And on top of that, we provided data schemas for the two properties `list_settable_by_member` and `entry_settable_by_member` (written for `ajv`). This means, that when someone adds or sets entries to or in those properties, the incoming data is validated before actually encrypting and storing it.

To allow other users to work on the contract, they have to be invited with:

```

await dataContract.inviteToContract(businessCenterDomain, contract.options.address, ↪
  ↪accounts[0], accounts[1]);

```

Now the user `accounts[1]` can use functions from the contract, but to actually store data, the user needs access to the data key for the DataContract. This can be done via updating the contracts sharing:

```

const blockNr = await web3.eth.getBlockNumber();
const contentKey = await sharing.getKey(contract.options.address, accounts[0], '*', ↪
  ↪blockNr);
await sharing.addSharing(contract.options.address, accounts[0], accounts[1], '*', ↪
  ↪blockNr, contentKey);

```

Now the contract has been created, has a sharing and another user has been granted access to it. Variable names from this section will be used in the rest of the document as example values.

4.5.3 createSharing

```
dataContract.createSharing(accountId);
```

Create initial sharing for contract.

Parameters

1. `accountId` - string: owner of the new contract

Returns

Promise returns any: sharing info with { `contentKey`, `hashKey`, `sharings`, `sharingsHash`, }

Example

```
const sharing = await dataContract.createSharing(profileReceiver);
```

4.5.4 = Entries =

4.5.5 setEntry

```
dataContract.setEntry(contract, entryName, value, accountId[, dfsStorage,   
↪encryptedHashes, encryption]);
```

Set entry for a key.

Parameters

1. contract - any|string: contract or contractId
2. entryName - string: entry name
3. value - any: value to set
4. accountId - string: Ethereum account id
5. dfsStorage - Function (optional): store values in dfs, defaults to true
6. encryptedHashes - boolean (optional): encrypt hashes from values, defaults to true
7. encryption - string (optional): encryption algorithm to use, defaults to defaultCryptoAlgo (set in constructor)
8. encryptionContext - string (optional): plain text name of an encryption context, defaults to accountId

Returns

Promise returns void: resolved when done

Example

```
const sampleValue = 123;  
await dataContract.setEntry(contract, 'entry_settable_by_owner', sampleValue,   
↪accounts[0]);
```

Entries are automatically encrypted before setting it in the contract. If you want to use values as is, without encrypting them, you can add them in raw mode, which sets them as bytes32 values:

```
const sampleValue = '0x000000000000000000000000000000000000000000000000000000000000007b';  
await dataContract.setEntry(contract, 'entry_settable_by_owner', sampleValue,   
↪accounts[0], true);
```

4.5.6 getEntry

```
dataContract.getEntry(contract, entryName, accountId[, dfsStorage, encryptedHashes]);
```

Return entry from contract.

Parameters

1. contract - any|string: contract or contractId
2. entryName - string: entry name
3. accountId - string: Ethereum account id
4. dfsStorage - Function (optional): store values in dfs, defaults to true
5. encryptedHashes - boolean (optional): decrypt hashes from values, defaults to true

Returns

Promise returns any: entry

Example

Entries can be retrieved with:

```
const retrieved = await dataContract.getEntry(contract, 'entry_settable_by_owner',  
↪accounts[0]);
```

Raw values can be retrieved in the same way:

```
const retrieved = await dataContract.getEntry(contract, 'entry_settable_by_owner',  
↪accounts[0], true);
```

4.5.7 = List Entries =

4.5.8 addListEntries

```
dataContract.addListEntries(contract, listName, values, accountId[, dfsStorage,  
↪encryptedHashes, encryption]);
```

Add list entries to lists.

List entries support the raw mode as well. To use raw values, pass `true` in the same way as wehn using the entries functions.

List entries can be added in bulk, so the value argument is an array with values. This array can be arbitrarily large **up to a certain degree**. Values are inserted on the blockchain side and adding very large arrays this way may take more gas during the contract transaction, than may fit into a single transaction. If this is the case, values can be added in chunks (multiple transactions).

Parameters

1. `contract` - `any|string`: contract or `contractId`
2. `listName` - `string`: name of the list in the data contract
3. `values` - `any[]`: values to add
4. `accountId` - `string`: Ethereum account id
5. `dfsStorage` - `string` (optional): store values in dfs, defaults to `true`
6. `encryptedHashes` - `boolean` (optional): encrypt hashes from values, defaults to `true`
7. `encryption` - `string` (optional): encryption algorithm to use, defaults to `defaultCryptoAlgo` (set in constructor)
8. `encryptionContext` - `string` (optional): plain text name of an encryption context, defaults to `accountId`

Returns

Promise `returns void`: resolved when done

Example

```
const sampleValue = {
  foo: 'sample',
  bar: 123,
};
await dataContract.addListEntries(contract, 'list_settable_by_member', [sampleValue], ↵
↵accounts[0]);
```

When using lists similar to tagging list entries with metadata, entries can be added in multiple lists at once by passing an array of list names:

```
const sampleValue = {
  foo: 'sample',
  bar: 123,
};
await dataContract.addListEntries(contract, ['list_1', 'list_2'], [sampleValue], ↵
↵accounts[0]);
```

4.5.9 getListEntryCount

```
dataContract.getListEntryCount(contract, listName, index, accountId[, dfsStorage, ↵
↵encryptedHashes]);
```

Return number of entries in the list. Does not try to actually fetch and decrypt values, but just returns the count.

Parameters

1. `contract` - any|string: contract or contractId
2. `listName` - string: name of the list in the data contract

Returns

Promise returns number: list entry count

Example

```
await dataContract.getListEntryCount(contract, 'list_settable_by_member');
```

4.5.10 getListEntries

```
dataContract.getListEntries(contract, listName, accountId[, dfsStorage,   
→encryptedHashes, count, offset, reverse]);
```

Return list entries from contract. Note, that in the current implementation, this function retrieves the entries one at a time and may take a longer time when querying large lists, so be aware of that, when you retrieve lists with many entries.

Parameters

1. `contract` - any|string: contract or contractId
2. `listName` - string: name of the list in the data contract
3. `accountId` - string: Ethereum account id
4. `dfsStorage` - string (optional): store values in dfs, defaults to true
5. `encryptedHashes` - boolean (optional): decrypt hashes from values, defaults to true
6. `count` - number (optional): number of elements to retrieve, defaults to 10
7. `offset` - number (optional): skip this many items when retrieving, defaults to 0
8. `reverse` - boolean (optional): retrieve items in reverse order, defaults to false

Returns

Promise returns any[]: list entries

Example

```
await dataContract.getListEntries(contract, 'list_settable_by_member', accounts[0]);
```

4.5.11 getListEntry

```
dataContract.getListEntry(contract, listName, index, accountId[, dfsStorage,
↵encryptedHashes]);
```

Return a single list entry from contract.

Parameters

1. `contract` - any|string: contract or contractId
2. `listName` - string: name of the list in the data contract
3. `index` - number: list entry id to retrieve
4. `accountId` - string: Ethereum account id
5. `dfsStorage` - string (optional): store values in dfs, defaults to `true`
6. `encryptedHashes` - boolean (optional): decrypt hashes from values, defaults to `true`

Returns

Promise returns any: list entry

Example

```
const itemIndex = 0;
await dataContract.getListEntry(contract, 'list_settable_by_member', itemIndex,
  ↪accounts[0]));
```

4.5.12 removeListEntry

```
dataContract.removeListEntry(contract, listName, entryIndex, accountId);
```

Remove list entry from list.

This will reposition last list entry into emptied slot.

Attention

If the data contract was created by the *container* by the `DigitalTwin` API you need to grant the permissions for removing list entries manually. To do this you can use the `RightsAndRoles` API. The following example shows how to grant permissions to delete list entries from list *exampleList* to group 0.

[illegible]

(continues on next page)

(continued from previous page)

```

contractOwner,          // account, that can change permissions
0,                      // role id, uint8 value
'exampleList',          // name of the object
PropertyType.ListEntry, // what type of element is modified
ModificationType.Remove, // type of the modification
true,                   // grant this capability
);

```

Parameters

1. `contract` - any|string: contract or contractId
2. `listName` - string: name of the list in the data contract
3. `index` - number: index of the entry to remove from list
4. `accountId` - string: Ethereum account id

Returns

Promise returns void: resolved when done

Example

```

const listName = 'list_removable_by_owner'
const itemIndexInList = 1;
await dataContract.removeListEntry(contract, listNameF, itemIndexInList, accounts[0]);

```

4.5.13 moveListEntry

```

dataContract.moveListEntry(contract, listNameFrom, entryIndex, listNamesTo, ↵
↵accountId);

```

Move one list entry to one or more lists.

Note, that moving items requires the executing account to have `remove` permissions on the list `listNameFrom`. If this isn't the case, the transaction will not be executed and no updates will be made.

Parameters

1. `contract` - any|string: contract or contractId
2. `listNameFrom` - string: origin list
3. `index` - number: index of the entry to move in the origin list
4. `listNamesTo` - string: lists to move data into
5. `accountId` - string: Ethereum account id

Returns

Promise returns void: resolved when done

Example

```
const listNameFrom = 'list_removable_by_owner';
const listNameTo = 'list_settable_by_member';
const itemIndexInFromList = 1;
await dataContract.moveListEntry(contract, listNameFrom, itemIndexInFromList, ↵
↵[listNameTo], accounts[0]);
```

4.5.14 = Mappings =

4.5.15 setMappingValue

```
dataContract.setMappingValue(contract, mappingName, entryName, value, accountId[, ↵
↵dfsStorage, encryptedHashes, encryption]);
```

Set entry for a key in a mapping. Mappings are basically dictionaries in data contracts. They are a single permittable entry, that allows to set any keys to it. This can be used for properties, that should be extended during the contracts life as needed, but without the need to update its permission settings.

Parameters

1. contract - any|string: contract or contractId
2. mappingName - string: name of a data contracts mapping property
3. entryName - string: entry name (property in the mapping)
4. value - any: value to add
5. accountId - string: Ethereum account id
6. dfsStorage - string (optional): store values in dfs, defaults to true
7. encryptedHashes - boolean (optional): encrypt hashes from values, defaults to true
8. encryption - string (optional): encryption algorithm to use, defaults to defaultCryptoAlgo (set in constructor)
9. encryptionContext - string (optional): plain text name of an encryption context, defaults to accountId

Returns

Promise returns void: resolved when done

Example

```
await dataContract.setMappingValue(  
  contract,  
  'mapping_settable_by_owner',  
  'sampleKey',  
  'sampleValue',  
  accounts[0],  
  storeInDfs,  
);
```

4.5.16 getMappingValue

```
dataContract.getMappingValue(contract, listName, index, accountId[, dfsStorage,  
↪encryptedHashes]);
```

Return a value from a mapping. Looks up a single key from a mapping and returns its value.

Parameters

1. contract - any|string: contract or contractId
2. mappingName - string: name of a data contracts mapping property
3. entryName - string: entry name (property in the mapping)
4. accountId - string: Ethereum account id
5. dfsStorage - string (optional): store values in dfs, defaults to true
6. encryptedHashes - boolean (optional): encrypt hashes from values, defaults to true
7. encryption - string (optional): encryption algorithm to use, defaults to defaultCryptoAlgo (set in constructor)

Returns

Promise returns any: mappings value for given key

Example

```
const value = await dataContract.getMappingValue(  
  contract,  
  'mapping_settable_by_owner',  
  'sampleKey',  
  accounts[0],  
  storeInDfs,  
);
```

4.5.17 = Encryption =

4.5.18 encrypt

```
dataContract.encrypt(toEncrypt, contract, accountId, propertyName, block[,   
↪ encryption]);
```

Encrypt incoming envelope.

Parameters

1. toEncrypt - Envelope: envelope with data to encrypt
2. contract - any: contract instance or contract id
3. accountId - string: encrypting account
4. propertyName - string: property in contract, the data is encrypted for
5. block - block: block the data belongs to
6. encryption - string: encryption name, defaults to defaultCryptoAlgo (set in constructor)

Returns

Promise returns string: encrypted envelope or hash as string

Example

```
const data = {  
  public: {  
    foo: 'example',  
  },  
  private: {  
    bar: 123,  
  },  
  cryptoInfo: cryptor.getCryptoInfo(nameResolver.soliditySha3(accounts[0])),  
};  
const encrypted = await dataContract.encrypt(data, contract, accounts[0], 'list_  
↪ settable_by_member', 12345);
```

4.5.19 decrypt

```
dataContract.decrypt(toDecrypt, contract, accountId, propertyName, block[,   
↪ encryption]);
```

Decrypt input envelope return decrypted envelope.

Parameters

1. toDecrypt - string: data to decrypt
2. contract - any: contract instance or contract id
3. accountId - string: account id that decrypts the data
4. propertyName - string: property in contract that is decrypted

Returns

Promise returns Envelope: decrypted envelope

Example

```
const encrypted = await dataContract.decrypt(encrypted, contract, accounts[0], 'list_
↪settable_by_member');
```

4.5.20 encryptHash

```
dataContract.encryptHash(toEncrypt, contract, accountId);
```

Encrypt incoming hash. This function is used to encrypt DFS file hashes, uses AES ECB for encryption.

Parameters

1. toEncrypt - Envelope: hash to encrypt
2. contract - any: contract instance or contract id
3. accountId - string: encrypting account

Returns

Promise returns string: hash as string

Example

```
const hash = '0x1111111111111111111111111111111111111111111111111111111111111111';
const encrypted = await dataContract.encryptHash(hash, contract, accounts[0]);
```

4.5.21 decryptHash

```
dataContract.encrypt(toEncrypttoDecrypt, contract, accountId, propertyName, block[, encryption]);
```

Decrypt input hash, return decrypted hash. This function is used to decrypt encrypted DFS file hashes, uses AES ECB for decryption.

Parameters

1. `toDecrypt` - `Envelope`: hash to decrypt
2. `contract` - `any`: **contract** instance or contract id
3. `accountId` - `string`: **encrypting** account

Returns

Promise returns string: decrypted hash

Example

```
const encryptedHash =  
    ↪ '0x2222222222222222222222222222222222222222222222222222222222222222';  
const encrypted = await dataContract.decryptHash(encryptedHash, contract, ↵  
    ↪ accounts[0]);
```

4.5.22 clearSharings

```
dataContract.clearSharings();
```

Clears cached sharing information of this contract Can be used when sharings have been updated externally and new sharings should be fetched.

Example

```
dataContract.clearSharings();
```

4.6 Service Contract

Class Name	ServiceContract
Extends	Logger
Source	service-contract.ts
Examples	service-contract.spec.ts

4.6.1 constructor

```
new ServiceContract(options);
```

Creates a new `ServiceContract` instance.

Parameters

1. **options - ServiceContractOptions:** options for `ServiceContract` constructor.

- `cryptoProvider` - `CryptoProvider`: `CryptoProvider` instance
- `dfs` - `DfsInterface`: `DfsInterface` instance
- `keyProvider` - `KeyProvider`: `KeyProvider` instance
- `sharing` - `Sharing`: `Sharing` instance
- `web3` - `Web3`: `Web3` instance
- `defaultCryptoAlgo` - string (optional): crypto algorithm name from `CryptoProvider`, defaults to `aes`
- `log` - Function (optional): function to use for logging: `(message, level) => {...}`
- `logLevel` - `LogLevel` (optional): messages with this level will be logged with `log`
- `logLog` - `LogLogInterface` (optional): container for collecting log messages
- `logLogLevel` - `LogLevel` (optional): messages with this level will be pushed to `logLog`

Returns

`ServiceContract` instance

Example

```
const serviceContract = new ServiceContract({
  cryptoProvide,
  dfs,
  executor,
  keyProvider,
  loader,
  nameResolver,
  sharing,
  web3,
});
```

4.6.2 create

```
serviceContract.create(accountId, businessCenterDomain, service[, ↵
↵descriptionDfsHash]);
```

create and initialize new contract

Parameters

1. `accountId` - string: owner of the new contract and transaction executor
2. `businessCenterDomain` - string: ENS domain name of the business center
3. `service` - any: service definition
4. `descriptionHash` - string (optional): bytes2 hash of DBCP description, defaults to `0x00`

Returns

Promise returns any: contract instance

Example

```
const serviceContract = await serviceContract.create(accounts[0], ↵  
↵businessCenterDomain, sampleService);
```

4.6.3 = Service =

The service is the communication pattern definition for the `ServiceContract`. A single service contract can only have one service definition and all calls and answers must follow its defition.

To create calls and answers with different patterns, create a new `ServiceContract` and use an updated service definition there.

4.6.4 setService

```
serviceContract.setService(contract, accountId, service, businessCenterDomain[, ↵  
↵skipValidation]);
```

Set service description.

Parameters

1. `contract` - any|string: smart contract instance or contract ID
2. `accountId` - string: Ethereum account ID
3. `service` - any: service to set
4. `businessCenterDomain` - string: domain of the business the service contract belongs to
5. `skipValidation` - bool (optional): skip validation of service definition, validation is enabled by default

Returns

Promise returns void: resolved when done

Example

```
await serviceContract.setService(contract, accounts[0], sampleService,   
↪businessCenterDomain);
```

4.6.5 getService

```
serviceContract.getService(contract, accountId);
```

Gets the service of a service contract.

Parameters

1. contract - any|string: smart contract instance or contract ID
2. accountId - string: Ethereum account ID

Returns

Promise returns string: service description as JSON string

Example

```
const service = await sc.getService(contract, accounts[0]);
```

4.6.6 = Calls =

Calls are the requests done by authors, that initiate a service conversation. They are basically the first part of conversations and allow answers to be added to them. Calls are usually broadcasted or multicasted.

Samples for calls are:

- capacity requests
- information requests
- information broadcasts

4.6.7 sendCall

```
serviceContract.sendCall(contract, accountId, call);
```

Send a call to a service.

Parameters

1. `contract` - `any|string`: smart contract instance or contract ID
2. `accountId` - `string`: Ethereum account ID
3. `call` - `any`: call to send

Returns

Promise **returns** number: id of new call

Example

```
const callId = await serviceContract.sendCall(contract, accounts[0], sampleCall);
```

4.6.8 getCalls

```
serviceContract.getCalls(contract, accountId[, count, offset, reverse]);
```

Get all calls from a contract.

Parameters

1. `contract` - `any|string`: smart contract instance or contract ID
2. `accountId` - `string`: Ethereum account ID
3. `count` - `number` (optional): number of elements to retrieve, defaults to 10
4. `offset` - `number` (optional): skip this many elements, defaults to 0
5. `reverse` - `boolean` (optional): retrieve last elements first, defaults to `false`

Returns

Promise **returns** `any[]`: the calls

Example

```
const calls = await serviceContract.getCalls(contract, accounts[0]);
```

4.6.9 getCall

```
serviceContract.getCall(contract, accountId, callId);
```

Get a call from a contract.

Parameters

1. `contract` - `any|string`: smart contract instance or contract ID
2. `accountId` - `string`: Ethereum account ID
3. `callId` - `number`: index of the call to retrieve

Returns

Promise **returns** any: a single call

Example

```
const call = await serviceContract.getCall(contract, accounts[0], 12);
```

4.6.10 getCallCount

```
serviceContract.getCallCount(contract);
```

Get number of calls of a contract.

Parameters

1. `contract` - `any|string`: smart contract instance or contract ID

Returns

Promise **returns** number: number of calls

Example

```
let callCount = await serviceContract.getCallCount(contract);
console.log(callCount);
// Output:
// 2
await serviceContract.sendCall(contract, accounts[0], sampleCall);
callCount = await serviceContract.getCallCount(contract);
console.log(callCount);
// Output:
// 3
```

4.6.11 getCallOwner

```
serviceContract.getCallOwner(contract, callId);
```

Gets the owner/creator of a call.

Parameters

1. `contract` - any|string: smart contract instance or contract ID
2. `callId` - number: index of the call to retrieve owner for

Returns

Promise returns string: account id of call owner

Example

```
console.log(await serviceContract.getCallOwner(contract, 2));  
// Output:  
0x0000000000000000000000000000000000000000000000000000000000000001
```

4.6.12 addToCallSharing

```
serviceContract.addToCallSharing(contract, accountId, callId, to[, hashKey,   
↪contentKey, section]);
```

Adds list of accounts to a calls sharings list.

Parameters

1. `contract` - any|string: smart contract instance or contract ID
2. `accountId` - string: Ethereum account ID
3. `callId` - number: id of the call to retrieve
4. `to` - string[]: accountIds, to add sharings for
5. `hashKey` - string (optional): hash key to share, if omitted, key is retrieved with `accountId`
6. `contentKey` - string (optional): content key to share, if omitted, key is retrieved with `accountId`
7. `section` - string (optional): section to share key for, defaults to '*'

Returns

Promise returns void: resolved when done

Example

```
// account[0] adds accounts[2] to a sharing
await serviceContract.addToCallSharing(contract, accounts[0], callId, [accounts[2]]);
```

4.6.13 = Answers =

Answers are replies to calls. Answers can only be created as answers to calls. Answers are usually directed to the author of a call.

Examples are

- capacity replies
- information responses

4.6.14 sendAnswer

```
serviceContract.sendAnswer(contract, accountId, answer, callId, callAuthor);
```

Send answer to service contract call.

Parameters

1. contract - any|string: smart contract instance or contract ID
2. accountId - string: Ethereum account ID
3. answer - any: answer to send
4. callId - number: index of the call to which the answer was created
5. callAuthor - string: Ethereum account ID of the creator of the initial call

Returns

Promise returns number: id of new answer

Example

```
await serviceContract.inviteToContract(businessCenterDomain, contract.options.address,
  ↳ accounts[0], accounts[2]);
const contentKey = await sharing.getKey(contract.options.address, accounts[0], '*',
  ↳ 0);
await sharing.addSharing(contract.options.address, accounts[0], accounts[2], '*', 0,
  ↳ contentKey);
await serviceContract.sendCall(contract, accounts[0], sampleCall);
const call = await serviceContract.getCall(contract, accounts[0], 0);
const answerId = await serviceContract.sendAnswer(contract, accounts[2], sampleAnswer,
  ↳ 0, call.metadata.author);
```

4.6.15 getAnswers

```
serviceContract.getAnswers(contract, accountId, callId[, count, offset, reverse]);
```

Retrieves answers for a given call.

Parameters

1. `contract` - `any|string`: smart contract instance or contract ID
2. `accountId` - `string`: Ethereum account ID
3. `callId` - `number`: index of the call to which the answers were created
4. `count` - `number` (optional): number of elements to retrieve, defaults to 10
5. `offset` - `number` (optional): skip this many elements, defaults to 0
6. `reverse` - `boolean` (optional): retrieve last elements first, defaults to `false`

Returns

Promise `returns any[]`: the answers

Example

```
const answers = await serviceContract.getAnswers(contract, accounts[0], 12);
```

4.6.16 getAnswer

```
serviceContract.getAnswer(contract, accountId, answerIndex);
```

Get a answer from a contract.

Parameters

1. `contract` - `any|string`: smart contract instance or contract ID
2. `accountId` - `string`: Ethereum account ID
3. `callId` - `number`: index of the call to which the answer was created
4. `answerIndex` - `number`: index of the answer to retrieve

Returns

Promise `returns any`: a single answer

Example

```
const answer = await serviceContract.getAnswer(contract, accounts[0], 12, 2);
```

4.6.17 getAnswerCount

```
serviceContract.getAnswerCount(contract, callId);
```

Retrieves number of answers for a given call.

Parameters

1. `contract` - any|string: smart contract instance or contract ID
2. `callId` - number: index of the call to which the answer was created

Returns

Promise **returns** number: number of answers

Example

```
const sampleCallId = 3;
let answerCount = await serviceContract.getAnswerCount(contract, sampleCallId);
console.log(answerCount);
// Output:
// 2
await serviceContract.sendAnswer(contract, accounts[0], sampleAnswer, sampleCallId,
↪accounts[1]);
answerCount = await serviceContract.getAnswerCount(contract, sampleCallId);
console.log(answerCount);
// Output:
// 3
```

4.7 Digital Twin Usage Examples

This section shows a few usage examples that can occur in common digital twin usage scenarios and cover handling of the modules *DigitalTwin* and *Container*. The examples in here are build around the management of data for heavy construction machines and cover common events link storing data, sharing data, etc.

- *manufacturer*: original manufacturer of the heavy machine
- *customer*: a user, that has bought the physical heavy machine
- *service-technician*: hired by the *customer* to perform maintenance on the heavy machine

The examples in this section use these variables for aforementioned users:

```
const manufacturer = '0x0000000000000000000000000000000000000000000000000000000000000001';
const customer = '0x0000000000000000000000000000000000000000000000000000000000000002';
const serviceTechnician = '0x0000000000000000000000000000000000000000000000000000000000000003';
```

4.7.1 Create a Digital Twin

Digital Identities are collections of data related to a “thing”. A “thing” can be basically anything, a bird, a plane or someone from the planet Krypton. Jokes aside, it most commonly describes a physical object - like in our example here a heavy machine.

So let’s create a digital twin for our heavy machine “Big Crane 250”, which is done with the `DigitalTwin.create` function:

```
const bigCrane250 = await DigitalTwin.create(runtime, { accountId: manufacturer });
```

This creates a new digital twin for the account `manufacturer`, which can now add containers to it or set other properties.

The `DigitalTwin.create` *config* argument function supports more properties than just *accountId*.

You can **and should** give your digital twin a *DBCP* description. To do this pass it to the new digital twin in the *config* property.

```
const description = {
  name: 'Big Crane 250',
  description: 'Digital Twin for my heavy machine "Big Crane 250"',
  author: 'Manufacturer',
  version: '0.1.0',
  dbcpVersion: 2,
};
const bigCrane250 = await DigitalTwin.create(
  runtime, { accountId: manufacturer, description });
```

If you do not set a description, at creation time, a default description is set. This description is available at the `DigitalTwin.defaultDescription` and can be used as a starting point for your own description. A description can be updated later on as well, see `digitalTwin.setDescription`.

So let’s say, we have created a digital twin for our heavy machine with the setup from the last code example. We now have the following setup:

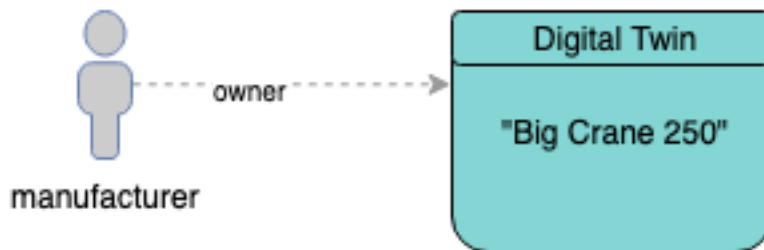


Fig. 1: manufacturer created a digital twin

4.7.2 Add Containers to Digital Twin

Continuing with the digital twin from the last section we add a container, that holds manufacturers private data with information about the production process and a link to a manual file. This can be done with the `digitalTwin.createContainers` function:

```
const { data } = await bigCrane250.createContainers({
  data: {},
});
```

The manufacturer account now has created a *Container* instance called `data`. This can be customized as described at `Container.create`.

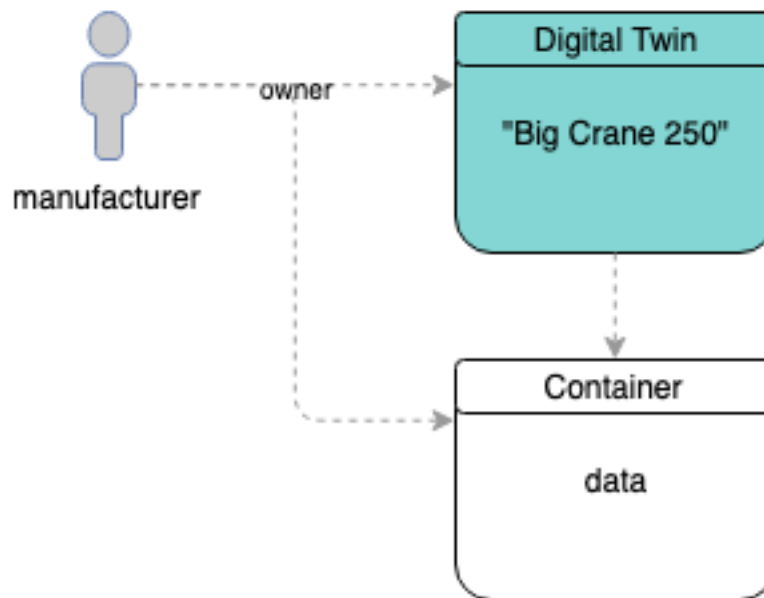


Fig. 2: manufacturer added a container to the twin

4.7.3 Add Data to the Container

Continuing the example, the manufacturer adds data to the container.

```
await data.setEntry(
  'productionProfile',
  {
    id: 'BC250-4711',
    dateOfManufacturing: '1554458858126',
    category: 'hem-c',
  },
);
await data.setEntry('manual', 'https://a-link-the-manual...');
```

As these properties are new, `container.setEntry` adds a role for each property and the owner of the digital twin joins this role. During this role 0 to 63 are skipped as they are system reserved and can be used for more complex contract role setups. So the roles 64 (for `productionProfile`) and 65 (for `manual`) are created.

For each new property a new encryption key is generated and stored in the contracts *Sharings*. When new properties are added, this key is only shared for the owner of the digital twin, so only the owner can access the data stored in the contract.

Data can be read from the containers with `container.getEntry`:

```
const productionProfile = await data.getEntry('productionProfile');
```

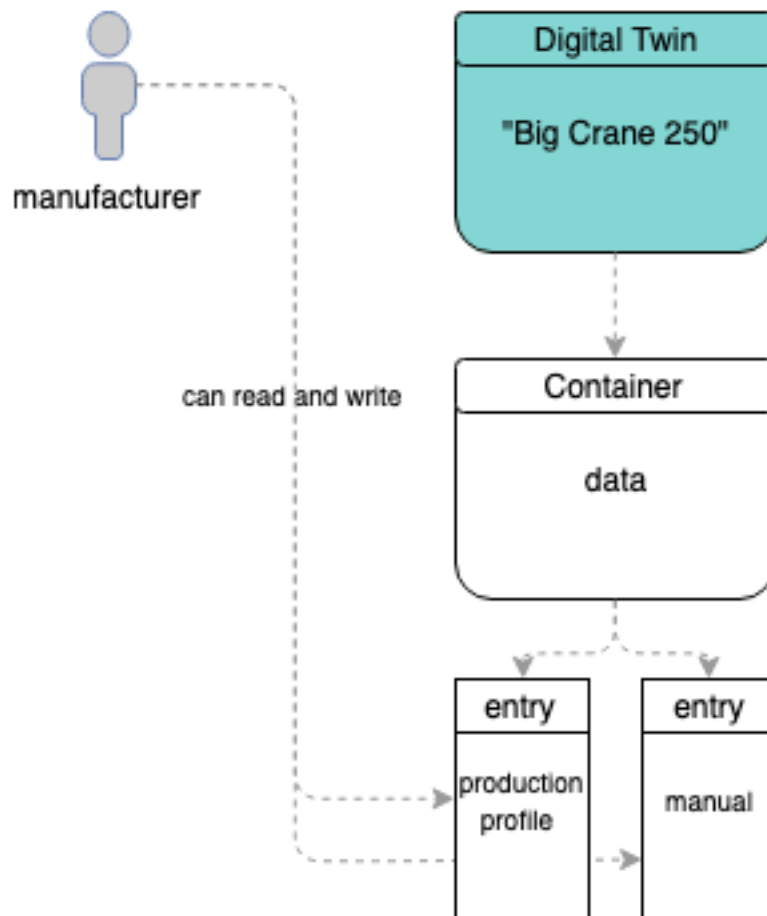


Fig. 3: manufacturer added entries to the container

4.7.4 Share Container Properties

As already said, the manufacturer wants to keep production data for own usage and share a link to the manual to the account `customer`. When not explicitly shared, properties are kept private, so nothing to do for the field `productionProfile`. To allow other accounts to access `manual`, encryption keys have to be shared, which can be done with `container.shareProperties`:

```
await data.shareProperties([
  { accountId: customer, read: ['manual'] }
]);
```

With this call, the account `customer` is added to the role 1 (member), which allows basic contract interaction but not necessarily access to the data. And because `manual` has been specified as a read (-only) field, this account receives an encryption key for the property `manual`, so it is now able to read data from this field.

To load data from the twins, customer can now fetch the container from the digital twin and load its data. Let's assume manufacturer has communicated the address of the digital twin (e.g. 0x00000000000000000000000000000000c1) to customer and the customer can access the link to the manual with:

```
const bigCrane250LoadedFromCustomer = new DigitalTwin(
    runtime, { accountId: customer, address: '0x00000000000000000000000000000000c1' });
const dataLoadedFromCustomer = await bigCrane250LoadedFromCustomer.getEntry('data');
const link = await dataLoadedFromCustomer.getEntry('manual');
```

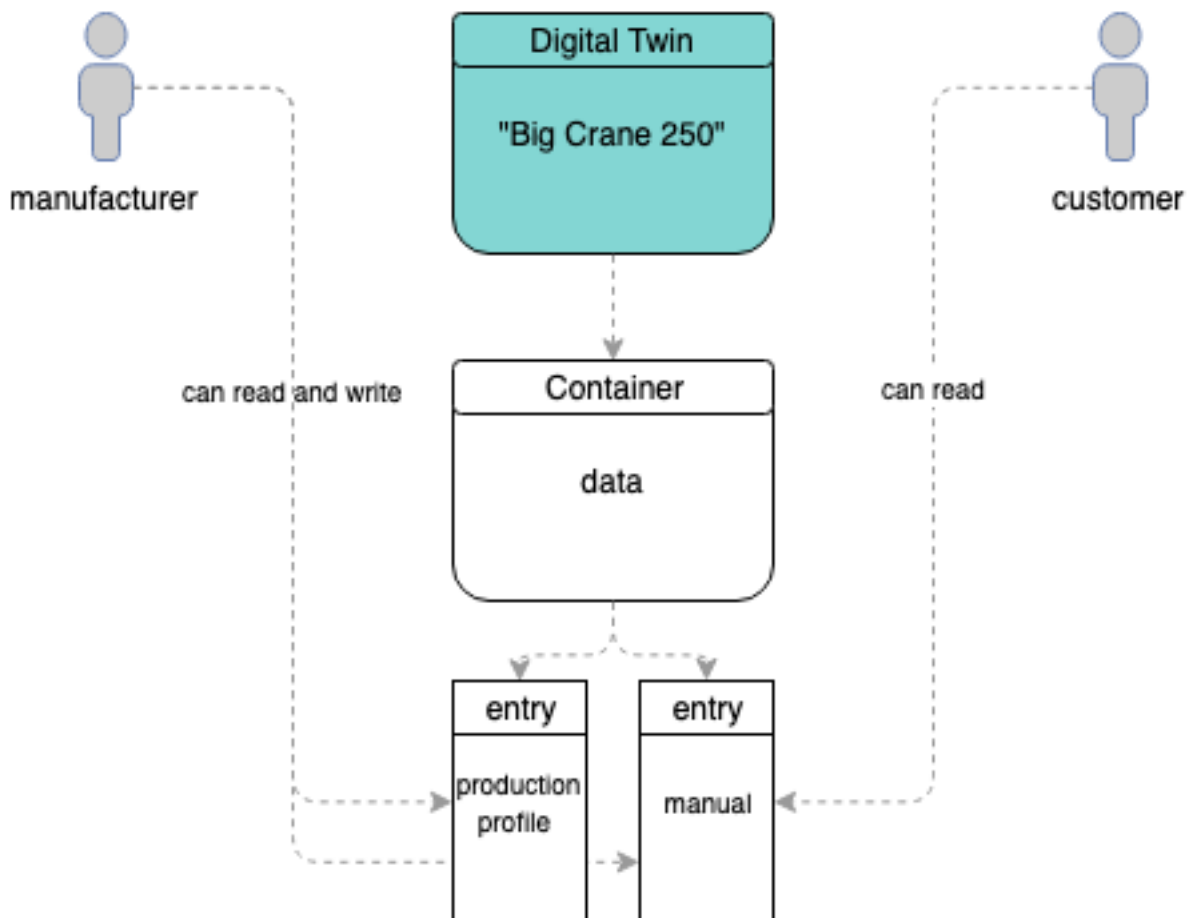


Fig. 4: customer can read entry “manual”

4.7.5 Cloning Containers

If `customer` wants to re-use data from a data container or an entire data container but have ownership over it, it can clone it and use it in an own digital twin contract. This can be done with `Container.clone`:

```
const dataClone = await Container.clone(
  runtime, { accountId: customer }, dataLoadedFromCustomer);
```

This clone can be linked to a digital twin owner by `customer`. So let's create a new one and add the clone to it:

```
const customersDescription = {
  name: 'My own Big Crane 250',
  description: 'I bought a Big Crane 250 and this is my collection of data for it',
  author: 'Customer',
  version: '0.1.0',
  dbcpVersion: 2,
};
const customersBigCrane250 = await DigitalTwin.create(
  runtime, { accountId: customer, description: customersDescription });

await customersBigCrane250.setEntry(
  'machine-data',
  dataClone,
  DigitalTwinEntryType.Container,
);
```

Note that the container is not named `data` like in the original twin but called `machine-data` here. Names can be reassigned as desired.

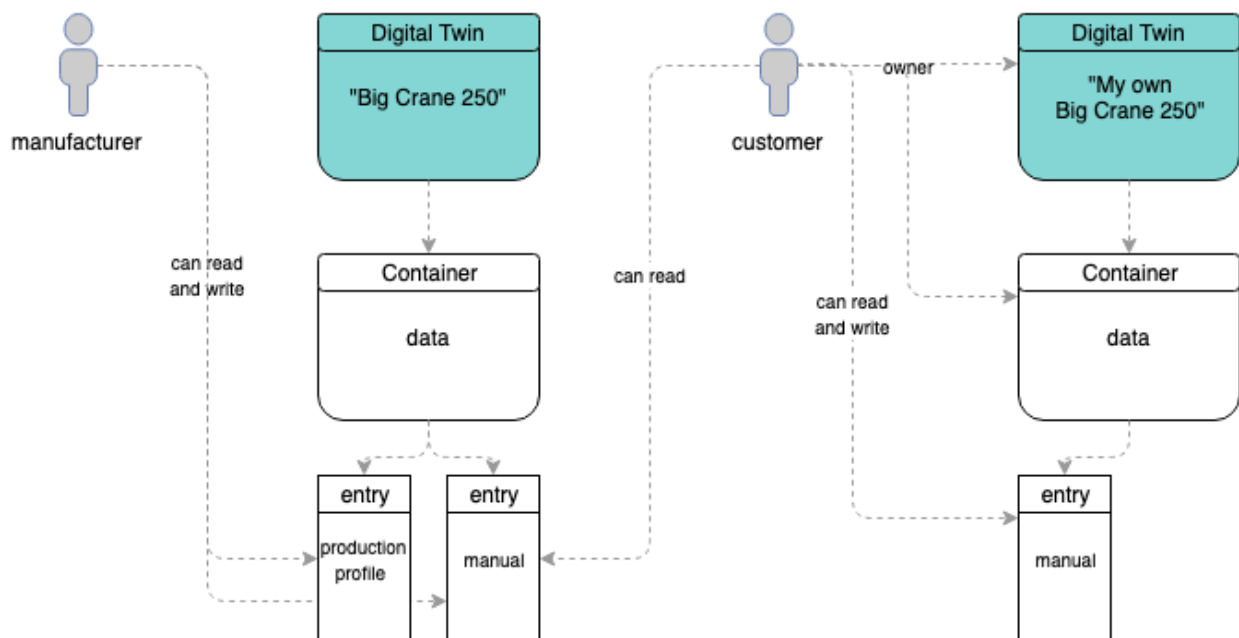


Fig. 5: customer cloned data container

4.7.6 Granting Write Access

Properties at *Containers* can be “entries” as used in the last examples or “list entries”. To add data to lists call `container.addListEntries`:

```
await dataClone.addListEntries(
  'usagelog',
  [ 'I started using my new Big Crane 250' ]
);
```

Now customer wants to invite serviceTechnician and allow this account to add entries to the list usagelog as well. To do this, the list is shared the same way as in the previous example, but the field is shared as readWrite:

```
await dataClone.shareProperties([
  { accountId: customer, readWrite: ['usagelog'] }
]);
```

serviceTechnician can now write to the list usagelog and we now have the following setup:

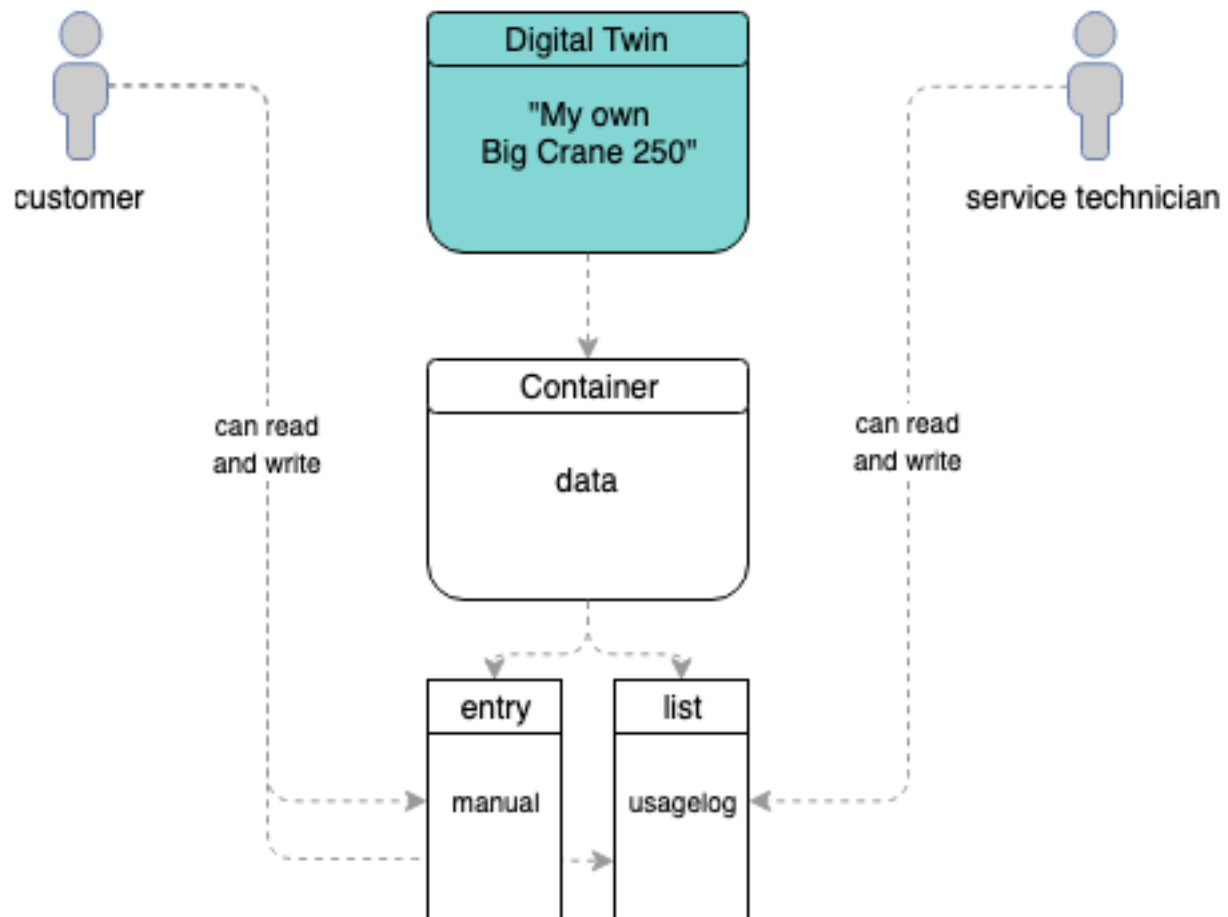


Fig. 6: customer invited service technician

4.7.7 Handling Files

Containers can hold files as well. File handling follows a few simple principles:

- files are stored encrypted (as everything in containers is stored encrypted)

- files are always stored as an array of files (think of it like a folder with files)
- files are encrypted, uploaded and a reference is stored as a file at the contract (sounds like the default [Hybrid Storage](#)) approach, but is a reapplication to itself, as encrypted additional files with references to the original encrypted files are stored at the contract

Okay, let's add some files to a container (taken from our [tests](#)).

A file needs to be provided as a buffer. In NodeJs, this can be done with `fs.readFile`

```
import { promisify } from 'util';
import { readFile } from 'fs';

const file = await promisify(readFile)(
  `${__dirname}/testfiles/animal-animal-photography-cat-96938.jpg`);
```

The file is expected to be wrapped in a specific container format, which is defined in the [ContainerFile](#) interface. So let's build such a file object and store it in an object including a property called `files`, as files are always provided as arrays of [ContainerFile](#) instances to the API:

```
const sampleFiles = {
  files:[{
    name: 'animal-animal-photography-cat-96938.jpg',
    fileType: 'image/jpeg',
    file,
  }]
};
```

If not already done, create (or load) a container:

```
const container = await Container.create(runtime, config);
```

If not already done, add a field for files to our container, for this the static property `Container.defaultTemplates` can be useful:

```
await container.ensureProperty('sampleFiles', Container.defaultSchemas.filesEntry);
```

So now everything is set up and we can store our file:

```
await container.setEntry('sampleFiles', sampleFiles);
```

And later on we can retrieve our file with:

```
await container.getEntry('sampleFiles');
```

That's it for the simple case. If you want to get fancy, you can have a look at the more complex examples in the tests. With the build in file handling you can:

- store lists of files in an entry (this example) test path: [Container/when setting entries/can handle files](#)
- store lists of files in complex objects (e.g. if you want to annotate them) test path: [Container/when setting entries/can handle files in complex objects](#)
- store a list of lists of files (hands up, who tripped, when reading this, *me too*, it's basically a list of directories), this can be used to store different versions of files or separate file groups which have no relations between them test path: [Container/when setting list entries/can handle files](#)
- store a list of lists of files (a combination between lists and complex objects) test path: [Container/when setting list entries/can handle files in complex objects](#)

4.8 Digital Twin

Class Name	DigitalTwin
Extends	Logger
Source	digital-twin.ts
Examples	digital-twin.spec.ts

TL;DR: usage examples and a data flow can be found [here](#).

A `DigitalTwin` is a wrapper, that holds data or references to data. This data can be

- a `Container` - This is the most common use case, actual data is stored in the container and the `DigitalTwin` merely holds a reference to the container.
- an `DigitalTwin` - `DigitalTwins` can be linked together, so an entry can be another `DigitalTwin`, which allows to navigate through twins to retrieve data properties from linked twins, see [getEntry](#)
- an account address / a contract address
- `bytes32` hashes (e.g. file hashes)

4.8.1 constructor

```
new DigitalTwin(options, config);
```

Create new `DigitalTwin` instance. This will not create a smart contract contract but is used to load existing digital twins. To create a new contract, use the static [create](#) function.

Parameters

1. `options` - `DigitalTwinOptions`: runtime-like object with required modules

- `contractLoader` - `ContractLoader`: `ContractLoader` instance
- `cryptoProvider` - `CryptoProvider`: `CryptoProvider` instance
- `dataContract` - `DataContract`: `DataContract` instance
- `description` - `Description`: `Description` instance
- `executor` - `Executor`: `Executor` instance
- `log` - Function (optional): function to use for logging: `(message, level) => {...}`
- `logLevel` - `LogLevel` (optional): messages with this level will be logged with `log`
- `logLog` - `LogLogInterface` (optional): container for collecting log messages
- `logLogLevel` - `LogLevel` (optional): messages with this level will be pushed to `logLog`
- `nameResolver` - `NameResolver`: `NameResolver` instance
- `profile` - `Profile`: `Profile` instance

- `rightsAndRoles` - `RightsAndRoles`: `RightsAndRoles` instance
- `sharing` - `Sharing`: `Sharing` instance
- `verifications` - `Verifications`: `Verifications` instance
- `web3` - `Web3`: `Web3` instance

2. `config` - `DigitalTwinConfig`: digital twin related config

- `accountId` - string: account id of user, that interacts with digital twin
- `containerConfig` - `ContainerConfig`: address of a `DigitalTwin` instance, can be ENS or contract address
- `address` - string (optional): address of an `DigitalTwin` instance, can be ENS or contract address
- `description` - string: description has to be passed to `.create` to apply it to contract
- `factoryAddress` - string (optional): factory address can be passed to `.create` for customer digital twin factory

Returns

`DigitalTwin` instance

Example

```
const digitalTwin = new DigitalTwin(
  runtime,
  {
    accountId: '0x0000000000000000000000000000000000000000',
    address: 'sample-digital-twin.somewhere.evan',
  },
);
```

4.8.2 = Creating Digital Identities =

4.8.3 create

```
DigitalTwin.create(runtime, config);
```

Create digital twin contract.

Note, that this function is static. It is used on the `DigitalTwin` class object and returns a `DigitalTwin` class instance.

The options argument has the same structure as the options object that is passed to the constructor as it is used for the new `DigitalTwin` instance. The `config` argument requires a proper value for the property description.

4.8.4 = Container =

4.8.5 createContainers

```
digitalTwin.createContainers(containers);
```

Create new *Container* instances and add them as entry to twin.

When a container entry fetched with *getEntry* or *getEntry*, the value will become a *Container* instance and can be used as such.

Parameters

1. containers - { [id: string]: Partial<ContainerConfig> }: object with containers to create, name is used as entry name in twin

Returns

Promise returns { [id: string]: Container }: map with Container instances

Example

```
const containers = await twin.createContainers({
  entry1: { template: 'metadata' },
  entry2: { template: 'metadata' },
});
```

4.8.6 = Entries =

4.8.7 setEntry

```
digitalTwin.setEntry(name, value, entryType);
```

Set entry in index contract; entries are unique, setting the same name a second time will overwrite the first value.

Parameters

1. name - string: entry name
2. value - string: value to set
3. entryType - DigitalTwinType: type of given value

Returns

Promise returns void: resolved when done

Example

```
await digitalTwin.setEntry('myId', accountId, DigitalTwinEntryType.AccountId);
console.log((await digitalTwin.getEntry('myId')).value);
// Output:
// 0x0000000000000000000000000000000000000000000000000000000000000001
```

4.8.8 setEntries

```
digitalTwin.setEntries(arguments);
```

Set multiple entries at index contract.

Parameters

1. entries - {[id: string]: DigitalTwinIndexEntry}: entries to set

Returns

Promise returns void: resolved when done

Example

```
const sampleContractId = '0x00000000000000000000000000000000000000000000000000000000000000c0274ac7';
await digitalTwin.setEntries({
  'account': { value: accountId, entryType: DigitalTwinEntryType.AccountId },
  'contract': { value: sampleContractId, entryType: DigitalTwinEntryType.
↳GenericContract },
});

const result = (await digitalTwin.getEntries()).map(entry => value);
console.log(result.account.value);
// Output:
// 0x0000000000000000000000000000000000000000000000000000000000000001
console.log(result.contract.value);
// Output:
// 0x00000000000000000000000000000000000000000000000000000000000000c0274ac7
```

4.8.9 getEntry

```
digitalTwin.getEntry(name);
```

Get single entry from index contract.

When this twin has other twins as its entries, properties from those can be selected by building a path of properties.

For example a twin called `car` may have a link to another twin under the name `tire`. The twin `tire` has an entry called `metadata`. It is possible to retrieve this entry from the twin `car` with:

```
const tireMetadata = await car.getEntry('tire/metadata');
```

Parameters

1. name - string: entry name or path to data in linked twin

Returns

Promise returns `DigitalTwinIndexEntry`: entry object

Example

```
await digitalTwin.setEntry('myId', accountId, DigitalTwinEntryType.AccountId);
console.log((await digitalTwin.getEntry('myId')).value);
// Output:
// 0x0000000000000000000000000000000000000000000000000000000000000001
```

4.8.10 getEntries

```
digitalTwin.getEntries();
```

Get all entries from index contract.

Returns

Promise returns `{[id: string]: DigitalTwinIndexEntry}`: key-value map with all entries

Example

```
const sampleContractId = '0x000000000000000000000000000000000000000000000c0274ac7';
await digitalTwin.setEntries({
  'account': { value: accountId, entryType: DigitalTwinEntryType.AccountId },
  'contract': { value: sampleContractId, entryType: DigitalTwinEntryType.
↳GenericContract },
});

const result = (await digitalTwin.getEntries()).map(entry => value);
console.log(result.account.value);
// Output:
// 0x0000000000000000000000000000000000000000000000000000000000000001
console.log(result.contract.value);
// Output:
// 0x000000000000000000000000000000000000000000000c0274ac7
```

4.8.11 = Verifications =

4.8.12 addVerifications

```
digitalTwin.addVerifications(verifications);
```

Add verifications to this twin; this will also add verifications to contract description.

If the calling account is the owner of the identity of the digital twin

- the description will be automatically updated with tags for verifications
- verifications issued with this function will be accepted automatically

See interface `DigitalTwinVerificationEntry` for input data format.

Parameters

1. `verifications - DigitalTwinVerificationEntry[]`: list of verifications to add

Returns

Promise returns void: resolved when done

Example

```
await digitalTwin.addVerifications([ { topic: 'exampleVerification' } ]);
```

4.8.13 getVerifications

```
digitalTwin.getVerifications();
```

Gets verifications from description and fetches list of verifications for each of them.

See [Verifications](#) documentation for details on output data format.

Returns

Promise returns any: list of verification lists from [Verifications](#), `getVerifications`

Example

```
await digitalTwin.addVerifications([ { topic: 'exampleVerification' } ]);  
const verifications = await digitalTwin.getVerifications();
```

4.8.14 = Descriptions =

4.8.15 getDescription

```
digitalTwin.getDescription();
```

Returns description from digital twin.

Returns

Promise returns any: public part of the description

Example

```
const description = await container.getDescription();
console.dir(description);
// Output:
// { name: 'test twin',
//   description: 'twin from test run',
//   author: 'evan GmbH',
//   version: '0.1.0',
//   dbcpVersion: 2,
//   tags: [ 'evan-digital-twin' ],
//   identity:
//     '0x1a496043385fec8d52f61e2b700413f8e12eb6e7e11649f80c8f4716c1063d06' }
```

4.8.16 setDescription

```
digitalTwin.setDescription(description);
```

Write given description to digital twins DBCP.

Parameters

1. description - any: description (public part)

Returns

Promise returns void: resolved when done

Example

```
// get current description
const description = await digitalTwin.getDescription();
console.dir(description);
// Output:
// { name: 'test twin',
```

(continues on next page)

(continued from previous page)

```
//  description: 'twin from test run',
//  author: 'evan GmbH',
//  version: '0.1.0',
//  dbcpVersion: 2,
//  tags: [ 'evan-digital-twin' ],
//  identity:
//    '0x1a496043385fec8d52f61e2b700413f8e12eb6e7e11649f80c8f4716c1063d06' }

// update description
description.version = '0.1.1';
await digitalTwin.setDescription(description);

// fetch again
console.dir(await digitalTwin.getDescription());
// Output:
// { name: 'test twin',
//   description: 'twin from test run',
//   author: 'evan GmbH',
//   version: '0.1.1',
//   dbcpVersion: 2,
//   tags: [ 'evan-digital-twin' ],
//   identity:
//     '0x1a496043385fec8d52f61e2b700413f8e12eb6e7e11649f80c8f4716c1063d06' }
```

4.8.17 = Profile =

4.8.18 addAsFavorite

```
digitalTwin.addAsFavorite();
```

Add the digital twin with given address to profile.

Returns

Promise returns void: resolved when done

Example

```
const digitalTwin = new DigitalTwin(options.config);
if (await digitalTwin.isFavorite()) {
  console.log('I know this digital twin!');
} else {
  await digitalTwin.addToFavorites();
  console.log('bookmarked digital twin');
}
```


4.8.20 isFavorite

```
digitalTwin.isFavorite();
```

Check if this digital twin is bookmarked in profile.

Returns

Promise returns boolean: true if bookmarked

Example

```
const digitalTwin = new DigitalTwin(options.config);
if (await digitalTwin.isFavorite()) {
  console.log('I know this digital twin!');
} else {
  await digitalTwin.addToFavorites();
  console.log('bookmarked digital twin');
}
```

4.8.21 removeFromFavorites

```
digitalTwin.removeFromFavorites();
```

Removes the current twin from the favorites in profile.

Returns

Promise returns void: resolved when done

Example

```
const digitalTwin = new DigitalTwin(options.config);
if (await digitalTwin.isFavorite()) {
  await digitalTwin.removeFromFavorites();
  console.log('removed digital twin from favorites');
}
```

4.8.22 = Utilities =

4.8.23 getValidity

```
DigitalTwin.getValidity(options, ensAddress);
```

Check if a valid contract is located under the specified address, which allows to check for twins before actually loading them.

Return value properties have the following meaning:

- `valid`: `true` if contract could not be found or if it doesn't have the tag "evan-digital-twin"
- `exists`: `true` if a contract address could be found at given ENS address
- `error`: an error object, if one of the other properties is `false`

Note, that this function is called on the Class `DigitalTwin` and not on an instance of it.

Parameters

1. `options` - `DigitalTwinOptions`: twin runtime options

- `contractLoader` - `ContractLoader`: `ContractLoader` instance
- `cryptoProvider` - `CryptoProvider`: `CryptoProvider` instance
- `dataContract` - `DataContract`: `DataContract` instance
- `description` - `Description`: `Description` instance
- `executor` - `Executor`: `Executor` instance
- `nameResolver` - `NameResolver`: `NameResolver` instance
- `rightsAndRoles` - `RightsAndRoles`: `RightsAndRoles` instance
- `sharing` - `Sharing`: `Sharing` instance
- `verifications` - `Verifications`: `Verifications` instance
- `web3` - `Web3`: `Web3` instance
- `log` - `Function` (optional): function to use for logging: `(message, level) => {...}`
- `logLevel` - `LogLevel` (optional): messages with this level will be logged with `log`
- `logLog` - `LogLogInterface` (optional): container for collecting log messages
- `logLogLevel` - `LogLevel` (optional): messages with this level will be pushed to `logLog`

2. `ensAddress` - `string`: ens address that should be checked

Returns

Promise `returns` { `valid`: `boolean`, `exists`: `boolean`, `error`: `Error` }: resolved when done

Example

```
const { valid } = await DigitalTwin.getValidity(runtime, address);
if (!valid) {
  throw new Error(`no valid digital twin found at "${address}"`);
}
```


4.8.26 Additional Components

4.8.27 Public Properties

defaultDescription (static)

Default description used when no specific description is given to *.create*.

```
console.dir(DigitalTwin.defaultDescription);  
// Output:  
// {  
//   name: 'Digital Identity',  
//   description: 'Digital Identity Contract',  
//   author: '',  
//   version: '0.1.0',  
//   dbcpVersion: 2  
// }
```

4.8.28 Enums

DigitalTwinEntryType

possible entry types for entries in index

1. AccountId
2. Container
3. FileHash
4. GenericContract
5. Hash
6. DigitalTwin

4.8.29 Interfaces

DigitalTwinConfig

config for digital twin

1. accountId - string: account id of user, that interacts with digital twin
2. containerConfig - ContainerConfig: address of a DigitalTwin instance, can be ENS or contract address
3. address - string (optional): address of an DigitalTwin instance, can be ENS or contract address
4. description - string (optional): description has to be passed to *.create* to apply it to to contract
5. factoryAddress - string (optional): factory address can be passed to *.create* for customer digital twin factory

DigitalTwinIndexEntry

container for digital twin entry values

1. `entryType` - `DigitalTwinEntryType` (optional): type of entry in index
2. `raw` - any (optional): raw value (bytes32 hash)
3. `value` - any (optional): decrypted/loaded value

DigitalTwinVerificationEntry

data for verifications for digital twins

1. `topic` - string: name of the verification (full path)
2. `descriptionDomain` - string (optional): domain of the verification, this is a subdomain under 'verifications.evan', so passing 'example' will link verifications
3. `disableSubVerifications` - boolean (optional): if true, verifications created under this path are invalid, defaults to false
4. `expirationDate` - number (optional): expiration date, for the verification, defaults to 0 (does not expire)
5. `verificationValue` - any (optional): json object which will be stored in the verification

4.9 Container

Class Name	Container
Extends	Logger
Source	container.ts
Examples	container.spec.ts

TL;DR: usage examples and a data flow can be found [here](#).

The `Container` is an API layer over `DataContract` and combines its functionalities into a more use case oriented straight forward interface.

To reduce complexity the most common usage patterns from `DataContract` have been set as fixed in the `Container` implementation. Therefore the `Container` follows these principles:

- data is always encrypted
- each entry gets an own key for encryption
- each entry get an own role for granting write permissions
- an identity always is created for the container
- adding verifications adds the verification topic to the contract description to allow listing of all verifications of this container
- a property called `type` is added to the `Container` at creation type and marks its template type

4.9.1 constructor

```
new Container(options, config);
```

Create new `Container` instance. This will not create a smart contract contract but is used to load existing containers. To create a new contract, use the static `create` function.

Parameters

1. options - ContainerOptions: runtime for new container

- `contractLoader` - `ContractLoader`: `ContractLoader` instance
- `cryptoProvider` - `CryptoProvider`: `CryptoProvider` instance
- `dataContract` - `DataContract`: `DataContract` instance
- `description` - `Description`: `Description` instance
- `executor` - `Executor`: `Executor` instance
- `nameResolver` - `NameResolver`: `NameResolver` instance
- `rightsAndRoles` - `RightsAndRoles`: `RightsAndRoles` instance
- `sharing` - `Sharing`: `Sharing` instance
- `verifications` - `Verifications`: `Verifications` instance
- `web3` - `Web3`: `Web3` instance
- `log` - `Function` (optional): function to use for logging: `(message, level) => {...}`
- `logLevel` - `LogLevel` (optional): messages with this level will be logged with `log`
- `logLog` - `LogLogInterface` (optional): container for collecting log messages
- `logLogLevel` - `LogLevel` (optional): messages with this level will be pushed to `logLog`

2. config - DigitalTwinconfig: config for new container

- `accountId` - `string`: account id of user, that interacts with container
- `address` - `string`: address of a `DataContract` instance, can be ENS or contract address

Returns

`Container` instance

Example

```
const container = new Container(  
  runtime,  
  {  
    accountId: '0x0000000000000000000000000000000000000000',  
    address: 'samplecontainer.somewhere.evan',  
  },  
);
```


4.9.2 = Creating Containers =

4.9.3 create

```
Container.create(runtime, config);
```

Creates a new digital container contract on the blockchain.

Note, that this function is static. It is used on the `Container` class object and returns a `Container` class instance.

The options argument has the same structure as the options object that is passed to the constructor as it is used for the new `Container` instance. The `config` argument requires a proper value for the property description.

Parameters

1. options - ContainerOptions: runtime for new container

- `contractLoader` - `ContractLoader`: `ContractLoader` instance
- `cryptoProvider` - `CryptoProvider`: `CryptoProvider` instance
- `dataContract` - `DataContract`: `DataContract` instance
- `description` - `Description`: `Description` instance
- `executor` - `Executor`: `Executor` instance
- `nameResolver` - `NameResolver`: `NameResolver` instance
- `rightsAndRoles` - `RightsAndRoles`: `RightsAndRoles` instance
- `sharing` - `Sharing`: `Sharing` instance
- `verifications` - `Verifications`: `Verifications` instance
- `web3` - `Web3`: `Web3` instance
- `log` - `Function` (optional): function to use for logging: `(message, level) => {...}`
- `logLevel` - `LogLevel` (optional): messages with this level will be logged with `log`
- `logLog` - `LogLogInterface` (optional): container for collecting log messages
- `logLogLevel` - `LogLevel` (optional): messages with this level will be pushed to `logLog`

2. config - DigitalTwinconfig: config for new container

- `accountId` - `string`: account id of user, that interacts with container
- `address` - `string`: ENS address used for container
- `description` - `string`: description has to be passed to `.create` to apply it to contract
- `factoryAddress` - `string` (optional): factory address can be passed to `.create` for customer container factory
- `plugin` - `string|ContainerPlugin` (optional): plugin to be used in `.create`, can be string with name or a `ContainerPlugin`

Returns

Promise returns `Container`: new instance bound to new `DataContract`

- `web3` - `Web3`: `Web3` instance
- `log` - `Function` (optional): function to use for logging: `(message, level) => {...}`
- `logLevel` - `LogLevel` (optional): messages with this level will be logged with `log`
- `logLog` - `LogLogInterface` (optional): container for collecting log messages
- `logLogLevel` - `LogLevel` (optional): messages with this level will be pushed to `logLog`

2. config-DigitalTwinconfigIg: config for new container

- `accountId - string`: account id of user, that interacts with container
- `address - string`: ENS address used for container
- `description - string`: description has to be passed to `.create` to apply it to contract
- `factoryAddress - string (optional)`: factory address can be passed to `.create` for customer container factory
- `plugin - string|ContainerPlugin (optional)`: plugin to be used in `.create`, can be string with name or a `ContainerPlugin`

3. source - Container: container to clone

4. `copyValues` - boolean: copy entry values from source contract to new contract

Returns

Promise returns Container: new instance bound to new DataContract and a copy of source

Example

[illegible]

4.9.5 deleteContainerPlugin

```
container.deleteContainerPlugin(profile);
```

Remove a container plugin from a users profile.

Parameters

1. Profile - `Profile`: profile instance
2. name - string: plugin name

Returns

Promise **returns** void

Example

```
await Container.deleteContainerPlugin(profile, 'awesomeplugin');
```

4.9.6 getContainerPlugin

```
container.getContainerPlugin(profile, name);
```

Get one container plugin for a users profile by name.

Parameters

1. Profile - `Profile`: profile instance
2. name - string: plugin name

Returns

Promise **returns** ContainerPlugin

Example

```
const accountId1 = '0x0000000000000000000000000000000000000000000000000000000000000001';
const plugin = await Container.getContainerPlugin(profile, 'awesomeplugin');

// create container with accountId
const container = await Container.create(options, {
  ...config,
  accountId: accountId1,
  description: plugin.description,
  plugin: plugin,
});
```

4.9.7 getContainerPlugins

```
container.getContainerPlugins(profile);
```

Get all container plugins for a users profile

Parameters

1. Profile - `Profile`: profile instance
2. `loadContracts` - boolean (default = true): run `loadBcContract` directly for all saved entries (if false, unresolved ipld tree will be returned as value)

Returns

Promise returns `Array<ContainerPlugin>`

Example

```
const accountId1 = '0x0000000000000000000000000000000000000000000000000000000000000001';
const plugins = await Container.getContainerPlugins(profile);

// create container with accountId
const container = await Container.create(options, {
  ...config,
  accountId: accountId1,
  description: plugins['awesomeplugin'].description,
  plugin: plugins['awesomeplugin'],
});
```

4.9.8 saveContainerPlugin

```
container.saveContainerPlugin(profile, name, plugin);
```

Persists a plugin including an dbcp description to the users profile.

Parameters

1. Profile - `Profile`: profile instance
2. name - string: plugin name
3. plugin - `ContainerPlugin`: container plugin object
4. beforeName - string: remove previous plugin instance when it was renamed

Returns

Promise returns void

Example

```
const plugins = await Container.saveContainerPlugin(
  profile,
  'awesomeplugin',
  { ... }
);
```

4.9.9 toPlugin

```
container.toPlugin([getValues]);
```

Export current container state as plugin. If `getValues` is `true`, exports entry values as well.

This plugin can be passed to [create](#) and used to create new containers.

Parameters

1. `getValues` - boolean: export entry values or not (list entries are always excluded)

Returns

Promise returns `ContainerPlugin`: plugin build from current container

Example

```
const sampleValue = 123;
await container.setEntry('numberField', sampleValue);

console.dir(await container.toPlugin(true));
```

4.9.10 = Entries =

4.9.11 setEntry

```
container.setEntry(entryName, value);
```

Set a value for an entry.

Parameters

1. `entryName` - string: name of an entry in the container
2. `value` - any: value to set

Returns

Promise returns `void`: resolved when done

Example

```
const sampleValue = 123;
await container.setEntry('numberField', sampleValue);
console.log(await container.getEntry('numberField'));
// Output:
// 123
```

4.9.12 getEntry

```
container.getEntry(entryName);
```

Return entry from contract.

Parameters

1. entryName - string: entry name

Returns

Promise returns any: entry value

Example

Entries can be retrieved with:

```
const sampleValue = 123;
await container.setEntry('numberField', sampleValue);
console.log(await container.getEntry('numberField'));
// Output:
// 123
```

4.9.13 removeEntries

```
container.removeEntries(entries);
```

Remove multiple entries from the container, including data keys and sharings. Can also pass a single property instead of an array. Retrieves dynamically all sharings for the passed entries and runs *unshareProperties* for them.

Parameters

1. entries - string / string[]: name / list of entries, that should be removed

Returns

Promise returns void: resolved when done

Example

```
const accountId1 = '0x0000000000000000000000000000000000000000000000000000000000000001';
const accountId2 = '0x0000000000000000000000000000000000000000000000000000000000000002';

// open container with accountId1
const container = new Container(options, { ...config, accountId: accountId1 });

// assuming, that entry 'myField' has been shared with accountId2
// remove the whole property from the container
await container.removeEntries(['myField']);

// fetch value with accountId2 and with accountId1
const accountId2Container = new Container(options, { ...config, accountId: accountId2,
↪});
let value;
try {
  value = await accountId2Container.getEntry('myField');
  console.log(value);
} catch (ex) {
  console.error('could not get entry');
}

// also the owner cannot get this entry anymore
try {
  value = await container.getEntry('myField');
  console.log(value);
} catch (ex) {
  console.error('could not get entry');
}

// Output:
// could not get entry
```

4.9.14 = List Entries =

4.9.15 addListEntries

```
container.addListEntries(listName, values);
```

Add list entries to a list list property.

List entries can be added in bulk, so the value argument is an array with values. This array can be arbitrarily large **up to a certain degree**. Values are inserted on the blockchain side and adding very large arrays this way may take more gas during the contract transaction, than may fit into a single transaction. If this is the case, values can be added in chunks (multiple transactions).

Parameters

1. listName - string: name of the list in the container
2. values - any[]: values to add

Returns

Promise **returns** void: resolved when done

Example

```

const listName = 'exampleList';
console.log(await container.getListEntryCount(listName));
// Output:
// 0

const sampleValue = {
  foo: 'sample',
  bar: 123,
};
await container.addListEntries(listName, [sampleValue]);
console.log(await container.getListEntryCount(listName));
// Output:
// 1

console.dir(await container.getListEntries(listName));
// Output:
// [{
//   foo: 'sample',
//   bar: 123,
// }]

```

4.9.16 getListEntryCount

```
container.getListEntryCount(listName);
```

Return number of entries in the list. Does not try to actually fetch and decrypt values, but just returns the count.

Parameters

1. listName - string: name of a list in the container

Returns

Promise **returns** number: list entry count

Example

```

const listName = 'exampleList';
console.log(await container.getListEntryCount(listName));
// Output:
// 0

const sampleValue = {

```

(continues on next page)

(continued from previous page)

```
    foo: 'sample',
    bar: 123,
  };
await container.addListEntries(listName, [sampleValue]);
console.log(await container.getListEntryCount(listName));
// Output:
// 1

console.dir(await container.getListEntries(listName));
// Output:
// [{
//   foo: 'sample',
//   bar: 123,
// }]
```

4.9.17 getListEntries

```
container.getListEntries(contract, listName, accountId[, dfsStorage, encryptedHashes, ↵
↵count, offset, reverse]);
```

Return list entries from contract. Note, that in the current implementation, this function retrieves the entries one at a time and may take a longer time when querying large lists, so be aware of that, when you retrieve lists with many entries.

Parameters

1. listName - string: name of the list in the container
2. count - number (optional): number of elements to retrieve, defaults to 10
3. offset - number (optional): skip this many items when retrieving, defaults to 0
4. reverse - boolean (optional): retrieve items in reverse order, defaults to false

Returns

Promise returns any[]: list entries

Example

```
const listName = 'exampleList';
console.log(await container.getListEntryCount(listName));
// Output:
// 0

const sampleValue = {
  foo: 'sample',
  bar: 123,
};
await container.addListEntries(listName, [sampleValue]);
```

(continues on next page)

(continued from previous page)

```
console.log(await container.getListEntryCount(listName));  
// Output:  
// 1  
  
console.dir(await container.getListEntries(listName));  
// Output:  
// [{  
//   foo: 'sample',  
//   bar: 123,  
// }]
```

4.9.18 getListEntry

```
container.getListEntry(listName, index);
```

Return a single list entry from contract.

Parameters

1. listName - string: name of the list in the container
2. index - number: list entry id to retrieve

Returns

Promise returns any: list entry

Example

```
const listName = 'exampleList';  
console.log(await container.getListEntryCount(listName));  
// Output:  
// 0  
  
const sampleValue = {  
  foo: 'sample',  
  bar: 123,  
};  
await container.addListEntries(listName, [sampleValue]);  
const count = await container.getListEntryCount(listName);  
console.log(count);  
// Output:  
// 1  
  
console.dir(await container.getListEntry(listName, count - 1));  
// Output:  
// {  
//   foo: 'sample',  
//   bar: 123,  
// }
```

4.9.19 = Store multiple properties =

4.9.20 storeData

```
container.storeData(data);
```

Store data to a container. This allows to

- – store data into already existing entries and/or list entries
- – implicitly create new entries and/or list entries (the same logic for deciding on their type is applied as in *setEntry/addListEntries* is applied here)
- – in case of entries, their value is overwritten
- – in case of list entries, given values are added to the list

Parameters

1. data - object: object with keys, that are names of lists or entries and values, that are the values to store to them

Returns

Promise returns void: resolved when done

Example

```
const sampleValue = 123;
await container.storeData({
  'numberField': sampleValue,
});
console.log(await container.getEntry('numberField'));
// Output:
// 123
```

4.9.21 = Share Container Data =

4.9.22 shareProperties

```
container.shareProperties(shareConfigs);
```

Share entry/list to another user; this handles role permissions, role memberships.

Share configurations are given per user, that receives gets data shared with. The properties have the following meaning

- accountId:

- account, that gets properties shared
- this user will be invited to the contract as a consumer (role 1)
- read:
 - list of properties, that are shared read-only
 - for each property here, a key sharing for the user will be added if not already done so
 - this field will always be expanded by the field `type`, which is read only accessible for every member by default, even if `read` is omitted
 - if not already done so, a hash key sharing will be added for given user
- readWrite:
 - properties listed here will be threaded the same way as those in the field `read`
 - additionally the following applies:
 - * if not already done so, a role, that has `Set` permissions will be added for this field
 - * given `accountId` will be added to the group responsible for this field
 - * aforementioned roles start at role 64, the first 64 roles are system reserved for smart contract custom logic or in-detail role configurations
 - * possible roles can go up to 255, so it is possible to add up to 192 properties to a container
- removeListEntries:
 - properties listed here will be threaded the same way as those in the field `read`
 - additionally the following applies:
 - * if not already done so, a role, that has `Remove` permissions will be added for this field
 - * given `accountId` will be added to the group responsible for this field
 - * aforementioned roles start at role 64, the first 64 roles are system reserved for smart contract custom logic or in-detail role configurations
 - * possible roles can go up to 255, so it is possible to add up to 192 properties to a container

Parameters

1. `shareConfigs - ContainerShareConfig[]`: list of share configs

Returns

Promise returns void: resolved when done

Example

```
const accountId1 = '0x0000000000000000000000000000000000000000000000000000000000000001';
const accountId2 = '0x0000000000000000000000000000000000000000000000000000000000000002';

// create container with accountId1
const container = await Container.create(options, { ...config, accountId: accountId1 }
↪);
```

(continues on next page)

(continued from previous page)

```

await container.setEntry('myField', 123);
console.log(await container.getEntry('myField'));
// Output:
// 123

// share field from accountId1 to accountId2
await container.shareProperties([
  {
    accountId: accountId2,
    read: ['myField'],
  }
]);

// fetch value with accountId2
const accountId2Container = new Container(options, { ...config, accountId: accountId2,
  ↪});
console.log(await accountId2Container.getEntry('myField'));
// Output:
// 123

```

4.9.23 unshareProperties

```
container.unshareProperties(unshareConfigs);
```

Remove keys and/or permissions for a user; this also handles role permissions, role memberships.

Please note: To prevent ‘dead’ and inaccessible container entries, the API will throw an error by trying to unshare properties for an owner of a container. If you are sure and really want remove the owner from a property, you need to set the ‘force’ attribute of the *ContainerUnshareConfig* for the owner to true. If you want to remove a property for all invited users, please use the *removeEntries* function.

Parameters

1. unshareConfigs - *ContainerUnshareConfig*: list of account-field setups to remove permissions/keys for

Returns

Promise returns void: resolved when done

Example

```

const accountId1 = '0x0000000000000000000000000000000000000000000000000000000000000001';
const accountId2 = '0x0000000000000000000000000000000000000000000000000000000000000002';

// open container with accountId1
const container = new Container(options, { ...config, accountId: accountId1 });

// assuming, that entry 'myField' has been shared with accountId2
// unshare field from accountId1 to accountId2
await container.unshareProperties([
  {
    accountId: accountId2,

```

(continues on next page)

(continued from previous page)

```

    read: ['myField'],
  }]);

// fetch value with accountId2
const accountId2Container = new Container(options, { ...config, accountId: accountId2,
↪});
let value;
try {
  value = await accountId2Container.getEntry('myField');
  console.log(value);
} catch (ex) {
  console.error('could not get entry');
}
// Output:
// could not get entry

```

4.9.24 getContainerShareConfigForAccount

```
container.getContainerShareConfigForAccount(accountId);
```

Check permissions for given account and return them as ContainerShareConfig object.

Parameters

1. accountId - string: account to check permissions for

Returns

Promise returns ContainerShareConfig: resolved when done

Example

```

const accountId1 = '0x0000000000000000000000000000000000000000000000000000000000000001';
const accountId2 = '0x0000000000000000000000000000000000000000000000000000000000000002';

// create container with accountId1
const container = await Container.create(options, { ...config, accountId: accountId1,
↪});
await container.setEntry('myField', 123);
console.log(await container.getEntry('myField'));
// Output:
// 123

// share field from accountId1 to accountId2
await container.shareProperties([
  {
    accountId: accountId2,
    read: ['myField'],
  }
]);

```

(continues on next page)

(continued from previous page)

[illegible]

4.9.25 getContainerShareConfigs

```
container.getContainerShareConfigs();
```

Check permissions for given account and return them as ContainerShareConfig object.

Returns

Promise **returns** ContainerShareConfig[]: resolved when done

Example

```
const accountId1 = '0x0000000000000000000000000000000000000000000000000000000000000001'; // account in runtime
const accountId2 = '0x0000000000000000000000000000000000000000000000000000000000000002'; // account to invite

const container = await Container.create(runtime, defaultConfig);
const randomString1 = Math.floor(Math.random() * 1e12).toString(36);
await container.setEntry('testField1', randomString1);
const randomString2 = Math.floor(Math.random() * 1e12).toString(36);
await container.setEntry('testField2', randomString2);

await container.shareProperties([
  { accountId: accountId2, readWrite: ['testField1'], read: ['testField2'] },
]);

console.dir(await container.getContainerShareConfigs());
// Output:
// [ { accountId: '0x0000000000000000000000000000000000000000000000000000000000000001',
//   readWrite: [ 'testField1', 'testField2' ] },
//   { accountId: '0x0000000000000000000000000000000000000000000000000000000000000002',
//     read: [ 'testField2' ],
//     readWrite: [ 'testField1' ] } ]
```


4.9.26 setContainerShareConfigs

```
container.setContainerShareConfigs(newConfigs, originalConfigs);
```

Takes a full share configuration for an accountId (or a list of them), share newly added properties and unshare removed properties from the container. Also accepts a list or instance of the original sharing configurations so that duplicated loading can be avoided.

Parameters

1. newConfigs - *ContainerShareConfig* / *ContainerShareConfig* []: sharing configurations that should be persisted
2. originalConfigs - *ContainerShareConfig* / *ContainerShareConfig* [] (optional): pass original share configurations to check differences; better performance if provided, automatically fetched if omitted

Returns

Promise returns void: resolved when done

Example

```
const accountId1 = '0x0000000000000000000000000000000000000000000000000000000000000001';
const accountId2 = '0x0000000000000000000000000000000000000000000000000000000000000002';

// open container with accountId1
const container = new Container(options, { ...config, accountId: accountId1 });

await container.shareProperties([
  {
    accountId: '0x0000000000000000000000000000000000000000000000000000000000000002',
    read: [ 'testField', ],
    readWrite: [ 'testField2', ]
  }
]);

console.dir(await container.getContainerShareConfigForAccount(accountId2))
// {
//   accountId: '0x0030C5e7394585400B1FB193DdbCb45a37Ab916E',
//   read: [ 'testField' ],
//   readWrite: [ 'testField2' ]
// }

shareConfig.readWrite = [ 'testField3' ];
await container.setContainerShareConfigs(shareConfig);

console.dir(await container.getContainerShareConfigForAccount(accountId2))
// {
//   accountId: '0x0030C5e7394585400B1FB193DdbCb45a37Ab916E',
//   read: [ 'testField' ],
//   readWrite: [ 'testField3' ]
// }
```

4.9.27 = Verifying Containers =

4.9.28 addVerifications

```
container.addVerifications(verifications);
```

Add verifications to this container; this will also add verifications to contract description.

If the calling account is the owner of the identity of the container

- the description will be automatically updated with tags for verifications
- verifications issued with this function will be accepted automatically

See interface `ContainerVerificationEntry` for input data format.

Parameters

1. `verifications - ContainerVerificationEntry[]`: list of verifications to add

Returns

Promise returns void: resolved when done

Example

```
await container.addVerifications([{ topic: 'exampleVerification' }]);
```

4.9.29 getOwner

```
container.getOwner();
```

Gets the owner account id for the container.

Returns

Promise returns string: owner account id

Example

```
const isOwner = (await container.getOwner()) === runtime.activeAccount;
```

4.9.30 getVerifications

```
container.getVerifications();
```

Gets verifications from description and fetches list of verifications for each of them.

See [Verifications](#) documentation for details on output data format.

Returns

Promise returns any: list of verification lists from [Verifications](#), `getVerifications`

Example

```
await container.addVerifications([ { topic: 'exampleVerification' } ]);
const verifications = await container.getVerifications();
```

4.9.31 = Working with Container Descriptions =

4.9.32 getDescription

```
container.getDescription();
```

Get description from container contract.

Returns

Promise returns any: public part of the description

Example

```
const description = await container.getDescription();
console.dir(description);
// Output:
// { name: 'test container',
//   description: 'container from test run',
//   author: 'evan GmbH',
//   version: '0.1.0',
//   dbcpVersion: 2,
//   identity:
//     '0x70c969a64e880fc904110ce9ab72ba5f95f706a252ac085ae0525bd7a284337c',
//   dataSchema: { type: { type: 'string', '$id': 'type_schema' } } }
```

4.9.33 setDescription

```
container.setDescription(description);
```

Write given description to containers DBCP.

Parameters

1. `description` - any: description (public part)

Returns

Promise returns void: resolved when done

Example

```
// get current description
const description = await container.getDescription();
console.dir(description);
// Output:
// { name: 'test container',
//   description: 'container from test run',
//   author: 'evan GmbH',
//   version: '0.1.0',
//   dbcpVersion: 2,
//   identity:
//     '0x70c969a64e880fc904110ce9ab72ba5f95f706a252ac085ae0525bd7a284337c',
//   dataSchema: { type: { type: 'string', '$id': 'type_schema' } } }

// update description
description.version = '0.1.1';
await container.setDescription(description);

// fetch again
console.dir(await container.getDescription());
// Output:
// { name: 'test container',
//   description: 'container from test run',
//   author: 'evan GmbH',
//   version: '0.1.1',
//   dbcpVersion: 2,
//   identity:
//     '0x70c969a64e880fc904110ce9ab72ba5f95f706a252ac085ae0525bd7a284337c',
//   dataSchema: { type: { type: 'string', '$id': 'type_schema' } } }
```

4.9.34 = Utilities =

4.9.35 getContractAddress

```
container.getContractAddress();
```

Get contract address of underlying DataContract.

Returns

Promise returns string: address of the DataContract

ContainerShareConfig

config for sharing multiple fields to one account (read and/or readWrite access)

1. `accountId - string`: account, that gets properties shared
2. `read - string[]` (optional): list of properties, that are shared read-only
3. `readWrite - string[]` (optional): list of properties, that are shared readable and writable

ContainerUnshareConfig

config for unsharing multiple fields from one account (write and/or readWrite access)

1. `accountId - string`: account, that gets properties unshared
2. `readWrite - string[]` (optional): list of properties, that are unshared (read and write permissions)
3. `removeListEntries - string[]` (optional): list of properties, that are losing the rights to remove list-entries
4. `write - string[]` (optional): list of properties, for which write permissions should be removed
5. `force - boolean` (optional): Without force flag, removal of the owner will throw an error. By setting to true, force will even remove the owner. **Important: By removing the owner from a property, the encryptions keys get lost and cannot be recovered. As the result of this, the data isn't readable anymore and must be overwritten by creating new encryption keys to encrypt future content.**

ContainerPlugin

base definition of a container instance, covers properties setup and permissions

1. `description - any`: type of the template (equals name of the template)
2. `template - ContainerTemplate`: template for container instances, covers properties setup and permissions

ContainerTemplate

template for container instances, covers properties setup and permissions

1. `type - string`: type of the template (equals name of the template)
2. `properties - { [id: string]: ContainerTemplateProperty; }` (optional): list of properties included in this template, key is field name, value is property setup

ContainerTemplateProperty

config for sharing multiple fields to one account (read and/or readWrite access)

1. `dataSchema - any`: Ajv data schema for field
2. `permissions - { [id: number]: string[] }`: permissions for this template, key is role id, value is array with 'set' and/or 'remove'
3. `type - string`: type of property (entry/list)
4. `value - any` (optional): value of property

ContainerVerificationEntry

data for verifications for containers

1. `topic - string`: verification path
 2. `descriptionDomain - string` (optional): domain, where the description of this verification is stored
 3. `disableSubverifications - boolean` (optional): if set, verification created in a sub-path are invalid by default, defaults to false
 4. `expirationDate - number` (optional): expiration date, verifications do not expire if omitted, defaults to 0
 5. `verificationValue - string` (optional): reference to additional verification details
-

4.9.39 Public Properties

defaultDescription (static)

Default description used when no specific description is given to *.create*.

defaultSchemas (static)

Predefined simple schemas, contains basic schemas for files, number, object, string entries and their list variants.

defaultTemplate (static)

Default template used when no specific description is given to *.create*. Default template is `metadata`.

profileTemplatesKey (static)

Key that is used in user profile to store templates, default is `templates.datacontainer.digitaltwin.evan`

templates (static)

Predefined templates for containers, currently only contains the `metadata` template.

This section includes modules, that deal with smart contract interactions, which includes:

- contract helper libraries, e.g. for
 - `BaseContract`
 - `DataContract`
 - `ServiceContract`
- permission management
- sharing keys for contracts

DFS (Distributed File System)

5.1 DFS Interface

Interface Name	DfsInterface
Source	dfs-interface.ts

The [DfsInterface](#) is used to add or get files from the distributed file system. It is the only class, that has to be used before having access to a runtime, when using the *createDefaultRuntime*.

Internally the modules use the [DfsInterface](#) to access data as well. As the actual implementation of the file access may vary, an instance of the interface has to be created beforehand and passed to the *createDefaultRuntime* function. An implementation called [Ipfs](#), that relies on the [IPFS framework](#) is included as in the package.

5.1.1 add

```
dfs.add(name, data);
```

add content to dfs file content is converted to Buffer (in NodeJS) or an equivalent “polyfill” (in browsers)

Parameters

1. `name - string`: name of the added file
2. `data - buffer`: data (as buffer) of the added file

Returns

`string`: hash of the data.

Example

```
const fileHash = await runtime.dfs.add(
  'about-maika-1.txt',
  Buffer.from('we have a cat called "Maika"', 'utf-8'),
);
console.log(fileHash);
// Output:
// 0x695adc2137f1f069ff697aa287d0eae486521925a23482f180b3ae4e6dbf8d70
```

5.1.2 addMultiple

```
dfs.addMultiple(files);
```

Multiple files can be added at once. This way of adding should be preferred for performance reasons, when adding files, as requests are combined.

Parameters

1. files - FileToAdd[]: array with files to add

Returns

Promise resolves to string[]: hash array of the data.

Example

```
const fileHashes = await runtime.dfs.addMultiple([
  {
    path: 'about-maika-1.txt',
    content: Buffer.from('we have a cat called "Maika"', 'utf-8'),
  }, {
    path: 'about-maika-2.txt',
    content: Buffer.from('she can be grumpy from time to time"', 'utf-8'),
  }
]);
console.dir(fileHashes);
// Output:
// [ '0x695adc2137f1f069ff697aa287d0eae486521925a23482f180b3ae4e6dbf8d70',
//   '0x6b85c8b24b59b12a630141143c05bbf40a8adc56a8753af4aa41ebacf108b2e7' ]
```

5.1.3 get

```
dfs.get(hash, returnBuffer);
```

get data from dfs by hash

Parameters

1. `hash - string`: hash (or bytes32 encoded) of the data
2. `returnBuffer - bool`: should the function return the plain buffer, defaults to `false`

Returns

Promise resolves to `string` | `buffer`: data as text or buffer.

Example

```
const fileBuffer = await runtime.dfs.get(
  ↪ '0x695adc2137f1f069ff697aa287d0eae486521925a23482f180b3ae4e6dbf8d70');
console.log(fileBuffer.toString('utf-8'));
// Output:
// we have a cat called "Maika"
```

5.1.4 remove

```
dfs.remove(hash);
```

removes data by hash reference

Parameters

1. `hash - string`: hash (or bytes32 encoded) of the data

Returns

Promise resolves to `void`: resolved when done

Example

```
const fileBuffer = await runtime.dfs.remove(
  ↪ '0x695adc2137f1f069ff697aa287d0eae486521925a23482f180b3ae4e6dbf8d70');
```

5.1.5 = Additional Components =

5.1.6 Interfaces

FileToAdd

1. `path - string`: name of the added file
2. `content - buffer`: data (as buffer) of the added file

5.2 IPFS

Class Name	Ipfs
Implements	DfsInterface
Extends	Logger
Source	ipfs.ts
Examples	ipfs.spec.ts

This is `DfsInterface` implementation, that relies on the `IPFS` framework.

5.2.1 constructor

```
new Ipfs(options);
```

Creates a new IPFS instance.

Parameters

1. **options - IpfsOptions:** options for IPFS constructor.

- `remoteNode` - any: ipfs-api instance to remote server
- `cache` - `DfsCacheInterface` (optional): `DfsCacheInterface` instance
- `log` - Function (optional): function to use for logging: `(message, level) => {...}`
- `logLevel` - `LogLevel` (optional): messages with this level will be logged with `log`
- `logLog` - `LogLogInterface` (optional): container for collecting log messages
- `logLogLevel` - `LogLevel` (optional): messages with this level will be pushed to `logLog`

Returns

Ipfs instance

Example

```
const ipfs = new Ipfs({
  remoteNode,
  cache
});
```

5.2.2 ipfsHashToBytes32

```
dfs.ipfsHashToBytes32(hash);
```

convert IPFS hash to bytes 32 see https://www.reddit.com/r/ethdev/comments/6lbnhy/a_practical_guide_to_cheap_ipfs_hash_storage_in

Parameters

1. hash - string: IPFS hash

Returns

string: bytes32 string.

Example

```
runtime.dfs.ipfsHashToBytes32('QmWmyoMocfbAaiEs2G46gpeUmhqFRDW6KWo64y5r581Vz')
// returns 0x7D5A99F603F231D53A4F39D1521F98D2E8BB279CF29BEBFD0687DC98458E7F89
```

5.2.3 bytes32ToIpfsHash

```
dfs.bytes32ToIpfsHash(str);
```

convert bytes32 to IPFS hash see https://www.reddit.com/r/ethdev/comments/6lbnhy/a_practical_guide_to_cheap_ipfs_hash_storage_in

Parameters

1. str - string: bytes32 string

Returns

string: IPFS Hash.

Example

```
runtime.dfs.ipfsHashToBytes32(
  ↪ '0x7D5A99F603F231D53A4F39D1521F98D2E8BB279CF29BEBFD0687DC98458E7F8')
// returns QmWmyoMocfbAaiEs2G46gpeUmhqFRDW6KWo64y5r581Vz
```

5.2.4 add

```
dfs.add(name, data);
```

add content to ipfs file content is converted to Buffer (in NodeJS) or an equivalent “polyfill” (in browsers)

Parameters

1. name - string: name of the added file
2. data - buffer: data (as buffer) of the added file

Returns

string: ipfs hash of the data.

Example

```
const fileHash = await runtime.dfs.add(
  'about-maika-1.txt',
  Buffer.from('we have a cat called "Maika"', 'utf-8'),
);
console.log(fileHash);
// Output:
// 0x695adc2137f1f069ff697aa287d0eae486521925a23482f180b3ae4e6dbf8d70
```

5.2.5 addMultiple

```
dfs.addMultiple(files);
```

Multiple files can be added at once. This way of adding should be preferred for performance reasons, when adding files, as requests are combined.

Parameters

1. files - FileToAdd[]: array with files to add

Returns

Promise resolves to string[]: ipfs hash array of the data.

Example

```
const fileHashes = await runtime.dfs.addMultiple([
  {
    path: 'about-maika-1.txt',
    content: Buffer.from('we have a cat called "Maika"', 'utf-8'),
  }, {
    path: 'about-maika-2.txt',
    content: Buffer.from('she can be grumpy from time to time"', 'utf-8'),
  }
]);
console.dir(fileHashes);
// Output:
// [ '0x695adc2137f1f069ff697aa287d0eae486521925a23482f180b3ae4e6dbf8d70',
//   '0x6b85c8b24b59b12a630141143c05bbf40a8adc56a8753af4aa41ebacf108b2e7' ]
```

5.2.6 pinFileHash

```
dfs.pinFileHash(hash);
```

pins file hashes on ipfs cluster

Parameters

1. hash - string: filehash of the pinned item

Returns

Promise resolves to void: resolved when done.

Example

```
const fileBuffer = await runtime.dfs.pinFileHash(
  ↪ 'QmWmyoMocTfbAaiEs2G46gpeUmhqFRDW6KW64y5r581Vz');
```

5.2.7 get

```
dfs.get(hash, returnBuffer);
```

get data from ipfs by ipfs hash

Parameters

1. hash - string: ipfs hash (or bytes32 encoded) of the data
2. returnBuffer - bool: should the function return the plain buffer, defaults to false

Returns

Promise resolves to string | buffer: data as text or buffer.

Example

```
const fileBuffer = await runtime.dfs.get(
  ↪ '0x695adc2137f1f069ff697aa287d0eae486521925a23482f180b3ae4e6dbf8d70');
console.log(fileBuffer.toString('utf-8'));
// Output:
// we have a cat called "Maika"
```

5.2.8 = Additional Components =

5.2.9 Interfaces

FileToAdd

1. `path - string`: name of the added file
2. `content - buffer`: data (as buffer) of the added file

5.3 IPLD

Class Name	<code>Ipld</code>
Extends	<code>Logger</code>
Source	<code>ipld.ts</code>
Examples	<code>ipld.spec.ts</code>

`IPLD` is a way to store data as trees. The used implementation relies on `js-ipld-graph-builder` for iterating over tree nodes and setting new subtrees, but uses a few modifications to the standard: - nodes are not stored as `IPFS DAGs`, but stored as play JSON IPFS files - nodes, that are encrypted, contain the property `cryptoInfo` for decryption (see Encryption)

5.3.1 constructor

```
new IPLD(options);
```

Creates a new `Ipld` instance.

Requires

Parameters

1. **options - `IpldOptions`**: The options used for calling

- `cryptoProvider` - `CryptoProvider`: `CryptoProvider` instance
- `defaultCryptoAlgo` - `string`: default encryption algorithm
- `ipfs` - `Ipfs`: `Ipfs` instance
- `keyProvider` - `KeyProviderInterface`: `KeyProviderInterface` instance
- `originator` - `string`: originator of tree (default encryption context)
- `nameResolver` - `NameResolver`: `NameResolver` instance
- `log` - `Function` (optional): function to use for logging: `(message, level) => {...}`
- `logLevel` - `LogLevel` (optional): messages with this level will be logged with `log`
- `logLog` - `LogLogInterface` (optional): container for collecting log messages
- `logLogLevel` - `LogLevel` (optional): messages with this level will be pushed to `logLog`

Returns

IpLdOptions instance

Example

```
const ipld = new IPLD(options);
```

5.3.2 store

```
ipld.store(toSet);
```

Store tree, if tree contains merklefied links, stores tree with multiple linked subtrees. Hashes returned from this function represent the the final tree, that can be stored as bytes32 hashes in smart contracts, etc.

Parameters

1. toSet - any: tree to store

Returns

Promise returns string: hash reference to a tree with with merklefied links

Example

```
const sampleObject = {
  personalInfo: {
    firstName: 'eris',
  },
};
const stored = await ipld.store(Object.assign({}, sampleObject));
console.log(stored);
// Output:
// 0x12f6526dbe223eddd6c6a0fb7df118c87c56d34bf0c845b54bdca2fec0f3017d
```

When storing nested trees created with `_ipld_set_`, subtrees at junction points are stored as separate trees, then converted to serialized buffers, which are automatically deserialized and cast back when calling `ipld_getLinkedGraph`.

```
console.log(JSON.stringify(extended, null, 2));
const extendedstored = await ipld.store(Object.assign({}, extended));
// Output:
// "0xc74f6946aacbbd1418ddd7dec83a5bcd3710b384de767d529e624f9f08cbf9b4"
const loaded = await ipld.getLinkedGraph(extendedstored, '');
console.log(JSON.stringify(IpLd.purgeCryptoInfo(loaded), null, 2));
// Output:
//
// "personalInfo": {
//   "firstName": "eris"
```

(continues on next page)

(continued from previous page)

```
// },
// "dapps": {
//   "/": {
//     "type": "Buffer",
//     "data": [ 18, 32, 246, 21, 166, 135, 236, 212, 70, 130, 94, 47, 81, 135, 153,
// ↪154, 201, 69, 109, 249, 97, 84, 252, 56, 214, 195, 149, 133, 116, 253, 19, 87, 217,
// ↪66 ]
//   }
// }
//
```

5.3.3 getLinkedGraph

```
ipld.getLinkedGraph(graphReference[, path]);
```

Get a path from a tree; resolve subtrees only if required (depends on requested path).

Parameters

1. graphReference - string | Buffer | any: hash/buffer to look up or a graph object
2. path - string (optional): path in the tree, defaults to ''

Returns

Promise returns any: linked graph

Example

To retrieve data from IPLD trees, use the *bytes32* hash from storing the data:

```
const stored = '0x12f6526dbe223eddd6c6a0fb7df118c87c56d34bf0c845b54bdca2fec0f3017d';
const loaded = await ipld.getLinkedGraph(stored, '');
console.dir(Ipld.purgeCryptoInfo(loaded));
// Output:
// { personalInfo: { firstName: 'eris' } }
```

For info about the `Ipld.purgeCryptoInfo` part see Encryption.

The second argument is the path inside the tree. Passing '' means "retrieve data from root level". To get more specific data, provide a path:

```
const stored = '0x12f6526dbe223eddd6c6a0fb7df118c87c56d34bf0c845b54bdca2fec0f3017d';
const loaded = await ipld.getLinkedGraph(stored, 'personalInfo');
console.dir(Ipld.purgeCryptoInfo(loaded));
// Output:
// { firstName: 'eris' }
```

```
const stored = '0x12f6526dbe223eddd6c6a0fb7df118c87c56d34bf0c845b54bdca2fec0f3017d';
const loaded = await ipld.getLinkedGraph(stored, 'personalInfo/firstName');
console.dir(Ipld.purgeCryptoInfo(loaded));
// Output:
// 'eris'
```

5.3.4 getResolvedGraph

```
ipld.getResolvedGraph(graphReference[, path, depth]);
```

Get a path from a tree; resolve links in paths up to depth (default is 10).

This function is for **debugging and analysis purposes only**, it tries to resolve the entire graph, which would be too much requests in most scenarios. If resolving graphs, prefer using *ipld_getLinkedGraph*, with specific queries into the tree, that limit the resolve requests.

Parameters

1. graphReference - string | Buffer | any: hash/buffer to look up or a graph object
2. path - string (optional): path in the tree, defaults to ''
3. depth - number (optional): resolve up to this many levels of depth, defaults to 10

Returns

Promise returns any: resolved graph

Example

```
const treeHash = '0xc74f6946aacbbd1418ddd7dec83a5bcd3710b384de767d529e624f9f08cbf9b4';
console.dir(await ipld.getResolvedGraph(treeHash, ''));
// Output:
// { personalInfo: { firstName: 'eris' },
//   dapps: { '/': { contracts: [Array], cryptoInfo: [Object] } },
//   cryptoInfo:
//     { originator: '0xd7c759941fa3962e4833707f2f44f8cb11b471916fb6f9f0facb03119628234e
//     ↪',
//       keyLength: 256,
//       algorithm: 'aes-256-cbc' } }
```

Compared to *ipld_getLinkedGraph*:

```
const treeHash = '0xc74f6946aacbbd1418ddd7dec83a5bcd3710b384de767d529e624f9f08cbf9b4';
console.dir(await ipld.getLinkGraph(treeHash, ''));
// Output:
// { personalInfo: { firstName: 'eris' },
//   dapps:
//     { '/':
//       Buffer [18, 32, 246, 21, 166, 135, 236, 212, 70, 130, 94, 47, 81, 135, 153,
//       ↪154, 201, 69, 109, 249, 97, 84, 252, 56, 214, 195, 149, 133, 116, 253, 19, 87, 217,
//       ↪66] },
```

(continues on next page)

(continued from previous page)

```
// cryptoInfo:
// { originator: '0xd7c759941fa3962e4833707f2f44f8cb11b471916fb6f9f0facb03119628234e
→',
//   keyLength: 256,
//   algorithm: 'aes-256-cbc' } }
```

5.3.5 set

```
ipld.set(tree, path, subtree[, plainObject, cryptoInfo]);
```

Set a value to a tree node; inserts new element as a linked subtree by default.

What's pretty useful about IPLD graphs is, that not only plain JSON trees can be stored, but that those trees can be linked to other graphs, which makes it possible to build very powerful tree structures, that consist of multiple separate trees, that can be used on their own or in a tree, that combines all of those. The resulting hash is again `bytes32` hash and this can be stored in smart contracts like any other IPFS hash.

This function adds the given subtree under a path in the existing tree. Different subtrees can be added by using this function multiple times. The final tree can then be stored to IPFS with *ipld_store*.

Parameters

1. `tree` - any: tree to extend
2. `path` - string: path of inserted element
3. `subtree` - any: element that will be added
4. `plainObject` - boolean (optional): do not link values as new subtree, defaults to `false`
5. `cryptoInfo` - `CryptoInfo` (optional): crypto info for encrypting subtree

Returns

Promise returns any: tree with merklefied links

Example

```
const sampleObject = {
  personalInfo: {
    firstName: 'eris',
  },
};
const sub = {
  contracts: ['0x01', '0x02', '0x03']
};
const extended = await ipld.set(
  sampleObject,           // extend this graph
  'dapps',                // attach the subgraph under the path "dapps"
  sub,                    // attach this graph as a subgraph
);
```

(continues on next page)

(continued from previous page)

```

console.log(JSON.stringify(extended, null, 2));
// Output:
// {
//   "personalInfo": {
//     "firstName": "eris"
//   },
//   "dapps": {
//     "/": {
//       "contracts": [
//         "0x01",
//         "0x02",
//         "0x03"
//       ]
//     }
//   }
// }

```

5.3.6 remove

```
ipld.remove(tree, path);
```

Delete a value from a tree node.

Parameters

1. tree - any: tree to extend
2. string - string: path of inserted element

Returns

Promise returns any: tree with merklefied links

Example

```

const treeHash = '0xc74f6946aacbbd1418ddd7dec83a5bcd3710b384de767d529e624f9f08cbf9b4';
const loaded = await ipld.getLinkedGraph(treeHash, '');
console.log(loaded);
// Output:
// { personalInfo: { firstName: 'eris' },
//   dapps:
//     { '/': <Buffer 12 20 f6 15 a6 87 ec d4 46 82 5e 2f 51 87 99 9a c9 45 6d f9 61
→ 54 fc 38 d6 c3 95 85 74 fd 13 57 d9 42> },
//   cryptoInfo:
//     { originator:
→ '0xd7c759941fa3962e4833707f2f44f8cb11b471916fb6f9f0facb03119628234e',
//       keyLength: 256,
//       algorithm: 'aes-256-cbc' } }

```

(continues on next page)

(continued from previous page)

```
const updated = await ipld.remove(loaded, 'dapps');
console.log(updated);
// Output:
// { personalInfo: { firstName: 'eris' },
//   cryptoInfo:
//     { originator:
//       ↪ '0xd7c759941fa3962e4833707f2f44f8cb11b471916fb6f9f0facb03119628234e',
//       keyLength: 256,
//       algorithm: 'aes-256-cbc' } } }
```

5.3.7 purgeCryptoInfo

```
Ipld.purgeCryptoInfo(toPurge);
```

(static class function)

Remove all cryptoInfos from tree.

Some example here use `Ipld.purgeCryptoInfo` to cleanup the objects before logging them. This is done, because IPLD graphs are encrypted by default, which has a few impact on the data stored:

- The root node of a tree is “encrypted” with the encryption algorithm “unencrypted”, resulting in the root node having its data stored as a Buffer. This is done to keep the root node in the same format as the other nodes, as:
- Nodes in the Tree are encrypted. This encryption is specified in the constructor as *defaultCryptoAlgo*.
- All nodes are en- or decrypted with the same account or “originator”. The originator, that is used, is specified in the constructor as “originator”. This means, that the IPLD instance is account bound and a new instance has to be created if another account should be used.

Parameters

1. toPurge - any: The options used for calling

Returns

void

Example

To show the difference, without purging:

```
const stored = '0x12f6526dbe223eddd6c6a0fb7df118c87c56d34bf0c845b54bdca2fec0f3017d';
const loaded = await ipld.getLinkedGraph(stored, '');
console.dir(loaded);
// Output:
// { personalInfo: { firstName: 'eris' },
//   cryptoInfo:
//     { originator:
//       ↪ '0xd7c759941fa3962e4833707f2f44f8cb11b471916fb6f9f0facb03119628234e',
//       keyLength: 256,
```

(continues on next page)

(continued from previous page)

```
//      algorithm: 'aes-256-cbc' } }  
//
```

With purging:

```
const stored = '0x12f6526dbe223eddd6c6a0fb7df118c87c56d34bf0c845b54bdca2fec0f3017d';  
const loaded = await ipld.getLinkedGraph(stored, '');  
console.dir(Ipld.purgeCryptoInfo(loaded));  
// Output:  
// { personalInfo: { firstName: 'eris' } }
```

The DFS section handles modules, that deal with managing data in distributed file system, like the DFS interface, its implementation for IPFS and the IPLD graph builder, that works on top of the IPFS impementation.

6.1 Encryption Wrapper

Class Name	EncryptionWrapper
Extends	Logger
Source	encryption-wrapper.ts
Examples	encryption-wrapper.spec.ts

Encryption processes often deal with the following questions:

- Where to get a key from?
- Do I need to generate a new key? How can I store this?
- How to fetch a matching crypto?
- How to encrypt data?

`EncryptionWrapper` handles these topics and offers a fast way to work with encryption.

6.1.1 constructor

```
new EncryptionWrapper(options);
```

Create new `EncryptionWrapper` instance.

Parameters

1. **options - `DigitalTwinOptions`: runtime-like object with required modules**
 - `cryptoProvider - CryptoProvider: CryptoProvider instance`

- `nameResolver` - `NameResolver`: `NameResolver` instance
- `profile` - `Profile`: `Profile` instance
- `sharing` - `Sharing`: `Sharing` instance
- `web3` - `Web3`: `Web3` instance
- `log` - `Function` (optional): function to use for logging: `(message, level) => {...}`
- `logLevel` - `LogLevel` (optional): messages with this level will be logged with `log`
- `logLog` - `LogLogInterface` (optional): container for collecting log messages
- `logLogLevel` - `LogLevel` (optional): messages with this level will be pushed to `logLog` factory

Returns

`EncryptionWrapper` instance

Example

```
const encryptionWrapper = new EncryptionWrapper(runtime);
```

6.1.2 = CryptoInfos =

6.1.3 getCryptoInfo

```
encryptionWrapper.getCryptoInfo(keyContext, keyType, cryptorType[, artifacts])
```

`CryptoInfo`s are descriptors for encrypted documents. They are stored alongside the encrypted data and provide a hint about the context of the data and how to decrypt them. In the `EncryptionWrapper` they are also used as an option parameter for the `encrypt` function.

Parameters

1. `keyContext` - any: used to identify key, can be any string (but must not have colons)
2. `keyType` - `EncryptionWrapperKeyType`: defines where keys are stored
3. `cryptorType` - `EncryptionWrapperCryptorType`: cryptor to use
4. `artifacts` - any: (optional) additional information for encryption may be required, depends on `keyType`, see section below for details

artifacts

Depending on `keyType` different properties are required. `artifacts` can be omitted, if used `keyType` is not listed below.

- `EncryptionWrapperKeyType.Sharing`:
 - `sharingContractId` - string: contract address of `Shared` or `MultiShared` contract

- `sharingId` - string (optional): id in a `MultiShared` contract and only used for them, defaults to `null`

Returns

Promise returns `CryptoInfo`: crypto info built out of input arguments

Example

```
const keyContext = 'my key 15';
const cryptoInfo = await encryptionWrapper.getCryptoInfo(
  keyContext,
  EncryptionWrapperKeyType.Profile,
  EncryptionWrapperCryptorType.Content,
);
console.dir(cryptoInfo);
// Output:
// { algorithm: 'aes-256-cbc',
//   block: 198543,
//   originator: 'profile:my key 15' }
```

6.1.4 = Key Handling =

6.1.5 generateKey

```
encryptionWrapper.generateKey(cryptoInfo);
```

Generates a new encryption key. Crypto algorithm in `cryptoInfo` is used to decide on which `Cryptor` to pick for this.

Parameters

1. `cryptoInfo` - `CryptoInfo`: details for encryption, can be created with *`getCryptoInfo`*

Returns

Promise returns any: key to encrypt/decrypt data

Example

```
const key = await encryptionWrapper.generateKey(cryptoInfo);
console.dir(key);
// Output:
// 'd387d41011a2f04f18930e982ad30c537d29bc12588164cb978d0f70a5d11b3f'
```

6.1.6 storeKey

```
encryptionWrapper.storeKey(cryptoInf[, artifacts]);
```

Store key in respective storage location, depending on given `cryptoInfo`, additional information may be required, which can be given via `artifacts`.

Parameters

1. `cryptoInfo` - `CryptoInfo`: details for encryption, can be created with *getCryptoInfo*
2. `key` - any: key to store
3. `artifacts` - any: (optional) additional information for encryption may be required, depends on `cryptoInfo.originator`, see section below for details

artifacts

Depending on `cryptoInfo.originator` different properties are required. `artifacts` can be omitted, if used `cryptoInfo.originator` schema is not listed below. Note, that `cryptoInfo.originator` schema depends on with which `EncryptionWrapperKeyType` *getCryptoInfo* was called.

- **sharing:.*:**
 - `accountId` - string: `accountId`, that is used to share keys from, executes the internal transaction
 - `receiver` - string (optional): `accountId`, that receives the key, defaults to `accountId`

Returns

Promise returns void: resolved when done

Example

```
const key = await encryptionWrapper.generateKey(cryptoInfo);
await encryptionWrapper.storeKey(cryptoInfo, key);
```

6.1.7 getKey

```
encryptionWrapper.getKey(cryptoInf[, artifacts]);
```

Get key for given `cryptoInfo`. Can when storing keys in custom storage locations.

Parameters

1. `cryptoInfo` - `CryptoInfo`: details for encryption, can be created with *getCryptoInfo*
2. `artifacts` - any: (optional) additional information for encryption may be required, depends on `cryptoInfo.originator`, see section below for details

artifacts

Depending on `cryptoInfo.originator` different properties are required. `artifacts` can be omitted, if used `cryptoInfo.originator` schema is not listed below. Note, that `cryptoInfo.originator` schema depends on with which `EncryptionWrapperKeyType` *getCryptoInfo* was called.

- `sharing:.*:`
 - `accountId` - string: `accountId`, that accesses data, is used to get shared keys with
 - `propertyName` - string (optional): property, that is decrypted, defaults to `'*'`
- `custom:.*:`
 - `key` - string: `accountId`, that accesses data, is used to get shared keys with

Returns

Promise `returns void`: resolved when done

Example

```
const keyContext = 'my key 15';
const cryptoInfo = await encryptionWrapper.getCryptoInfo(
  keyContext,
  EncryptionWrapperKeyType.Profile,
  EncryptionWrapperCryptorType.Content,
);
console.dir(await encryptionWrapper.getKey(cryptoInfo));
// Output:
// '08bca9594ebaa7812f030f299fa30b51c5a7c3e7b2b66cd0a18c5cf46314aab7'
```

6.1.8 = Encryption =

6.1.9 encrypt

```
encryptionHandler.encrypt(toEncrypt, cryptoInfo[, artifacts]);
```

Encrypt given object, depending on given `cryptoInfo`, additional information may be required, which can be given via `artifacts`

Parameters

1. `toEncrypt` - any: object to encrypt
2. `cryptoInfo` - `CryptoInfo`: details for encryption, can be created with `getCryptoInfos`
3. `artifacts` - any: (optional) additional information for encryption may be required, depends on `cryptoInfo.originator`, see section below for details

artifacts

Depending on `cryptoInfo.originator` different properties are required. artifacts can be omitted, if used `cryptoInfo.originator` schema is not listed below. Note, that `cryptoInfo.originator` schema depends on with which `EncryptionWrapperKeyType` *getCryptoInfo* was called.

- `sharing:.*:`
 - `accountId` - string: `accountId`, that accesses data, is used to get shared keys with
 - `propertyName` - string (optional): property, that is encrypted, defaults to `'*'`
- `custom:.*:`
 - `key` - string: `accountId`, that accesses data, is used to get shared keys with

Returns

Promise returns Envelope: envelope with encrypted data

Example

```
const sampleData = {
  foo: TestUtils.getRandomBytes32(),
  bar: Math.random(),
};
const keyContext = 'my key 15';
const cryptoInfo = await encryptionWrapper.getCryptoInfo(
  keyContext,
  EncryptionWrapperKeyType.Profile,
  EncryptionWrapperCryptorType.Content,
);
const encrypted = await encryptionWrapper.encrypt(sampleData, cryptoInfo);
// Output:
// { private:
//
// ↪ 'ec6a2e0401e6270c50a88db31d0a22b677516162925a87bb7ec11a80613275817b883e75ee4bc8f82fe681d3462cf8ad4
// ↪ ',
//   cryptoInfo:
//     { algorithm: 'aes-256-cbc',
//       block: 198573,
//       originator:
//         'profile:0xb1c492ee6085679497c73008100c3b3136a75a8519c2a0016fec686a05f1c7f0' }
// ↪ }
```

6.1.10 decrypt

```
encryptionHandler.decrypt(toDecrypt[, artifacts]);
```

Decrypt given Envelope.

Parameters

1. `toDecrypt` - any: encrypted envelop
2. `artifacts` - any: (optional) additional information for decrypting

artifacts

Depending on `cryptoInfo.originator` different properties are required. `artifacts` can be omitted, if used `cryptoInfo.originator` schema is not listed below. Note, that `cryptoInfo.originator` schema depends on with which `EncryptionWrapperKeyType` *[getCryptoInfo](#)* was called.

- `sharing:.*:`
 - `accountId` - string: `accountId`, that accesses data, is used to get shared keys with
 - `propertyName` - string (optional): property, that is decrypted, defaults to `'*'`
- `custom:.*:`
 - `key` - string: `accountId`, that accesses data, is used to get shared keys with

Returns

Promise returns `Envelope`: envelope with encrypted data

Example

```
const sampleData = {
  foo: TestUtils.getRandomBytes32(),
  bar: Math.random(),
};
const keyContext = 'my key 15';
const cryptoInfo = await encryptionWrapper.getCryptoInfo(
  keyContext,
  EncryptionWrapperKeyType.Profile,
  EncryptionWrapperCryptorType.Content,
);
const encrypted = await encryptionWrapper.encrypt(sampleData, cryptoInfo);
const decrypted = await encryptionWrapper.decrypt(encrypted);
console.dir(decrypted);
// Output:
// { foo:
//   '0x746dccef8a185d9e34a2778af51e8ee7e513e4035f7a5e2c2d122904a21f32e6',
//   bar: 0.618861426409717 }
```

6.1.11 Additional Components

6.1.12 Enums

DigitalTwinEntryType

- `Content`: content encryption is used for generic data (strings, in memory objects)

- **File:** file encryption is used for binary file data
- **Unencrypted:** unencrypted data encryption can be used to embed unencrypted data in encryption containers

EncryptionWrapperKeyType

- **Custom:** custom key handling means that the key is handled elsewhere and has to be given to profile
- **Profile:** key is stored in profile, usually in property “encryptionKeys”
- **Sharing:** key is stored in Shared or MultiShared contract

6.2 Key Provider

Class Name	KeyProvider
Implements	KeyProviderInterface
Extends	Logger
Source	key-provider.ts

The [KeyProvider](#) returns given decryption/encryption keys for a given CryptoInfo. They use a given `evan.network` profile to retrieve the needed keys to encrypt/decrypt the envelope

6.2.1 constructor

```
new KeyProvider(options);
```

Creates a new KeyProvider instance.

Parameters

1. **options - KeyProviderOptions:** options for KeyProvider constructor.

- **keys** - any (optional): object with key mappings of accounts
- **log** - Function (optional): function to use for logging: `(message, level) => {...}`
- **logLevel** - [LogLevel](#) (optional): messages with this level will be logged with log
- **logLog** - [LogLogInterface](#) (optional): container for collecting log messages
- **logLogLevel** - [LogLevel](#) (optional): messages with this level will be pushed to logLog

Returns

KeyProvider instance

Example


```
const keyProvider = new KeyProvider({
  keys: {
    '0x123': 'abcdeF9043'
  }
});
```

6.2.2 init

```
keyProvider.init(_profile);
```

initialize a new KeyProvider with a given `evan.network` Profile

Parameters

1. `_profile` - Profile: initialized `evan.network` profile

Example

```
runtime.keyProvider.init(runtime.profile);
```

6.2.3 getKey

```
keyProvider.getKey(info);
```

get a encryption/decryption key for a specific `CryptoInfo` from the associated `AccountStore` or the loaded `evan.network` profile

Parameters

1. `cryptoAlgo` - string: crypto algorithm

Returns

Promise resolves to `string`: the found key for the `cryptoinfo`.

Example

```
const cryptoInfo = {
  "public": {
    "name": "envelope example"
  },
  "private": "...",
  "cryptoInfo": {
```

(continues on next page)

(continued from previous page)

[illegible]

6.3 Crypto Provider

Class Name	CryptoProvider
Extends	CryptoProvider
Source	crypto-provider.ts

The **CryptoProvider** is a container for supported *Cryptors* and is able to determine, which *Cryptor* to use for encryption / decryption.

6.3.1 constructor

```
new CryptoProvider(cryptors);
```

Creates a new `CryptoProvider` instance.

Parameters

1. cryptors - any: object with available `Cryptors`.

Returns

CryptoProvider instance

Example

```
const serviceContract = new CryptoProvider({
  cryptors: {
    aes: new Aes(),
    unencrypted: new Unencrypted()
  }
});
```


(continued from previous page)

```
};  
const cryptor = runtime.cryptoProvider.getCryptorByCryptoInfo(cryptoInfo);
```

6.3.4 = Additional Components =

6.3.5 Interfaces

Cryptor

1. `options` - any: options which will be passed to the cryptor to work (like key for encryption)
2. `generateKey` - function: generates a random key for encryption/decryption
3. `getCryptoInfo` - function: returns an empty `CryptoInfo` object for the current `Cryptor`
4. `encrypt` - function: function to encrypt a given message
5. `decrypt` - function: function to decrypt a given message

Envelope

1. `algorithm` - string: algorithm used for encryption
2. `block` - number (optional): block number for which related item is encrypted
3. `cryptorVersion` - number (optional): version of the cryptor used. describes the implementation applied during decryption and not the algorithm version.
4. `originator` - string (optional): context for encryption, this can be
 - a context known to all parties (e.g. key exchange)
 - a key exchanged between two accounts (e.g. b-mails)
 - a key from a sharing's info from a contract (e.g. `DataContract`)defaults to 0
5. `keyLength` - number (optional): length of the key used in encryption

CryptoInfo

1. `public` - any (optional): unencrypted part of the data; will stay as is during encryption
2. `private` - any (optional): encrypted part of the data. If encrypting, this part will be encrypted, depending on the encryption. If already encrypted, this will be the encrypted value
3. `cryptoInfo` - `CryptoInfo`: describes used encryption

6.4 Cryptor - AES CBC

Class Name	Aes
Implements	Cryptor
Extends	Logger
Source	aes.ts
Examples	aes.spec.ts

The [AES](#) cryptor encodes and decodes content with aes-cbc.

6.4.1 constructor

```
new Aes(options);
```

Creates a new Aes instance.

Parameters

1. **options - AesOptions:** options for Aes constructor.

- `log` - `Function` (optional): function to use for logging: `(message, level) => {...}`
- `logLevel` - `LogLevel` (optional): messages with this level will be logged with `log`
- `logLog` - `LogLogInterface` (optional): container for collecting log messages
- `logLogLevel` - `LogLevel` (optional): messages with this level will be pushed to `logLog`

Returns

Aes instance

Example

```
const aes = new Aes();
```

6.4.2 getCryptoInfo

```
cryptor.getCryptoInfo(originator);
```

create new crypto info for this cryptor

Parameters

1. `originator` - `string`: originator or context of the encryption

Returns

CryptoInfo: details about encryption for originator with this cryptor.

Example

```
const cryptor = new Aes();
const cryptoInfo = cryptor.getCryptoInfo('0x123');
```

6.4.3 generateKey

```
cryptor.generateKey();
```

generate key for cryptor/decryption

Returns

Promise resolves to string: key used for encryption.

Example

```
const cryptor = new Aes();
const cryptoInfo = cryptor.generateKey();
```

6.4.4 encrypt

```
cryptor.encrypt(message, options);
```

‘encrypt’ a message (serializes message)

Parameters

1. `message - string`: message which should be encrypted
2. **`options - any: cryptor options`**
 - `key - string`: key used for encryption

Returns

Promise resolves to string: encrypted message.

Example

```
const cryptor = new Aes();
const cryptoInfo = cryptor.encrypt('Hello World', { key: '0x12345' });
```

6.4.5 decrypt

```
cryptor.decrypt(message, options);
```

‘decrypt’ a message (deserializes message)

Parameters

1. **message - Buffer:** message which should be decrypted
2. **options - any: cryptor options**
 - **key - string:** key used for encryption

Returns

Promise resolves to any: decrypted message.

Example

```
const cryptor = new Aes();
const cryptoInfo = cryptor.decrypt('afeweq4lf1e61e3f', { key: '0x12345' });
```

6.5 Cryptor - AES ECB

Class Name	AesEcb
Implements	Cryptor
Extends	Logger
Source	aes-ecb.ts
Examples	aes-ecb.spec.ts

The **AES ECB** cryptor encodes and decodes content with aes-ecb.

6.5.1 constructor

```
new AesEcb(options);
```

Creates a new AesEcb instance.

Parameters

1. options - **AesEcbOptions**: options for AesEcb constructor.

- log - Function (optional): function to use for logging: (message, level) => {...}
- logLevel - LogLevel (optional): messages with this level will be logged with log
- logLog - LogLogInterface (optional): container for collecting log messages
- logLogLevel - LogLevel (optional): messages with this level will be pushed to logLog

Returns

AesEcb instance

Example

```
const aesEcb = new AesEcb();
```

6.5.2 getCryptoInfo

```
cryptor.getCryptoInfo(originator);
```

create new crypto info for this cryptor

Parameters

1. originator - string: originator or context of the encryption

Returns

CryptoInfo: details about encryption for originator with this cryptor.

Example

```
const cryptor = new AesEcb();
const cryptoInfo = cryptor.getCryptoInfo('0x123');
```

6.5.3 generateKey

```
cryptor.generateKey();
```

generate key for cryptor/decryption

Returns

Promise resolves to `string`: key used for encryption.

Example

```
const cryptor = new Unencrypted();
const cryptoInfo = cryptor.generateKey();
```

6.5.4 encrypt

```
cryptor.encrypt(message, options);
```

‘encrypt’ a message (serializes message)

Parameters

1. `message` - `string`: message which should be encrypted
2. **options** - **any**: **cryptor options**
 - `key` - `string`: key used for encryption

Returns

Promise resolves to `string`: encrypted message.

Example

```
const cryptor = new Unencrypted();
const cryptoInfo = cryptor.encrypt('Hello World', { key: '0x12345' });
```

6.5.5 decrypt

```
cryptor.decrypt(message, options);
```

‘decrypt’ a message (deserializes message)

Parameters

1. `message` - `Buffer`: message which should be decrypted
2. **options** - **any**: **cryptor options**
 - `key` - `string`: key used for encryption

Returns

Promise resolves to any: decrypted message.

Example

```
const cryptor = new Unencrypted();
const cryptoInfo = cryptor.decrypt('afeweq4lf1e61e3f', { key: '0x12345' });
```

6.6 Cryptor - AES Blob

Class Name	AesBlob
Implements	Cryptor
Extends	Logger
Source	aes-blob.ts
Examples	aes-blob.spec.ts

The [AES Blob](#) cryptor encodes and decodes content with aes-cbc.

6.6.1 constructor

```
new AesBlob(options);
```

Creates a new AesBlob instance.

Parameters

1. **options - AesBlobOptions:** options for AesBlob constructor.

- `dfs - DfsInterface:` `DfsInterface` instance
- `log - Function (optional):` function to use for logging: `(message, level) => {...}`
- `logLevel - LogLevel (optional):` messages with this level will be logged with `log`
- `logLog - LogLogInterface (optional):` container for collecting log messages
- `logLogLevel - LogLevel (optional):` messages with this level will be pushed to `logLog`

Returns

AesBlob instance

Example

```
const aesBlob = new AesBlob({
  dfs
});
```

6.6.2 getCryptoInfo

```
cryptor.getCryptoInfo(originator);
```

create new crypto info for this cryptor

Parameters

1. `originator - string`: originator or context of the encryption

Returns

`CryptoInfo`: details about encryption for originator with this cryptor.

Example

```
const cryptor = new AesBlob();
const cryptoInfo = cryptor.getCryptoInfo('0x123');
```

6.6.3 generateKey

```
cryptor.generateKey();
```

generate key for cryptor/decryption

Returns

Promise resolves to `string`: key used for encryption.

Example

```
const cryptor = new AesBlob();
const cryptoInfo = cryptor.generateKey();
```

6.6.4 encrypt

```
cryptor.encrypt(message, options);
```

‘encrypt’ a message (serializes message)

Parameters

1. `message` - `string`: message which should be encrypted
2. **options** - **any**: **cryptor options**
 - `key` - `string`: key used for encryption

Returns

Promise resolves to `string`: encrypted message.

Example

```
const cryptor = new AesBlob();
const cryptoInfo = cryptor.encrypt('Hello World', { key: '0x12345' });
```

6.6.5 decrypt

```
cryptor.decrypt(message, options);
```

‘decrypt’ a message (deserializes message)

Parameters

1. `message` - `Buffer`: message which should be decrypted
2. **options** - **any**: **cryptor options**
 - `key` - `string`: key used for encryption

Returns

Promise resolves to `any`: decrypted message.

Example

```
const cryptor = new AesBlob();
const cryptoInfo = cryptor.decrypt('afeweq41f1e61e3f', { key: '0x12345' });
```

6.7 Cryptor - Unencrypted

Class Name	Unencrypted
Implements	Cryptor
Extends	Logger
Source	unencrypted.ts

The `Unencrypted` cryptor encodes and decodes content “unencrypted” this means no encryption is applied to the content. So simply the content is public, only HEX encoded.

6.7.1 constructor

```
new Unencrypted(options);
```

Creates a new `Unencrypted` instance.

Parameters

1. **options - UnencryptedOptions: options for Unencrypted constructor.**

- `log` - `Function` (optional): function to use for logging: `(message, level) => {...}`
- `logLevel` - `LogLevel` (optional): messages with this level will be logged with `log`
- `logLog` - `LogLogInterface` (optional): container for collecting log messages
- `logLogLevel` - `LogLevel` (optional): messages with this level will be pushed to `logLog`

Returns

`Unencrypted` instance

Example

```
const unencrypted = new Unencrypted();
```

6.7.2 getCryptoInfo

```
cryptor.getCryptoInfo(originator);
```

create new crypto info for this cryptor

Parameters

1. `originator` - `string`: originator or context of the encryption

Returns

CryptoInfo: details about encryption for originator with this cryptor.

Example

```
const cryptor = new Unencrypted();
const cryptoInfo = cryptor.getCryptoInfo('0x123');
```

6.7.3 generateKey

```
cryptor.generateKey();
```

generate key for cryptor/decryption

Returns

Promise resolves to string: key used for encryption.

Example

```
const cryptor = new Unencrypted();
const cryptoInfo = cryptor.generateKey();
```

6.7.4 encrypt

```
cryptor.encrypt(message, options);
```

‘encrypt’ a message (serializes message)

Parameters

1. message - string: message which should be encrypted
2. options - any: cryptor options
 - key - string: key used for encryption

Returns

Promise resolves to string: encrypted message.

The “public” section contains data, that is visible without any knowledge of the encryption key. The “private” section can only be decrypted if the user that tries to read the data has access to the encryption key. The *cryptoInfo* part is used to determine which decryption algorithm to use and where to look for it.

When decrypted, the *private* section takes precedence over the *public* section. This can lead to the private section overwriting sections of the *public* part. For example a public title may be replace with a “true” title (only visible for a group of people) from the private section.

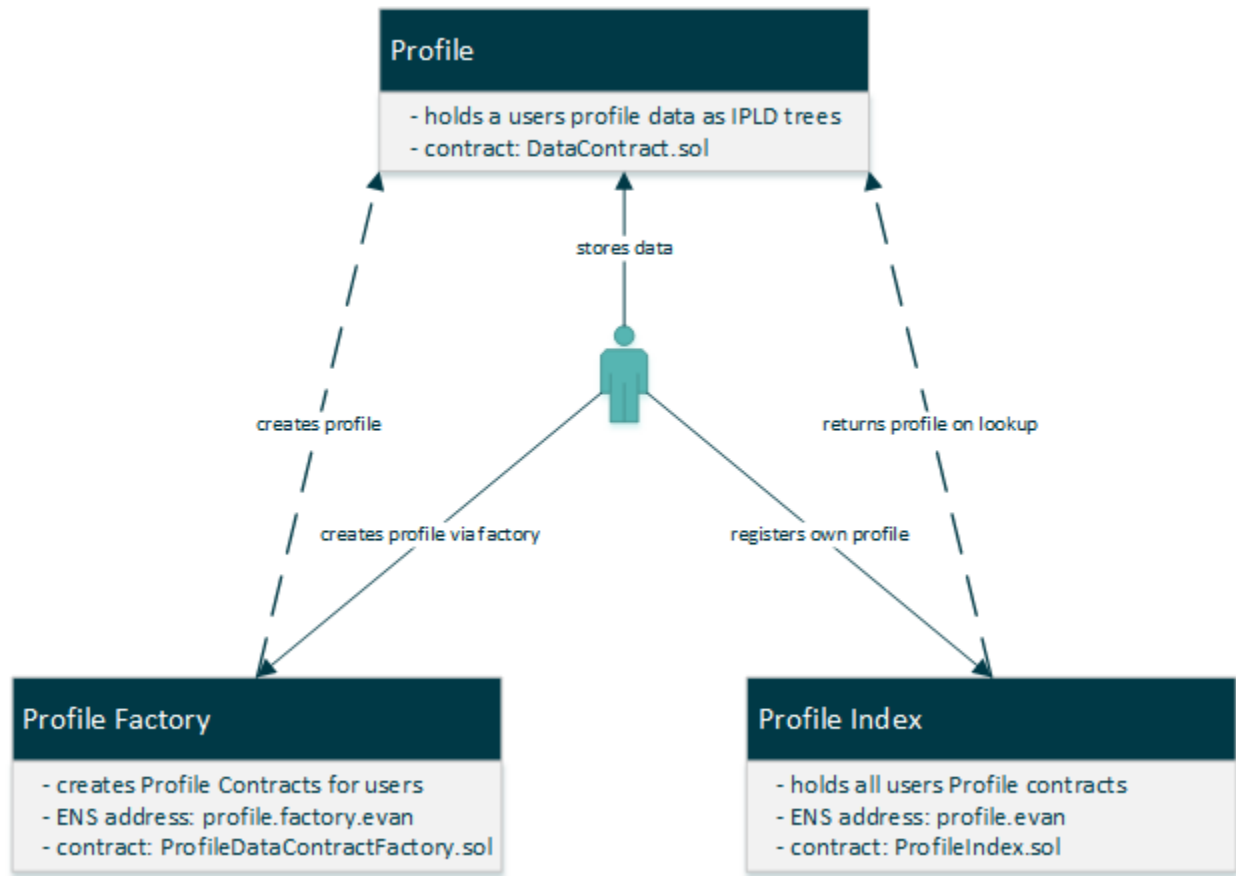
7.1 Profile

Class Name	Profile
Extends	Logger
Source	profile.ts
Examples	profile.spec.ts

A users profile is its personal storage for - contacts - encryption keys exchanged with contacts - an own public key for exchanging keys with new contacts - bookmarked ÐAPPs - created contracts

This data is stored as an [IPLD Graphs](#) per type and stored in a users profile contract. These graphs are independant from each other and have to be saved separately.

This contract is a [DataContract](#) and can be created via the factory at *profile.factory.evan* and looked up at the global profile index *profile.evan*. The creation process and landmap looks like this:



7.1.1 Basic Usage

```
// the bookmark we want to store
const sampleDesc = {
  title: 'sampleTest',
  description: 'desc',
  img: 'img',
  primaryColor: '#FFFFFF',
};

// create new profile, set private key and keyexchange partial key
await profile.createProfile(keyExchange.getDiffieHellmanKeys());

// add a bookmark
await profile.addDappBookmark('sample1.test', sampleDesc);

// store tree to contract
await profile.storeForAccount(profile.treeLabels.bookmarkedDapps);
```

7.1.2 constructor

```
new Profile(options);
```

Creates a new Profile instance.

Parameters

1. options - ProfileOptions: options for Profile constructor

- accountId - string: account, that is the profile owner
- contractLoader - `ContractLoader`: `ContractLoader` instance
- dataContract - `DataContract`: `DataContract` instance
- executor - `Executor`: `Executor` instance
- ipld - `Ipld`: `Ipld` instance
- nameResolver - `NameResolver`: `NameResolver` instance
- defaultCryptoAlgo - string (optional): crypto algorithm name from `CryptoProvider`, defaults to aes
- log - Function (optional): function to use for logging: (message, level) => {...}
- logLevel - `LogLevel` (optional): messages with this level will be logged with log
- logLog - `LogLogInterface` (optional): container for collecting log messages
- logLogLevel - `LogLevel` (optional): messages with this level will be pushed to logLog
- trees - object (optional): precached profile data, defaults to {}

Returns

Profile instance

Example

```
const profile = new Profile({
  accountId: accounts[0],
  contractLoader,
  dataContract,
  executor,
  ipld,
  nameResolver,
});
```

7.1.3 createProfile

```
profile.createProfile(keys)
```

Create new profile, store it to profile index initialize addressBook and publicKey.

Parameters

1. **keys - any: diffie hell man keys for account, created by `KeyExchange`**
 - `privateKey` - Buffer: private key for key exchange
 - `publicKey` - Buffer: combination of shared secret and own private key

Returns

Promise returns void: resolved when done

Example

```
await profile.createProfile(keyExchange.getDiffieHellmanKeys());
```

7.1.4 checkCorrectProfileData

```
profile.checkCorrectProfileData();
```

Check if profile data is correct, according to a specific profile type. Throws, when the data is invalid.

Parameters

1. `data` - object: profile data (`accountDetails`, `registration`, `contact`, ...) (see `setProfileProperties` for example values)
2. `type` - string: profileType (`user`, `company`, `device`)

Returns

Promise returns void: `true` if a contract was registered, `false` if not

Example

```
console.log(await Profile.checkCorrectProfileData({
  "accountDetails": {
    "accountName": "Test",
    "profileType": "company"
  },
  "registration": {
    "company": "Company Name",
    "court": "register court1",
    "register": "hrb",
    "registerNumber": "Registration Number",
    "salesTaxID": "tax id"
  },
  "contact": {
    "country": "DE",
```

(continues on next page)

(continued from previous page)

```
"city": "City",
"postalCode": "12345",
"streetAndNumber": "street",
"website": "https://evan.network"
}
}, 'company'));
// Output:
// true
```

7.1.5 exists

```
profile.exists();
```

Check if a profile has been stored for current account.

Parameters

1. options - object: The options used for calling

Returns

Promise returns void: true if a contract was registered, false if not

Example

```
console.log(await profile.exists());
// Output:
// true
```

7.1.6 getContactKnownState

```
profile.getContactKnownState(accountId);
```

Check, known state for given account.

Parameters

1. accountId - string: account id of a contact

Returns

Promise returns void: true if known account

Example

```
console.log(await profile.getContactKnownState(accountId));  
// Output:  
// true
```

7.1.7 getProfileProperty

```
profile.getProfileProperty(property);
```

Return the saved profile information according to the specified profile type. No type directly uses “user” type.

Parameters

1. `property` - string: Restrict properties that should be loaded.

Returns

Promise returns void: true if known account

Example

```
console.log(await profile.getProfileProperty('accountDetails'));  
// Output:  
// {  
//   "accountName": "Test",  
//   "profileType": "company"  
// }
```

7.1.8 setContactKnownState

```
profile.setContactKnownState(accountId, contactKnown);
```

Store given state for this account.

Parameters

1. `accountId` - string: account id of a contact
2. `contactKnown` - boolean: true if known, false if not

Returns

Promise returns void: resolved when done

Example

```
// mark accountId as a known contact
profile.setContactKnownState(accountId, true);
```

7.1.9 setProfileProperties

```
profile.setProfileProperties(payload);
```

Takes a set of profile properties and saves them into the profile DataContainer. Throws errors, if not the correct properties are applied for the specified account type.

Parameters

1. payload - any: Object that should saved. Each entry will be saved as seperated entry.

Returns

Promise returns void: resolved when done

Example

```
// mark accountId as a known contact
profile.setProfileProperties({
  "accountDetails": {
    "accountName": "Test",
    "profileType": "company"
  },
  "registration": {
    "company": "Company Name",
    "court": "register court1",
    "register": "hrb",
    "registerNumber": "Registration Number",
    "salesTaxID": "tax id"
  },
  "contact": {
    "country": "DE",
    "city": "Cirty",
    "postalCode": "12345",
    "streetAndNumber": "street",
    "website": "https://evan.network"
  }
});
```

7.1.10 loadForAccount

```
profile.loadForAccount([tree]);
```

Load profile for given account from global profile contract, if a tree is given, load that tree from ipld as well.

Parameters

1. `tree` - string (optional): tree to load ('bookmarkedDapps', 'contracts', ...), `profile.treeLabels` properties can be passed as arguments

Returns

Promise returns void: resolved when done

Example

```
await profile.loadForAccount(profile.treeLabels.contracts);
```

7.1.11 storeForAccount

```
profile.storeForAccount(tree);
```

Stores profile tree or given hash to global profile contract.

Parameters

1. `tree` - string: tree to store ('bookmarkedDapps', 'contracts', ...)
2. `ipldHash` - string (optional): store this hash instead of the current tree for account

Returns

Promise returns void: resolved when done

Example

```
await profile.storeForAccount(profile.treeLabels.contracts);
```

7.1.12 loadFromIpld

```
profile.loadFromIpld(tree, ipldIpfsHash);
```

Load profile from ipfs via ipld dag via ipfs file hash.

Parameters

1. `tree - string`: tree to load ('bookmarkedDapps', 'contracts', ...)
2. `ipldIpfsHash - string`: ipfs file hash that points to a file with ipld a hash

Returns

Promise returns Profile: this profile

Example

```
await profile.loadFromIpld(profile.treeLabels.contracts, ipldIpfsHash);
```

7.1.13 storeToIpld

```
profile.storeToIpld(tree);
```

Store profile in ipfs as an ipfs file that points to a ipld dag.

Parameters

1. `tree - string`: tree to store ('bookmarkedDapps', 'contracts', ...)

Returns

Promise returns string: hash of the ipfs file

Example

```
const storedHash = await profile.storeToIpld(profile.treeLabels.contracts);
```

7.1.14 = addressBook =

7.1.15 addContactKey

```
profile.addContactKey(address, context, key);
```

Add a key for a contact to bookmarks.

Parameters

1. `address - string`: account key of the contact
2. `context - string`: store key for this context, can be a contract, bc, etc.
3. `key - string`: communication key to store

Returns

Promise returns void: resolved when done

Example

```
await profile.addContactKey(accounts[0], 'context a', 'key 0x01_a');
```

7.1.16 addProfileKey

```
profile.addProfileKey(address, key, value);
```

Add a profile value to an account.

Parameters

1. address - string: account key of the contact
2. key - string: store key for the account like alias, etc.
3. value - string: value of the profile key

Returns

Promise returns void: resolved when done

Example

```
await profile.addProfileKey(accounts[0], 'email', 'sample@example.org');  
await profile.addProfileKey(accounts[0], 'alias', 'Sample Example');
```

7.1.17 getAddressBookAddress

```
profile.getAddressBookAddress(address);
```

Function description

Parameters

1. address - string: contact address

Returns

Promise returns any: bookmark info

Example

```
await profile.getAddressBookAddress(accounts[0]);
```

7.1.18 getAddressBook

```
profile.getAddressBook();
```

Get the whole addressBook.

Parameters

(none)

Returns

any: entire address book

Example

```
await profile.getAddressBook();
```

7.1.19 getContactKey

```
profile.getContactKey(address, context);
```

Get a communication key for a contact from bookmarks.

Parameters

1. address - string` : account key of the contact
2. context - string` : store key for this context, can be a contract, bc, etc.

Returns

Promise returns void: matching key

Example

```
await profile.getContactKey(accounts[0], 'exampleContext');
```

7.1.20 getProfileKey

```
profile.getProfileKey(address, key);
```

Get a key from an address in the address book.

Parameters

1. address - string: address to look up
2. key - string: type of key to get

Returns

Promise returns any: key

Example

```
const alias = await profile.getProfileKey(accountId, 'alias');
```

7.1.21 removeContact

```
profile.removeContact(address);
```

Remove a contact from bookmarkedDapps.

Parameters

1. address - string: account key of the contact

Returns

Promise returns void: resolved when done

Example

```
await profile.removeContact(address);
```

7.1.22 = bookmarkedDapps =

7.1.23 addDappBookmark

```
profile.addDappBookmark(address, description);
```

Add a bookmark for a dapp.

Parameters

1. address - string: ENS name or contract address (if no ENS name is set)
2. description - DappBookmark: description for bookmark

Returns

Promise returns void: resolved when done

Example

```
const bookmark = {
  "name": "taskboard",
  "description": "Create todos and manage updates.",
  "i18n": {
    "description": {
      "de": "Erstelle Aufgaben und überwache Änderungen",
      "en": "Create todos and manage updates"
    },
    "name": {
      "de": "Task Board",
      "en": "Task Board"
    }
  },
  "imgSquare": "...",
  "standalone": true,
  "primaryColor": "#e87e23",
  "secondaryColor": "#fffaf5",
};
await profile.addDappBookmark('sampletaskboard.evan', bookmark);
```

7.1.24 getDappBookmark

```
profile.getDappBookmark(address);
```

Get a bookmark for a given address if any.

Parameters

1. address - string: ENS name or contract address (if no ENS name is set)

Returns

Promise returns any: bookmark info

Example

```
await profile.getDappBookmark('sample1.evan');
```

7.1.25 getBookmarkDefinition

```
profile.getBookmarkDefinition();
```

Get all bookmarks for profile.

Parameters

(none)

Returns

Promise returns any: all bookmarks for profile

Example

```
await profile.getBookmarkDefinitions();
```

7.1.26 removeDappBookmark

```
profile.removeDappBookmark(address);
```

Remove a dapp bookmark from the bookmarkedDapps.

Parameters

1. address - string: address of the bookmark to remove

Returns

Promise returns void: resolved when done

Example

```
await profile.removeDappBookmark(address);
```

7.1.27 setDappBookmarks

```
profile.setDappBookmarks(bookmarks);
```

Set bookmarks with given value.

Parameters

1. bookmarks - any: The options used for calling

Returns

Promise returns void: resolved when done

Example

```
const bookmarks = await profile.getBookmarkDefinitions();  
// update bookmarks  
// ...  
await profile.setDappBookmarks(bookmarks);
```

7.1.28 = contracts =

7.1.29 addContract

```
profile.addContract(address, data);
```

Add a contract to the current profile.

Parameters

1. address - string: contract address
2. data - any: bookmark metadata

Returns

Promise returns void: resolved when done

Example

```
await profile.addContract('0x...', contractDescription);
```

7.1.30 getContracts

```
profile.getContracts();
```

Get all contracts for the current profile.

Parameters

(none)

Returns

Promise returns any: contracts info

Example

```
await profile.getContracts();
```

7.1.31 getContract

```
profile.getContract(address);
```

Get a specific contract entry for a given address.

Parameters

1. address - string: contact address

Returns

Promise returns any: bookmark info

Example

```
await profile.getContract('testbc.evan');
```

7.1.32 addBcContract

```
profile.addBcContract(bc, address, data)
```

Add a contract (task contract etc.) to a business center scope of the current profile

Parameters

1. `bc - string`: business center ens address or contract address
2. `address - string`: contact address
3. `data - any`: bookmark metadata

Returns

Promise `returns void`: resolved when done

Example

```
await profile.addBcContract('testbc.evan', '0x...', contractDescription);
```

7.1.33 getBcContract

```
profile.getBcContract(bc, address);
```

Get a specific contract entry for a given address.

Parameters

1. `bcc - string`: business center ens address or contract address
2. `address - string`: contact address

Returns

Promise `returns any`: bookmark info

Example

```
await profile.getBcContract('testbc.evan', '0x...');
```

7.1.34 getBcContracts

```
profile.getBcContracts(bc, address);
```

Get all contracts grouped under a business center.

Parameters

1. `bcc - string`: business center ens address or contract address

Returns

Promise returns any: bookmark info

Example

```
await profile.getBcContracts('testbc.evan');
```

7.1.35 removeContract

```
profile.removeBcContract(address, data);
```

removes a contract (task contract etc.) from a business center scope of the current profile

Parameters

1. bc - string: business center ens address or contract address
2. address - any: contact address

Returns

Promise returns void: resolved when done

Example

```
await profile.removeBcContract('testbc.evan', '0x');
```

7.1.36 = publicKey =

7.1.37 addPublicKey

```
profile.addPublicKey(key);
```

Add a key for a contact to bookmarks.

Parameters

1. key - string: public Diffie Hellman key part to store

Returns

Promise returns void: resolved when done

Example

```
await profile.addPublicKey('...');
```

7.1.38 getPublicKey

```
profile.getPublicKey();
```

Get public key of profiles.

Parameters

(none)

Returns

Promise returns any: public key

Example

```
const key = await profile.getPublicKey();
```

7.1.39 loadActiveVerifications

```
profile.loadActiveVerifications();
```

Load all verificationss that should be displayed for this profile within the ui.

Parameters

(none)

Returns

Promise returns string[]: array of topics of verificationss that should be displayed (e.g. ['/company/tuev', '/test/1234'])

Example

```
const topics = await bcc.profile.loadActiveVerifications();
```

7.1.40 setActiveVerifications

```
profile.setActiveVerifications(bookmarks);
```

Save an array of active verificationss to the profile.

Parameters

1. bookmarks - string[]: bookmarks to set

Returns

Promise returns void: resolved when saving is done

Example

```
await bcc.profile.setActiveVerifications([ '/company/tuev', '/test/1234' ]);
```

7.1.41 setPlugins

```
profile.setPlugins(plugins);
```

Save set of plugins to profile.

Parameters

1. plugins - any: entire collections of plugins to store in profile

Returns

Promise returns void: resolved when done

Example

```
await profile.setPlugins({ customMetadata: { } });
```

7.1.42 getPlugins

```
profile.getPlugins();
```

Get entire set of plugins from profile.

Returns

Promise returns any: all plugins from profile

Example

```
const plugins = await profile.getPlugins();
```

7.2 Business Center Profile

Class Name	BusinessCenterProfile
Extends	Logger
Source	business-center-profile.ts
Examples	business-center-profile.spec.ts

The BusinessCenterProfile module allows to create profiles in a business center.

These profiles are like business cards for specific contexts and can be used to share data like contact data or certificates under this business centers context.

7.2.1 Basic Usage

```
// update/set contact card locally
await profile.setContactCard(JSON.parse(JSON.stringify(sampleProfile)));

// store to business center
await profile.storeForBusinessCenter(businessCenterDomain, accounts[0]);

// load from business center
await profile.loadForBusinessCenter(businessCenterDomain, accounts[0]);
const loadedProfile = await profile.getContactCard();
```

7.2.2 constructor

```
new BusinessCenterProfile(options);
```

Creates a new BusinessCenterProfile instance.

Parameters

- options - BusinessCenterProfileOptions: options for BusinessCenterProfile constructor.**
 - bcAddress - string: ENS address (domain name) of the business center, the module instance is scoped to
 - cryptoProvider - CryptoProvider: CryptoProvider instance

- `defaultCryptoAlgo` - string: crypto algorithm name from `CryptoProvider`
- `ipld` - `IpIpld`: `IpIpld` instance
- `nameResolver` - `NameResolver`: `NameResolver` instance
- `ipldData` - any (optional): preloaded profile data
- `log` - Function (optional): function to use for logging: `(message, level) => {...}`
- `logLevel` - `LogLevel` (optional): messages with this level will be logged with `log`
- `logLog` - `LogLogInterface` (optional): container for collecting log messages
- `logLogLevel` - `LogLevel` (optional): messages with this level will be pushed to `logLog`

Returns

`BusinessCenterProfile` instance

Example

```
const businessCenterProfile = new BusinessCenterProfile({
  ipld,
  nameResolver,
  defaultCryptoAlgo: 'aes',
  bcAddress: businessCenterDomain,
  cryptoProvider,
});;
```

7.2.3 setContactCard

```
businessCenterProfile.setContactCard();
```

Set contact card on current profile.

Parameters

1. `contactCard` - any: contact card to store

Returns

Promise returns any: updated tree

Example

```
const updated = await businessCenterProfile.setContactCard(contactCard);
```

7.2.4 getContactCard

```
businessCenterProfile.getContactCard();
```

Get contact card from.

Parameters

(none)

Returns

Promise returns any: contact card

Example

```
const loadedProfile = await businessCenterProfile.getContactCard();
```

7.2.5 storeForBusinessCenter

```
businessCenterProfile.storeForBusinessCenter(businessCenterDomain, account);
```

Stores profile to business centers profile store.

Parameters

1. businessCenerDomain - string: ENS domain name of a business center
2. account - string: Ethereum account id

Returns

Promise returns void: resolved when done

Example

```
await businessCenterProfile.setContactCard(contactCard);  
await businessCenterProfile.storeForBusinessCenter(businessCenterDomain, accounts[0]);
```

7.2.6 loadForBusinessCenter

```
businessCenterProfile.loadForBusinessCenter(businessCenterDomain, account);
```

Function description

Parameters

1. `businessCenerDomain - string`: ENS domain name of a business center
2. `account - string`: Ethereum account id

Returns

Promise returns void: resolved when done

Example

```
await newProfilebusinessCenterProfile.loadForBusinessCenter(businessCenterDomain, ↪accounts[0]);  
const contactCard = await businessCenterProfile.getContactCard();
```

7.2.7 storeToIpId

```
businessCenterProfile.storeToIpId();
```

Store profile in ipfs as an ipfs file that points to a ipld dag.

Parameters

(none)

Returns

Promise returns string: hash of the ipfs file

Example

```
await businessCenterProfile.storeToIpId();
```

7.2.8 loadFromIpId

```
businessCenterProfile.loadFromIpId(tree, ipIdIpfsHash);
```

Load profile from ipfs via ipld dag via ipfs file hash.

Parameters

1. `ipIdIpfsHash - string`: ipfs file hash that points to a file with ipld a hash

Returns

Promise **returns** `BusinessCenterProfile`: this profile

Example

```
businessCenterProfile.loadFromIpld(ipIpldHash);
```

7.3 Onboarding

Class Name	Onboarding
Extends	Logger
Source	onboarding.ts
Examples	onboarding.spec.ts

The onboarding process is used to enable users to invite other users, where no blockchain account id is known. It allows to send an email to such contacts, that contains a link. This link points to a `evan.network` DApp, that allows accept the invitation by either creating a new account or by accepting it with an existing account.

It uses the [Key Exchange](#) module described in the last section for its underlying key exchange process but moves the process of creating a new communication key to the invited user.

To get in contact with a user via email, a smart agent is used. This smart agent has to be added as a contact and a regular key exchange with the smart agent is performed. The agent accepts the invitation automatically and the inviting user sends a bmail (blockchain mail) with the contact details of the user, that should be invited, and an amount of welcome EVEs to the smart agent.

The onboarding smart creates a session on his end and sends an email to the invited user, that includes the session token, with which the invited user can claim the welcome EVEs.

The invited user now creates or confirms an account and start the key exchange process on his or her end. The rest of the flow is as described in [Key Exchange](#).

To start the process at from the inviting users side, make sure that this user has exchanged keys with the onboarding smart agent.

7.3.1 constructor

```
new Onboarding(options);
```

Creates a new Onboarding instance.

Parameters

1. **options - OnboardingOptions: options for Onboarding constructor**
 - `executor` - `Executor`: `Executor` instance
 - `mailbox` - `Mailbox`: `Mailbox` instance

- `smartAgentId` - string: account id of onboarding smart agent
- `log` - Function (optional): function to use for logging: (message, level) => {...}
- `logLevel` - `LogLevel` (optional): messages with this level will be logged with log
- `logLog` - `LogLogInterface` (optional): container for collecting log messages
- `logLogLevel` - `LogLevel` (optional): messages with this level will be pushed to logLog

Returns

Onboarding instance

Example

```
const onboarding = new Onboarding({
  mailbox,
  smartAgentId: config.smartAgents.onboarding.accountId,
  executor,
});
```

7.3.2 sendInvitation

```
onboarding.sendInvitation(invitation, weiToSend);
```

Send invitation to another user via smart agent that sends a mail.

Parameters

1. `invitation` - invitation: mail that will be sent to invited person
2. `weiToSend` - string: amount of ETC to transfert to new member, can be created with `web3.utils.toWei(10, 'ether')` [web3 >=1.0] / `web.toWei(10, 'ether')` [web3 < 1.0]

Returns

Promise returns void: resolved when done

Example

```
await onboarding.sendInvitation({
  fromAlias: 'example inviter',
  to: 'example invitee <example.invitee@evan.network>',
  lang: 'en',
  subject: 'evan.network Onboarding Invitation',
  body: 'I\'d like to welcome you on board.',
}, web3.utils.toWei('1'));
```

7.3.3 createMnemonic

```
Onboarding.createMnemonic();
```

(static class function)

Generates a new random Mnemonic

Returns

string

Example

To show the difference, without purging:

```
const mnemonic = Onboarding.createMnemonic();
console.log(mnemonic);
// prints out a random 12 word mnemonic
```

7.3.4 createNewProfile

```
Onboarding.createNewProfile(mnemonic, password, profileProperties);
```

(static class function)

Creates a new full blown profile on a given even network (testcore/core) and returns the mnemonic, password and a configuration for the runtime initialization

Parameters

1. mnemonic - string: 12 word mnemonic as string
2. password - string: password of the new created profile
3. profileProperties - any: Properties for the profile to be created

Returns

Promise returns any: object with the mnemonic, password and the config object for the runtime

Example

```
const originRuntime = await TestUtils.getRuntime(accounts[0]);
const mnemonic = Onboarding.createMnemonic();
await Onboarding.createNewProfile(originRuntime, mnemonicNew, 'Test1234', {
  accountDetails: {
    profileType: 'company',
    accountName: 'test account'
  })
});
```

7.4 Key Exchange

Class Name	KeyExchange
Extends	Logger
Source	keyExchange.ts
Examples	keyExchange.spec.ts

The `KeyExchange` module is used to exchange communication keys between two parties, assuming that both have created a profile and have a public facing partial Diffie Hellman key part (the combination of their own secret and the shared secret). The key exchange consists of three steps:

1. create a new communication key, that will be used by both parties for en- and decryption and store it on the initiators side
 2. look up the other parties partial Diffie Hellman key part and combine it with the own private key to create the exchange key
 3. use the exchange key to encrypt the communication key and send it via `bmail` (blockchain mail) to other party
-

7.4.1 Basic Usage

Starting the Key Exchange Process

This example retrieves public facing partial Diffie Hellman key part from a second party and sends an invitation mail to it:

```
// account, that initiates the invitation
const account1 = '0x0000000000000000000000000000000000000000000000000000000000000001';
// account, that will receive the invitation
const account2 = '0x0000000000000000000000000000000000000000000000000000000000000002';
// profile from user, that initiates key exchange
const profile1 = {};
await profile1.loadForAccount();
// profile from user, that is going to receive the invitation
const profile2 = {};
await profile2.loadForAccount();
// key exchange instance for account1
const keyExchange1 = {};
// key exchange instance for account2
const keyExchange2 = {};

const foreignPubkey = await profile2.getPublicKey();
const commKey = await keyExchange1.generateCommKey();
await keyExchange1.sendInvite(account2, foreignPubkey, commKey, {
  fromAlias: 'Bob', // initiating user states, that his name is 'Bob'
});
await profile1.addContactKey(account2, 'commKey', commKey);
await profile1.storeForAccount(profile1.treeLabels.addressBook);
```

Finishing the Key Exchange Process

Let's assume that the communication key from the last example has been successfully sent to the other party and continue at there end from here. To keep the roles from the last example, the variables profile1, profile2 will belong to the same accounts:

```
const encryptedCommKey = '...';           // key sent by account1
const profile1 = await profile1.getPublicKey();
const commSecret = keyExchange2.computeSecretKey(profile1);
const commKey = await keyExchange2.decryptCommKey(encryptedCommKey, commSecret.
  ↪toString('hex'));
```

7.4.2 constructor

```
new KeyExchange(options);
```

Creates a new KeyExchange instance.

Parameters

1. options - KeyExchangeOptions: options for KeyExchange constructor.

- account - string: account, that will perform actions
- cryptoProvider - `CryptoProvider`: `CryptoProvider` instance
- defaultCryptoAlgo - string: default encryption algorithm
- keyProvider - `KeyProviderInterface`: `KeyProviderInterface` instance
- mailbox - `Mailbox`: `Mailbox` instance
- log - Function (optional): function to use for logging: (message, level) => {...}
- logLevel - `LogLevel` (optional): messages with this level will be logged with log
- logLog - `LogLogInterface` (optional): container for collecting log messages
- logLogLevel - `LogLevel` (optional): messages with this level will be pushed to logLog
- privateKey - object (optional): private key for key exchange, if privateKey or publicKey is omitted, new keys are generated
- publicKey - object (optional): public key for key exchange, if privateKey or publicKey is omitted, new keys are generated

Returns

KeyExchange instance

Example

```
const keyExchange = new KeyExchange({
  mailbox,
  cryptoProvider,
  defaultCryptoAlgo: 'aes',
  account: accounts[0],
  keyProvider,
});
```

7.4.3 computeSecretKey

```
keyExchange.computeSecretKey(partialKey);
```

Combines given partial key from another profile with own private key.

Parameters

1. partialKey - string: The options used for calling

Returns

string combined exchange key

Example

```
// encrypted communication key sent from account 1 to account 2
const encryptedKey = '...'
// (profile 1 belongs to account 1, keyExchange 2 to account 2)
const publicKeyProfile1 = await profile1.getPublicKey();
const commSecret = keyExchange2.computeSecretKey(publicKeyProfile1);
commKey = await keyExchange2.decryptCommKey(encryptedKey, commSecret.toString('hex'));
```

7.4.4 decryptCommKey

```
keyExchange.decryptCommKey(encryptedCommKey, exchangeKey);
```

Decrypts a given communication key with an exchange key.

Parameters

1. encryptedCommKey - string: encrypted communications key received from another account
2. exchangeKey - string: Diffie Hellman exchange key from computeSecretKey

Returns

Promise returns Buffer: commKey as a buffer

Example

```
// encrypted communication key sent from account 1 to account 2
const encryptedKey = '...'
// (profile 1 belongs to account 1, keyExchange 2 to account 2)
const publicKeyProfile1 = await profile1.getPublicKey();
const commSecret = keyExchange2.computeSecretKey(publicKeyProfile1);
commKey = await keyExchange2.decryptCommKey(encryptedKey, commSecret.toString('hex'));
```

7.4.5 getDiffieHellmanKeys

```
keyExchange.getDiffieHellmanKeys();
```

Returns the public and private key from the diffieHellman.

Parameters

(void)

Returns

Promise returns any: object with public and private keys

Example

```
console.dir(await keyExchange.getDiffieHellmanKeys());
// Output:
// {
//   private: '...',
//   public: '...',
// }
```

7.4.6 generateCommKey

```
keyExchange.generateCommKey();
```

Generates a new communication key and returns the hex string.

Parameters

(none)

Returns

Promise returns string: comm key as string

Example

```
console.dir(await keyExchange.generateCommKey());  
// Output:  
// '1c967697c192235680efbb24b980981b4778c8058b5e0864f1471fc1d941499d'
```

7.4.7 getExchangeMail

```
keyExchange.getExchangeMail(from, mailContent[, encryptionCommKey]);
```

Creates a bmail for exchanging comm keys.

Parameters

1. from - string: sender accountId
2. mailContent - any: bmail metadata
3. encryptedCommKey - string (optional): comm key, that should be exchanged

Returns

Promise returns Mail: mail for key exchange

Example

```
const commKey = '1c967697c192235680efbb24b980981b4778c8058b5e0864f1471fc1d941499d';  
const mail = keyExchange.getExchangeMail(  
  '0x0000000000000000000000000000000000000000000000000000000000000001',  
  { fromAlias: 'user 1', fromMail: 'user1@example.com', title: 'sample', body: 'sample',  
    ↪ }  
);  
console.log(mail);  
// Output:  
// { content:  
//   { from: '0x0000000000000000000000000000000000000000000000000000000000000001',  
//     fromAlias: 'user 1',  
//     fromMail: 'user1@example.com',  
//     title: 'sample',  
//     body: 'sample',  
//     attachments: [ [Object] ] } } }
```

7.4.8 sendInvite

```
keyExchange.sendInvite(targetAccount, targetPublicKey, commKey, mailContent);
```

Sends a mailbox mail to the target account with the partial key for the key exchange.

Parameters

1. string - targetAccount: receiver of the invitation
2. string - targetPublicKey: combination of shared secret plus targetAccounts private secret
3. string - commKey: communication key between sender and targetAccount
4. any - mailContent: mail to send

Returns

Promise returns void: resolved when done

Example

```
const foreignPubkey = await profile2.getPublicKey();
const commKey = await keyExchange1.generateCommKey();
await keyExchange1.sendInvite(accounts[1], foreignPubkey, commKey, { fromAlias: 'Bob',
  ↪ });
await profile.addContactKey(accounts[1], 'commKey', commKey);
await profile.storeForAccount(profile.treeLabels.addressBook);
```

7.4.9 setPublicKey

```
keyExchange.setPublicKey(publicKey, privateKey);
```

Set the private and public key on the current diffieHellman object.

Parameters

1. publicKey - string: public Diffie Hellman key
2. privateKey - string: private Diffie Hellman key

Returns

(no return value)

Example

```
keyExchange.setPublicKey('...', '...');
```

7.5 Mailbox

Class Name	Mailbox
Extends	Logger
Source	mailbox.ts
Examples	mailbox.spec.ts

The [Mailbox](#) module is used for sending and retrieving bmails (blockchain mails) to other even.network members. Sending regular bmails between to parties requires them to have completed a [Key Exchange](#) before being able to send encrypted messages. When exchanging the keys, bmails are encrypted with a commonly known key, that is only valid in this case and the underlying messages, that contain the actual keys are encrypted with Diffie Hellman keys, to ensure, that keys are exchanged in a safe manner (see [Key Exchange](#) for details).

The mailbox is a [smart contract](#), that holds

- bytes32 hashes, that are the encrypted contents of the mails
- basic metadata about the mails, like
 - recipient of a mail
 - sender of a mail
 - amount of EVEs, that belongs to the bmail
- if the mail is an answer to another mail, the reference to the original mail

7.5.1 constructor

```
new Mailbox(options);
```

Creates a new Mailbox instance.

Instances created with the constructor are **not usable** right from the start. They require the [init\(\)](#) function to be called, before they are ready to use.

Parameters

1. **options - MailboxOptions: options for Mailbox constructor.**

- contractLoader - [ContractLoader](#): [ContractLoader](#) instance
- cryptoProvider - [CryptoProvider](#): [CryptoProvider](#) instance
- defaultCryptoAlgo - string: crypto algorithm name from [CryptoProvider](#)
- ipfs - [Ipfs](#): [Ipfs](#) instance
- keyProvider - [KeyProviderInterface](#): [KeyProviderInterface](#) instance
- mailboxOwner - string: account, that will be used, when working with the mailbox
- nameResolver - [NameResolver](#): [NameResolver](#) instance
- executor - [Executor](#) (optional): [Executor](#) instance
- log - Function (optional): function to use for logging: (message, level) => {...}

- `logLevel` - `LogLevel` (optional): messages with this level will be logged with `log`
- `logLog` - `LogLogInterface` (optional): container for collecting log messages
- `logLogLevel` - `LogLevel` (optional): messages with this level will be pushed to `logLog`

Returns

Mailbox instance

Example

```
const mailbox = new Mailbox({
  mailboxOwner,
  nameResolver,
  ipfs,
  contractLoader,
  cryptoProvider,
  keyProvider,
  defaultCryptoAlgo: 'aes',
});
await mailbox.init();
```

7.5.2 init

```
mailbox.init();
```

Initialize mailbox module.

This function needs to be called, before the mailbox module can be used.

Parameters

(none)

Returns

Promise returns void: resolved when done

Example

```
const mailbox = new Mailbox({
  mailboxOwner,
  nameResolver,
  ipfs,
  contractLoader,
  cryptoProvider,
  keyProvider,
  defaultCryptoAlgo: 'aes',
```

(continues on next page)

(continued from previous page)

```
});  
await mailbox.init();
```

7.5.3 sendMail

```
mailbox.sendMail(mail, from, to[, value, context]);
```

Sends a mail to given target.

Parameters

1. mail - Mail: a mail to send
2. from - string: account id to send mail from
3. to - string: account id to send mail to
4. value - string (optional): amount of EVEs to send with mail in Wei, can be created with `web3[.utils].toWei(...)`, defaults to 0
5. context - string (optional): encryption context for bmail, if a special context should be used (e.g. `keyExchange`)

Returns

Promise returns void: resolved when done

Example

```
// account, that sends the mail  
const account1 = '0x0000000000000000000000000000000000000000000000000000000000000001';  
// account, that receives the mail  
const account2 = '0x0000000000000000000000000000000000000000000000000000000000000002';  
// mailbox of the sender  
const mailbox1 = {};  
// mailbox of the receiver  
const mailbox2 = {};  
  
const bmail = {  
  content: {  
    from: account1,  
    to,  
    title: 'Example bmail',  
    body: 'This is a little example to demonstrate sending a bmail.',  
    attachments: [ ]  
  }  
};  
await mailbox1.sendMail(bmail, account1, account2);
```

7.5.4 sendAnswer

```
mailbox.sendAnswer(mail, from, to[, value, context]);
```

Send answer to a mail.

Parameters

1. mail - Mail: a mail to send, mail.parentId must be set to mailId of mail, that is answered
2. from - string: account id to send mail from
3. to - string: account id to send mail to
4. value - string (optional): amount of EVEs to send with mail in Wei, can be created with web3[.utils].toWei(...), defaults to 0
5. context - string (optional): encryption context for bmail, if a special context should be used (e.g. keyExchange)

Returns

Promise returns void: resolved when done

Example

```
// account, that sends the answer
const account1 = '0x0000000000000000000000000000000000000000000000000000000000000001';
// account, that receives the answer
const account2 = '0x0000000000000000000000000000000000000000000000000000000000000002';
// mailbox of the sender
const mailbox1 = {};
// mailbox of the receiver
const mailbox2 = {};

const bmail = {
  content: {
    from: account1,
    to,
    title: 'Example bmail',
    body: 'This is a little example to demonstrate sending a bmail.',
    attachments: [ ]
  },
  parentId: '0x0000000000000000000000000000000000000000000000000000000000000012',
};
await mailbox1.sendAnswer(bmail, account1, account2);
```

7.5.5 getMails

```
mailbox.getMails([count, offset, type]);
```

Gets the last n mails, resolved contents.

Parameters

1. count - number (optional): retrieve up to this many answers (for paging), defaults to 10
2. offset - number (optional): skip this many answers (for paging), defaults to 0
3. type - string (optional): retrieve sent or received mails, defaults to 'Received'

Returns

Promise returns any: resolved mails

Example

```
const received = await mailbox2.getMails();
console.dir(JSON.stringify(received[0], null, 2));
// Output:
// {
//   "mails": {
//     "0x0000000000000000000000000000000000000000e": {
//       "content": {
//         "from": "0x00000000000000000000000000000000000000001",
//         "to": "0x00000000000000000000000000000000000000002",
//         "title": "Example bmail",
//         "body": "This is a little example to demonstrate sending a bmail.",
//         "attachments": [ ],
//         "sent": 1527083983148
//       },
//       "cryptoInfo": {
//         "originator":
// ↪ "0x549704d235e1fe5cd7326aleb0c44c1e0a5434799ba6ff2370c2955730b66e2b",
//         "keyLength": 256,
//         "algorithm": "aes-256-cbc"
//       }
//     },
//     "totalResultCount": 9
//   }
// }
```

Results can be paged with passing arguments for page size and offset to the `getMails` function:

```
const received = await mailbox2.getMails(3, 0);
console.dir(JSON.stringify(received[0], null, 2));
// Output:
// { mails:
//   { '0x0000000000000000000000000000000000000000e': { content: [Object], ↵
// ↪ cryptoInfo: [Object] },
//     '0x0000000000000000000000000000000000000000d': { content: [Object], ↵
// ↪ cryptoInfo: [Object] },
//     '0x0000000000000000000000000000000000000000c': { content: [Object], ↵
// ↪ cryptoInfo: [Object] } },
//   totalResultCount: 9 }
// }
```

To get bmails *sent* by an account, use (the example account hasn't sent any bmail yet):

```
mailbox.getMail(mail);
```

Parameters

- ```
const mailId = '0x0012';
const bmail = await mailbox.getMail(mailId);
```

```
mailbox.getAnswersForMail(mailId[, count, offset]);
```

259

## Example

```
const mailId = '0x0012';
const answers = await mailbox.getAnswersForMail(mailId);
```

---

## 7.5.8 getBalanceFromMail

```
mailbox.getBalanceFromMail(mailId);
```

Returns amount of EVE deposited for a mail.

Bmails can contain EVEs for the recipient as well. Because retrieving bmails is a reading operation, funds send with a bmail have to be retrieved separately.

### Parameters

1. mailId - string: mail to resolve

### Returns

Promise returns string: balance of the mail in Wei, can be converted with `web3[utils].fromWei(...)`

## Example

```
const bmail = {
 content: {
 from: account1,
 to,
 title: 'Example bmail',
 body: 'This is a little example to demonstrate sending a bmail.',
 attachments: []
 }
};
await mailbox1.sendMail(bmail, account1, account2, web3.utils.toWei('0.1', 'Ether'));
const received = await mailbox2.getMails(1, 0);
const mailBalance = await mailbox2.getBalanceFromMail(Object.keys(received)[0]);
console.log(mailBalance);
// Output:
// 1000000000000000000
```

---

## 7.5.9 withdrawFromMail

```
mailbox.withdrawFromMail(mailId, recipient);
```

Funds from bmails can be claimed with the account, that received the bmail. Funds are transferred to a specified account, which can be the claiming account or another account of choice.



## Parameters

1. `mailId` - string: mail to resolve
2. `recipient` - string: account, that receives the EVEs

## Returns

Promise returns void: resolved when done

## Example

```
const received = await mailbox2.getMails(1, 0);
const mailBalance = await mailbox2.getBalanceFromMail(Object.keys(received)[0]);
console.log(mailBalance);
// Output:
// 1000000000000000000
await mailbox2.withdrawFromMail(received[0], accounts2);
const mailBalance = await mailbox2.getBalanceFromMail(Object.keys(received)[0]);
console.log(mailBalance);
// Output:
// 0
```

## 7.6 Verifications

| Class Name | Verifications                         |
|------------|---------------------------------------|
| Extends    | <a href="#">Logger</a>                |
| Source     | <a href="#">verifications.ts</a>      |
| Tests      | <a href="#">verifications.spec.ts</a> |

The Verifications module allows to

- issue verifications about oneself or about other parties
- confirm or delete verifications about oneself

Verifications have a pattern similar to file paths, a verification for an account called “foo” being an employee of a company called “bar” may look like this:

```
/company/bar/employee
```

Under this “path” a set of values can be found. These value describe the verification, the subject of the verification and optional its response to it. Basically an issuer creates a verification about a subject The values are:

- `verification` (name) full path to a verification, for example `/company/bar/employee/foo`, settable by the subject of the parent verification `/company/bar/employee`
- `subject` an account, a verification has been issued for, can be a group/wallet or an externally owned account being the subject of a verification basically means to be the owner of the verification and allows to create subverifications below the own verification path
- `issuer` an account (group/wallet or externally owned) that creates a verification, to be able to issue a verification, the issuer has to be the subject of the parent verification `/company/bar/employee`

- `data` The hash of the verification data, sitting in another location, a bit-mask, call data, or actual data based on the verification scheme.
- `uri` The location of the verification, this can be HTTP links, swarm hashes, IPFS hashes, and such.
- `status` this represents a verifications status, values are `uint8` range from 0 to 255, the currently used values are: - 0: Issued - 1: Confirmed
- `signature` Signature which is the proof that the verification issuer issued a verification of topic for this identity. It MUST be a signed message of the following structure: `keccak256(address identityHolder_address, uint256 _topic, bytes data)`
- `creationDate` `creationDate` of the verification
- `id` id of the current verification
- `valid` check if the verification has a valid signature

For a explanation on how to use verification API, possible flows and meaning of the results have a look at the [verifications usage example](#).

---

### 7.6.1 constructor

```
new Verifications(options);
```

Creates a new Verifications instance.

Note, that the option properties `registry` and `resolver` are optional but should be provided in most cases. As the module allows to create an own ENS structure, that includes an own ENS registry and an own default resolver for it, setting them beforehand is optional.

#### Parameters

##### 1. `options - VerificationsOptions`: options for Verifications constructor.

- `accountStore` - `AccountStore`: `AccountStore` instance
- `config` - any: config object with `NameResolver` config
- `contractLoader` - `ContractLoader`: `ContractLoader` instance
- `description` - `Description`: `Description` instance
- `dfs` - `DfsInterface`: `DfsInterface` instance
- `executor` - `Executor`: `Executor` instance
- `nameResolver` - `NameResolver`: `NameResolver` instance
- `log` - Function (optional): function to use for logging: `(message, level) => {...}`
- `logLevel` - `LogLevel` (optional): messages with this level will be logged with `log`
- `logLog` - `LogLogInterface` (optional): container for collecting log messages
- `logLogLevel` - `LogLevel` (optional): messages with this level will be pushed to `logLog`
- `registry` - string (optional): contract address of the identity storage registry
- `storage` - string (optional): contract address of the identity storage registry

## Returns

Verifications instance

## Example

```
const verifications = new Verifications({
 accountStore,
 config,
 contractLoader,
 description,
 dfs,
 executor,
 nameResolver,
 storage: '0x0001',
});
```

## 7.6.2 = Issuers =

### 7.6.3 createIdentity

```
verifications.createIdentity(accountId[, contractId, updateDescription]);
```

Creates a new identity for account or contract and registers them on the storage. Returned identity is either a 20B contract address (for account identities) or a 32B identity hash contract identities.

## Parameters

1. accountId - string: account that runs transaction, receiver of identity when omitting the other arguments
2. contractId - string: (optional) contract address to create the identity for, creates account identity for accountId if omitted
3. updateDescription - boolean (optional): update description of contract, defaults to true

## Returns

Promise returns void: resolved when done

## Example

```
const identity = await verifications.createIdentity(accounts[0]);
console.log(identity);
// Output:
// 0x1fE5F7235f1989621135466Ff8882287C63A5bae
```

## 7.6.4 identityAvailable

```
verifications.identityAvailable(subject);
```

Checks if a account has already an identity contract.

### Parameters

1. subject - string: target subject to check

### Returns

Promise returns boolean: true if identity exists, otherwise false

### Example

```
console.log(await verifications.identityAvailable(accounts[0]));
// Output:
// false

await await verifications.createIdentity(accounts[0]);

console.log(await verifications.identityAvailable(accounts[0]));
// Output:
// true
```

---

## 7.6.5 getIdentityForAccount

```
verifications.getIdentityForAccount(subject);
```

Gets the identity contract for a given account id or contract.

### Parameters

1. subject - string: target subject to get identity for

### Returns

Promise returns any: identity contract instance

### Example

```
const identityContract = await verifications.getIdentityForAccount(accounts[0]);
```

### 7.6.6 setVerification

```
verifications.setVerification(issuer, subject, topic, expirationDate,
↪verificationValue, descriptionDomain, disableSubVerifications);
```

Sets or creates a verification; this requires the issuer to have permissions for the parent verification (if verification name seen as a path, the parent ‘folder’).

The “verificationValue” field can also be set to a custom JSON object with any data. For example DID, VC’s, documents or any other custom value you want to attach to the verification.

#### Parameters

1. issuer - string: issuer of the verification
2. subject - string: subject of the verification and the owner of the verification node
3. topic - string: name of the verification (full path)
4. expirationDate - number (optional): expiration date, for the verification, defaults to 0 (does not expire)
5. verificationValue - any (optional): json object which will be stored in the verification
6. descriptionDomain - string (optional): domain of the verification, this is a subdomain under ‘verifications.evan’, so passing ‘example’ will link verifications description to ‘example.verifications.evan’, unset if omitted
7. disableSubVerifications - boolean (optional): invalidate all verifications that gets issued as children of this verification (warning will include the disableSubVerifications warning)
8. isIdentity - boolean (optional): boolean value if the subject is already a identity address (default false)
9. uri - string (optional): adds a specific URI to the verification

#### Returns

Promise returns string: id of new verification

#### Example

```
// accounts[0] issues verification '/company' for accounts[1]
const firstVerification = await verifications.setVerification(accounts[0],
↪accounts[1], '/company');

// accounts[0] issues verification '/companyData' for accounts[1] with additional
↪data attached
const secondVerification = await verifications.setVerification(accounts[0],
↪accounts[1], '/companyData', 0, {
 additionalDocument: <binary buffer>
 vcDid: {
 "@context": [
 "https://www.w3.org/2018/credentials/v1",
 "https://www.w3.org/2018/credentials/examples/v1"
],
 "id": "http://example.edu/credentials/3732",
 "type": ["VerifiableCredential", "UniversityDegreeCredential"],
```

(continues on next page)

(continued from previous page)

```

 "credentialSubject": {
 "id": "did:example:ebfeb1f712ebc6f1c276e12ec21",
 "degree": {
 "type": "BachelorDegree",
 "name": "Bachelor of Science and Arts"
 }
 },
 "proof": { }
 }
});

// accounts[0] issues verification '/company' for accounts[1], sets an expiration date
// and links to description domain 'sample'
const thirdVerification = await verifications.setVerification(
 accounts[0], accounts[1], '/company', expirationDate, verificationValue, 'example');

```

### 7.6.7 getVerifications

```
verifications.getVerifications(subject, topic, isIdentity];
```

Gets verification information for a verification name from a given account; results has the following properties: creationBlock, creationDate, data, description, expirationDate, id, issuer, name, signature, status, subject, topic, uri, valid.

#### Parameters

1. `subject` - string: subject of the verifications
2. `topic` - string: name (/path) of a verification
3. `isIdentity` - string (optional): indicates if the subject is already an identity

#### Returns

Promise returns `any[]`: verification info array,

Verifications have the following properties:

1. `creationBlock` - string: block number at which verification was issued
2. `creationDate` - string: UNIX timestamp (in seconds), at which verification was issued
3. `data` - string: 32Bytes hash of data stored in DFS
4. `description` - any: DBCP description
5. `disableSubVerifications` - boolean: true if this verification does not allow verifications at subtopics
6. `expirationDate` - string: UNIX timestamp (in seconds), null if verification does not expire
7. `expired` - boolean: ticket expiration state
8. `id` - string: 32Bytes id of verification
9. `issuer` - string: account address of issuers identity contract

10. name - string: topic of verification
11. rejectReason - any: object with information from subject about rejection
12. signature - string: arbitrary length hex string with signature of verification data, signed by issuer
13. status - number: 0 (Issued) || 1 (Confirmed) || 2 (Rejected)
14. subject - string: accountId of subject
15. topic - string: keccak256 hash of the topic name, converted to uint256
16. uri - string: link to ipfs file of data
17. valid - boolean: true if issuer has been correctly confirmed as the signer of signature and if signature is related to subject, topic and data

### Example

```
const verificationId = await verifications.setVerification(
 accounts[0], accounts[1], '/example1');
console.log(verificationId);
// Output:
// 0xb4843ed5177433312dd2c7c4f8065ce84f37bf96c04db2775c16c9455ad96270

const issued = await verifications.getVerifications(accounts[1], '/example1');
console.dir(issued);
// Output:
// [{
// creationBlock: '186865',
// creationDate: '1558599441',
// data: '0x00',
// description: null,
// disableSubVerifications: false,
// expirationDate: null,
// expired: false,
// id: '0xb4843ed5177433312dd2c7c4f8065ce84f37bf96c04db2775c16c9455ad96270',
// issuer: '0xe560eF0954A2d61D6006E8547EC769fAc322bbCE',
// name: '/example1',
// rejectReason: undefined,
// signature:
 ↪ '0x6a2b41714c1faac09a5ec06024c8931ad6e3aa902c502e3d1bc5d5c4577288c04e9be136c149b569e0456dfec9d50a2',
 ↪ ',
// status: 0,
// subject: '0x0030C5e7394585400B1FB193DdbCb45a37Ab916E',
// topic:
 ↪ '34884897835812838038558016063403566909277437558805531399344559176587016933548',
// uri: '',
// valid: true
// }]
```

## 7.6.8 getNestedVerifications

```
getNestedVerifications(subject, topic, isIdentity);
```

Get all the verifications for a specific subject, including all nested verifications for a deep integrity check.

## Parameters

1. `subject` - string: subject to load the verifications for.
2. `topic` - string: topic to load the verifications for.
3. `isIdentity` - boolean: optional indicates if the subject is already a identity

## Returns

Promise returns `Array<any>`: all the verifications with the following properties.

## Example

```
const nestedVerifications = await getNestedVerifications('0x123...', '/test')

// will return
[
 {
 // creator of the verification
 issuer: '0x1813587e095cDdfd174DdB595372Cb738AA2753A',
 // topic of the verification
 name: '/company/b-s-s/employee/swo',
 // -1: Not issued => no verification was issued
 // 0: Issued => status = 0, warning.length > 0
 // 1: Confirmed => issued by both, self issued state is 2, values match
 status: 2,
 // verification for account id / contract id
 subject: subject,
 // ???
 value: '',
 // ???
 uri: '',
 // ???
 signature: ''
 // icon for cards display
 icon: 'icon to display',
 // if the verification was rejected, a reject reason could be applied
 rejectReason: '' || { },
 // subject type
 subjectType: 'account' || 'contract',
 // if it's a contract, it can be an contract
 owner: 'account' || 'contract',: 'account' || 'contract',
 // warnings
 [
 // parent verification does not allow subverifications
 'disableSubVerifications',
 // verification has expired
 'expired',
 // signature does not match requirements, this could be because it hasn't been
 ↪signed by
 // correct account or underlying checksum does not match
 // ``subject``, ``topic`` and ``data``
 'invalid',
 // verification has been issued, but not accepted or rejected by subject
 'issued',
]
 }
]
```

(continues on next page)



(continued from previous page)

```

 // verification has not been issued
 'missing',
 // given subject has no identity
 'noIdentity',
 // verification path has a trusted root verification topic, but this_
 ↪ verification is not
 // signed by a trusted instance
 'notEnsRootOwner',
 // parent verification is missing in path
 'parentMissing',
 // verification path cannot be traced back to a trusted root verification
 'parentUntrusted',
 // verification has been issued and then rejected by subject
 'rejected',
 // verification issuer is the same account as the subject
 'selfIssued',
],
 parents: [...],
 parentComputed: [...]
}
]

```

## 7.6.9 getNestedVerificationsV2

```
getNestedVerificationsV2(subject, topic[, isIdentity, queryOptions]);
```

Get verifications and their parent paths for a specific subject, then format it to updated result format.

### Parameters

1. subject - string: subject (account/contract or identity)
2. topic - string: topic (verification name) to check
3. isIdentity - boolean: true if subject is identity
4. queryOptions - VerificationsQueryOptions: options for query and status computation

### Returns

Promise returns VerificationsResultV2: verification result object with status, verification data and tree

### Example

```

const validationOptions: VerificationsValidationOptions = {
 disableSubVerifications: VerificationsStatusV2.Red,
 expired: VerificationsStatusV2.Red,
 invalid: VerificationsStatusV2.Red,
 issued: VerificationsStatusV2.Yellow,
 missing: VerificationsStatusV2.Red,
}

```

(continues on next page)

(continued from previous page)

```

noIdentity: VerificationsStatusV2.Red,
notEnsRootOwner: VerificationsStatusV2.Yellow,
parentMissing: VerificationsStatusV2.Yellow,
parentUntrusted: VerificationsStatusV2.Yellow,
rejected: VerificationsStatusV2.Red,
selfIssued: VerificationsStatusV2.Yellow,
};
const queryOptions: VerificationsQueryOptions = {
 validationOptions: validationOptions,
};
const nestedVerificationsV2 = await verifications.getNestedVerificationsV2(
 accounts[1], '/example1', false, queryOptions);
console.dir(nestedVerificationsV2);
// Output:
// { verifications:
// [{ details:
// { creationDate: 1561722858000,
// ensAddress:
// '4d2027082fdec4ee253363756eccb1b5492f61fb6329f25d8a7976d7909c10ac.
↳example1.verifications.evan',
// id:
// '0x855a3c10b9cd6d42da5fd5e9b61e0f98a5af79b1acbfee57a9e4f3c9721f9c5d',
// issuer: '0x5035aEe29ea566F3296cdD97C29baB2b88C17c25',
// issuerIdentity: '0xD2860FeC7A198A646f9fD1207B59aD42f00c3189',
// subject: '0x9aE6533e7a2C732863C0aF792D5EA358518cd757',
// subjectIdentity: '0x9F870954c615E4457660D22BE0F38FE0200b1Ed9',
// subjectType: '0x9F870954c615E4457660D22BE0F38FE0200b1Ed9',
// topic: '/example1',
// status: 'green' },
// raw:
// { creationBlock: '224038',
// creationDate: '1561722858',
// data:
// '0x00',
// disableSubVerifications: false,
// signature:
↳'0x941f316d77f5c1dc8b38000ecbb60304554ee2fb36453487ef7822ce6d8c7ce5267bb62396cfb08191028099de2e28d
↳',
// status: 1,
// topic:
//
↳'34884897835812838038558016063403566909277437558805531399344559176587016933548' },
// statusFlags: [] }],
// levelComputed:
// { subjectIdentity: '0x9F870954c615E4457660D22BE0F38FE0200b1Ed9',
// subjectType: 'account',
// topic: '/example1',
// subject: '0x9aE6533e7a2C732863C0aF792D5EA358518cd757' },
// status: 'green' }

```

### 7.6.10 computeVerifications

```
bcService.computeVerifications(topic, verifications);
```

Takes an array of verifications and combines all the states for one quick view.

#### Parameters

1. `topic` - string: topic of all the verifications
2. `verifications` - Array<any>: all verifications of a specific topic

#### Returns

any: computed verification including latest creationDate, displayName

#### Example

```
// load all sub verifications
verification.parents = await verifications.getNestedVerifications(verification.
 ↳issuerAccount, verification.parent || '/', false);

// use all the parents and create a viewable computed tree
const computed = verifications.computeVerifications(verification.topic, verification.
 ↳parents)

// returns =>
// const computed:any = {
// verifications: verifications,
// creationDate: null,
// displayName: topic.split('/').pop() || 'evan',
// loading: verifications.filter(verification => verification.loading).length > 0,
// name: topic,
// status: -1,
// subjects: [],
// warnings: [],
// }
```

### 7.6.11 getComputedVerification

```
getComputedVerification(subject, topic, isIdentity);
```

Loads a list of verifications for a topic and a subject and combines to a single view for a simple verification status check, by combining `getNestedVerifications` with `computeVerifications`.

#### Parameters

1. `subject` - string: subject to load the verifications for.
2. `topic` - string: topic to load the verifications for.

3. `isIdentity` - boolean: optional indicates if the subject is already a identity

## Returns

any: computed verification including latest `creationDate`, `displayName`

## Example

```
// use all the parents and create a viewable computed tree
const computed = verifications.getComputedVerification(subject, topic)

// returns =>
// const computed:any = {
// verifications: verifications,
// creationDate: null,
// displayName: topic.split('/').pop() || 'evan',
// loading: verifications.filter(verification => verification.loading).length > 0,
// name: topic,
// status: -1,
// subjects: [],
// warnings: [],
// }
```

---

## 7.6.12 validateVerification

```
verifications.validateVerification(subject, verificationId, isIdentity]);
```

validates a given `verificationId` in case of integrity

## Parameters

1. `subject` - string: subject of the verifications
2. `verificationId` - string: The verification identifier
3. `isIdentity` - boolean (optional): indicates if the subject is already an identity, defaults to false

## Returns

Promise returns boolean: resolves with true if the verification is valid, otherwise false

## Example

```
console.dir(await verifications.validateVerification(
 accounts[1]),
 '0x00',
);
// Output:
true
```

### 7.6.13 deleteVerification

```
verifications.deleteVerification(accountId, subject, verificationId[, isIdentity]);
```

Delete a verification. This requires the **accountId** to have permissions for the parent verification (if verification name seen as a path, the parent ‘folder’). Subjects of a verification may only delete it, if they are the issuer as well. If not, they can only react to it by confirming or rejecting the verification.

#### Parameters

1. `accountId` - string: account, that performs the action
2. `subject` - string: the subject of the verification
3. `verificationId` - string: id of a verification to delete
4. `isIdentity` - bool (optional): true if given subject is an identity, defaults to false

#### Returns

Promise returns void: resolved when done

#### Example

```
const verificationId = await verifications.setVerification(accounts[0], accounts[1],
 ↳ '/company');
await verifications.deleteVerification(accounts[0], accounts[1], verificationId);
```

### 7.6.14 = Subjects =

### 7.6.15 confirmVerification

```
verifications.confirmVerification(accountId, subject, verificationId[, isIdentity]);
```

Confirms a verification; this can be done, if a verification has been issued for a subject and the subject wants to confirm it.

#### Parameters

1. `accountId` - string: account, that performs the action
2. `subject` - string: verification subject
3. `verificationId` - string: id of a verification to confirm
4. `isIdentity` - bool (optional): true if given subject is an identity, defaults to false

#### Returns

Promise returns void: resolved when done

### Example

```
const newVerification = await verifications.setVerification(accounts[0], accounts[1],
 ↪ '/company');
await verifications.confirmVerification(accounts[0], accounts[1], newVerification);
```

---

## 7.6.16 rejectVerification

```
verifications.rejectVerification(accountId, subject, verificationId[, rejectReason, ↵
 ↪ isIdentity]);
```

Reject a Verification. This verification will be marked as rejected but not deleted. This is important for tracking reasons. You can also optionally add a reject reason as JSON object to track additional informations about the rejection. Issuer and Subject can reject a special verification.

### Parameters

1. `accountId` - string: account, that performs the action
2. `subject` - string: the subject of the verification
3. `verificationId` - string: id of a verification to delete
4. `rejectReason` - object (optional): JSON Object of the rejection reason
5. `isIdentity` - bool (optional): true if given subject is an identity, defaults to false

### Returns

Promise returns void: resolved when done

### Example

```
const verificationId = await verifications.setVerification(accounts[0], accounts[1],
 ↪ '/company');
await verifications.rejectVerification(accounts[0], accounts[1], verificationId, { ↵
 ↪ rejected: "because not valid anymore" });
```

---

## 7.6.17 = Delegated Verifications =

## 7.6.18 signSetVerificationTransaction

```
verifications.signSetVerificationTransaction(issuer, subject, topic[, expirationDate, ↵
 ↪ verificationValue, descriptionDomain, disableSubVerifications, isIdentity, ↵
 ↪ executionNonce]);
```

Signs a verification (offchain) and returns data, that can be used to submit it later on. Return value can be passed to `executeVerification`.

Note that, when creating multiple signed verification transactions, the `nonce` argument **has to be specified and incremented between calls**, as the nonce is included in transaction data and restricts the order of transactions, that can be made.

### Parameters

1. `issuer - string`: issuer of the verification
2. `subject - string`: subject of the verification and the owner of the verification node
3. `topic - string`: name of the verification (full path)
4. `expirationDate - number (optional)`: expiration date, for the verification, defaults to 0 (does not expire)
5. `verificationValue - any (optional)`: json object which will be stored in the verification
6. `descriptionDomain - string (optional)`: domain of the verification, this is a subdomain under 'verifications.evan', so passing 'example' will link verifications description to 'example.verifications.evan', unset if omitted
7. `disableSubVerifications - boolean (optional)`: invalidate all verifications that gets issued as children of this verification (warning will include the `disableSubVerifications` warning)
8. `isIdentity - boolean (optional)`: true if given subject is identity, defaults to `false`
9. `executionNonce - number (optional)`: current execution nonce of issuer identity contract, defaults to -1 (fetch dynamically)
10. `uri - string (optional)`: adds a specific URI to the verification

### Returns

Promise returns `VerificationsDelegationInfo`: data for submitting delegated verifications

### Example

```
// accounts[0] wants to issue a verification for accounts[1] via delegation
const txInfo = await verifications.signSetVerificationTransaction(
 accounts[0], accounts[1], '/company');
```

## 7.6.19 executeVerification

```
verifications.executeVerification(accountId, txInfo);
```

Executes a pre-signed verification transaction with given account. This account will be the origin of the transaction and not of the verification. Second argument is generated with `signSetVerificationTransaction`.

## Parameters

1. accountId - string: account, that submits the transaction
2. txInfo - VerificationsDelegationInfo: information with verification tx data

## Returns

Promise returns string: id of new verification

## Example

```
// accounts[0] wants to issue a verification for accounts[1] via delegation
const txInfo = await verifications.signSetVerificationTransaction(
 accounts[0], accounts[1], '/company');

// accounts[2] submits transaction, that actually issues verification
const verificationId = await verifications.executeVerification(accounts[2], txInfo);
```

## 7.6.20 getExecutionNonce

```
verifications.getExecutionNonce(issuer[, isIdentity]);
```

Gets current execution nonce for an identity or an accounts identity.

Nonce is returned as string. When using nonces for preparing multiple transactions, small nonces can just be parsed to a number and then incremented as needed. Consider using BigNumber or similar modules to deal with large numbers if required.

## Parameters

1. issuer - string: account or identity to get execution nonce for
2. isIdentity - boolean (optional): true if given issuer is an identity, defaults to false

## Returns

Promise returns string: execution nonce

## Example

```
// nonce in this example is relatively small, so we can just parse it and use it as a
↪number
// consider using BigNumber or similar to deal with larger numbers if required
let nonce = JSON.parse(await verifications.getExecutionNonce(accounts[0]));
const txInfos = await Promise.all(['/example1', '/example2', '/example3'].map(
 topic => verifications.signSetVerificationTransaction(
 accounts[0], accounts[1], topic, 0, null, null, false, false, nonce++)
));
```



## 7.6.21 = Delegated Transactions =

### 7.6.22 signTransaction

```
verifications.signTransaction(contract, functionName, options[, ...args]);
```

Signs a transaction from an identity (offchain) and returns data, that can be used to submit it later on. Return value can be passed to `executeTransaction`.

Note that, when creating multiple signed transactions, the `nonce` argument **has to be specified and incremented between calls**, as the nonce is included in transaction data and restricts the order of transactions, that can be made.

#### Parameters

1. `contract` - any: target contract of transaction or null if just sending funds
2. `functionName` - string: function for transaction or null if just sending funds
3. `options` - any: options for transaction, supports from, to, nonce, input, value
4. `args` - any[] (optional): arguments for function transaction

#### Returns

Promise returns `VerificationsDelegationInfo`: prepared transaction for `executeTransaction`

#### Example

```
// create test contract
const testContract = await executor.createContract(
 'TestContract', ['old data'], { from: accounts[0], gas: 500000 });
let data = await executor.executeContractCall(testContract, 'data');

// on account[0]'s side
const txInfo = await verifications.signTransaction(
 testContract,
 'setData',
 { from: accounts[0] },
 'new data',
);

// on account[2]'s side
await verifications.executeTransaction(accounts[2], txInfo);

// now check
data = await executor.executeContractCall(testContract, 'data');
```

### 7.6.23 executeTransaction

```
verifications.executeTransaction(accountId, txInfo);
```

Executes a pre-signed transaction from `signTransaction` of an identity. This can be and usually is a transaction, that has been prepared by the identity owner and is now submitted to the chain and executed by another account.

## Parameters

1. accountId - string: account, that sends transaction to the blockchain and pays for it
2. txInfo - VerificationsDelegationInfo: details about the transaction
3. partialOptions - any (optional): data for handling event triggered by this transaction

## Returns

Promise returns any: result of transaction (if received from event)

## Example

```
// create test contract
const testContract = await executor.createContract(
 'TestContract', ['old data'], { from: accounts[0], gas: 500000 });
let data = await executor.executeContractCall(testContract, 'data');

// on account[0]'s side
const txInfo = await verifications.signTransaction(
 testContract,
 'setData',
 { from: accounts[0] },
 'new data',
);

// on account[2]'s side
await verifications.executeTransaction(accounts[2], txInfo);

// now check
data = await executor.executeContractCall(testContract, 'data');
```

---

## 7.6.24 = Descriptions =

### 7.6.25 setVerificationDescription

```
verifications.setVerificationDescription(accountId, topic, domain, description);
```

Set description for a verification under a domain owned by given account. This sets the description at the ENS endpoint for a verification.

Notice, that this will **not** insert a description at the verification itself. Consider it as setting a global registry with the description for your verifications and not as a label attached to a single verification.

So a setting a description for the verification `/some/verification` the subdomain `example` registers this at the ENS path `$_sha3('/some/verification')example.verifications.evan'`.

When this description has been set, it can be used when setting verifications, e.g. with

```
verifications.setVerification(accounts[0], accounts[1], '/some/verification',
↪expirationDate, verificationValue, 'example');
```

A description can be setup even after verifications have been issued. So it is recommended to use the verification domain when setting up verifications, even if the description isn't required at the moment, when verifications are set up.

## Parameters

1. `accountId` - string: `accountId`, that performs the description update
2. `topic` - string: name of the verification (full path) to set description
3. `domain` - string: domain of the verification, this is a subdomain under 'verifications.evan', so passing 'example' will link verifications description to 'example.verifications.evan'
4. `description` - string: DBCP description of the verification; can be an Envelope but only public properties are used

## Returns

Promise returns void: resolved when done

## Example

```
const sampleVerificationsDomain = 'sample';
const sampleVerificationTopic = '/company';
const sampleDescription = {
 name: 'sample verification',
 description: 'I\'m a sample verification',
 author: 'evan.network',
 version: '1.0.0',
 dbcpVersion: 1,
};
await verifications.setVerificationDescription(accounts[0], sampleVerificationTopic,
↪sampleVerificationsDomain, sampleDescription);
await verifications.setVerification(accounts[0], accounts[1], sampleVerificationTopic,
↪ null, null, sampleVerificationsDomain);
const verificationsForAccount = await verifications.getVerifications(accounts[1],
↪sampleVerificationTopic);
const last = verificationsForAccount.length - 1;
console.dir(verificationsForAccount[last].description);
// Output:
// {
// name: 'sample verification',
// description: 'I\'m a sample verification',
// author: 'evan.network',
// version: '1.0.0',
// dbcpVersion: 1,
// }
```

### 7.6.26 getVerificationEnsAddress

```
verifications.getVerificationEnsAddress(topic);
```

Map the topic of a verification to its default ens domain.

### Parameters

1. `topic - string`: verification topic

### Returns

`string`: The verification ens address

### Example

```
const ensAddress = verifications.getVerificationEnsAddress('/evan/test');
// will return test.verifications.ens
```

---

## 7.6.27 ensureVerificationDescription

```
verifications.ensureVerificationDescription(verification);
```

Gets and sets the default description for a verification if it does not exists.

### Parameters

1. `verification - any`: verification topic

### Example

```
verifications.ensureVerificationDescription(verification);
```

---

## 7.6.28 = Utilities =

## 7.6.29 createStructure

```
verifications.createStructure(accountId);
```

Create a new verifications structure; this includes a userregistry and the associated libraries. This isn't required for creating a module instance, its is solely used for creating new structures on the blockchain.

### Parameters

1. `accountId - string`: account, that execute the transaction and owner of the new registry

### Returns

`Promise` returns `any`: object with property 'storage', that is a web3js contract instance

### Example

```
const verificationsStructure = await verifications.createStructure(accountId);
console.log(verificationsStructure.storage.options.address);
// Output:
// 0x000a
```

## 7.6.30 deleteFromVerificationCache

```
verifications.deleteFromVerificationCache(subject, topic);
```

Delete a single entry from the verification cache object using subject and topic

### Parameters

1. subject - string: the subject that should be removed
2. topic - string: the topic that should be removed

### Example

```
verifications.deleteFromVerificationCache(accounts[1], '/some/verification');
```

## 7.6.31 trimToStatusTree

```
verifications.trimToStatusTree(result);
```

Trim VerificationsResultV2 result down to statusFlags and status values for analysis purposes and debugging.

### Parameters

1. inputResult - VerificationsResultV2: result to trim down

### Returns

Promise returns any: trimmed down tree

### Example

```
const validationOptions: VerificationsValidationOptions = {
 disableSubVerifications: VerificationsStatusV2.Red,
 expired: VerificationsStatusV2.Red,
 invalid: VerificationsStatusV2.Red,
 issued: VerificationsStatusV2.Yellow,
 missing: VerificationsStatusV2.Red,
 noIdentity: VerificationsStatusV2.Red,
 notEnsRootOwner: VerificationsStatusV2.Yellow,
```

(continues on next page)

(continued from previous page)

```

parentMissing: VerificationsStatusV2.Yellow,
parentUntrusted: VerificationsStatusV2.Yellow,
rejected: VerificationsStatusV2.Red,
selfIssued: VerificationsStatusV2.Yellow,
};
const queryOptions: VerificationsQueryOptions = {
 validationOptions: validationOptions,
};
const nestedVerificationsV2 = await verifications.getNestedVerificationsV2(
 accounts[1], '/example1', false, queryOptions);
console.dir(nestedVerificationsV2);
// Output:
// { verifications:
// [{ details:
// { creationDate: 1561722858000,
// ensAddress:
// '4d2027082fdec4ee253363756eccb1b5492f61fb6329f25d8a7976d7909c10ac.
↪example1.verifications.evan',
// id:
// '0x855a3c10b9cd6d42da5fd5e9b61e0f98a5af79b1acbfee57a9e4f3c9721f9c5d',
// issuer: '0x5035aEe29ea566F3296cdD97C29baB2b88C17c25',
// issuerIdentity: '0xD2860FeC7A198A646f9fD1207B59aD42f00c3189',
// subject: '0x9aE6533e7a2C732863C0aF792D5EA358518cd757',
// subjectIdentity: '0x9F870954c615E4457660D22BE0F38FE0200b1Ed9',
// subjectType: '0x9F870954c615E4457660D22BE0F38FE0200b1Ed9',
// topic: '/example1',
// status: 'green' },
// raw:
// { creationBlock: '224038',
// creationDate: '1561722858',
// data:
// '0x00',
// disableSubVerifications: false,
// signature:
// '0x941f316d77f5c1dc8b38000ecbb60304554ee2fb36453487ef7822ce6d8c7ce5267bb62396cfb08191028099de2e28d
↪',
// status: 1,
// topic:
// '34884897835812838038558016063403566909277437558805531399344559176587016933548' },
// statusFlags: [] }],
// levelComputed:
// { subjectIdentity: '0x9F870954c615E4457660D22BE0F38FE0200b1Ed9',
// subjectType: 'account',
// topic: '/example1',
// subject: '0x9aE6533e7a2C732863C0aF792D5EA358518cd757' },
// status: 'green' }

const trimmed = verifications.trimToStatusTree(nestedVerificationsV2);
console.dir(trimmed);
// Output:
// { status: 'green',
// verifications:
// [{ details: { status: 'green', topic: '/example1' },
// statusFlags: [] }] }

```

## 7.6.32 Enums

### VerificationsStatus

verification status from blockchain

1. **Issued:** issued by a non-issuer parent verification holder, self issued state is 0
2. **Confirmed:** issued by a non-issuer parent verification holder, self issued state is 0
3. **Rejected:** verification rejected status

### VerificationsStatusFlagsV2

status annotations about verification, depending on defined `VerificationsQueryOptions`, this may lead to the verification to be invalid or less trustworthy

1. **disableSubVerifications:** parent verification does not allow subverifications
2. **expired:** verification has expired
3. **invalid:** signature does not match requirements, this could be because it hasn't been signed by correct account or underlying checksum does not match subject, topic and data
4. **issued:** verification has been issued, but not accepted or rejected by subject
5. **missing:** verification has not been issued
6. **noIdentity:** given subject has no identity
7. **notEnsRootOwner:** verification path has a trusted root verification topic, but this verification is not signed by a trusted instance
8. **parentMissing:** parent verification is missing in path
9. **parentUntrusted:** verification path cannot be traced back to a trusted root verification
10. **rejected:** verification has been issued and then rejected by subject
11. **selfIssued:** verification issuer is the same account as the subject

### VerificationsStatusV2

represents the status of a requested verification topic after applying rules in `VerificationsQueryOptions`

1. **Green:** verification is valid according to `VerificationsQueryOptions`
  2. **Yellow:** verification may be valid but more checks may be required more for trusting it, see status flags for details
  3. **Red:** verification is invalid, see status flags for details
- 

## 7.6.33 Interfaces

### VerificationsDelegationInfo

information for submitting a delegated transaction, created with `signSetVerificationTransaction` consumed by `executeVerification`

1. `sourceIdentity` - string: address of identity contract, that issues verification
2. `value` - number: value to transfer, usually 0
3. `input` - string: abi encoded input for transaction
4. `signedTransactionInfo` - string: signed data from transaction
5. `nonce` - “string”(optional): source identity contract execution nonce for this transaction
6. `targetIdentity` - string``(optional): address of identity contract, that receives verification, either this or ``to has to be given
7. `to` - string (optional): address of target of transaction, either this or *targetIdentity* has to be given

## VerificationsQueryOptions

options for `getNestedVerificationsV2`, define how to calculate status of verification

1. `statusComputer` - `VerificationsStatusComputer` (optional): function for setting verification with custom logic
2. `validationOptions` - `VerificationsValidationOptions` (optional): specification of how to handle status flags of each single verification

## VerificationsResultV2

result of a verification query

1. `status` - `VerificationsStatusV2`: overall status of verification
2. `verifications` - `VerificationsVerificationEntry[]`: (optional): list of verifications on same topic and subject
3. **`levelComputed` (optional): consolidated information about verification**
  - `subjectIdentity` - string: identity contract address or hash of subject
  - `subjectType` - string: type of subject (account/contract)
  - `topic` - string: topic (name) of verification
  - `expirationDate` - number (optional): timestamp, when verification will expire, js timestamp
  - `parents` - `VerificationsResultV2` (optional): verifications of parent path, issued for all issuers of verifications on this level
  - `subject` - number (optional): subject accountId/contractId (if query was issued with `isIdentity` set to false)

## VerificationsVerificationEntry

a single verification; usually used in `VerificationsResultV2`

1. **`details`: details about verification**
  - `creationDate` - number: js timestamp of verification creation
  - `ensAddress` - string: ens address of description for this verification
  - `id` - string: id in verification holder / verifications registry
  - `issuer` - string: account id of verification issuer



- `issuerIdentity` - string: issuers identity contract id
- `subjectIdentity` - string: identity (contract or identity hash) of subject
- `subjectType` - string: type of subject (account/contract)
- `topic` - string: topic of identity (name)
- `data` - any (optional): 32B data hash string of identity
- `description` - any (optional): only if actually set
- `expirationDate` - number (optional): expiration date of verification (js timestamp)
- `rejectReason` - string (optional): if applicable, reason for verification rejection
- `status` - `VerificationsStatusV2` (optional): status of verification, is optional during result computation and required when done
- `subject` - string (optional): subject accountId/contractId (if query was issued with `isIdentity` set to false)

2. **raw (optional): raw data about verification from contract**

- `creationBlock` - string: block in which verification was issued
- `creationDate` - string: unix timestamp is s when verification was issued
- `data` - string: 32B data hash string of identity, bytes32 zero if unset
- `disableSubVerifications` - boolean: true if subverification are not allowed
- `signature` - string: signature over verification data
- `status` - number: status of verification, (issued, accepted, rejected, etc.)
- `topic` - string: uint string of verification name (topic), is uint representation of sha3 of name

3. `statusFlags` - string[] (optional): all found flags, those may not have impact on status, depends on `VerificationsStatusFlagsV2`

## VerificationsVerificationEntryStatusComputer

Computes status for a single verification. verification, partialResult.

### Parameters

1. `verification` - `Partial<VerificationsVerificationEntry>`: current verification result (without status)
2. `partialResult` - `Partial<VerificationsResultV2>`: options for verifications query

### Returns

Promise returns `VerificationsStatusV2`: status for this verification

## VerificationsStatusComputer

Computes status from overall verifications result. This function is applied after each verification has received an own computed status.

## Parameters

1. `partialResult` - `Partial<VerificationsResultV2>`: current verification result (without status)
2. `queryOptions` - `VerificationsQueryOptions`: options for verifications query
3. `currentStatus` - `VerificationsStatusV2`: current status of verification

## Returns

Promise returns `Promise<VerificationsStatusV2>`: updated status, will be used at verification status

## VerificationsValidationOptions

Options for verification status computation. Keys are string representations of `VerificationsStatusFlagsV2`, values can be `VerificationsStatusV2` or functions. If value is `VerificationsStatusV2`, then finding given status flag sets verification value to given `VerificationsStatusV2` (if not already at a higher trust level). If value is function, pass verification to this function and set verification status to return value (if not already at a higher trust level).

1. `[id: string] - VerificationsStatusV2 | VerificationsVerificationEntryStatusComputer`: each key is a status flag and usually an enum value from `VerificationsStatusFlagsV2`, though it is possible to use custom status flags

## 7.7 Verifications Examples

This sections aims to help getting started with the verifications service. Herein are shown a few examples on

- how to set verifications
- how to get verifications
- how to handle verifications for externally owned accounts and contracts
- how to check verifications for validity

Verifications can be issued with the *Verification API*. Their smart contract implementation follow the principles outlined in [ERC-725](#) and [ERC-735](#).

---

### 7.7.1 About the Code Examples

Many code examples are taken from the [verification tests](#). You can a look at those for more examples or to have a look at in which context the test are run in.

Code examples on this page use an initialized module called `verifications`, which is an instance of the verification module *Verification API*. When using a *runtime* this module is available as `runtime.verifications`.

Many code examples here use variable naming from the tests. So there are a few assumptions about variables used here:

- `accounts` is an array with addresses of externally owned accounts, of which private keys are known to the runtime/executor/signer instance to make transaction with them
- `accounts[0]` usually takes the role of issuing verifications

- `accounts[1]` is usually an account that responds to actions from `accounts[0]`
- `accounts[2]` is usually an account, that has no direct relation to the former two accounts
- `contractId` refers to the address of a contract, that is owned by `accounts[0]`, this contract usually has a [DBCP description](#)

## 7.7.2 Different Types of Identities

Okay, this page is about verifications, so why does the first heading talk about some kind of identities? The answer for this is pretty simple: Verifications are issued by identities.

So what exactly is an identity? An identity is an instance, that is able to issue and hold verifications (therefore the technical name [VerificationHolder](#)).

Technically speaking, identities are like Wallet Contracts, that have a certificate store attached to them. Identities can be traced back to an externally owned account, that owns this wallet contract and vice versa the matching identity (wallet) contract can be looked up for any given externally owned account, if this account has created and registered an identity contract.

Because of their wallet-like behavior, [identity contracts](#) require a calling instance, for example an externally owned account or another wallet to call them for managing their verifications. For example to issue a verification for another identity, a user makes a transaction against its own identity contract, which itself makes a transaction to create a verification in the other identity.

This works pretty well and comes with useful features like preparing “outgoing” transactions to be made from ones own identity, that can be paid by another party, but has some implications when applied as “smart contract identities” instead of “account identities”. These smart contracts would need to be owner of their own identities and working with verifications would require the contracts to have functions for issuing, rejecting, etc. verification or offer wallet like functionalities themselves, which would make it often pretty hard to introduce verifications in an existing smart contract infrastructure and it would often have an impact on these contracts costs as these extra functionalities have to be added to the contract.

To allow for a more easy integration, contract identities have been added as a 32 Bytes identities or pseudonyms, which can be created with a registry contract. These 32 Bytes identities are IDs in a central [registry](#), that offers basically the same functionalities as the single [identity contracts](#). Identities at the registry are owned by the account, that created them (ownership can be transfered) and ownership over such an identity grants the permission to make transactions for it the same way as for the own account identity.

So let's get to the code examples.

### Account Identities

Note that the creation example can be omitted when using an account creating during the [visual onboarding flow](#), as this includes creating an identity.

If you want to check if your account already has an identity, you can try to get its identity contracts address with [identityAvailable](#):

```
if (await verifications.identityAvailable(accounts[0])) {
 console.log(`account "${accounts[0]}" has an identity`);
} else {
 console.log(`account "${accounts[0]}" does not have an identity`);
}
```

Account identities can be created straight forward by calling `createIdentity` and passing the account to it, an identity should be created for.

```
const identity = await verifications.createIdentity(accounts[0]);
console.log(identity);
// Output:
// 0x1fE5F7235f1989621135466Ff8882287C63A5bae
```

This returns the 40Bytes contract address of the accounts identity contract.

The given account now has an identity attached to it, which is a requirement for interacting with the rest of the `verifications API` and only has to be done once per externally owned account.

## Contract Identities / Pseudonym Identities

Contract identities are created “on behalf of” a contract. An externally owned account, often the owner of the contract, usually does the following:

- creating an identity for the contract
- linking contract and identity
- providing the information of which identity belongs to a contract to other parties

Creating a contract identity registers a new contract identity at the `registry`, this identity is then owned by the executing `accountId`.

Linking is done by registering the contract address as the receiver of the identity, this is done at the `registry` as well.

When third parties want to check verifications of a contract, they need a way to get the identity for a contract. This can be done by adding an extra function to the contract, by setting up a lookup mechanism, etc. `DBCP` is a description language used throughout `evan.network`, which provides a way to add meta information to smart contracts. The contract identity is usually added to this `DBCP description` of a contract.

The aforementioned three steps are covered by the `createIdentity` function, which can be called with:

```
const contractIdentity = await verifications.createIdentity(accounts[0], contractId);
console.log(idencontractIdentity);
// Output:
// 0x4732281e708aadb3e13f0bf4dd616de86df3d3edb3ead21604a354101de45316
```

When using contracts without descriptions or when handling the relation between contracts and an identity elsewhere, the process of updating the description can be omitted. For this set the `updateDescription` argument to `false`:

```
const contractIdentity = await verifications.createIdentity(accounts[0], contractId,
↪ false);
console.log(idencontractIdentity);
// Output:
// 0x4732281e708aadb3e13f0bf4dd616de86df3d3edb3ead21604a354101de45316
```

Pseudonyms can be handled the same way. Just set the flag to link given identity to `false`:

```
const contractIdentity = await verifications.createIdentity(accounts[0], null, false,
↪ false);
console.log(idencontractIdentity);
// Output:
// 0x4732281e708aadb3e13f0bf4dd616de86df3d3edb3ead21604a354101de45316
```

This returns an identity, that is owned by `accountId` and can be used to issue verifications for.

### 7.7.3 Issue verifications

Verifications are statements, issued by an account called `issuer`, towards target, called `subject`. This basically means something like “The person `issuer` says, a statement applies to `subject`”. The subject may or may not react to it by confirming or rejecting it. Technically speaking, the `issuer` identity issues the verification with the statement to `subject`’s identity and the `subject`’s identity may react to it.

#### Issue verifications for an account

```
const verificationId = await verifications.setVerification(
 accounts[0], accounts[1], '/example1');
console.log(verificationId);
// Output:
// 0xb4843ed5177433312dd2c7c4f8065ce84f37bf96c04db2775c16c9455ad96270

const issued = await verifications.getVerifications(accounts[1], '/example1');
console.dir(issued);
// Output:
// [{
// creationBlock: '186865',
// creationDate: '1558599441',
// data: '0x00',
// description: null,
// disableSubVerifications: false,
// expirationDate: null,
// expired: false,
// id: '0xb4843ed5177433312dd2c7c4f8065ce84f37bf96c04db2775c16c9455ad96270',
// issuer: '0xe560eF0954A2d61D6006E8547EC769fAc322bbCE',
// name: '/example1',
// rejectReason: undefined,
// signature:
 ↪ '0x6a2b41714c1faac09a5ec06024c8931ad6e3aa902c502e3d1bc5d5c4577288c04e9be136c149b569e0456dfec9d50a2',
 ↪ ',
// status: 0,
// subject: '0x0030C5e7394585400B1FB193DdbCb45a37Ab916E',
// topic:
 ↪ '34884897835812838038558016063403566909277437558805531399344559176587016933548',
// uri: '',
// valid: true
// }]
```

Have a look at [getVerifications](#) or the section on [this page](#) for the meaning of the returned values, for how to find out, if the returned verification is trustworthy, have a look at [Validating Verifications](#).

#### Issue verifications for a contract with a description

```
const verificationId = await verifications.setVerification(
 accounts[0], contractId, '/example2');
console.log(verificationId);
// Output:
// 0x2bc6d5fdb937f6808252b837437220d8e16b92a974367f224260d028413e7c6e
```

(continues on next page)

(continued from previous page)

```

const issued = await verifications.getVerifications(contractId, '/example2');
console.dir(issued);
// [{
// creationBlock: '187823',
// creationDate: '1558621998',
// data: '0x00',
// description: null,
// disableSubVerifications: false,
// expirationDate: null,
// expired: false,
// id: '0x2bc6d5fdb937f6808252b837437220d8e16b92a974367f224260d028413e7c6e',
// issuer: '0xe560eF0954A2d61D6006E8547EC769fAc322bbCE',
// name: '/example2',
// rejectReason: undefined,
// signature:
 ↪ '0x0f4f19a369645a0ec2795bd2836fad0857ef82169c7e5800d7a06fb162583c9c14a731f4e942cf30d67fb10a551d906',
 ↪,
// status: 0,
// subject: '0x005C5FF57D4d6Bf105Bf3bF16ffCd8Ac143B3Ef0',
// topic:
 ↪ '107276559880603231420598591656057035604273757486333915273364042567965107775848',
// uri: '',
// valid: true
// }]

```

Have a look at [getVerifications](#) or the section on [this page](#) for the meaning of the returned values, for how to find out, if the returned verification trustworthy, have a look at [Validating Verifications](#).

Note that for contracts with descriptions the contractId can be given to setVerification and getVerifications. The contract identity is fetched from the contract description automatically.

### Issue verifications for a contract without using a description

```

// assume, we have created an identity for our contract and stored this identity as
 ↪ the variable `contractIdentity`
const verificationId = await verifications.setVerification(
 accounts[0], contractIdentity, '/example3', 0, null, null, false, true);
console.log(verificationId);
// Output:
// 0x2bc6d5fdb937f6808252b837437220d8e16b92a974367f224260d028413e7c6e

const issued = await verifications.getVerifications(contractIdentity, '/example3',
 ↪ true);
console.dir(issued);
// [{
// creationBlock: '187823',
// creationDate: '1558621998',
// data: '0x00',
// description: null,
// disableSubVerifications: false,
// expirationDate: null,
// expired: false,
// id: '0x2bc6d5fdb937f6808252b837437220d8e16b92a974367f224260d028413e7c6e',
// issuer: '0xe560eF0954A2d61D6006E8547EC769fAc322bbCE',

```

(continues on next page)

(continued from previous page)

```
// name: '/example2',
// rejectReason: undefined,
// signature:
↪ '0x0f4f19a369645a0ec2795bd2836fad0857ef82169c7e5800d7a06fb162583c9c14a731f4e942cf30d67fb10a551d906
↪ ',
// status: 0,
// subject: '0x005C5FF57D4d6Bf105Bf3bF16ffCd8Ac143B3Ef0',
// topic:
↪ '107276559880603231420598591656057035604273757486333915273364042567965107775848',
// uri: '',
// valid: true
// }]
```

In case you're wondering: `contractIdentity` is the same identity as returned in our Contract Identities / Pseudonym Identities example.

Have a look at [getVerifications](#) for the meaning of the returned values, for how to find out, if the returned verification trustworthy, have a look at [Validating Verifications](#).

Note that for contracts without descriptions `contractIdentity` is given and the last argument (`isIdentity`) is set to true. The functions `setVerification` and `getVerifications` support passing a contract identity to them as well and they also have the argument `isIdentity`, which is set to true, when passing contract identities to them.

## Delegated verification issuing

The transaction that issues a verification can be done by an account, that is neither `issuer` nor `subject`. This means, that it is possible to let another account pay transaction costs but issuing the verification itself is done from the original identity.

For example: Alice wants to issue a verification to Bob, but should not pay for the transaction costs. Alice can now prepare the transaction to be done from her identity contract towards Bobs identity contract and send the prepared transaction data to Clarice. Clarice then can submit this data to Alice's identity contract, which will issue the verification.

```
const [alice, bob, clarice] = accounts;

// on Alice's side
const txInfo = await verifications.signSetVerificationTransaction(alice, bob, '/
↪ example');

// on Clarice's side
const verificationId = await verifications.executeVerification(clarice, txInfo);
```

Note that transactions prepared with `signSetVerificationTransaction` can only be executed once and only with the arguments of the original data. To prevent multiple repetitions of the transaction, a nonce at the issuers identity contract is used. This nonce is retrieved from the identity contract automatically when calling `signSetVerificationTransaction`, but when preparing multiple transactions and not submitting them immediately, the nonce would stay the same. Therefore the nonce has to be increased by hand when preparing multiple transactions from the same identity contract.

Nonces determine the order in which prepared transactions can be performed from issuers identity contract, so execute prepared transactions in order of their nonces.

```

const [alice, bob, clarice] = accounts;

// on Alice's side
// nonce in this example is relatively small, so we can just parse it and use it as a
↳number
// consider using BigNumber or similar to deal with larger numbers if required
let nonce = JSON.parse(await verifications.getExecutionNonce(alice));
const txInfos = await Promise.all(['/example1', '/example2', '/example3'].map(
 topic => verifications.signSetVerificationTransaction(
 alice, bob, topic, 0, null, null, false, false, nonce++
));

// on Clarice's side
const verificationIds = [];
for (let txInfo of txInfos) {
 verificationIds.push(await verifications.executeVerification(clarice, txInfo));
}

```

## 7.7.4 Validating Verifications

Verifications can be retrieved with two different functions:

- *getVerifications*: simple “fetch all” verifications for a topic, returns all verifications and detailed validity checks have to be made by hand
- *getNestedVerification*: return verifications with default checks and inspects parent verifications as well, used for verifications, that should be traced back to a trusted root verifier

### getVerifications

The example for *getVerifications* is the same we used when creating a verification for and account:

```

const verificationId = await verifications.setVerification(
 accounts[0], accounts[1], '/example1');
console.log(verificationId);
// Output:
// 0xb4843ed5177433312dd2c7c4f8065ce84f37bf96c04db2775c16c9455ad96270

const issued = await verifications.getVerifications(accounts[1], '/example1');
console.dir(issued);
// Output:
// [{
// creationBlock: '186865',
// creationDate: '1558599441',
// data: '0x00',
// description: null,
// disableSubVerifications: false,
// expirationDate: null,
// expired: false,
// id: '0xb4843ed5177433312dd2c7c4f8065ce84f37bf96c04db2775c16c9455ad96270',
// issuer: '0xe560eF0954A2d61D6006E8547EC769fAc322bbCE',
// name: '/example1',
// rejectReason: undefined,

```

(continues on next page)



(continued from previous page)

```
// signature:
→ '0x6a2b41714c1faac09a5ec06024c8931ad6e3aa902c502e3d1bc5d5c4577288c04e9be136c149b569e0456dfec9d50a2'
→ ',
// status: 0,
// subject: '0x0030C5e7394585400B1FB193DdbCb45a37Ab916E',
// topic:
→ '34884897835812838038558016063403566909277437558805531399344559176587016933548',
// uri: '',
// valid: true
// }]
```

As stated above, only basic validations have been made on the data of the verifications, so conclusions have to be drawn by based on the data returned here. For a full list of explanations to the properties have a look at the [API documentation](#), but the ones you will be most probably using the most are:

- status and rejectReason:
  - status - number:
    - 0 (Issued) || 1 (Confirmed) || 2 (Rejected)
    - reflects how the subject responded (1|2) to the verification or if no response has been made (0)
    - rejectReason - any: object with information from subject about rejection
- valid - boolean:
  - true if issuer has been correctly confirmed as the signer of signature
  - also checks if provided signature has been correctly built as checksum over subject, topic and data
- expired and expirationDate:
  - expired - boolean: ticket expiration state
  - expirationDate - string: UNIX timestamp (in seconds), null if verification does not expire
- issuer - string:
  - account address of issuers identity contract, can be used to check if the issuer is an account, that you trust
- data and uri:
  - data - string: 32Bytes hash of data stored in DFS
  - uri - string: link to ipfs file of data
  - these two properties point to data, that has been attached to your verification (attaching data is optional)
  - the data referred here is the data provided as verificationValue in [setVerification](#)
  - data content handling, especially encryption and key management has be be handled in custom logic and is not covered in here

A sample, on how these properties can be used to determine the trustworthiness of a verification can be found at [hem workshop project](#).

## getNestedVerifications

For this section we take the last example and issue two subverifications. We add /example1/example1\_child as the direct child of it and /example1/example1\_child/example1\_grandchild as a subverification below the first this child.

```

const verificationId = await verifications.setVerification(
 accounts[0], accounts[0], '/example4');
const verificationId = await verifications.setVerification(
 accounts[0], accounts[0], '/example4/child');
const verificationId = await verifications.setVerification(
 accounts[0], accounts[0], '/example4/child/grandchild');

const issued = await verifications.getNestedVerifications(accounts[0], '/example1/
↳example1_child/example1_grandchild');
console.dir(issued);
// Output:
// [{
// name: '/example4/child/grandchild',
// parent: '/example4/child',
// warnings: ['issued', 'selfIssued', 'parentUntrusted'],
// id: '0x1ade7f60f5a8d153aeeda7a6e4f8c950fa93b0cb5d3218c6a9389cd05f5f7f6',
// issuer: '0xe560eF0954A2d61D6006E8547EC769fAc322bbCE',
// status: 0,
// subject: '0x001De828935e8c7e4cb56Fe610495cAe63fb2612',
// subjectIdentity: '0xe560eF0954A2d61D6006E8547EC769fAc322bbCE',
// subjectType: 'account',
// issuerAccount: '0x001De828935e8c7e4cb56Fe610495cAe63fb2612',
// parents:
// [{
// name: '/example4/child',
// parent: '/example4',
// warnings: ['issued', 'selfIssued', 'parentUntrusted'],
// id: '0x28e1df758883bb3d4d5e7e0fa978ff673bc749ade0a3d78ad952a30d0a0e2a01',
// issuer: '0xe560eF0954A2d61D6006E8547EC769fAc322bbCE',
// status: 0,
// subject: '0x001De828935e8c7e4cb56Fe610495cAe63fb2612',
// subjectIdentity: '0xe560eF0954A2d61D6006E8547EC769fAc322bbCE',
// subjectType: 'account',
// issuerAccount: '0x001De828935e8c7e4cb56Fe610495cAe63fb2612',
// parents:
// [{
// name: '/example4',
// parent: '',
// warnings: ['issued'],
// id:
// ↳'0x18fb0ef05d96cba2a57c6de6d8cfd031e16367f6484f20797a39d25a3e76e20a',
// issuer: '0xe560eF0954A2d61D6006E8547EC769fAc322bbCE',
// status: 0,
// subject: '0x001De828935e8c7e4cb56Fe610495cAe63fb2612',
// subjectIdentity: '0xe560eF0954A2d61D6006E8547EC769fAc322bbCE',
// subjectType: 'account',
// issuerAccount: '0x001De828935e8c7e4cb56Fe610495cAe63fb2612',
// parents: [],
// }],
// }],
// }],
// }]

```

The output above has been heavily trimmed down to show differences between both functions and highlight parent to child relations and warnings. To view full output have a look at the [full output](#).

To create a simple chain of verifications, we have used the following structure: - `accounts[0]` creates a verification for itself, called `/example4` - then creates a subverification under called `/example4/child` for itself under this - then creates another subverification (under the first subverification) called `/example4/child/grandchild` for

itself

The call `verifications.getNestedVerifications(accounts[0], '/example1/example1_child/example1_grandchild')` now inspects what verifications and possible relations to parent verifications exists and it finds the following possible issues:

- `/example4/child/grandchild:` `warnings: [ 'issued', 'selfIssued', 'parentUntrusted' ]`
  - `issued` means, that is is only issued and not confirmed, we can see its status is 0, so yes, it is unconfirmed (and if we look above, we actually didn't confirm the verification)
  - `selfIssued`, yes, `issuer` equals `subjectIdentity`, therefore `selfIssued` and thinking back, we did issue this verification to ourself
  - `parentUntrusted`, this means that the parent verification hasn't been accepted, its status is 0, so yes, only issued and not confirmed
- following the parent verifications, we find basically the same on the next level
  - `/example4/child`:` `warnings: [ 'issued', 'selfIssued', 'parentUntrusted' ]`
  - the same reasons and explanations apply here, so let's continue to the last on in the verification path
- `/example4: warnings: [ 'issued' ]`
  - `issued`: yep, status is 0 therefore it is only issued and not confirmed
  - no `parentUntrusted`? as this verification is a root verification, there is not parent
  - no `selfIssued`?
    - \* the path is `/example4`, which makes this verification a root verification
    - \* root verifications can be issued by any party without being flagged as `selfIssued`, to allow building own verification chains
    - \* to narrow this down to a limited set of trusts, there are basically two solutions:
      - own checks can be made, e.g. check if the issuer of the root verification is a well known and trusted account
      - use `/evan` derived verification paths, the root verification `/evan` is only trusted, if it is issued by a trusted root issuer, get in contact with us via [info@evan.team](mailto:info@evan.team) for details on how to obtain a subverification like `/evan/myOwnTrustedVerification`, that can be used for building widely accepted verification paths

## getNestedVerificationsV2

The output from the last example has some aspects, where it could perform better:

- the output is a bit lengthy and looks unstructured, some of the information here may not be useful in most situations and relies on predefined defaults (e.g. a default description, that isn't actually set is returned if no description is defined, which may lead to the opinion, that a description has been set)
- it is quite hard to determine with a simple query and not further processing to determine if a valid verification is present or not
- documentation about properties returned in verification is a bit sparse in some points

An updated version of the `getNestedVerifications` has been added as `getNestedVerificationsV2`. This version will replace the current one soon, but is available for now under the V2 name. This version is under

development and may undergo further changes but the basic behavior will not change and it will replace the regular one at some point of time.

A usage example:

```
const validationOptions: VerificationsValidationOptions = {
 disableSubVerifications: VerificationsStatusV2.Red,
 expired: VerificationsStatusV2.Red,
 invalid: VerificationsStatusV2.Red,
 issued: VerificationsStatusV2.Yellow,
 missing: VerificationsStatusV2.Red,
 noIdentity: VerificationsStatusV2.Red,
 notEnsRootOwner: VerificationsStatusV2.Yellow,
 parentMissing: VerificationsStatusV2.Yellow,
 parentUntrusted: VerificationsStatusV2.Yellow,
 rejected: VerificationsStatusV2.Red,
 selfIssued: VerificationsStatusV2.Yellow,
};
const queryOptions: VerificationsQueryOptions = {
 validationOptions: validationOptions,
};
const nestedVerificationsV2 = await verifications.getNestedVerificationsV2(
 accounts[1], '/example1', false, queryOptions);
console.dir(nestedVerificationsV2);
// Output:
// { verifications:
// [{ details:
// { creationDate: 1561722858000,
// ensAddress:
// '4d2027082fdec4ee253363756eccb1b5492f61fb6329f25d8a7976d7909c10ac.'
// },
// id:
// '0x855a3c10b9cd6d42da5fd5e9b61e0f98a5af79b1acbfee57a9e4f3c9721f9c5d',
// issuer: '0x5035aEe29ea566F3296cdD97C29baB2b88C17c25',
// issuerIdentity: '0xD2860FeC7A198A646f9fD1207B59aD42f00c3189',
// subject: '0x9aE6533e7a2C732863C0aF792D5EA358518cd757',
// subjectIdentity: '0x9F870954c615E4457660D22BE0F38FE0200b1Ed9',
// subjectType: '0x9F870954c615E4457660D22BE0F38FE0200b1Ed9',
// topic: '/example1',
// status: 'green' },
// raw:
// { creationBlock: '224038',
// creationDate: '1561722858',
// data:
// '0x00',
// disableSubVerifications: false,
// signature:
// '0x941f316d77f5c1dc8b38000ecbb60304554ee2fb36453487ef7822ce6d8c7ce5267bb62396cfb08191028099de2e28d...'
// },
// status: 1,
// topic:
// '34884897835812838038558016063403566909277437558805531399344559176587016933548' },
// statusFlags: [] }],
// levelComputed:
// { subjectIdentity: '0x9F870954c615E4457660D22BE0F38FE0200b1Ed9',
// subjectType: 'account',
```

(continues on next page)

(continued from previous page)

```
// topic: '/example1',
// subject: '0x9aE6533e7a2C732863C0aF792D5EA358518cd757' },
// status: 'green' }
```

The variable `validationOptions` from the example is a set of rules for about how to interpret the `statusFlags` from the verifications. The last example no flags, but possible issues are tracked as status flags and are evaluated by given rules. The rules are explained in the respective interface and mostly match the warnings explained in the section below.

## 7.7.5 Warnings in Verifications

`getNestedVerification` returns a set of different warnings, that can be used to decide if a certification is valid or not. Those warnings are stored in the `.warnings` property, warnings, that can be returned are:

- `disableSubVerifications`: parent verification does not allow subverifications
- `expired`: verification has expired
- `invalid`: signature does not match requirements, this could be because it hasn't been signed by correct account or underlying checksum does not match subject, topic and data
- `issued`: verification has been issued, but not accepted or rejected by subject
- `missing`: verification has not been issued
- `noIdentity`: given subject has no identity
- `notEnsRootOwner`: verification path has a trusted root verification topic, but this verification is not signed by a trusted instance
- `parentMissing`: parent verification is missing in path
- `parentUntrusted`: verification path cannot be traced back to a trusted root verification
- `rejected`: verification has been issued and then rejected by subject
- `selfIssued`: verification issuer is the same account as the subject

## 7.7.6 Data in Verifications

### Unencrypted Data in Verifications

Additional data can be given when creating a verification. For this pass an object, that can be serialized to JSON as the `verificationValue` argument to `setVerification`. As this argument is placed after the `expirationDate` argument, we set this argument as well.

```
const verificationId = await verifications.setVerification(
 accounts[0], accounts[1], '/example1', 0, { foo: 'bar' });
console.log(verificationId);
// Output:
// 0x5ea689a7ed1d56d948dc8223dcd60866746bc7bea47617c19b63df75d63c9194

const issued = await verifications.getVerifications(accounts[1], '/example1');
```

(continues on next page)

(continued from previous page)

```

console.dir(issued);
// Output:
// [{ creationBlock: '198673',
// creationDate: '1559913567',
// data:
// '0xc710c57357d3862f351c00ff77a5ef90bb4491851f11c3e8ea010c16745c468e',
// description: null,
// disableSubVerifications: false,
// expirationDate: null,
// expired: false,
// id:
// '0x5ea689a7ed1d56d948dc8223dcd60866746bc7bea47617c19b63df75d63c9194',
// issuer: '0x6d2b20d6bf2B848D64dFE0B386636CDbFC521d4f',
// name: '/example1',
// rejectReason: undefined,
// signature:
// '0xf7ce3cc2f50ef62783ef293f8f45814b3ae868e614042cc05154853d00a694c176f8bdd94700736a137f92ff9a87639',
// status: 0,
// subject: '0x0030C5e7394585400B1FB193DdbCb45a37Ab916E',
// topic:
// '34884897835812838038558016063403566909277437558805531399344559176587016933548',
// uri:
// 'https://ipfs.test.evan.network/ipfs/
Qmbjig3cZbUUuFWqCEfzyCppqdnmQj3RoDjJWomnqYGylf',
// valid: true }]

const data = JSON.parse(await dfs.get(issued[0].data));
console.dir(data);
// Output:
// { foo: 'bar' }

```

## Encrypted Data in Verifications

Data added to the verification can be encrypted as well. Encryption is done outside of the verification service and has to be done before setting a verification and after getting the verification.

As key handling, storage and encryption itself is handled outside of the verification service, there are different ways for doing this. The suggested way to do this though, is using the [EncryptionWrapper](#). See the example below and its documentation for how it can be used.

```

const unencrypted = {foo: 'bar'};
const cryptoInfo = await encryptionWrapper.getCryptoInfo('test',
 EncryptionWrapperKeyType.Custom);
const key = await encryptionWrapper.generateKey(cryptoInfo);
const encrypted = await encryptionWrapper.encrypt(unencrypted, cryptoInfo, { key });

const verificationId = await verifications.setVerification(
 accounts[0], accounts[1], '/example1', 0, encrypted);
console.log(verificationId);
// Output:
// 0xdaa700acd52af1690c394445cc7908d01bef9a6c0c209dd4590cf869aa801586

```

(continues on next page)

(continued from previous page)

```

const issued = await verifications.getVerifications(accounts[1], '/example1');
console.dir(issued);
// Output:
// [{ creationBlock: '198706',
// creationDate: '1559915070',
// data:
// '0xb2eca508b635094d642950d3715783d744eac6771ff665303196040c6778cbc3',
// description: null,
// disableSubVerifications: false,
// expirationDate: null,
// expired: false,
// id:
// '0xdaa700acd52af1690c394445cc7908d01bef9a6c0c209dd4590cf869aa801586',
// issuer: '0x6d2b20d6bf2B848D64dFE0B386636CDbFC521d4f',
// name: '/example1',
// rejectReason: undefined,
// signature:
// '0x8ce1f239b254f2a4453e704cf5bd50f1aef215c5843408dc94ba3d128bba75d346a0b7945dd49f78b16cfd312ba51f6',
// status: 0,
// subject: '0x0030C5e7394585400B1FB193DdbCb45a37Ab916E',
// topic:
// '34884897835812838038558016063403566909277437558805531399344559176587016933548',
// uri:
// 'https://ipfs.test.evan.network/ipfs/
QmaP6Zyz2Mw4uBX1veuxQJSnvZnG3MLFxLGrPxbc2Y4pnn',
// valid: true }]

const retrieved = JSON.parse(await dfs.get(issued[0].data));
console.dir(retrieved);
// Output:
// { private:
// '017b0c07256180a69457f5c9a4e52431424532f698deaf401b754414bb070649',
// cryptoInfo:
// { algorithm: 'aes-256-cbc',
// block: 198705,
// originator: 'custom:test' } }

const decrypted = await encryptionWrapper.decrypt(retrieved, { key });
console.dir(decrypt);
// Output:
// { foo: 'bar' }

```

## 7.8 Verifications Examples - Full Output

This page shows the full output from the first nested verification example on *verification-usage-examples*.

```

[{ creationBlock: '188119',
 creationDate: 1558697814000,
 data:
 '0x00',

```

(continues on next page)

(continued from previous page)

[illegible]



(continued from previous page)

```

 status: 0,
 subject: '0x001De828935e8c7e4cb56Fe610495cAe63fb2612',
 topic:

→ '77917584097763165263567038986179239309321992426417843471841530443014147675245',
 uri: '',
 valid: true,
 displayName: 'child',
 parent: '/example4',
 warnings: ['issued', 'selfIssued', 'parentUntrusted'],
 subjectIdentity: '0xe560eF0954A2d61D6006E8547EC769fAc322bbCE',
 subjectType: 'account',
 issuerAccount: '0x001De828935e8c7e4cb56Fe610495cAe63fb2612',
 ensAddress:
 'ac43ca2dd23f1c08f2c2c6b22dcf233346588a5043ab7c4380656a2f7ac0146d.example4.
→ verifications.evan',
 topLevelEnsOwner: '0x00',
 parents:
 [{ creationBlock: '188117',
 creationDate: 1558697790000,
 data:
 '0x00',
 description:
 { author: '0x00',
 dbcpVersion: 1,
 description: '/example4',
 name: '/example4',
 version: '1.0.0',
 i18n: { name: { en: 'example4' } } },
 disableSubVerifications: false,
 expirationDate: null,
 expired: false,
 id:
 '0x18fb0ef05d96cba2a57c6de6d8cfd031e16367f6484f20797a39d25a3e76e20a',
 issuer: '0xe560eF0954A2d61D6006E8547EC769fAc322bbCE',
 name: '/example4',
 rejectReason: undefined,
 signature:

→ '0x081dec25292eeea7a969ec662a85b4160b8090227afff998c71d14d49f669e192569b4b76d5adbf4e7f9883bc560fdbb
→ ',
 status: 0,
 subject: '0x001De828935e8c7e4cb56Fe610495cAe63fb2612',
 topic:

→ '66865638538889962651480090943794875041354346360740147141539983678449951391661',
 uri: '',
 valid: true,
 displayName: 'example4',
 parent: '',
 warnings: ['issued'],
 subjectIdentity: '0xe560eF0954A2d61D6006E8547EC769fAc322bbCE',
 subjectType: 'account',
 issuerAccount: '0x001De828935e8c7e4cb56Fe610495cAe63fb2612',
 ensAddress:
 '93d49c39618452a1dcf4a52f91c7d50634851b5082233671e10070a3c73b3fad.
→ example4.verifications.evan',

```

(continues on next page)

(continued from previous page)

```

 topLevelEnsOwner: '0x00',
 parents: [],
 levelComputed:
 { creationDate: 1558697790000,
 disableSubVerifications: false,
 displayName: 'example4',
 loading: false,
 name: '/example4',
 status: 0,
 subjects: ['0x001De828935e8c7e4cb56Fe610495cAe63fb2612'],
 verifications: [Circular],
 warnings: ['issued'],
 description:
 { author: '0x00',
 dbcpVersion: 1,
 description: '/example4',
 name: '/example4',
 version: '1.0.0',
 i18n: { name: { en: 'example4' } } },
 ensAddress:
 '93d49c39618452a1dcf4a52f91c7d50634851b5082233671e10070a3c73b3fad.
→example4.verifications.evan',
 topLevelEnsOwner: '0x00',
 expirationDate: null } }],
 parentComputed:
 { creationDate: 1558697790000,
 disableSubVerifications: false,
 displayName: 'example4',
 loading: false,
 name: '/example4',
 status: 0,
 subjects: ['0x001De828935e8c7e4cb56Fe610495cAe63fb2612'],
 verifications:
 [{ creationBlock: '188117',
 creationDate: 1558697790000,
 data:
 '0x00
→',
 description:
 { author: '0x00',
 dbcpVersion: 1,
 description: '/example4',
 name: '/example4',
 version: '1.0.0',
 i18n: { name: { en: 'example4' } } },
 disableSubVerifications: false,
 expirationDate: null,
 expired: false,
 id:
 '0x18fb0ef05d96cba2a57c6de6d8cfd031e16367f6484f20797a39d25a3e76e20a
→',
 issuer: '0xe560eF0954A2d61D6006E8547EC769fAc322bbCE',
 name: '/example4',
 rejectReason: undefined,
 signature:

```

```

→ '0x081dec25292eaea7a969ec662a85b4160b8090227afff998c71d14d49f669e192569b4b76d5adbf4e7f9883bc560fdbb
→',

```

(continues on next page)

(continued from previous page)

```

 status: 0,
 subject: '0x001De828935e8c7e4cb56Fe610495cAe63fb2612',
 topic:

→ '66865638538889962651480090943794875041354346360740147141539983678449951391661',
 uri: '',
 valid: true,
 displayName: 'example4',
 parent: '',
 warnings: ['issued'],
 subjectIdentity: '0xe560eF0954A2d61D6006E8547EC769fAc322bbCE',
 subjectType: 'account',
 issuerAccount: '0x001De828935e8c7e4cb56Fe610495cAe63fb2612',
 ensAddress:
 '93d49c39618452a1dcf4a52f91c7d50634851b5082233671e10070a3c73b3fad.
→example4.verifications.evan',
 topLevelEnsOwner: '0x00',
 parents: [],
 levelComputed:
 { creationDate: 1558697790000,
 disableSubVerifications: false,
 displayName: 'example4',
 loading: false,
 name: '/example4',
 status: 0,
 subjects: ['0x001De828935e8c7e4cb56Fe610495cAe63fb2612'],
 verifications: [Circular],
 warnings: ['issued'],
 description:
 { author: '0x00',
 dbcpVersion: 1,
 description: '/example4',
 name: '/example4',
 version: '1.0.0',
 i18n: { name: { en: 'example4' } } },
 ensAddress:

→ '93d49c39618452a1dcf4a52f91c7d50634851b5082233671e10070a3c73b3fad.example4.
→verifications.evan',
 topLevelEnsOwner: '0x00',
 expirationDate: null } }],
 warnings: ['issued'],
 description:
 { author: '0x00',
 dbcpVersion: 1,
 description: '/example4',
 name: '/example4',
 version: '1.0.0',
 i18n: { name: { en: 'example4' } } },
 ensAddress:
 '93d49c39618452a1dcf4a52f91c7d50634851b5082233671e10070a3c73b3fad.
→example4.verifications.evan',
 topLevelEnsOwner: '0x00',
 expirationDate: null },
 levelComputed:
 { creationDate: 1558697805000,
 disableSubVerifications: false,

```

(continues on next page)

(continued from previous page)

```

 displayName: 'child',
 loading: false,
 name: '/example4/child',
 status: 0,
 subjects: ['0x001De828935e8c7e4cb56Fe610495cAe63fb2612'],
 verifications: [Circular],
 warnings: ['issued', 'selfIssued', 'parentUntrusted'],
 description:
 { author: '0x00',
 dbcpVersion: 1,
 description: '/example4/child',
 name: '/example4/child',
 version: '1.0.0',
 il8n: { name: { en: 'child' } } },
 ensAddress:
 'ac43ca2dd23f1c08f2c2c6b22dcf233346588a5043ab7c4380656a2f7ac0146d.
↪example4.verifications.evan',
 topLevelEnsOwner: '0x00',
 expirationDate: null } }],
parentComputed:
{ creationDate: 1558697805000,
 disableSubVerifications: false,
 displayName: 'child',
 loading: false,
 name: '/example4/child',
 status: 0,
 subjects: ['0x001De828935e8c7e4cb56Fe610495cAe63fb2612'],
 verifications:
 [{ creationBlock: '188118',
 creationDate: 1558697805000,
 data:
 '0x00',
 description:
 { author: '0x00',
 dbcpVersion: 1,
 description: '/example4/child',
 name: '/example4/child',
 version: '1.0.0',
 il8n: { name: { en: 'child' } } },
 disableSubVerifications: false,
 expirationDate: null,
 expired: false,
 id:
 '0x28e1df758883bb3d4d5e7e0fa978ff673bc749ade0a3d78ad952a30d0a0e2a01',
 issuer: '0xe560eF0954A2d61D6006E8547EC769fAc322bbCE',
 name: '/example4/child',
 rejectReason: undefined,
 signature:
 '0x568770823b2b541c24deddfa403f82f883ac73cbdaae99d3b4a112bb36ec391d374c824f15e279474f072f180b0aa94
↪',
 status: 0,
 subject: '0x001De828935e8c7e4cb56Fe610495cAe63fb2612',
 topic:
 '77917584097763165263567038986179239309321992426417843471841530443014147675245',
 uri: ''

```

(continues on next page)

(continued from previous page)

```

 valid: true,
 displayName: 'child',
 parent: '/example4',
 warnings: ['issued', 'selfIssued', 'parentUntrusted'],
 subjectIdentity: '0xe560eF0954A2d61D6006E8547EC769fAc322bbCE',
 subjectType: 'account',
 issuerAccount: '0x001De828935e8c7e4cb56Fe610495cAe63fb2612',
 ensAddress:
 'ac43ca2dd23f1c08f2c2c6b22dcf233346588a5043ab7c4380656a2f7ac0146d.
↪example4.verifications.evan',
 topLevelEnsOwner: '0x00',
 parents:
 [{ creationBlock: '188117',
 creationDate: 1558697790000,
 data:
 '0x00
↪',
 description:
 { author: '0x00',
 dbcpVersion: 1,
 description: '/example4',
 name: '/example4',
 version: '1.0.0',
 i18n: { name: { en: 'example4' } } },
 disableSubVerifications: false,
 expirationDate: null,
 expired: false,
 id:
 '0x18fb0ef05d96cba2a57c6de6d8cfd031e16367f6484f20797a39d25a3e76e20a
↪',
 issuer: '0xe560eF0954A2d61D6006E8547EC769fAc322bbCE',
 name: '/example4',
 rejectReason: undefined,
 signature:

↪'0x081dec25292eeea7a969ec662a85b4160b8090227afff998c71d14d49f669e192569b4b76d5adbf4e7f9883bc560fdb5
↪',
 status: 0,
 subject: '0x001De828935e8c7e4cb56Fe610495cAe63fb2612',
 topic:

↪'66865638538889962651480090943794875041354346360740147141539983678449951391661',
 uri: '',
 valid: true,
 displayName: 'example4',
 parent: '',
 warnings: ['issued'],
 subjectIdentity: '0xe560eF0954A2d61D6006E8547EC769fAc322bbCE',
 subjectType: 'account',
 issuerAccount: '0x001De828935e8c7e4cb56Fe610495cAe63fb2612',
 ensAddress:
 '93d49c39618452a1dcf4a52f91c7d50634851b5082233671e10070a3c73b3fad.
↪example4.verifications.evan',
 topLevelEnsOwner: '0x00',
 parents: [],
 levelComputed:
 { creationDate: 1558697790000,

```

(continues on next page)

(continued from previous page)

```

 disableSubVerifications: false,
 displayName: 'example4',
 loading: false,
 name: '/example4',
 status: 0,
 subjects: ['0x001De828935e8c7e4cb56Fe610495cAe63fb2612'],
 verifications: [Circular],
 warnings: ['issued'],
 description:
 { author: '0x00',
 dbcpVersion: 1,
 description: '/example4',
 name: '/example4',
 version: '1.0.0',
 i18n: { name: { en: 'example4' } } },
 ensAddress:

 ↪ '93d49c39618452a1dcf4a52f91c7d50634851b5082233671e10070a3c73b3fad.example4.
 ↪ verifications.evan',
 topLevelEnsOwner: '0x00',
 expirationDate: null } }],
 parentComputed:
 { creationDate: 1558697790000,
 disableSubVerifications: false,
 displayName: 'example4',
 loading: false,
 name: '/example4',
 status: 0,
 subjects: ['0x001De828935e8c7e4cb56Fe610495cAe63fb2612'],
 verifications:
 [{ creationBlock: '188117',
 creationDate: 1558697790000,
 data:

 ↪ '0x00',
 description:
 { author: '0x00',
 dbcpVersion: 1,
 description: '/example4',
 name: '/example4',
 version: '1.0.0',
 i18n: { name: { en: 'example4' } } },
 disableSubVerifications: false,
 expirationDate: null,
 expired: false,
 id:

 ↪ '0x18fb0ef05d96cba2a57c6de6d8cfd031e16367f6484f20797a39d25a3e76e20a',
 issuer: '0xe560eF0954A2d61D6006E8547EC769fAc322bbCE',
 name: '/example4',
 rejectReason: undefined,
 signature:

 ↪ '0x081dec25292eeea7a969ec662a85b4160b8090227afff998c71d14d49f669e192569b4b76d5adbf4e7f9883bc560fdb',
 ↪ '
 status: 0,
 subject: '0x001De828935e8c7e4cb56Fe610495cAe63fb2612',

```

(continues on next page)

(continued from previous page)

```

 topic:

↳ '66865638538889962651480090943794875041354346360740147141539983678449951391661',
 uri: '',
 valid: true,
 displayName: 'example4',
 parent: '',
 warnings: ['issued'],
 subjectIdentity: '0xe560eF0954A2d61D6006E8547EC769fAc322bbCE',
 subjectType: 'account',
 issuerAccount: '0x001De828935e8c7e4cb56Fe610495cAe63fb2612',
 ensAddress:

↳ '93d49c39618452a1dcf4a52f91c7d50634851b5082233671e10070a3c73b3fad.example4.
↳ verifications.evan',
 topLevelEnsOwner: '0x00',
 parents: [],
 levelComputed:
 { creationDate: 1558697790000,
 disableSubVerifications: false,
 displayName: 'example4',
 loading: false,
 name: '/example4',
 status: 0,
 subjects: ['0x001De828935e8c7e4cb56Fe610495cAe63fb2612'],
 verifications: [Circular],
 warnings: ['issued'],
 description:
 { author: '0x00',
 dbcpVersion: 1,
 description: '/example4',
 name: '/example4',
 version: '1.0.0',
 i18n: { name: { en: 'example4' } } },
 ensAddress:

↳ '93d49c39618452a1dcf4a52f91c7d50634851b5082233671e10070a3c73b3fad.example4.
↳ verifications.evan',
 topLevelEnsOwner: '0x00',
 expirationDate: null } }],
 warnings: ['issued'],
 description:
 { author: '0x00',
 dbcpVersion: 1,
 description: '/example4',
 name: '/example4',
 version: '1.0.0',
 i18n: { name: { en: 'example4' } } },
 ensAddress:
 '93d49c39618452a1dcf4a52f91c7d50634851b5082233671e10070a3c73b3fad.
↳ example4.verifications.evan',
 topLevelEnsOwner: '0x00',
 expirationDate: null },
 levelComputed:
 { creationDate: 1558697805000,
 disableSubVerifications: false,
 displayName: 'child',

```

(continues on next page)

(continued from previous page)

```

 loading: false,
 name: '/example4/child',
 status: 0,
 subjects: ['0x001De828935e8c7e4cb56Fe610495cAe63fb2612'],
 verifications: [Circular],
 warnings: ['issued', 'selfIssued', 'parentUntrusted'],
 description:
 { author: '0x00',
 dbcpVersion: 1,
 description: '/example4/child',
 name: '/example4/child',
 version: '1.0.0',
 i18n: { name: { en: 'child' } } },
 ensAddress:
 'ac43ca2dd23f1c08f2c2c6b22dcf233346588a5043ab7c4380656a2f7ac0146d.
↪example4.verifications.evan',
 topLevelEnsOwner: '0x00',
 expirationDate: null } }],
 warnings: ['issued', 'selfIssued', 'parentUntrusted'],
 description:
 { author: '0x00',
 dbcpVersion: 1,
 description: '/example4/child',
 name: '/example4/child',
 version: '1.0.0',
 i18n: { name: { en: 'child' } } },
 ensAddress:
 'ac43ca2dd23f1c08f2c2c6b22dcf233346588a5043ab7c4380656a2f7ac0146d.example4.
↪verifications.evan',
 topLevelEnsOwner: '0x00',
 expirationDate: null },
 levelComputed:
 { creationDate: 1558697814000,
 disableSubVerifications: false,
 displayName: 'grandchild',
 loading: false,
 name: '/example4/child/grandchild',
 status: 0,
 subjects: ['0x001De828935e8c7e4cb56Fe610495cAe63fb2612'],
 verifications: [Circular],
 warnings: ['issued', 'selfIssued', 'parentUntrusted'],
 description:
 { author: '0x00',
 dbcpVersion: 1,
 description: '/example4/child/grandchild',
 name: '/example4/child/grandchild',
 version: '1.0.0',
 i18n: { name: { en: 'grandchild' } } },
 ensAddress:
 '00b41b33a2396efb6fd434bc67d84b47a32eee9a9a275ee6384f5c32f2b12b93.example4.
↪verifications.evan',
 topLevelEnsOwner: '0x00',
 expirationDate: null } }]

```



## 7.9 DID

|            |             |
|------------|-------------|
| Class Name | Did         |
| Extends    | Logger      |
| Source     | did.ts      |
| Tests      | did.spec.ts |

The *Did* module allows to interact with DIDs on *evan.network*. As development of identity and DID handling on *evan.network* is an ongoing process, this document describes the current interoperability of DIDs on *evan.network* and can be seen as a work-in-progress state of the current implementation.

### 7.9.1 constructor

```
new Did(options);
```

Creates a new *Did* instance.

#### Parameters

##### 1. options - DidOptions: options for Did constructor.

- `contractLoader` - *ContractLoader*: *ContractLoader* instance
- `dfs` - *DfsInterface*: *DfsInterface* instance
- `executor` - *Executor*: *Executor* instance
- `nameResolver` - *NameResolver*: *NameResolver* instance
- `signerIdentity` - *SignerIdentity*: *SignerIdentity* instance
- `web3` - *Web3*: *Web3* instance
- `log` - *Function* (optional): function to use for logging: `(message, level) => {...}`
- `logLevel` - *LogLevel* (optional): messages with this level will be logged with `log`
- `logLog` - *LogLogInterface* (optional): container for collecting log messages
- `logLogLevel` - *LogLevel* (optional): messages with this level will be pushed to `logLog`

#### Returns

Did instance

#### Example

```
const did = new Did({
 contractLoader,
 dfs,
 executor,
 nameResolver,
```

(continues on next page)

(continued from previous page)

```
 signerIdentity,
 web3,
 });
```

## 7.9.2 = Working with DID documents =

### 7.9.3 setDidDocument

```
did.setDidDocument(did, document);
```

Store given DID document for given DID.

#### Parameters

1. `did - string`: DID to store DID document for
2. `document - any`: DID document to store, `getDidDocumentTemplate` can be used as a starting point for DID documents

#### Returns

Promise returns void: resolved when done

#### Example

```
const identity = await runtime.verifications.getIdentityForAccount(accountsId, true);
const did = await runtime.did.convertIdentityToDid(identity);
const document = await runtime.did.getDidDocumentTemplate();
await runtime.did.setDidDocument(did, document);
```

### 7.9.4 getDidDocument

```
did.getDidDocument([did]);
```

Get DID document for given DID.

#### Parameters

1. `did - string`: DID to fetch DID document for.

#### Returns

Promise returns any: a DID document that MAY resemble *DidDocumentTemplate* format

## Example

```
const identity = await runtime.verifications.getIdentityForAccount(accountsId, true);
const did = await runtime.did.convertIdentityToDid(identity);
const document = await runtime.did.getDidDocumentTemplate();
await runtime.did.setDidDocument(did, document);
const retrieved = await runtime.did.getDidDocument(did);
```

## 7.9.5 setService

```
did.setService(service[, did]);
```

Sets service in DID document.

### Parameters

1. did - string: DID name to set service for
2. service - DidServiceEntry[] | DidServiceEntry: service to set

### Returns

Promise returns void: resolved when done

## Example

```
const document = await runtime.did.getDidDocumentTemplate();
const identity = await runtime.verifications.getIdentityForAccount(account, true);
const did = await runtime.did.convertIdentityToDid(identity);
await runtime.did.setDidDocument(did, document);
const service = [{
 id: `${did}#randomService`,
 type: `randomService-${random}`,
 serviceEndpoint: `https://openid.example.com/${random}`,
}];
await runtime.did.setService(did, service);
```

## 7.9.6 getService

```
did.getService([did]);
```

Get service from DID document.

### Parameters

1. did - string: DID to fetch DID service for.

## Returns

```
Promise returns DidServiceEntry[] | DidServiceEntry: service
```

### Example

```
const document = await runtime.did.getDocumentTemplate();
const identity = await runtime.verifications.getIdentityForAccount(account, true);
const did = await runtime.did.convertIdentityToDid(identity);
await runtime.did.setDidDocument(did, document);
const service = [{
 id: `${did}/#randomService`,
 type: `randomService-${random}`,
 serviceEndpoint: `https://openid.example.com/${random}`,
}];
await runtime.did.setService(did, service);
const retrieved = await runtime.did.getService(did);
```

### 7.9.7 = utilities =

### 7.9.8 convertDidToIdentity

```
did.convertDidToIdentity(did);
```

Converts given DID to a `evan.network` identity.

## Parameters

- [illegible]

## Returns

[illegible]

### Example

[illegible]

### 7.9.9 convertIdentityToDid



```
const document = await runtime.did.getDidDocumentTemplate();
console.log(JSON.stringify(document, null, 2));
// Output:
// {
// "@context": "https://w3id.org/did/v1",
// "id": "did:evan:testcore:0x126E901F6F408f5E260d95c62E7c73D9B60fd734",
// "publicKey": [
// {
// "id": "did:evan:testcore:0x126E901F6F408f5E260d95c62E7c73D9B60fd734#key-1",
// "type": [
// "Secp256k1SignatureVerificationKey2018",
// "ERC725ManagementKey"
//],
// "publicKeyHex":
→ "045adfd502c0bc55f4fcb90eea36368d7e19c5b3045aa6f51dfa3699046e9751251d21bc6bdd06c1ff0014fcbbf9f1d83
→ "
// }
//],
// "authentication": [
// "did:evan:testcore:0x126E901F6F408f5E260d95c62E7c73D9B60fd734#key-1"
//]
// }
```

## 7.10 VC

| Class Name | Vc         |
|------------|------------|
| Extends    | Logger     |
| Source     | vc.ts      |
| Tests      | vc.spec.ts |

The *Vc* module allows to create, store, retrieve and revoke VCs on evan.network. As development of identities, and DID and VC handling on evan.network is an ongoing process, this document describes the current interoperability of VCs on evan.network and can be seen as a work-in-progress state of the current implementation.

### 7.10.1 constructor

```
new Vc(options, did);
```

Creates a new *Vc* instance.

#### Parameters

##### 1. options - VcOptions: options for Vc constructor.

- *accountStore* - *AccountStore*: *AccountStore* instance
- *activeAccount* - string: ID of the active account
- *contractLoader* - *ContractLoader*: *ContractLoader* instance
- *dfs* - *DfsInterface*: *DfsInterface* instance

- `did` - `Did: Did` instance for resolving and validating
- `executor` - `Executor: Executor` instance
- `nameResolver` - `NameResolver: NameResolver` instance
- `signerIdentity` - `SignerIdentity: SignerIdentity` instance
- `verifications` - `Verifications: Verifications` instance
- `web3` - `Web3: Web3` instance
- `log` - `Function (optional)`: function to use for logging: `(message, level) => {...}`
- `logLevel` - `LogLevel` (optional): messages with this level will be logged with `log`
- `logLog` - `LogLogInterface` (optional): container for collecting log messages
- `logLogLevel` - `LogLevel` (optional): messages with this level will be pushed to `logLog`

## 2. `config - VcConfig`: custom configuration for `Vc` constructor.

- `credentialStatusEndpoint` - `string`: URL of the credential status endpoint

## Returns

`Vc` instance

## Example

```
const vc = new Vc(
 {
 accountStore,
 activeIdentity
 contractLoader,
 dfs,
 did,
 executor,
 nameResolver,
 signerIdentity,
 verifications,
 web3,
 },
 { credentialStatusEndpoint },
);
```

## 7.10.2 = Working with VC documents =

### 7.10.3 `createId`

Claim a new ID in the VC registry which can be used later to store a VC **on-chain**.

```
vc.createId();
```

## Returns

Promise **returns** string: A new ID string

## Example

```
const newRegisteredId = await runtime.vc.createId();
const myVcDocument = {
 // Data here,
 id: newRegisteredId
};
await runtime.vc.storeVc(myVcDocument);
```

---

### 7.10.4 createVc

Create a signed **off-chain** VC document

```
vc.createVc(vcData);
```

## Parameters

1. `vcData` - *DocumentTemplate*: Collection of mandatory and optional VC properties to store in the VC document

## Returns

Promise **returns** VcDocument: The final VC document

## Example

```
const minimalVcData = {
 id: 'randomCustomId',
 issuer: {
 did: 'someDid',
 },
 credentialSubject: {
 did: 'someOtherDid',
 },
 validFrom: new Date(Date.now()).toISOString()
};
const offchainVc = await runtime.vc.createVc(minimalVcData);
```

---

### 7.10.5 getVc

Get VC document for given VC ID.



```
vc.getVc(vcId, encryptionInfo);
```

### Parameters

1. `vcId` - string: ID to fetch VC document for. Can be either a full VC URI (starting with `vc:evan:`) or just the VC ID (starting with `0x`)
2. `encryptionInfo` - *EncryptionInfo*: (optional): Information required for decryption

### Returns

Promise returns `VcDocument`: A VC document

### Example

```
const storedVcDoc = await vc.getVc(
 ↪ '0x2a838a6961be98f6a182f375bb9158848ee9760ca97a379939ccdf03fc442a23');
const otherStoredVcDoc = await vc.getVc(
 ↪ 'vc:evan:testcore:0x2a838a6961be98f6a182f375bb9158848ee9760ca97a379939ccdf03fc442a23
 ↪ ');

// using encryption
encryptionInfo = { key: vcKey };
const EncryptedVcDoc = await vc.getVc(
 ↪ 'vc:evan:testcore:0x5f7514378963d3a1211a3b015c51dd9fbd1e52d66a2fbb411fcdf80fd7b7bd4
 ↪ ', encryptionInfo);
```

## 7.10.6 storeVc

```
vc.storeVc(vcData, encryptionInfo);
```

Create a new VC that holds the given data and **store it on the chain**. Whether a new ID should be registered with the VC registry or the given ID in the document should be used depends of if `vcData.id` is set. If set, the method calls `createId()` to generate a new ID.

### Parameters

1. `vcData` - *DocumentTemplate*: Collection of mandatory and optional VC properties to store in the VC document
2. `encryptionInfo` - *EncryptionInfo*: (optional): Information required for encryption

### Returns

Promise returns `VcDocument`: Returns the VC document as stored on the chain.

## Example

```
const minimalVcData = {
 issuer: {
 did: 'someDid',
 },
 credentialSubject: {
 did: 'someOtherDid',
 },
 validFrom: new Date(Date.now()).toISOString()
};
const createdVcDoc = await runtime.vc.storeVc(minimalVcData);
const permanentVcAddress = createdVcDoc.id;
```

```
const myRegisteredId = await runtime.vc.createId();
const minimalVcData = {
 issuer: {
 did: 'someDid',
 },
 credentialSubject: {
 did: 'someOtherDid'
 },
 validFrom: new Date(Date.now()).toISOString()
};
minimalVcData.id = myRegisteredId;
const createdVcDoc = await runtime.vc.storeVc(minimalVcData);
const permanentVcAddress = createdVcDoc.id;
```

---

## 7.10.7 revokeVc

```
vc.revokeVc(vcId);
```

Sets a revoke status flag for the VC.

### Parameters

1. vcId - string: ID for VC document to be revoked.

### Returns

Promise returns void: resolved when done

## Example

```
const storedVcDoc = await vc.getVc(permanentVcAddress);
const vcId = storedVcDoc.id;

const revokeProcessed = await vc.revokeVc(vcId);
```

### 7.10.8 getRevokeVcStatus

```
vc.getRevokeVcStatus(vcId);
```

Gets the revoke status flag for the VC.

#### Parameters

1. `vcId` - string: ID for VC document whose status needs to be retrieved.

#### Returns

Promise returns bool: true for revoked, false for not revoked

#### Example

```
const storedVcDoc = await vc.getVc(permanentVcAddress);
const vcId = storedVcDoc.id;

const vcRevokeStatus = await vc.getRevokeVcStatus(vcId);
```

## 7.10.9 Additional Components

### 7.10.10 Interfaces

#### EncryptionInfo

configuration settings required for the encryption and decryption

1. `key` - string: the encryption key required for encrypting and decrypting the VC

#### DocumentTemplate

Template for the VC document containing the relevant data

1. `id` - string: the id of the VC
2. `type` - string: set of unordered URIs
3. `issuer` - *VcIssuer*: VC issuer details
4. `validFrom` - string: date from which the VC is valid
5. `validUntil` - string (optional): date until which the VC is valid
6. `credentialSubject` - *VcCredentialSubject*: subject details of VC
7. `credentialStatus` - *VcCredentialStatus* (optional): details regarding the status of VC
8. `proof` - *VcProof* (optional): proof of the respective VC

### VcIssuer

Template for the VC Issuer containing the relevant data

1. `id-string`: the id of the issuer
2. `name-string` (optional): name of the issuer

### VcCredentialSubject

Template for the VC credential subject containing the relevant data

1. `id-string`: the id of the subject
2. `data-VcCredentialSubjectPayload` (optional): data payload for subject
3. `description-string` (optional): description about subject
4. `uri-string` (optional): uri of subject

### VcCredentialStatus

Template for the VC credential status containing the status data

1. `id-string`: the id of the VC
2. `type-string`: VC status type

### VcProof

proof for VC, contains JWS and metadata

1. `type-string`: VC status type
2. `created-string`: date when the proof was created
3. `proofPurpose-string`: purpose of the proof
4. `verificationmethod-string`: method used for verification
5. `jws-string`: JSON Web Signature

---

Profiles are personal data for accounts. They can be shared with other accounts or used for storing own data. This section contains modules for maintaining profile and interacting with profiles from other accounts.

Two types of profiles are supported:

- **personal profiles**, that hold users data like contacts, bookmarks, encryption keys
- **business center profiles**, that are like contact cards and hold data intended for other participants